KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ INŽINERIJOS STUDIJŲ PROGRAMA

GRETA KRIVICKAITĖ

UML BŪSENŲ MODELIO TAIKYMAS PROGRAMŲ TESTAVIMUI

Magistro baigiamasis darbas

Darbo vadovas
dr. E. Bareiša

KAUNAS, 2013

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ INŽINERIJOS STUDIJŲ PROGRAMA

GRETA KRIVICKAITĖ

UML BŪSENŲ MODELIO TAIKYMAS PROGRAMŲ TESTAVIMUI

Magistro baigiamasis darbas

Darbo vadovas
dr. E. Bareiša

Recenzentas:
dr. A. Ušaniov
2013-05-

Atliko:
IFM-1/2 gr. studentė
Greta Krivickaitė
2013-05

KAUNAS, 2013

AUTORIŲ GARANTINIS RAŠTAS

DĖL PATEIKIAMO KŪRINIO

2013 - 06 - 01 d.
Kaunas

Autoriai, GretaKrivickaitė
(vardas, pavardė)

,

patvirtina, kad Kauno technologijos universitetui pateiktas baigiamasis magistro darbas (toliau vadinama – Kūrinys) UML būsenų modelio taikymas programų testavimui
(kūrinio pavadinimas)

pagal Lietuvos Respublikos autorių ir gretutinių teisių įstatymą yra originalus ir užtikrina, kad

1) jį sukūrė ir parašė Kūrinyje įvardyti autoriai;
2) Kūrinys nėra ir nebus įteiktas kitoms institucijoms (universitetams) (tiek lietuvių, tiek užsienio kalba);
3) Kūrinyje nėra teiginių, neatitinkančių tikrovės, ar medžiagos, kuri galėtų pažeisti kito fizinio ar juridinio asmens intelektinės nuosavybės teises, leidėjų bei finansuotojų reikalavimus ir sąlygas;
4) visi Kūrinyje naudojami šaltiniai yra cituojami (su nuoroda į pirminį šaltinį ir autorių);
5) neprieštarauja dėl Kūrinio platinimo visomis oficialiomis sklaidos priemonėmis.
6) atlygins Kauno technologijos universitetui ir tretiesiems asmenims žalą ir nuostolius, atsiradusius dėl pažeidimų, susijusių su aukščiau išvardintų Autorių garantijų nesilaikymu;
7) Autoriai už šiame rašte pateiktos informacijos teisingumą atsako Lietuvos Respublikos įstatymų nustatyta tvarka.

Autoriai

Greta Krivickaitė		
(vardas, pavardė)	(parašas)	
(vardas, pavardė)	(parašas)	
(vardas, pavardė)	(parašas)	
(vardas, pavardė)	(parašas)	

SANTRAUKA

Programinės įrangos testavimas gali užimti iki 50% projekto laiko, tad yra automatizuojamas. Norint generuoti automatinius būsenų testus, reikia turėti testavimo atvejus ir metodą įvertinti gautų rezultatų teisingumo įvertinimui. Šiame darbe yra aprašoma, kaip galima generuoti automatinius testus taikant UML būsenų modelį.

UML būsenų modeliu galima modeliuoti klases dinaminiu požiūriu, nusakant galimas objekto būsenas ir perėjimus tarp jų. Yra siūloma, testavimo atvejų generavimui naudoti būsenų diagramose aprašytus perėjimus ir jų apribojimus, o šių sekų rezultatų įvertinimui taikyti būsenoms nurodytas sąlygas, kurias turi tenkinti objektas esant toje būsenoje.

Tokiu principu buvo suprojektuotas ir realizuotas įrankis, leidžiantis generuoti automatinius testus, kur nereikėtų keisti realizuotų klasių struktūros, kadangi būsenų sąlygoms tikrinti būtų naudojama refleksija, o kviečiamų metodų sekų generavimui taikomas Deikstros algoritmas, leidžiantis aplankyti visas būsenas ir kelius. Be to, testuojamų metodų kvietimui, būtų taikoma ribinių reikšmių analizė, kad būtų padengiama kuo daugiau kodo ir aptinkama daugiau klaidų.

Taip pat, buvo ištirta kiek kodo padengia ir kiek neteisingų programos realizacijų aptinka tokiu būdu sugeneruoti testai. Buvo nustatyta, kad tokiu būdu galima padengti net 100 % kodo ir aptikti visus mutantus, tačiau yra svarbu, kad būsenų modelis nusakytų visas galimas sistemos būsenas ir perėjimus tarp jų.

SUMMARY

Software testing can take up to 50 % of the project time. Thus, testing process is being automated. In order to generate automatic tests, you need to have test cases and a method to evaluate results of those test cases. This work describes how is it possible to generate automatic tests using UML state machine model.

UML state machine model allows to model dynamic behaviour of a class/system, by defining possible object‘s states and transitions among them. It is being proposed to use transitions and their constraints for test cases generation and use constraints specified in states for evaluations of results of test cases.

By using such technique, a tool was designed and realized allowing to generate automatic tests, where checking of constraints is done by using reflection, which does not require altering of already created classes, nor revealing of any fields of classes. Also, to generate test cases Deijkstra’s algorithm is used, which allows visiting all states and transitions. Moreover, boundary value analysis is used to generate test data for tested methods invocation, allowing to cover more code and detect more defects.

Also, this tool and technique was evaluated to check how much code it is possible to cover and how many incorrect program realizations tests are able to detect. It was determined that there is a possibility to cover 100 % of code and detect all incorrect program realizations. However, with the intent to achieve this, it is very important that state chart model would describe all possible system’s states and transitions.

TURINYS

PAVEIKSLŲ SĄRAŠAS

LENTELIŲ SĄRAŠAS

TERMINŲ IR SANTRUMPŲ ŽODYNAS

MDD (*angl. Model Driven Development*) – modeliu paremtas programavimas

UML (*angl. Unified Modeling Language*) – vieninga modeliavimo kalba

xml (*angl. Extensible Markup Language*) – bendros paskirties duomenų struktūrų bei jų turinio aprašomoji kalba

XMI (*angl. XML Metadata Interchange*) – formatas skirtas XML meta duomenų keitimuisi

OCL (*angl. Object Constraint Language*) – kalba, skirta aprašyti taisyklėms ir apribojimams, kurie taikomi UML modelio elementams

JUnit – testavimo karkasas Java programavimo kalbai

Refleksija – galimybė dinamiškai vykdyti klases, kviečiant metodus, kintamuosius netiesiogiai, o pagal pavadinimą. Žemiau pateikiamas Java kodo pavyzdys.

```
public class Lamp {
   public boolean ijungta = false;

   public void on() {
        ijungta = true;
   }

   public void off() {
        ijungta = false;
   }

   public static void main(String[] args) throws Exception {
        Class<?> clazz = Class.forName("Lamp");
        Object o = clazz.newInstance();
        Field f = o.getClass().getDeclaredField("ijungta");
        f.setAccessible(true);
        // ijungta kintamojo reiksmes gavimas ir spausdinimas
        System.out.println(f.get(o));
        // metodo on iskvietimas
        clazz.getMethod("on").invoke(o);
        // ijungta kintamojo reiksmes gavimas ir spausdinimas
        System.out.println(f.get(o));
   }
}
```

1. ĮVADAS

Šis darbas priklauso Magistrantūros studijų Programų sistemų inžinerijos specializacijai. Toliau šiame skyriuje bus pateikiama darbo problematika, uždaviniai ir aptariama darbo struktūra.

1.1. Darbo problematika ir aktualumas

Programinės įrangos kūrimas yra sudėtingas procesas, nes programinę įrangą yra labai lengva keisti ir šio proceso rezultatai nėra matomi iškart [1, p. 3]. O testavimas yra viena iš šio proceso dalių, kurio metu yra tikrinama sukurtos programinės įrangos kokybė ir atitikimas reikalavimus. Tad testavimas yra labai svarbi kūrimo proceso dalis, nes dėl testavimo metu neaptiktų klaidų, galima sulaukti daug vartotojų nusiskundimų arba net projektas gali žlugti.

Kadangi testavimas gali užimti iki 50% projekto laiko [2, p. 344], šį procesą stengiamasi automatizuoti. Tačiau, norint atlikti testavimą, reikia turėti testavimo atvejus ir metodą, rezultatų teisingumo patikrinimui, o taikant atsitiktinį generavimą negalima užtikrinti testų efektyvumo. Tad yra siūloma taikyti UML būsenų modelį automatinių testų generavimui, kuris padėtų sumažinti automatinių testų generavimo kaštus ir kurti geresnę programinę įrangą.

1.2. Darbo tikslas ir uždaviniai

Šio darbo tikslas yra sukurti ir įvertinti metodą, kuris leistų generuoti automatinius testus taikant UML būsenų modelį vartotojui turint UML būsenų modelį ir realizuotą klasę. Pagrindiniai uždaviniai:

1. Sukurti algoritmą parenkantį testavimo sekas taikant UML būsenų modelį
2. Parinkti metodą, kaip galima parinkti duomenis testų metodų kvietimui, taikant ribinių reikšmių analizę
3. Sukurti metodą, leidžiantį aplankyti visas būsenas ir kelius
4. Sukurti metodą, leidžiantį įvertinti atliktų testavimo sekų rezultatų teisingumą
5. Įvertinti sugeneruotų automatinių testų efektyvumą, taikant mutacinį testavimą
6. Nustatyti, kurie parametrai labiausiai įtakoja sugeneruotų testų efektyvumą

1.3. Darbo rezultatai ir jų svarba

Šio darbo rezultatas bus metodas, leidžiantis generuoti automatinius testus, taikant UML būsenų modelį. Šis metodas palengvintų automatinių testų kūrimą, nes būtų galima generuoti testus iš UML būsenų diagramų, kuriose būtų nurodytos galimos sistemos būsenos bei perėjimai tarp jų. Tokiu būdu generuojant testus būtų išspręstos 2 problemos: testavimo atvejų parinkimas ir šių rezultatų teisingumo įvertinimas. Tokiu būdu, būtų galima sutaupyti laiko kuriant automatinius testus, nes užtektų turėti realizuotą klasę ir UML būsenų diagramą, kurioje nurodytos sistemos būsenos bei perėjimai tarp jų.

1.4. Darbo struktūra

Šis darbas susideda iš panaudotų terminų žodyno, įvado, keturių pagrindinių dalių, išvadų, panaudotos literatūros sąrašo ir priedų:

- Terminų ir santrumpų žodyne pateikiamos pagrindinės sąvokos naudotos šiame darbe.
- Įvade pateikiamos darbo problemos, aktualumas ir šio darbo struktūra.

- Probleminės srities analizės skyriuje yra apžvelgiamas UML būsenų modelio taikymas automatinių testų generavimui ir aptariami įrankiai skirti automatinių testų generavimui naudojant UML būsenų modelį.
- Projektinėje dalyje yra pateikiami esminiai reikalavimai ir architektūriniai sprendimai įgalinantys automatinių testų generavimą, taikant UML būsenų modelį.
- Sprendimo realizacijos ir testavimo skyriuje pateikiami projektinėje dalyje sukurto sprendimo realizacijos testavimo rezultatai ir aptariami atlikti patobulinimai.
- Eksperimentinio tyrimo skyrelyje pateikiami ir aptariami rezultatai eksperimento, kurio metu buvo generuojami testai 5 realizuotoms klasėms taikant UML būsenų modelį.
- Išvadų skyriuje pateikiamos pagrindinės šio darbo išvados.
- Literatūros skyriuje išvardinami pagrindiniai šio darbo šaltiniai.
- Prieduose yra pateikiami eksperimento metu naudoti duomenys.

2. PROBLEMINĖS SRITIES ANALIZĖ

Šiame skyriuje bus apžvelgiamas UML būsenų modelio taikymas automatinių testų generavimui ir aptariami įrankiai skirti automatinių testų generavimui naudojant UML būsenų modelį.

2.1. Analizės tikslas

Programinės įrangos kūrimas yra sudėtingas procesas. Jį apsunkina tai, kad projekto progreso rezultatai nuo pat pradžių yra nematomi iki tam tikros ribos, o pavyzdžiui pradėjus statyti namą – galima iškart matyti padėtus pamatus. Taip pat kitas sunkinantis dalykas yra tai kad programinę įrangą yra lengva keisti, dėl ko galima nesunkiai įvelti klaidų, pavyzdžiui sistema neatitinka reikalavimų, dėl to projektas užtrunka ilgiau ar net žlunga [1, p. 3].

Norint kurti kuo kokybiškesnę programinę įrangą su kuo mažiau klaidų, kūrimo procesas yra planuojamas ir taikomi įvairūs modeliai. Pagal vieną iš jų - Krioklio modelį programinės įrangos kūrimo procesas susideda iš kelių dalių: reikalavimų rinkimo, architektūros kūrimo, programavimo, testavimo ir palaikymo [1, p. 6]. Pirmiausia surenkami reikalavimai, tada kuriama sistemos architektūra. Ją galima aprašyti taikant Unifikuotą Modeliavimo kalbą (toliau – UML). Šia kalba galima modeliuoti sistemas tiek statiniu, tiek dinaminiu požiūriu, pavyzdžiui klasių diagramomis, būsenų diagramomis ir kitomis. Tuomet turint reikalavimus, sistemos modelį ir realizuotą sistemą galima vykdyti testavimą.

Testavimas yra veikla atliekama norint įvertinti produkto kokybę ir ją pakelti, surandant defektus ir problemas [3]. Testavimas gali būti rankinis ir automatinis. Rankinis testavimas atliekamas testuotojo ir užima daug laiko, o automatinio testavimo metu yra vykdomi automatiniai testai. Šie testai gali būti kuriami programuotojų arba generuojami automatiškai.

Vienas iš būdų atlikti testavimą yra Modeliu paremtas testavimas (*angl. Model Driven Testing*), kurio metu sistemų modeliai naudojami testavimo atvejų generavimui ir testų rezultatų įvertinimui. Tačiau taip norint generuoti testus, reikia turėti sistemos modelį. Sistemą galima modeliuoti taikant įvairiuos metodus: kuriant baigtinius automatus, būsenų diagramas, Markovo grandines ar taikant UML kalbą [4, pp. 1 - 3].

Kadangi UML leidžia modeliuoti sistemas tiek statiniu, tiek dinaminiu požiūriu, toliau šiame skyrelyje bus panagrinėta, kaip galima generuoti automatinius testus turint UML būsenų modelį ir realizuotą sistemą.

2.2. Tyrimo objektas, sritis ir problema

Tyrimo objektas yra UML būsenų modelis ir kaip galima jį taikyti automatinių testų generavimui.

2.3. Unifikuotos modeliavimo kalbos analizė

UML 2.2 standartas aprašo 14 diagramų tipų [5], iš kurių 7 aprašo informaciją apie sistemos struktūrą:

- **Klasių** – apibūdina sistemos objektų tipus ir parodo klasių atributus bei sąryšius tarp tų klasių.
- **Komponentų** – apibūdina, sistemos komponentus ir parodo jų priklausomybes.
- **Sudedamųjų struktūrų** – leidžia sudalinti klasės vidinę struktūrą.

- **Išdėstymo** – naudojama atvaizduoti fizinį sistemos išdėstymą, atskleidžiant kokios reikia aparatūrinės įrangos, norint, kad veiktų sistema.
- **Objektų** – naudojama atvaizduoti konkrečius sistemos objektus tam tikru laiko momentu.
- **Paketų** – nusako į kokias grupes sistema išskaidyta ir kaip šios grupės priklauso viena nuo kitos.
- **Profilių** – naudojama UML meta-modelio lygmenyje ir parodo klasių stereotipus ir profilius.

O likusios 7 diagramos nusako veikimą:

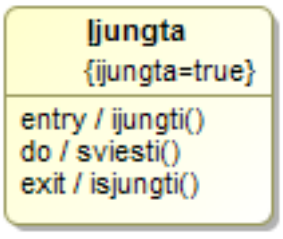
- **Veiklos** – naudojamos atvaizduoti procedūrų, verslo logiką ir veiksmų tėkmę.
- **Panaudos diagramos** – atvaizduoja sistemos panaudos atvejus ir kas juos naudoja.
- **Būsenų diagramos** – vaizduoja svarbiausias verslo ar veiklos sistemos būsenas, bei tų būsenos kitimą toje pačioje sistemoje. Ji nusako objektų būsenas ir jų pasikeitimus laike. Būsenos diagrama žingsnis po žingsnio atvaizduoja verslo ir operatyvinius sistemos komponentų srautus.
- **Sąveikų diagramos** –apibūdina duomenų cirkuliavimą tarp modeliuojamos sistemos elementų.
- **Bendradarbiavimo diagrama** – apibūdina sąveikas tarp objektų ar jų dalių, atvaizduoja informacijos kombinacijas, paimamas iš klasių, sekų ir panaudos atvejų diagramų apibūdinant ir statines struktūras, ir dinaminius sistemos veiksmus.
- **Sekų diagramos** – parodo, kaip objektai sąveikauja vienas su kitu. Šios diagramos iliustruoja objektų, jų būsenų, veiksmų lygiagretų išsidėstymą laike bei pranešimus tarp jų.
- **Laiko diagramos** – specifinės diagramos, orientuotos apibūdinti laiko apribojimus [6].

2.3.1. Būsenų modelio analizė

Būsenų diagramos naudojamos modeliuoti sistemų veikimą kaip baigtinį automatą. Jos vaizduoja galimas sistemos būsenas, bei tų būsenos kitimą sistemoje. Būsenų diagrama yra orientuoto grafo, kurio viršūnės žymi būsenas, o briaunos – perėjimus, grafinė reprezentacija. Tokia diagrama galima aprašyti visas įmanomas pasirinkto objekto būsenas ir kaip jos keičiasi įvykus tam tikriems įvykiams. Šiomis diagramomis galima modeliuoti tiek programas, tiek klases.

UML būsenų modelio pagrindiniai elementai yra šie:

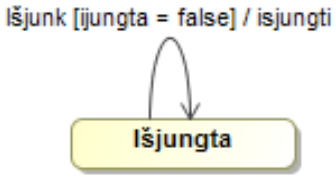
- Būsena (*angl. State*) – aprašo konkrečią objekto būseną. Yra prieš tai įvykusių įvykių ir veiksmų rezultatas. Būsena turi šiuos elementus:
  - Pavadinimą
  - Atributus
  - Veiksmus ir įvykius vykstančius atėjus, išeinant ir kitus į/iš šios būsenos



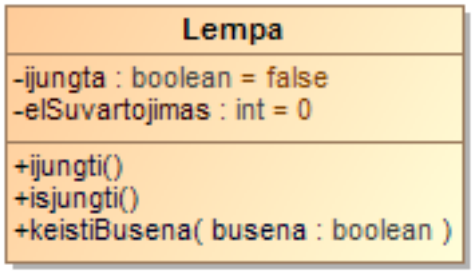
Taip pat būsenų diagramoje gali būti 2 specialios būsenos:

- Pradinė – nurodo pradinę būseną po objekto sukūrimo
- Galinė – žymi objekto sunaikinimą

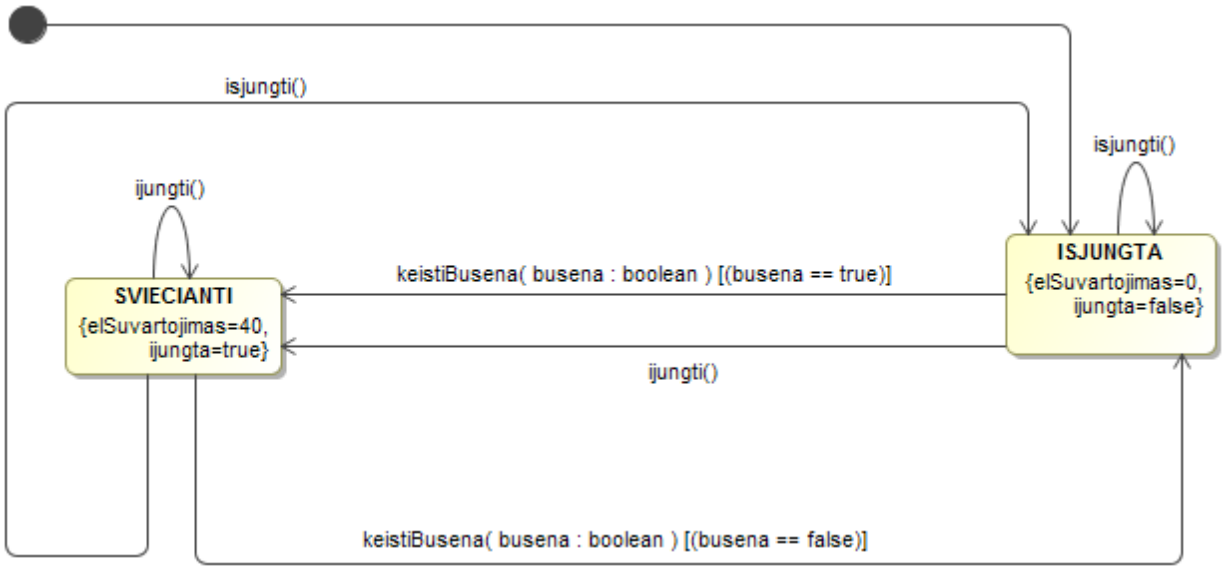
- Perėjimas (*angl. Transition*) – nusako kada objektas keičia būseną, t.y. aprašo objekto reakciją į tam tikrą įvykį. Perėjimas nusakomas:
  - Įvykiu (*angl. Event*) – nusako įvykį, kuris aktyvuoja perėjimą
  - Sąlyga (*angl. Guard*) – nusako sąlygas, kurios turi būti tenkinamos, kad perėjimas įvyktų
  - Veiksmu (*angl. Effect*) – nusako veiksmus susijusius su objektu



Žemiau yra pateikiamas pavyzdys, iliustruojantis sumodeliuotą sistemą, simuliuojančią stalinės lempos veikimą statiniu ir dinaminiu požiūriu, pasitelkus UML klasių ir būsenų diagramas.



1 pav. Lempos klasės klasių diagrama



2 pav. Lempos klasės būsenų diagrama

1 pav. pateiktoje klasių diagramoje pateikiama informacija apie lempos klasės laukus ir metodus, o 2 pav. pateikiamas lempos klasės būsenų modelis, kuriama demonstruojama kaip klasės

metodai keičia šios klasės objekto būsenas. Perėjimai tarp būsenų nusakomi metodais, kurie keičia būsenas, o būsenos – klasės laukais, kurie nusako objekto būseną esant toje būsenoje.

Taip UML būsenų diagramomis galima modeliuoti:

- būsenas, kuriose gali būti objektas ir sąlygas, kurias turi tenkinti objektas esant toje būsenoje
- įvykius, kuriems įvykus keičiasi objekto būseną
- sąlygas, kurios turi būti tenkinamos, prieš įvykstant perėjimui
- ir kitus veiksmus kurie vyksta objekto gyvavimo laiką, prieš ir po sąlygas bei kitus [7, pp. 83 - 87].

Turint šiuos duomenis galima generuoti automatinius testus, kur perėjimai būtų naudojami testavimo sekų generavimui, o būsenos – testavimo sekų rezultatų tikrinimui. Kaip tai galima atlikti, bus aptariama tolimesniuose skyreliuose.

2.4. Automatinių testų generavimas taikant UML būsenų modelį

Norint generuoti automatinius testus ir apskritai atlikti testavimą, reikia turėti testuojamus atvejus ir metodą patikrinti įvykdytų sekų gautų rezultatų teisingumui. Turint testuojamos klasės UML būsenų diagramą galima ją eksportuoti į XMI formato failą. XMI standartas leidžia atvaizduoti objektus XML formatu [8, pp. 3 - 5]. Tuomet nuskaičius šiame faile aprašytas būsenas ir perėjimus, galima generuoti testus, kuriuose testavimo atvejai būtų generuojami naudojant perėjimus, o jų pateikiamų rezultatų teisingumas būtų tikrinamas naudojant būsenose aprašytas sąlygas.

Tolimesniuose skyreliuose bus aptariami šie veiksmai detaliau.

2.4.1. Testavimo atvejų generavimas

Testavimo atvejų generavimas yra sudėtingas procesas, nes perėjimų tarp būsenų gali būti be galo daug, pavyzdžiui 2 pav. pateiktai sistemai galima aprašyti be galo daug kelių, kurie pavaizduoti žemiau esančioje lentelėje.

Lentelė 1. Galimos testavimo sekos stalinei lempai

Seka				Galutinė būsena
ijungti()				SVIECIANTI
ijungti()	isjungti()			ISJUNGTA
ijungti()	isjungti()	isjungti()		ISJUNGTA
ijungti()	isjungti()	isjungti()	ijungti()	SVIECIANTI
ijungti	keistiBusena(true)			SVIECIANTI

Taip pat ne visi keliai gali būti svarbūs, pavyzdžiui, jei esant ISJUNGTA būsenoje kartosime *isjungti()* metodo kvietimą daug kartų iš to didelės naudos nebus, nes šis metodas tik pakeičia kintamojo *ijungta* reikšmę į *false*.

Vis dėlto, žiūrint į būsenų diagramas, kaip į orientuotus grafus, galima taikyti grafų teorijos algoritmus ir bandyti generuoti testavimo sekas. Vienas iš taikytinų algoritmų yra Deikstros [9] algoritmas, kuris randa trumpiausius kelius tarp 2 viršūnių. Jo sudėtingumas yra $O(n^2)$, kur n viršūnių kiekis. Algoritmas pateikiamas žemiau.

```
{
s – pradinės viršūnė
```

```
d[1..n] kelių ilgių masyvas, d[k] trumpiausio kelio nuo viršūnės s iki viršūnės k
ilgis
prec[1..n] masyvas parodantis per kurias viršūnes keliai eina
}

sum:=infinity
d[i]:=sum; prec[i]:=0; i=1..n.
d[s]:= 0; {d(s,s)=0}
prec[s]:=s;
while „yra nenudažytų viršūnių“ do
     begin
          min:=sum
          for i:=1 to n do
               if „viršūnė i nenudažyta“ and min>d[i]
                    then
                         begin
                              min:=d[i]; k:= i;
                         end;

               if min = sum
                    then
                         begin
                              „grafas G – nejungusis“;
                              stop;
                         end;
                    else
                         begin
                              Nudažome viršūnę k;
                              For u viršūnes pasiekiamas iš k
                                   jei  u nenudažyta and d[u] > d[k] + c(k,u)
                                        then
                                             begin
                                                  d[u]:= d[k] + c(k,u)
                                                  prec[u]:=k
                                             end;
                                        end;
                              end;
          end;
end;
```

Tuomet, taikant šį algoritmą ir priskyrus svorius perėjimams (briaunoms) galima generuoti testavimo sekas, kur būtų imama pradinė viršūnė ir generuojami keliai iki visų kitų viršūnių.

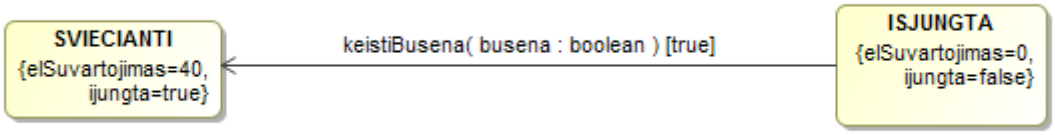
2.4.2. Testų duomenų generavimas

Vis dėlto turint testavimo atvejus ir testuojant realizuotą klasę, ne visada visi metodai bus be parametrų, pavyzdžiui, 1 pav. pavaizduotoje klasėje norint iškviesti metodą *keistiBusena()* reikia nurodyti loginio tipo kintamąjį. Taigi, generuojant automatinius testus reikia nurodyti konkrečias reikšmes naudojamas testų metodų kvietimui. Egzistuoja nemažai būdų tai atlikti, keletas iš jų:

- Atsitiktinis generavimas, kai generuojamos atsitiktinės reikšmės
- Genetinis programavimas, pvz.: kai stengiamasi rasti optimalų sprendimą iš keleto galimų
- Duomenų generavimas analizuojant programos kodą
- Generavimas taikant apribojimus

Vis dėl to, šį uždavinį galima palengvinti, nes UML būsenų diagramose galima aprašyti sąlygas, kurias tenkinant gali vykti perėjimai tarp būsenų. Nuskaičius šias vartotojo aprašytas sąlygas

bei pritaikius ribinių reikšmių analizę bei atsitiktinį generavimą, galima generuoti testų reikšmes. Pavyzdžiui, turint 3 pav. pavaizduotą perėjimą tarp būsenų SVIECIANTI ir ISJUNGTA, galima generuoti testavimo sekas, kur metodui bus nurodomos skirtingos reikšmės.



3 pav. Duomenų, metodų kvietimui, generavimo pavyzdys

Galima patikrinti, ar esant būsenoje ISJUNGTA iškvietus metodą *keistiBusena* su reikšme *true* bus pereita į būseną SVIECIANTI ir jei reikšmė bus lygi *false* – testuojamas objektas liks būsenoje ISJUNGTA. Papildžius testavimo atvejų generavimo algoritmą ribinių reikšmių analize, galima generuoti sekas, kur būtų ne vien tikrinama ar iš vienos būsenos pereinama į kitą, bet ir kas nutinka kai perėjimo sąlyga yra netenkinama.

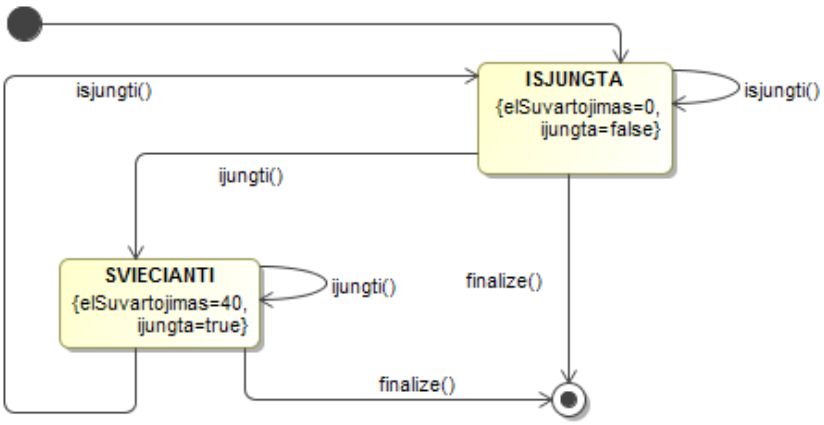
2.4.3. Būsenų tikrinimas

Įvykdžius testavimo seką, reikia patikrinti gautų rezultatų teisingumą. Ši problema kitaip dar vadinama Orakulo problema. Naudojant UML būsenų modelio diagramas, būsenoms galima aprašyti apribojimus, kuriuos turi tenkinti objektas esant toje būsenoje.

UML būsenų diagramose juos galima nusakyti sąlygomis, apribojimais ar prielaidomis. UML specifikacija neapriboja kaip jas reikėtų išreikši, tad norint jas nusakyti galima taikyti:

- Natūralią kalbą
- OCL
- Java kalbą
- Elementų reikšmes

Dažniausiai šie apribojimai nusakomi tam tikriems elementais kaip jų vaikai arba jų komentarai. Žemiau pateiktame paveiksliuke, būsenų sąlygos nusakomos konkrečiomis reikšmėmis, nurodant apriboto lauko pavadinimą ir reikšmę. Galima nurodyti *int, String, bool, null* reikšmes. Pavyzdžiui, būsenoje IJUNGTA yra nurodyta kad lempos klasės objekto lauko *ijungta* reikšmė bus lygu *true*, o esant būsenoje ISJUNGTA – *false*. Taip būsenas apribojant elementais ir reikšmėmis galima susieti klasės laukus ir būsenas, be to sumažėja rizika įvelti klaidas nei naudojant natūralią kalbą.



4 pav. UML būsenų apribojimų pavyzdys

Tuomet turint tokį modelį, galima nuskaityti būsenas bei jų apribojimus ir refleksijos būdu patikrinti reikšmes. Refleksijos metu kintamojo reikšmė pasiekiama pagal vardą:

```
Field f = lempa.getClass().getField("ijungta");

f.setAccessible(true);

Object reiksme = f.get(lempa)
```

2.5. Esamų problemos sprendimo metodų analizė

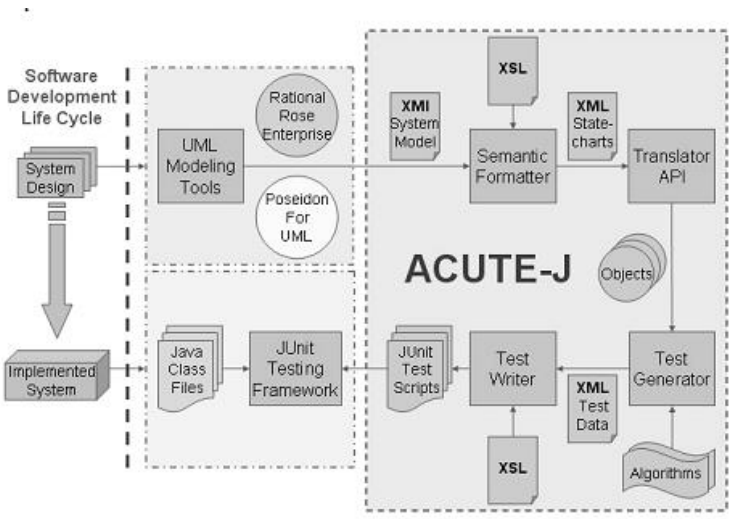
Šiuo metu rinkoje komercinio įrankio gebančio generuoti automatinius testus taikant UML būsenų modelį nėra, tačiau egzistuoja keletas sprendimų aprašančių, kaip tai galima atlikti. Jie yra:

- ACUTE-J (Automated stateChart Unit Testing Engine for Java)
- CBCDTOOL

2.5.1. Analizuojami metodai

2.5.1.1. ACUTE-J

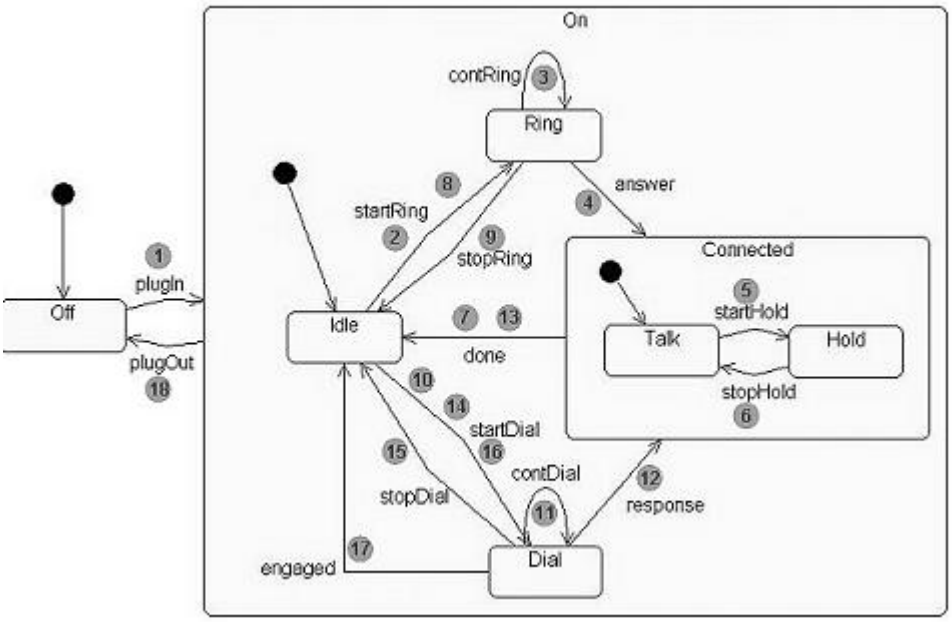
ACUTE-J (*angl.* Automated stateChart Unit Testing Engine for Java) aprašo kaip galima generuoti automatinius būsenų testus naudojant būsenų diagramas. Naudojant šį metodą būsenų diagrama yra nuosekliai konvertuojama į Junit vienetų testus, pritaikant grafų teoriją [10]. Jo veikimas pavaizduotas 5 pav.



5 pav. ACUTE-J generatoriaus veikimas

Testų generavimo procesas pradedamas nubraižant būsenų diagramą, įrankiu kuris palaiko XMI eksportavimą. Tuomet, ji perduodama ACUTE-J įrankiui. Diagrama nuskaitoma, pritaikomas Kinų paštininko (*angl. Chinese postman*) algoritmas ir generuojamas testas. Kinų paštininko algoritmu nustatoma kaip greičiausiai ir efektyviausiai aplankyti visas būsenas ir kelius. Tai užtikrina, kad kiekvienas perėjimas ir būsena būtų aplankyta.

6 pav. pavaizduota telefono veikimo būsenų diagrama. Tikrinamas kelias yra parodytas ir pažymėtas skaičiais nuo 1 iki 18.



6 pav. Telefono veikimo būsenų diagrama

Tokią diagramą eksportavus į XMI formatą ir nurodžius generatoriui, galima generuoti testą, kur įvykdomos testuojamos sekos ir tikrinamos būsenos, tačiau jų tikrinimui turi būti sukurtas atskiras kintamasis klasėje, nusakantis būseną.

```
1 import junit.framework.*;
```

```
2 import junit.textui.TestRunner;
3
4 public class TestPhone extends TestCase {
5   Phone phone0;
6
7   public TestPhone(String name) {  super(name); }
8
9   protected void setUp() { phone0 = new Phone(); }
10
11   protected void tearUp() { phone0 = null; }
12
13   public void testSetState() {
14     assertTrue(phone0.offState.equals(phone0.state));
15     phone0.plugIn();  // 1
16     assertTrue(phone0.onState.equals(phone0.state));
17     assertTrue(phone0.idleState.equals(phone0.state.state));
18     phone0.startRing();  // 2
19     assertTrue(phone0.onState.equals(phone0.state));
20     assertTrue(phone0.ringState.equals(phone0.state.state));
21     phone0.contRing();  // 3
22     assertTrue(phone0.onState.equals(phone0.state));
23     assertTrue(phone0.ringState.equals(phone0.state.state));
24     phone0.answer();  // 4
25     assertTrue(phone0.onState.equals(phone0.state));
26     assertTrue(phone0.connectedState.equals(phone0.state.state));
27     assertTrue(phone0.talkState.equals(phone0.state.state.state));
      .
      .
      .
28     phone0.startDial();  // 16
29     assertTrue(phone0.onState.equals(phone0.state));
30     assertTrue(phone0.dialState.equals(phone0.state.state));
31     phone0.engaged();  // 17
32     assertTrue(phone0.onState.equals(phone0.state));
33     assertTrue(phone0.idleState.equals(phone0.state.state));
34     phone0.plugOut();  // 18
35     assertTrue(phone0.offState.equals(phone0.state));
36   }
37
38   public static void main(String[] args) {
39     TestRunner.run(TestPhone.class);
40   }
41 }
```

Šio metodo pagrindiniai privalumai:

- Galima naudoti sub-būsenas

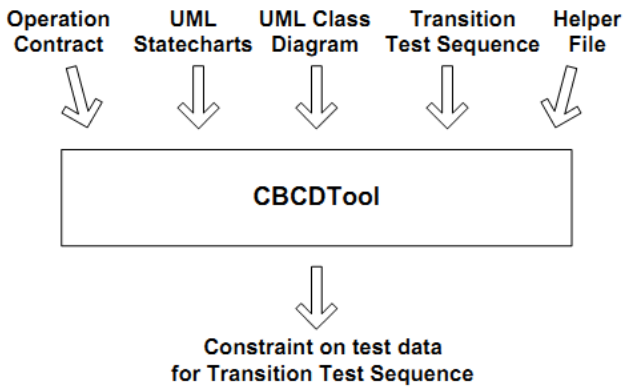
- Nereikia nurodyti testuojamų kelių, nes jie generuojami automatiškai taikant kinų paštininko algoritmą.

Šio metodo trūkumai:

- Reikia turėti kintamuosius, kurie nusako objekto būsenas
- Reikia atskleisti klasės kintamuosius

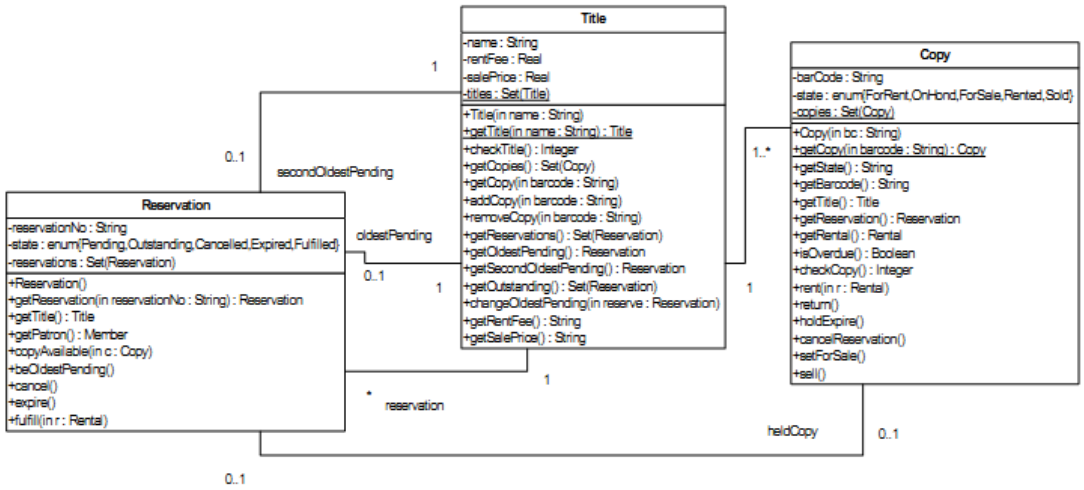
2.5.1.2. CBCDTOOL

Kitas metodas aprašo CBCDTool įrankį [11]. Šiam įrankiui nurodomos UML būsenų diagramos, UML klasių diagramos, perėjimų testų būsenos ir pagalbiniai failai. Šis įrankis naudoja OCL būsenų keitimosi validavimui. Jo veikimas parodytas 7 pav.

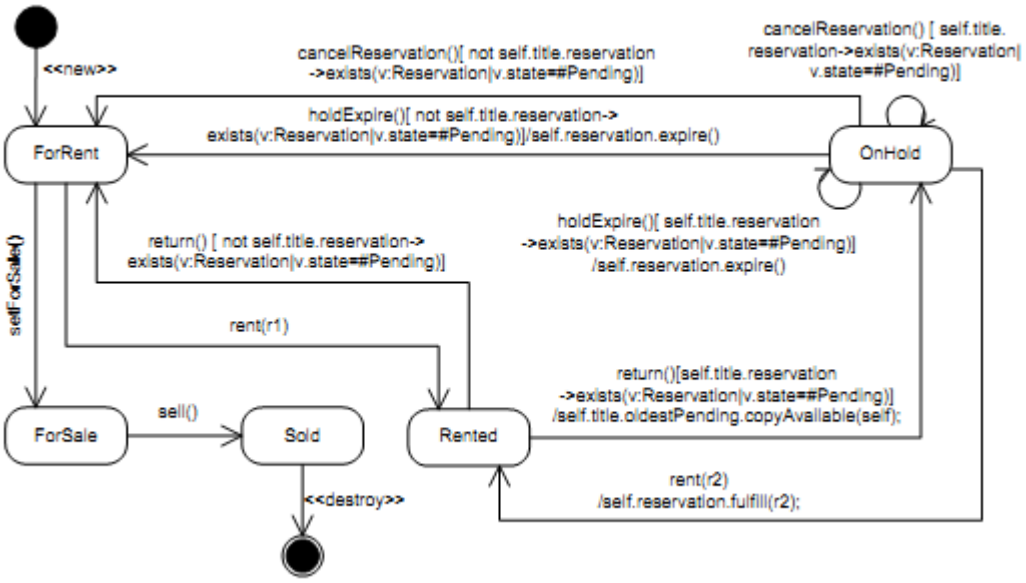


7 pav. CBCDTool veikimas

Pavyzdžiui, nurodoma knygų skolinimo klasių diagrama pavaizduota 8 pav.. ir būsenų diagrama 9 pav.



8 pav. Testuojamos sistemos klasės diagrama pateikiama CBCDTool



9 pav. Klasės Copy būsenų diagrama

Taip pat nurodoma testuojama perėjimų seka:

@ForRent@rent(r1)[true]

@Rented@return()[self.title.reservation->exists(v:Reservation|

v.state=#Pending)]/self.title.oldestPending.copyAvailable(c)

@OnHold@holdExpire()[self.title.reservation->exists(v:Reservation|

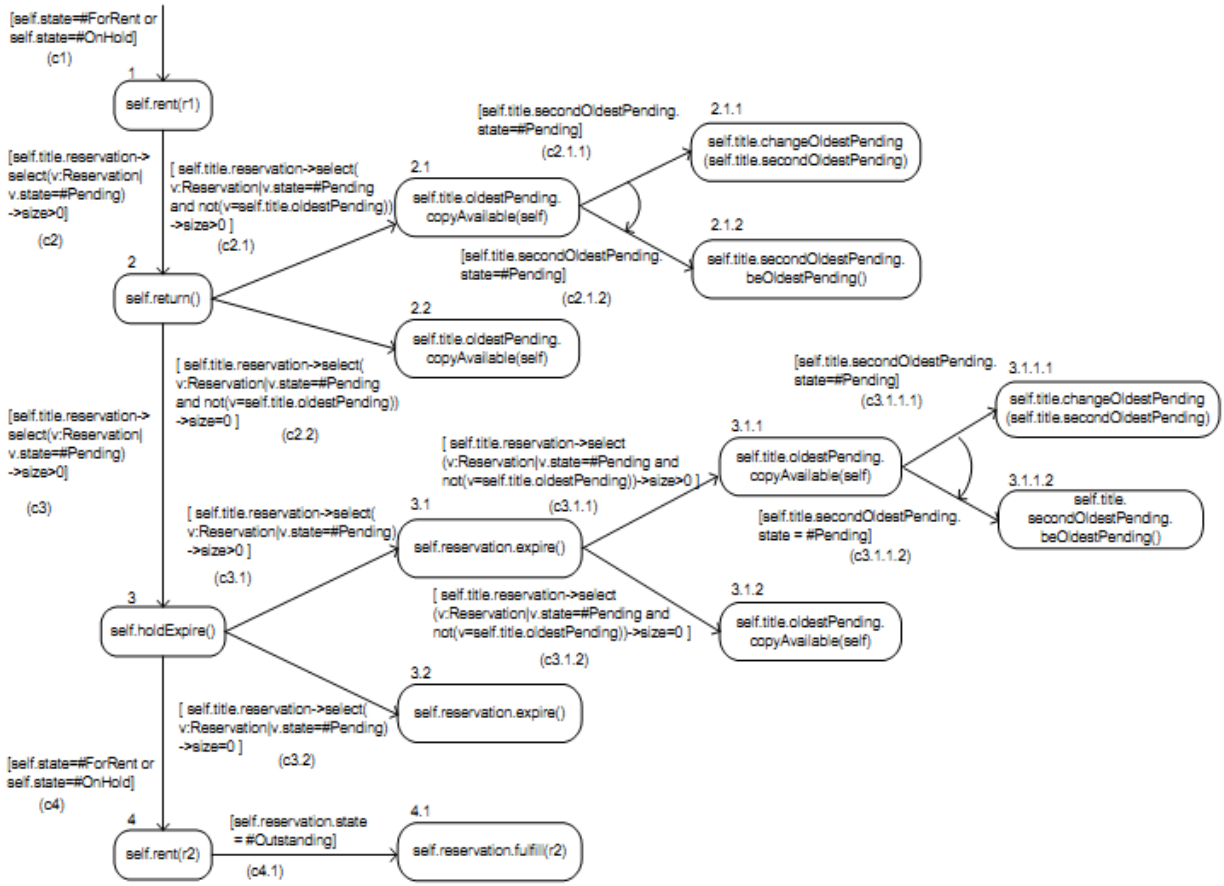
v.state=#Pending)]/self.reservation.expire()

@OnHold@rent(r2)/self.reservation.fulfill(r2)

@Rented

Šiame scenarijuje, yra paskolinama knygos kopija, ir po to prieš ją grąžinant, vėl yra atliekamas užsakymas. Tada ši rezervacija baigiasi taip jai ir neįvykus, bet prieš jai baigiantis yra atliekama dar viena rezervacija. Ši rezervacija nesibaigia, ir knyga yra paskolinama.

Aprašytam scenarijui įrankis sukuria ISM (Iškvietimų sekos medį), pavaizduotą 10 pav. Iš jo suminimizuojami perėjimų OCL apribojimai, kurie turi būti išpildomi, kol vykdoma seka. Šiuos apribojimus gali nuskaityti įrankis ir sugeneruoti testus.



10 pav. Iškvietimų sekos medis

Šio metodo privalumai:

- Šiuo metodu sugeneruoti testai leidžia tikrinti ar teisingai vyksta perėjimai tarp būsenų

Trūkumai:

- Reikia turėti ne vien būsenų diagramas, bet ir klasių diagramas

Sudėtinga nurodyti perėjimų būsenas, nes reikia OCL ir testuojamos sistemos architektūrinių ir veikimo žinių.

2.5.2. Išanalizuotų metodų santrauka

Buvo išanalizuoti ACUTE-J ir CBCDTOOL metodų taikymas automatinių testų generavimui. Santrauka pateikiama žemiau esančioje lentelėje.

Lentelė 2. Esamų sprendimų savybių santrauka

	ACUTE-J	CBCDTOOL	Siūlomas sprendimas
Visų būsenų padengimas	Taip	Taip	Taip
Visų šakų padengimas	Ne visai	Taip	Taip
Generavimo sudėtingumas	Nesudėtingas	Labai sudėtingas	Vidutiniškas
Realizuotos klasės laukų	Reikia atskleisti	Nereikia atskleisti	Nereikia atskleisti

atskleidimas			
Duomenų generavimas	Nepalaiko	Automatinis	Automatinis
Testų sekų generavimas	Automatinis	Reikia nurodyti sekas	Automatinis

2.6. Darbo tikslas, uždaviniai ir siekiami privalumai

Darbo tikslas yra sukurti sprendimą, leidžianti generuoti automatinius testus, kur būtų aplankomos visos būsenos ir keliai, neatskleidžiant klasių kintamųjų ir automatiškai generuojant reikšmes testų metodų kvietimui

2.7. Siekiamo sprendimo apibrėžimas

Siekiama sukurti sprendimą, kuris leistų generuoti automatinius testus, kuriuose testavimo sekos būtų generuojamos taikant UML būsenų modelio perėjimus, duomenys testų metodų kvietimui būtų parenkami taikant ribinių reikšmių analizę aprašytiems apribojimams ir testavimo sekų vykdymo rezultatų teisingumui tikrinti būtų taikomos būsenų modelyje būsenoms nurodytos sąlygos.

2.8. Analizės išvados

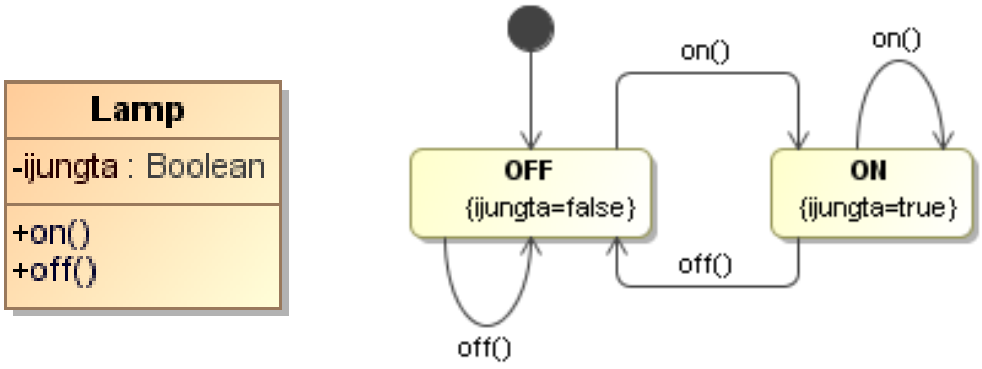
Analizės metu buvo žiūrima kaip galima generuoti automatinius testus taikant UML būsenų modelį. Buvo nustatyta, kad nubraižius testuojamai klasei būsenų diagramą, galima generuoti testus, kur testų sekų generavimui taikomi UML būsenų modelyje nurodyti perėjimai, kurie nusako kokius metodus reikia kviesti. Testavimo sekų generavimui galima taikyti Deikstros algoritmą, kurio pagalba galima būtų aplankyti visas būsenas ir kelius. O testų metodų kvietimui reikalingų reikšmių generavimui galima taikyti perėjimams nurodytus apribojimus, o įvykdytų rezultatų teisingumui, diagramose būsenoms aprašytas sąlygas kurias turi tenkinti objektas esant toje būsenoje.

3. PROJEKTINĖ DALIS

Šio skyriaus tikslas aprašyti automatinių būsenų testų generatoriaus, gebančio generuoti testus iš UML būsenų diagramų ir programinio kodo, projektinę dalį. Generatorius buvo realizuotas kaip Eclipse programavimo aplinkos įskiepis, kuriam nurodžius būsenų diagramą, būsenų perėjimų failą ir Eclipse projektą, kuriame yra realizuota testuojama klasė, yra generuojami testai. Toliau bus aprašyti jam iškelti reikalavimai ir pateiktas architektūros vaizdas.

3.1. Generatoriaus paskirtis

Vartotojas norėdamas sugeneruoti testus realizuotai klasei, turi nubraižyti UML būsenų diagramą, kurioje būti pavaizduotos klasės būsenos (jas aprašant pavadinimu ir kokios yra kintamųjų reikšmės esant toje būsenoje) ir kokiais metodais vyksta perėjimai tarp jų. Pavyzdžiui, turint lempos klasę, kurioje realizuoti *ijungti*() ir *isjungti*() metodai, galima nubraižyti UML būsenų diagramą pavaizduotą žemiau esančiame paveiksliuke.



11 pav. Pavyzdinės sistemos modelis

Tuomet tokią diagramą eksportavus į XMI formatą ir aprašius būsenų perėjimų failą galima generuoti testus. Būsenų perėjimo failo pavyzdys pateikiamas žemiau.

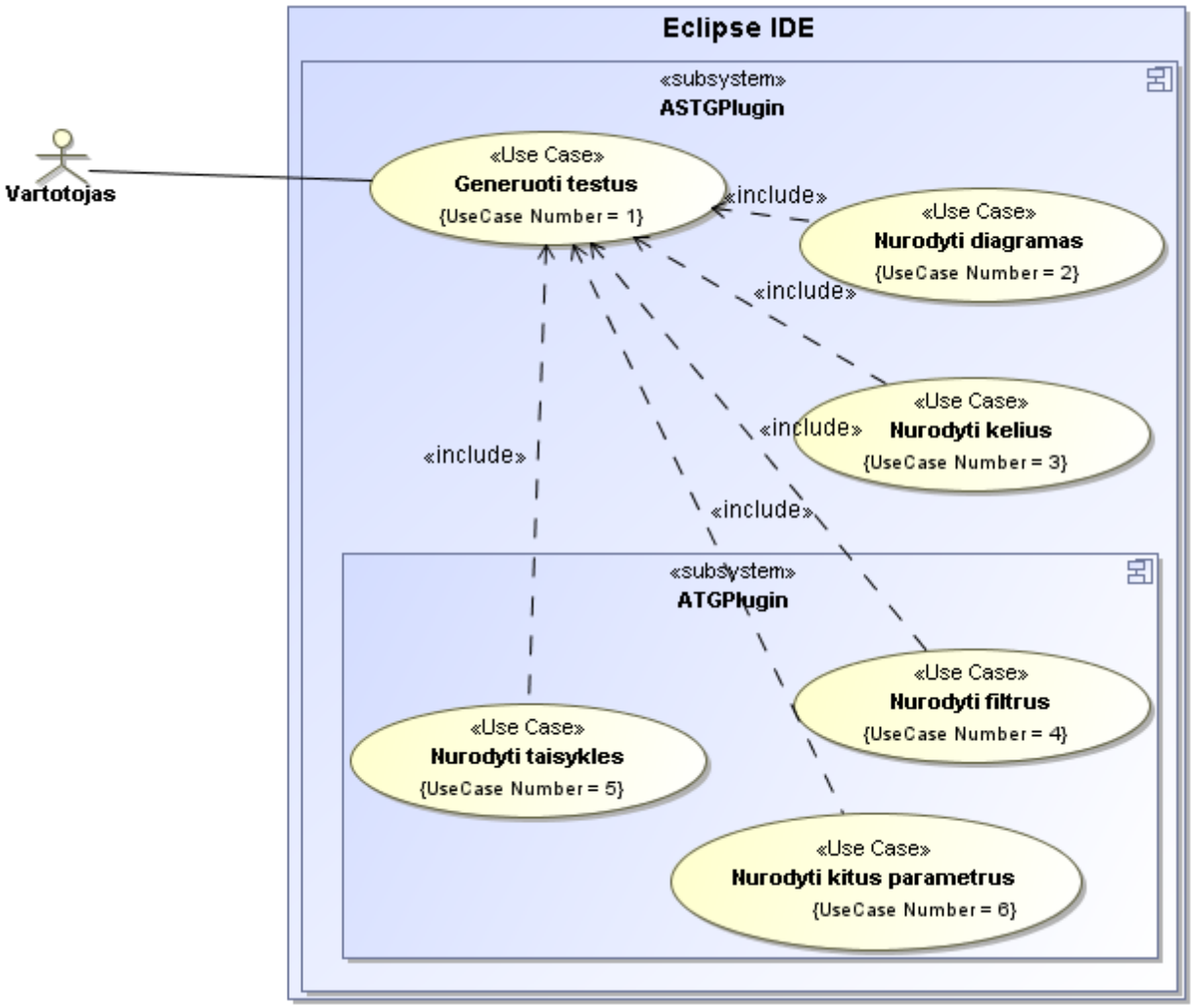
```
Lamp

On(), On() ON

On(),off(), OFF
```

3.2. Panaudos atvejai

Šiame skyrelyje išvardinti įskiepio panaudos atvejai.



12 pav. Sistemos panaudojimo atvejų diagrama

Šio įrankio vartotojo funkcijos:

- Generuoti testus – funkcija skirta testų generavimui.
- Nurodyti diagramas – skirta nurodyti UML būsenų diagramas.
- Nurodyti kelius – funkcija nurodyti kelių failus.
- Nurodyti filtrus – funkcija skirta nurodyti kokių klasių objektai bus konstruojami metodų kvietimui.
- Nurodyti taisykles – funkcija skirta nurodyti kokios konkrečios reikšmės bus naudojamos objektų kūrimui ir kvietimui.
- Nurodyti kitus parametrus – funkcija skirta nurodyti kur bus saugomi testai ir pan.

Lentelė 3. PA 1: Generuoti testus

Panaudos atvejis	Generuoti testus	**ID**	1
Aktoriai	• Vartotojas		
Tikslas	Sugeneruoti testus pagal parinktus nustatymus.		
Sužadinimo sąlyga	Vartotojas pasirenka generuoti testus punktą iš kontekstinio meniu arba paspaudžia mygtuką įrankių juostoje.		
Prieš sąlyga	Vartotojas turi būti įkėlęs projektą, kuriame egzistuotų testuojamos klasės, metodai ir atributai.		
Po sąlyga	Sugeneruojami testai.		
Pagrindinis scenarijus	1. Vartotojas pasirenka projektą, paketą ar klasę. 2. Vartotojas iškviečia testų generavimo vedlį. 3. Vartotojas nurodo būsenų diagramos failą ar failus, spaudžia Next mygtuką. 4. Vartotojas nurodo kelius, spaudžia Next mygtuką. 5. Vartotojas nurodo filtrus, spaudžia Next mygtuką. 6. Vartotojas nurodo taisykles, spaudžia Next mygtuką. 7. Vartotojas nurodo kitus parametrus, spaudžia Finish mygtuką. 8. Testai sugeneruojami į nurodytą katalogą.		
Alternatyvus scenarijus	• Parodoma klaida, jei diagramoje aprašyta klasė projekte neegzistuoja. • Parodoma klaida, jei nurodytas tuščias kelių failas. • Parodoma klaida, jei nurodyta diagrama tuščia.		

Lentelė 4. PA 2: Nurodyti diagramas

Panaudos atvejis	Nurodyti diagramas	**ID**	2
Aktoriai	• Vartotojas		
Tikslas	Nurodyti UML būsenų diagramas, pagal kurias bus generuojamos būsenos.		
Sužadinimo sąlyga	Vartotojas pasirenka generuoti testus punktą iš kontekstinio meniu arba paspaudžia mygtuką įrankių juostoje, taip atidarydamas generavimo vedlį.		
Prieš sąlyga	Turi būti iškviestas testų generavimo vedlys.		
Po sąlyga	Nurodomos diagramos.		
Pagrindinis scenarijus	1. Vartotojas pasirenka projektą, paketą ar klasę. 2. Vartotojas iškviečia testų generavimo vedlį. 3. Vartotojas nurodo būsenų diagramos failą ar failus.		
Alternatyvus scenarijus	• Parodoma klaida, jei nurodyta diagrama tuščia. • Parodoma klaida, jei diagramoje aprašyta klasė projekte neegzistuoja.		

Lentelė 5. PA 3: Nurodyti kelius

Panaudos atvejis	Nurodyti kelius	**ID**	3
Aktoriai	• Vartotojas		
Tikslas	Nurodyti kelius ir būsenas kurių bus tikimasi.		
Sužadinimo sąlyga	Vartotojas turi būti atidaręs testų generavimo vedlį, nurodęs diagramas.		
Prieš sąlyga	Turi būti nurodyta bent viena diagrama.		
Po sąlyga	Nurodytas bent vienas kelias.		
Pagrindinis scenarijus	1. Vartotojas pasirenka projektą, paketą ar klasę. 2. Vartotojas iškviečia testų generavimo vedlį. 3. Vartotojas nurodo būsenų diagramos failą ar failus, spaudžia Next mygtuką. 4. Vartotojas nurodo kelius.		
Alternatyvus scenarijus	• Parodoma klaida, jei nurodytas tuščias kelių failas.		

Lentelė 6. PA 4: Nurodyti filtrus

Panaudos atvejis	Nurodyti filtrus	**ID**	4
Aktoriai	• Vartotojas		
Tikslas	Nurodyti klasių, sąsajų, abstrakčių klasių ir sąrašų filtrus, kurie bus naudojami objektų konstravimui.		
Sužadinimo sąlyga	Vartotojas turi būti atidaręs testų generavimo vedlį, nurodęs diagramas, kelius.		
Prieš sąlyga	Turi būti parinktos diagramos, keliai.		
Po sąlyga	Parinkti filtrai.		
Pagrindinis scenarijus	1. Vartotojas pasirenka projektą, paketą ar klasę. 2. Vartotojas iškviečia testų generavimo vedlį. 3. Vartotojas nurodo būsenų diagramos failą ar failus, spaudžia Next mygtuką. 4. Vartotojas nurodo kelius, spaudžia Next mygtuką. 5. Vartotojas nurodo filtrus.		
Alternatyvus scenarijus			

Lentelė 7. PA 5: Nurodyti taisykles

Panaudos atvejis	Nurodyti taisykles	**ID**	5
Aktoriai	• Vartotojas		
Tikslas	Nurodyti taisykles, kokios reikšmės bus imamos generuojant reikšmes metodų kvietimui.		
Sužadinimo sąlyga	Vartotojas turi būti atidaręs testų generavimo vedlį, nurodęs diagramas, kelius, filtrus.		
Prieš sąlyga	Turi būti nurodytos diagramos, keliai, filtrai.		
Po sąlyga	Nurodytos taisyklės.		
Pagrindinis scenarijus	1. Vartotojas pasirenka projektą, paketą ar klasę. 2. Vartotojas iškviečia testų generavimo vedlį.		

	3. Vartotojas nurodo būsenų diagramos failą ar failus, spaudžia Next mygtuką. 4. Vartotojas nurodo kelius, spaudžia Next mygtuką. 5. Vartotojas nurodo filtrus, spaudžia Next mygtuką. 6. Vartotojas nurodo taisykles.
Alternatyvus scenarijus	

Lentelė 8. PA 6: Nurodyti kitus parametrus

Panaudos atvejis	Nurodyti kitus parametrus	**ID**	6
Aktoriai	• Vartotojas		
Tikslas	Parinkti kitus nustatymus, kaip testų generavimo gylis, saugojimo vieta ir pan.		
Sužadinimo sąlyga	Vartotojas turi būti atidaręs testų generavimo vedlį, nurodęs diagramas, kelius, filtrus, taisykles.		
Prieš sąlyga	Turi būti parinktos diagramos, keliai, filtrai, taisyklės.		
Po sąlyga	Parinkti kiti nustatymai.		
Pagrindinis scenarijus	1. Vartotojas pasirenka projektą, paketą ar klasę. 2. Vartotojas iškviečia testų generavimo vedlį. 3. Vartotojas nurodo būsenų diagramos failą ar failus, spaudžia Next mygtuką. 4. Vartotojas nurodo kelius, spaudžia Next mygtuką. 5. Vartotojas nurodo filtrus, spaudžia Next mygtuką. 6. Vartotojas nurodo taisykles, spaudžia Next mygtuką. 7. Vartotojas nurodo kitus parametrus.		
Alternatyvus scenarijus			

3.3. Reikalavimai

Šiame skyrelyje išvardinti iškelti funkciniai ir nefunkciniai reikalavimai automatiniam testų generatoriui.

3.3.1.1. Funkciniai

Lentelė 9. FR 1: Testų saugojimo vieta

Reikalavimas	1	**Reikalavimo tipas**	FR	**Įvykis/PA**	6
Aprašymas	Leidžiama nurodyti testų saugojimo vietą.				
Pagrindimas	Vartotojas gali norėti sugeneruotus testus saugoti atskirai.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi leisti nurodyti testų saugojimo vietą.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	2		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 10. FR 2: Klasių nagrinėjimo lygis

Reikalavimas	2	**Reikalavimo tipas**	FR	**Įvykis/PA**	6
Aprašymas	Leidžiama nurodyti testų generavimo gylį.				
Pagrindimas	Vartotojas gali norėti nurodyti gylį iki kol bus naudojamos NULL reikšmės objektų konstravimui.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi leisti nurodyti klasių paieškos gylį.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	2		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 11. FR 3: Testų metodų priešdėlis

Reikalavimas	3	**Reikalavimo tipas**	FR	**Įvykis/PA**	6
Aprašymas	Leidžiama nurodyti testų metodų priešdėlį.				
Pagrindimas	Vartotojas gali norėti nurodyti tam tikrą testų metodų priešdėlį.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi leisti nurodyti testų metodų priešdėlį.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	2		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 12. FR 4: Testų klasių galūnės pavadinimas

Reikalavimas	4	**Reikalavimo tipas**	FR	**Įvykis/PA**	6
Aprašymas	Leidžiama nurodyti testų klasių galūnės pavadinimą.				
Pagrindimas	Vartotojas gali norėti nurodyti tam tikrą testų klasių galūnės pavadinimą.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi leisti nurodyti testų klasių galūnę.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	2		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 13. FR 5: Testų generavimo taisyklės

Reikalavimas	5	**Reikalavimo tipas**	FR	**Įvykis/PA**	5
Aprašymas	Leidžiama nurodyti testų taisykles.				
Pagrindimas	Vartotojas gali norėti nurodyti tam tikras taisykles, kurios bus naudojamos konkrečių reikšmių generavimui kviečiant metodus.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi leisti nurodyti taisykles.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	2		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 14. FR 6: Testų filtrai

Reikalavimas	6	**Reikalavimo tipas**	FR	**Įvykis/PA**	4
Aprašymas	Leidžiama nurodyti testų filtrus.				
Pagrindimas	Vartotojas gali norėti nurodyti tam tikrus filtrus pvz.: kuriuos konstruktorius naudoti objektų konstravimui ir pan.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi leisti nurodyti filtrus.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	2		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 15. FR 7: Kelių failai

Reikalavimas	7	**Reikalavimo tipas**	FR	**Įvykis/PA**	3
Aprašymas	Leidžiama nurodyti kelių failus.				
Pagrindimas	Pagal vartotojo parinktus kelius bus kviečiami metodai ir tikrinamos būsenos.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi leisti nurodyti kelių failus.				
Užsakovo tenkinimas	4	**Užsakovo nepatenkinimas**	4		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 16. FR:8: Diagramų failai

Reikalavimas	8	**Reikalavimo tipas**	FR	**Įvykis/PA**	3
Aprašymas	Leidžiama nurodyti diagramų failus XMI formatu.				
Pagrindimas	Vartotojas iš šių diagramų generuos testus.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi leisti nurodyti diagramų failus XMI formatu.				
Užsakovo tenkinimas	4	**Užsakovo nepatenkinimas**	4		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 17. FR 9: Suvestinė po generavimo

Reikalavimas	9	**Reikalavimo tipas**	FR	**Įvykis/PA**	1
Aprašymas	Po testų generavimo turi būti parodyta kiek užtruko generavimas ir kokie testai buvo sugeneruoti.				
Pagrindimas	Vartotojas gali būti įdomu kaip pavyko generavimas.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi parodyti šią suvestinę po generavimo.				
Užsakovo tenkinimas	3	Užsakovo nepatenkinimas	1		
Priklausomybės		Konfliktai	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 18. FR 10: Testų negeneravimas

Reikalavimas	10	**Reikalavimo tipas**	FR	**Įvykis/PA**	1
Aprašymas	Testai turi būti negeneruojami, jei klasė nerealizuota projekte.				
Pagrindimas	Įrankis skirtas generuoti testus tik realizuotoms klasėms.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testavimo įrankis turi parodyti klaidą, jei bandoma generuoti testus nerealizuotoms klasėms.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	1		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

3.3.1.2. Nefunkciniai reikalavimai

Lentelė 19. NFR 11: Sąsaja

Reikalavimas	11	**Reikalavimo tipas**	NFR	**Įvykis/PA**	1-6
Aprašymas	Paprasta neperkrauta sąsaja.				
Pagrindimas	Vartotojui turi būti paprasta dirbti ir neintuityvi sąsaja neturi apsunkinti jo darbo.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Sąsaja turi būti intuityvi ir paprasta.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	1		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 20. NFR 12: Lengvas išmokimas

Reikalavimas	12	**Reikalavimo tipas**	NFR	**Įvykis/PA**	1-6
Aprašymas	Turi būti lengvai išmokstama naudotis įskiepiu.				
Pagrindimas	Vartotojui būtų paprasta dirbti ir neapsunkintų jo darbo neintuityvi sąsaja.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Sąsaja turi buti intuityvi ir paprasta.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	1		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 21. NFR 13: Generavimo greitis

Reikalavimas	13	**Reikalavimo tipas**	NFR	**Įvykis/PA**	1-6
Aprašymas	Testai turi būti generuojami kuo įmanoma greičiau.				
Pagrindimas	Vartotojas nelauks 2h kol bus sugeneruoti testai 1 klasei.				
Šaltinis	Užsakovas.				
Tikimo kriterijus	Testai generuojami greitai.				
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	1		
Priklausomybės		**Konfliktai**	Nėra		
Papildoma medžiaga					
Istorija	Užregistruotas 2012-03-05				

Lentelė 22. NFR 15: Plečiamumas

Reikalavimas	15	**Reikalavimo tipas**	NFR	**Įvykis/PA**	1-6
Aprašymas	Sistema turi būti paprastai taisoma ir plečiama.				
Pagrindimas	Gali tekti sistemą plėsti ir pildyti naujomis savybėmis.				

Šaltinis	Užsakovas.		
Tikimo kriterijus	Sistema naudoja sąsajas ir abstrakčias klases.		
Užsakovo tenkinimas	3	**Užsakovo nepatenkinimas**	1
Priklausomybės		**Konfliktai**	Nėra
Papildoma medžiaga			
Istorija	Užregistruotas 2012-03-05		

3.4. Architektūra

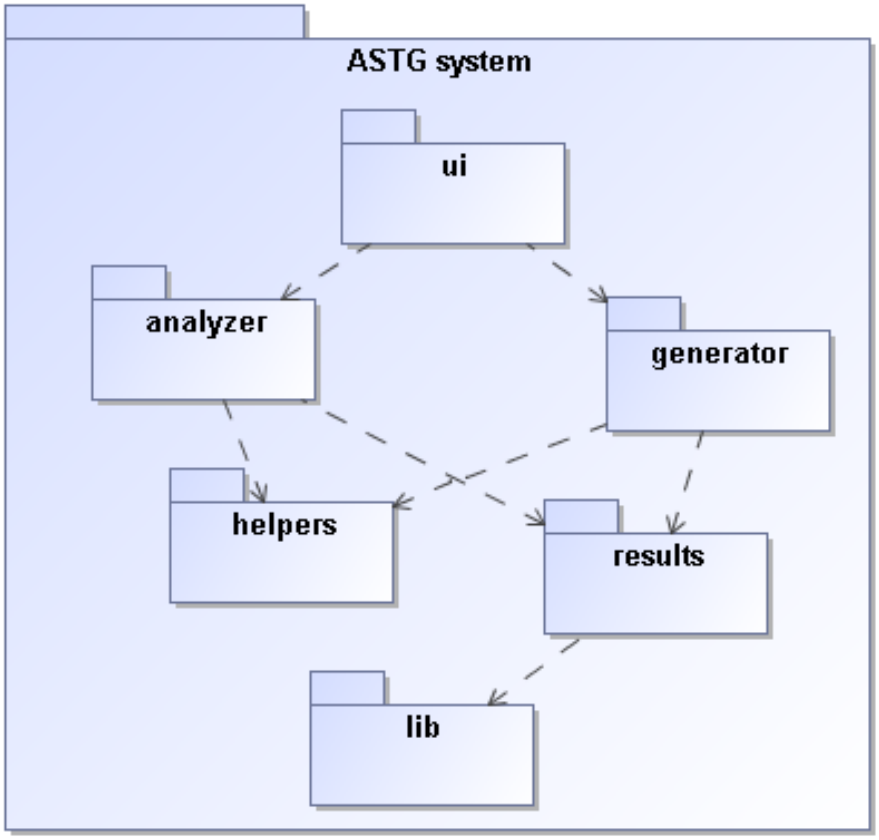
Šiame skyrelyje pateikiamas sukurto įskiepio sistemos architektūrinis vaizdas.

3.4.1. Statinis vaizdas

Sistema yra suskirstyta į 6 pagrindinius paketus:

- Analyzer (analizavimo)
- Generator (generavimo)
- Lib (testavimo bibliotekos)
- Results (rezultatų)
- UI (vartotojo sąsajos)
- Helpers (pagalbinio)

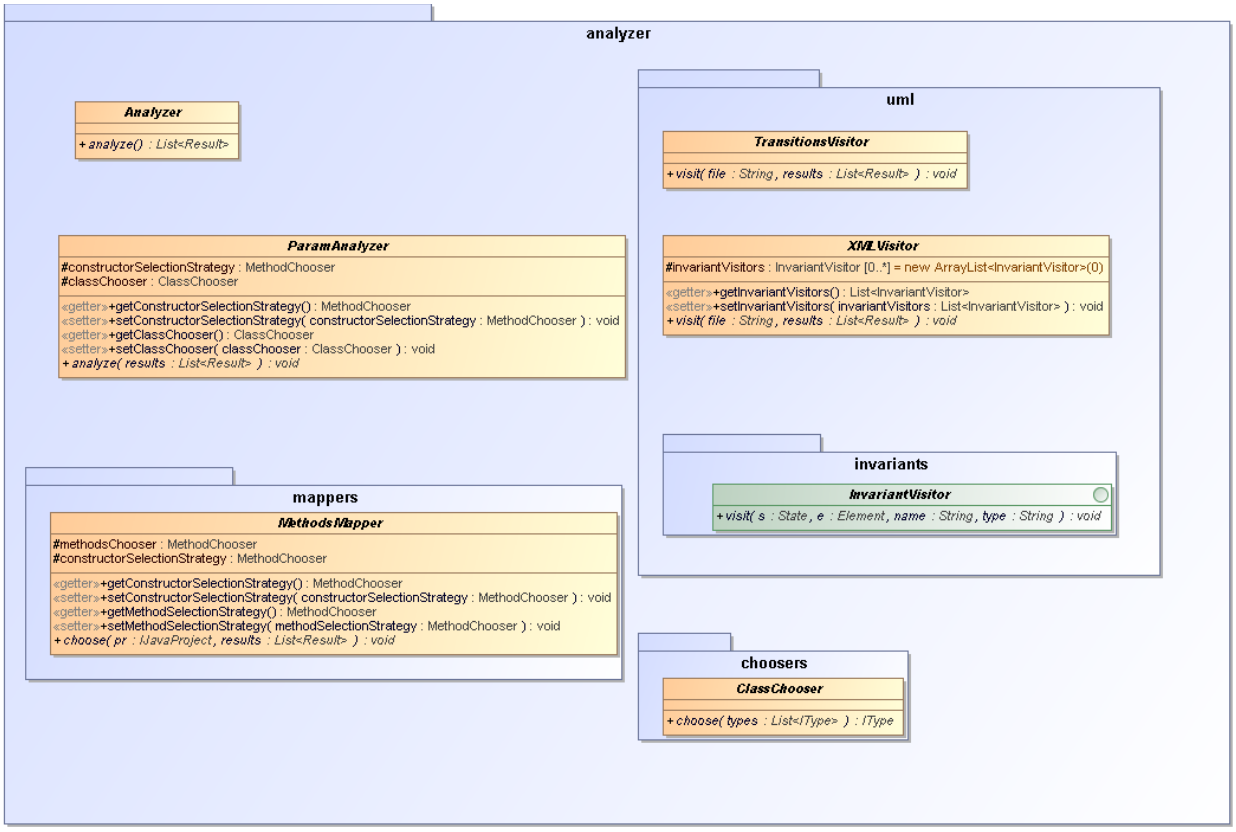
Jie pavaizduoti 13 pav.



13 pav. Realizuoto įskiepio paketų diagrama

3.4.1.1. Analyzer paketas

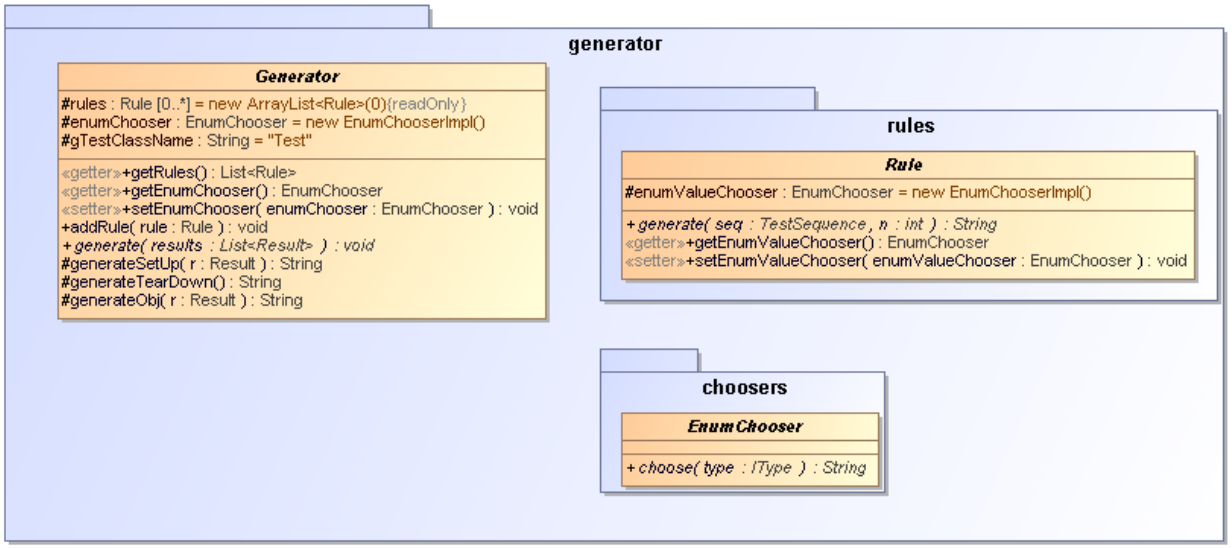
Analyzer – tai paketas atsakingas už kelių nagrinėjimą, klasių ir sąrašų parinkimą, kurie bus naudojami testų duomenims. Jam nurodomi analizuojamos klasės, jų būsenų sekų diagramos, perėjimų failai ir filtrai, kurie atsakingi už tai kokios klasės bus naudojamos generuojant reikšmes. Šio paketo statinis vaizdas pavaizduotas 14 pav.



14 pav. Analyzer paketas

3.4.1.2. Generator paketas

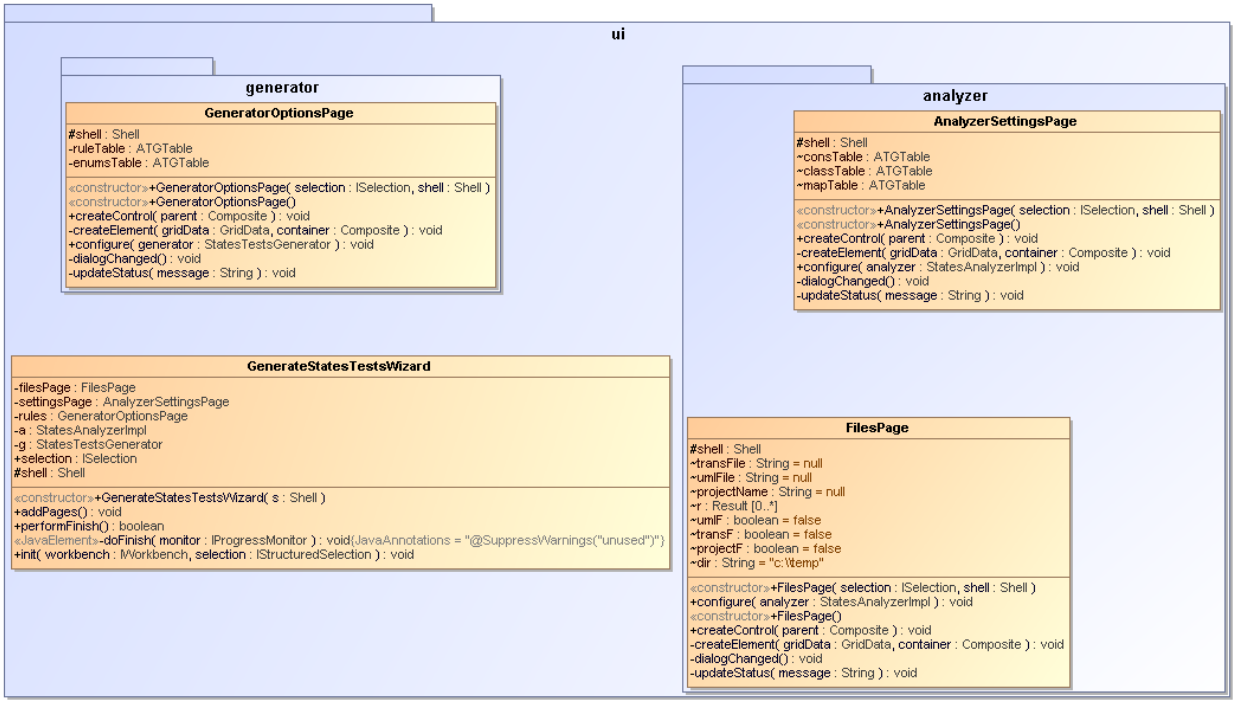
Generator – tai paketas skirtas testų ir jų kviečiamų metodų reikšmių generavimui. Jo vaizdas pateiktas 15 pav.



15 pav. Generator paketas

3.4.1.3. UI paketas

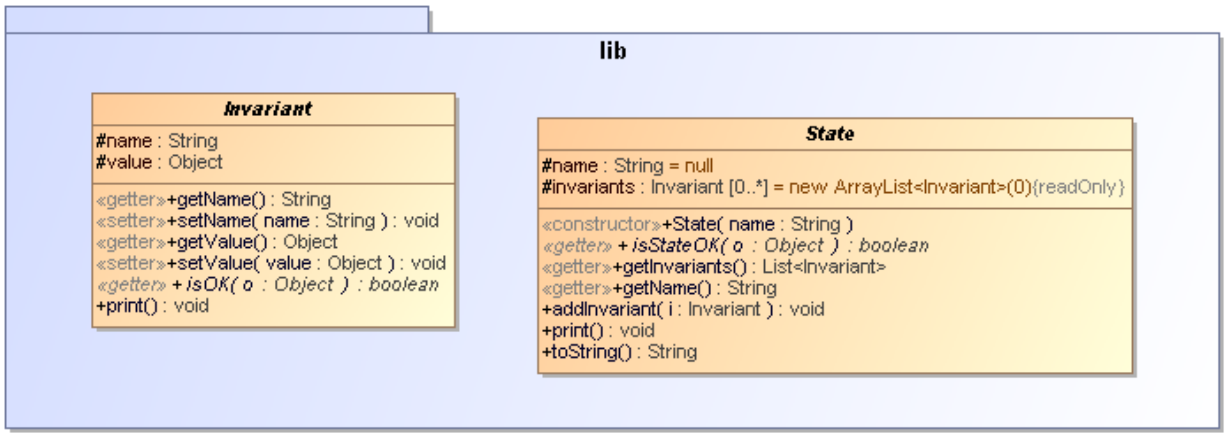
Šiame pakete aprašytos klasės atvaizduojančios vartotojo sąsaja.



16 pav. UI paketas

3.4.1.4. Lib paketas

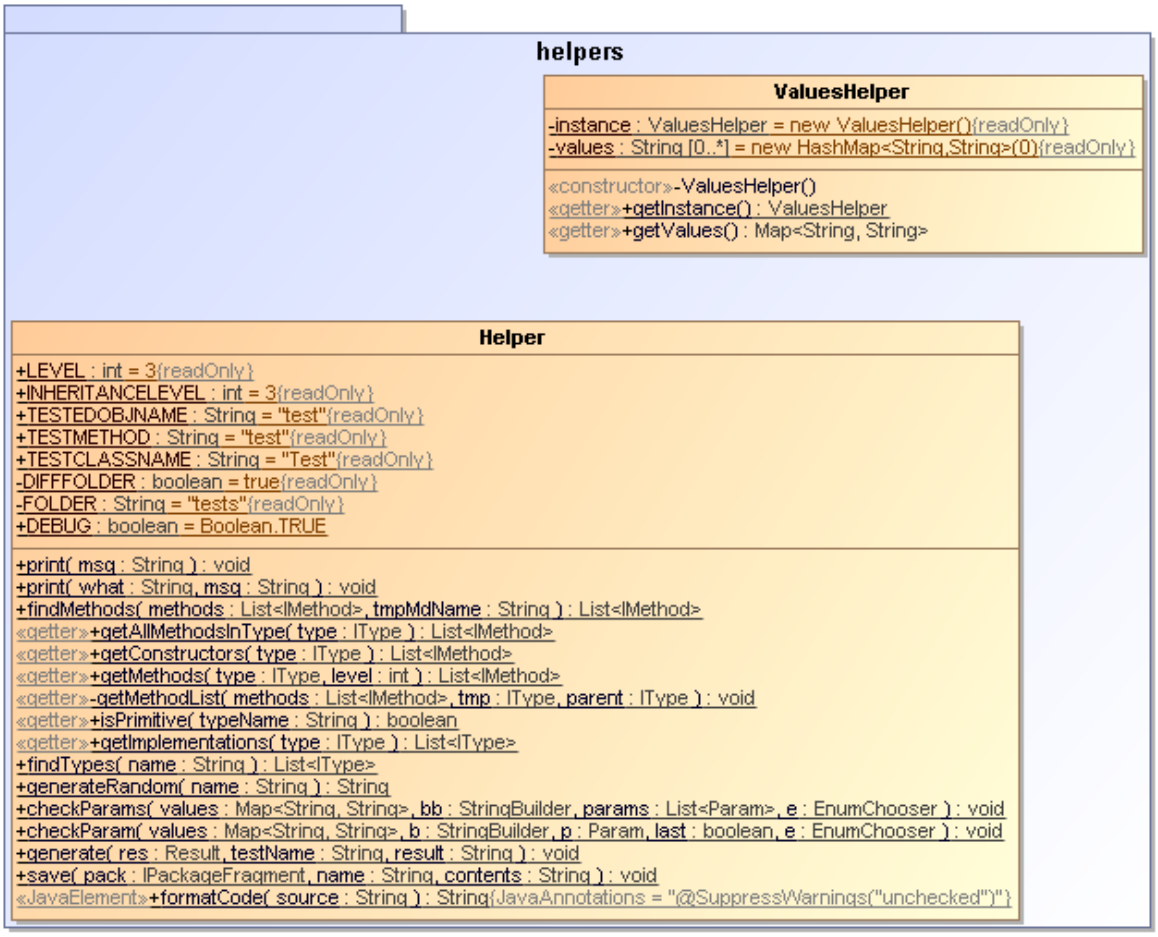
Šis paketas aprašo klases reikalingas objektų būsenų ir jų sąlygų aprašymui.



17 pav. Lib paketas

3.4.1.5. Helpers paketas

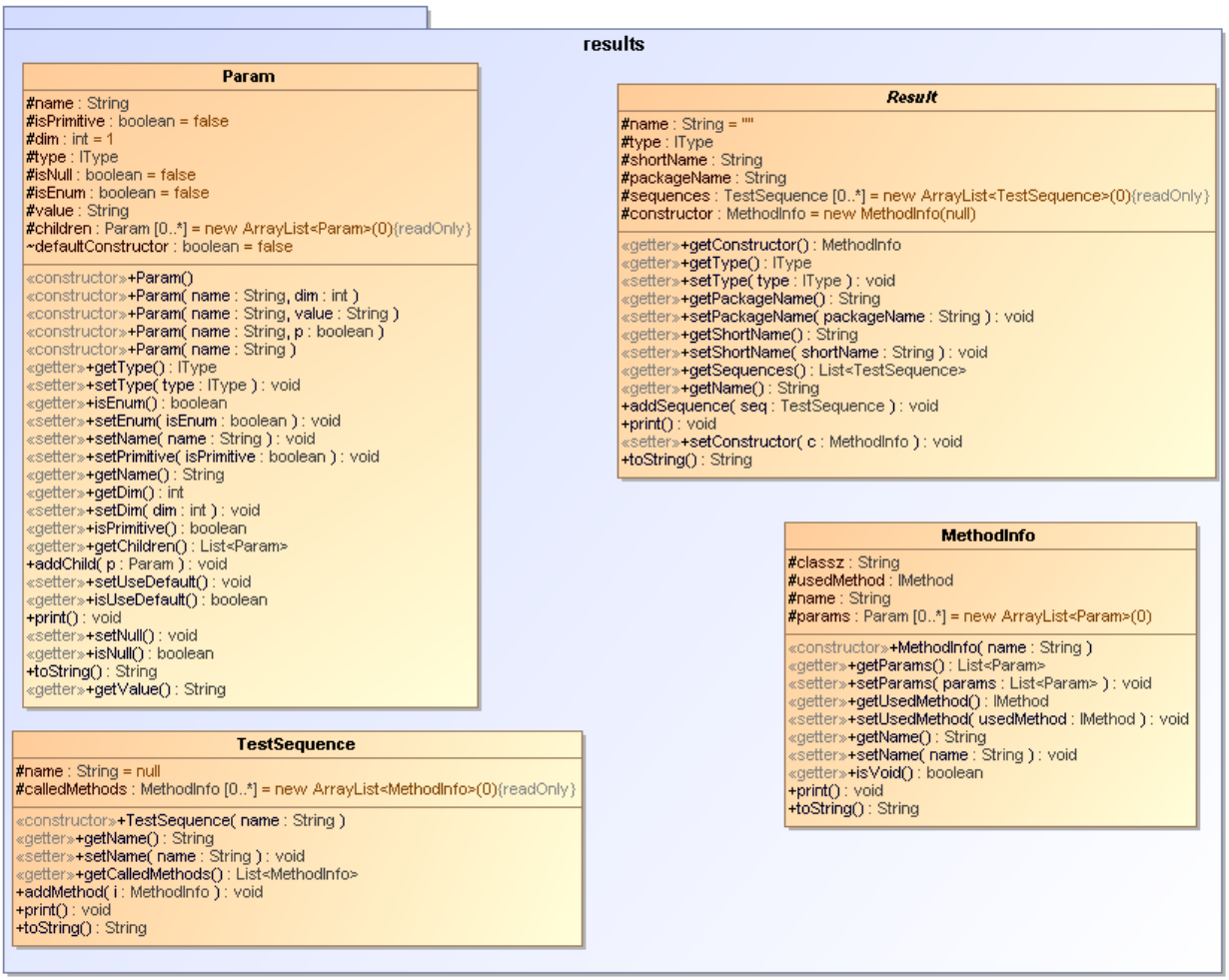
Šiame pakete saugomos pagalbinės klasės.



18 pav. Helpers paketas

3.4.1.6. Results paketas

Šiame pakete aprašytos klasės testuojamų klasių, metodų ir būsenų aprašymui, kurias naudoja Analyzer ir Generator paketai tarpusavio bendradarbiavimui.



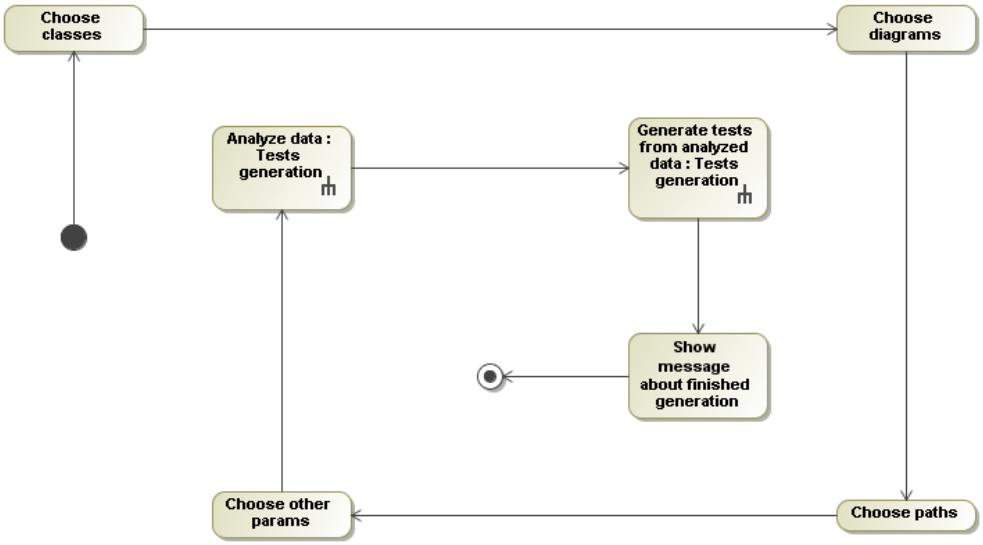
19 pav. Results paketas

3.4.2. Dinaminis sistemos vaizdas

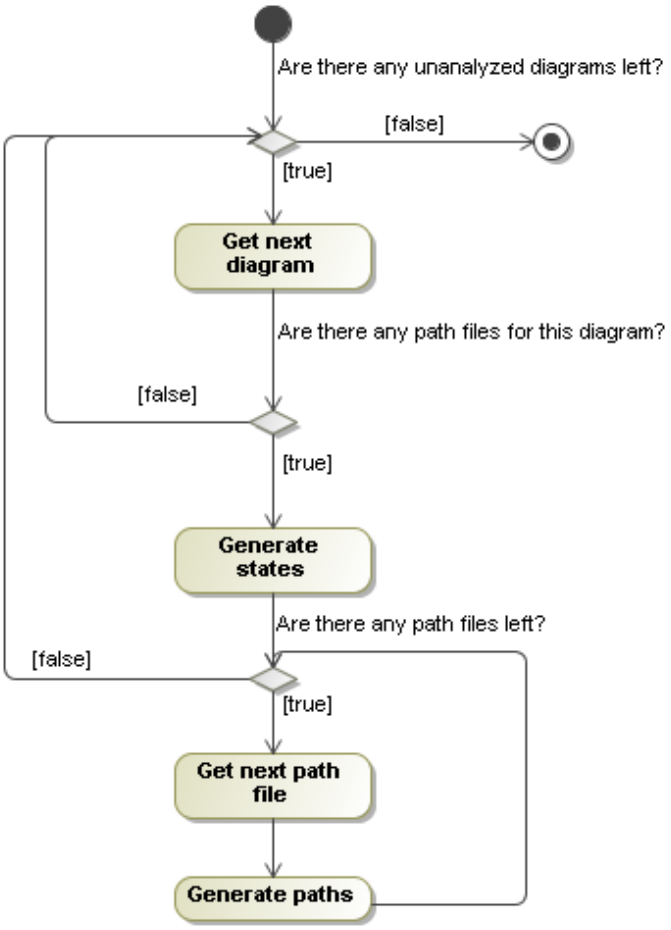
Šiame skyriuje pateikiamos sistemos veiklos ir būsenų diagramos nusakančios pagrindinius įskiepio veikimo principus.

3.4.2.1. Įskiepio veiklos diagramos

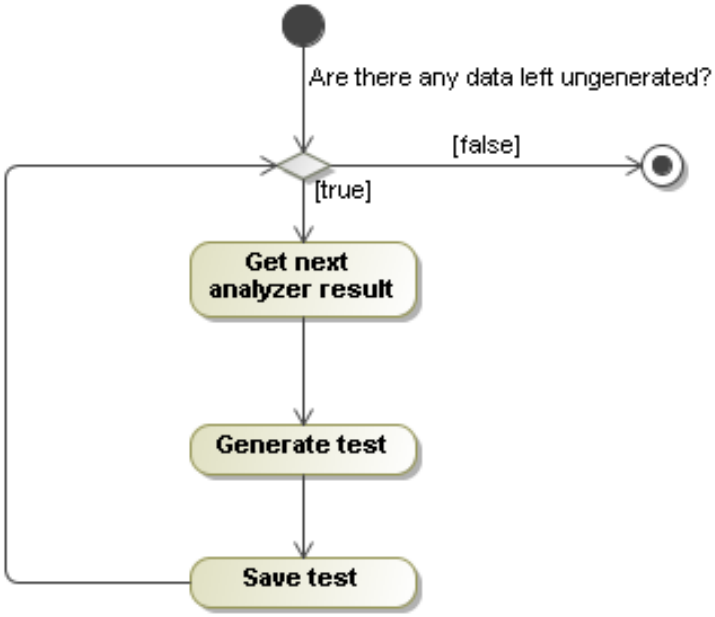
20 pav. pavaizduotas bendras sistemos veikimas. 21 pav. – detalizuotas duomenų analizės procesas, o 22 pav. – testų generavimo procesas.



20 pav. Testų generavimo schema



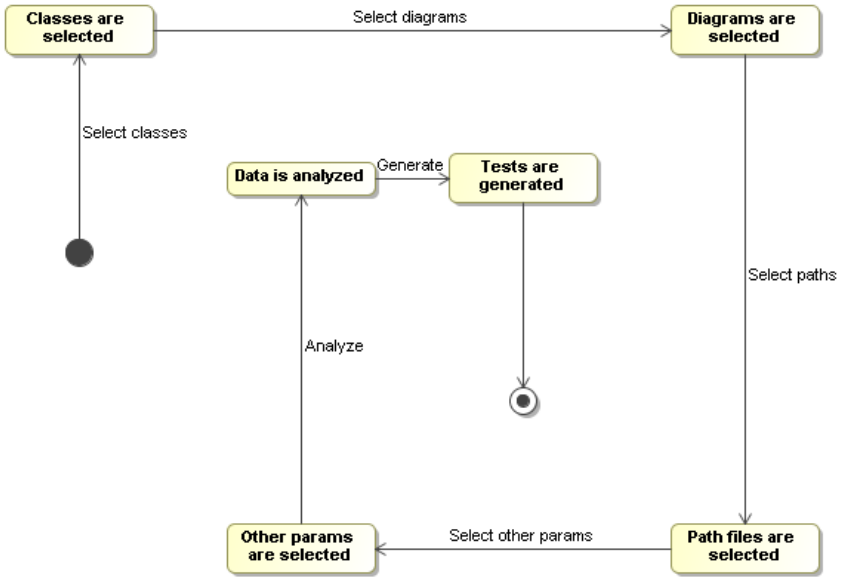
21 pav. Duomenų analizavimo schema



22 pav. Testų klasių generavimo schema

3.4.2.2. Įskiepio būsenų diagramos

Testų generavimą aprašo 23 pav. Egzistuoja 6 nuoseklios būsenos. Neparinkus klasių failų, negalima parinkti diagramų, neparinkus jų negalima nurodyti kelių ir pan.



23 pav. Testų generavimo būsenų diagrama

4. SPRENDIMO REALIZACIJA IR TESTAVIMAS

Šiame skyrelyje bus aptarta kaip buvo realizuotas ir kaip atitinka reikalavimus sukurtas sprendimas aprašytas 3 skyrelyje. Be to yra aptariami atlikti patobulinimai.

4.1. Sprendimo realizacijos ir veikimo aprašas

Sprendimas buvo realizuotas pagal 3-iame skyrelyje nurodytus reikalavimus. Vartotojas turėjo turėti UML būsenų modelio diagramą, perėjimų failą, bei realizuotą klasę norėdamas generuoti testus. Realizuoto įskiepio metrikos pateikiamos žemiau esančioje lentelėje.

Lentelė 23. Realizuoto įskiepio metrikos

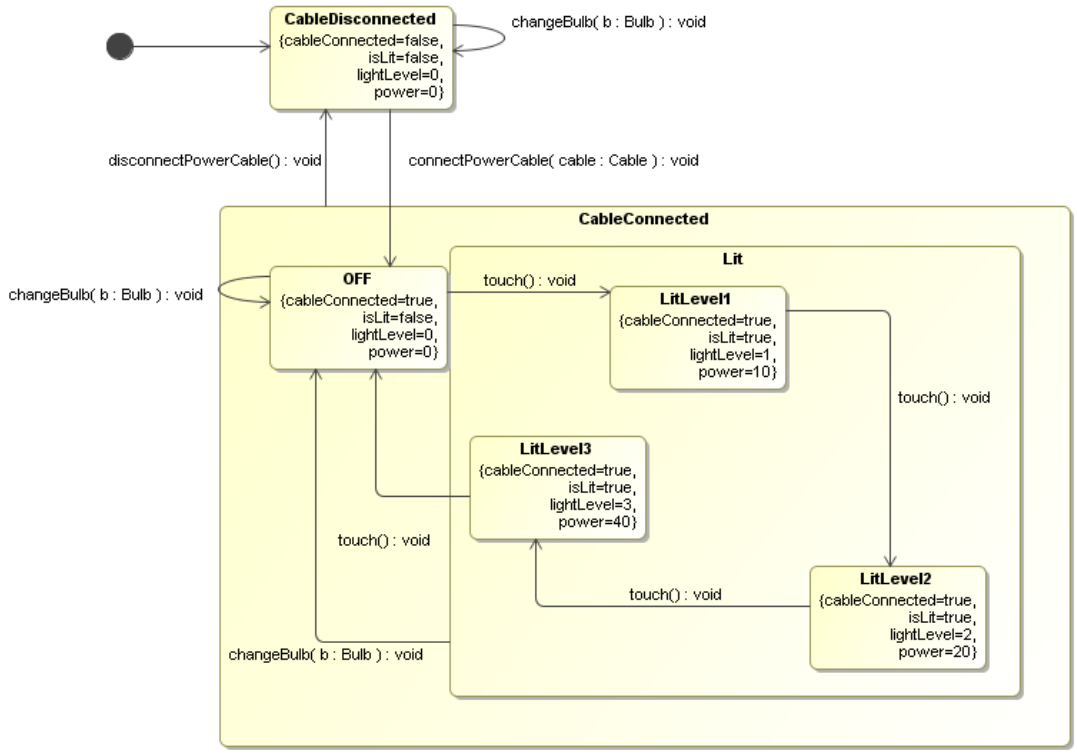
Metrika	**Reikšmė**
Bendras kodo eilučių kiekis (LOC)	3569
Kodo eilučių kiekis (MLOC)	2133
Paketų kiekis	34
Klasių kiekis	60
Metodų kiekis	273

4.2. Testavimo modelis, duomenys, rezultatai

Sukurto įskiepio metu testavimo buvo tikrinama ar nurodytai diagramai:

- sugeneruojami teisingi perėjimai
- teisingai sugeneruojamos būsenos
- testai veikia.

Įskiepiui buvo nurodyta žemiau pateikiama diagrama.



24 pav. Įskiepio tyrimui naudota diagrama

4.2.1. Perėjimų generavimas

Buvo tikrinama kaip įskiepis sugeneruoja perėjimus tarp būsenų iš vartotojo pateikto perėjimų failo.

Failo pavyzdys:

```
org.testing.lamp.TableLamp

CableDisconnected

connectPowerCable(),touch() LitLevel1

connectPowerCable()[OFF],touch()[LitLevel1],changeBulb() OFF
```

Sistema turėjo sugeneruoti 3 testų metodus, kuriuose kvietė metodus nurodytus faile. Tiek testų ir buvo sugeneruota. O šių sugeneruoti testai pateikiami žemiau esančioje lentelėje.

Lentelė 24. Įskiepio sugeneruotos sekos testuojamai klasei

Seka	Sugeneruotas testas

CableDisconnected	`@Test`
	`public void testSTR_CableDisconnected1() {`
	`assertTrue("Check CableDisconnected state", states.get("CableDisconnected").isStateOK(test));`
	`}`

Seka	Sugeneruotas testas

connectPowerCable(),touch() LitLevel1	`@Test`
	`public void testSTR_LitLevel15() {`
	`test.connectPowerCable(new org.testing.lamp.cables.Power220Cable(-1542495731));`
	`test.touch();`
	`assertTrue("Check LitLevel1 state", states.get("LitLevel1").isStateOK(test));`
	`}`

connectPowerCable()[OFF],touch()[LitLevel1],changeBulb() OFF	`@Test`
	`public void testSTR_OFF4() {`
	`test.connectPowerCable(new org.testing.lamp.cables.Power220Cable(-1784213145));`
	`assertTrue("Check OFF state", states.get("OFF").isStateOK(test));`
	`test.touch();`
	`assertTrue("Check LitLevel1 state", states.get("LitLevel1").isStateOK(test));`
	`test.changeBulb(new org.testing.lamp.bulbs.HalogenBulb(new java.lang.String(), 395629368, 1039236263, 17.206364896941505));`
	`assertTrue("Check OFF state", states.get("OFF").isStateOK(test));`
	`}`

Taigi, įskiepis teisingai sugeneravo sekas.

4.2.2. Būsenų generavimas

Tiriant įskiepį taip pat buvo patikrinta ar jis teisingai sugeneruoja būsenų sąrašus. Jam buvo nurodyta 24 pav. pavaizduota diagrama. Įskiepis sugeneravo 5 būsenas, nevertinant sudėtinių būsenų. Sąlygos buvo nurodytos teisingai. Viena iš sugeneruotų būsenų pateikiama žemiau.

```
{
        State s = new StateImpl("CableDisconnected");
        s.addInvariant(new InvariantImpl("isCableConnected", "false"));
        s.addInvariant(new InvariantImpl("isLit", "false"));
        s.addInvariant(new InvariantImpl("usedPowerInW", "0"));
        s.addInvariant(new InvariantImpl("lightLevel", "0"));
        states.put("CableDisconnected", s);
}
```

4.2.3. Testų tikrinimas

Būsenų tikrinimo metu, buvo paleistas sugeneruotas testas, patikrinant ar teisingai aptinkamos būsenos, tuomet pakeičiamos būsenų sąlygos ir tikrinama ar testai tai aptinka. Testai sėkmingai aptikdavo pasikeitusias būsenų sąlygas.

4.3. Atlikti patobulinimai

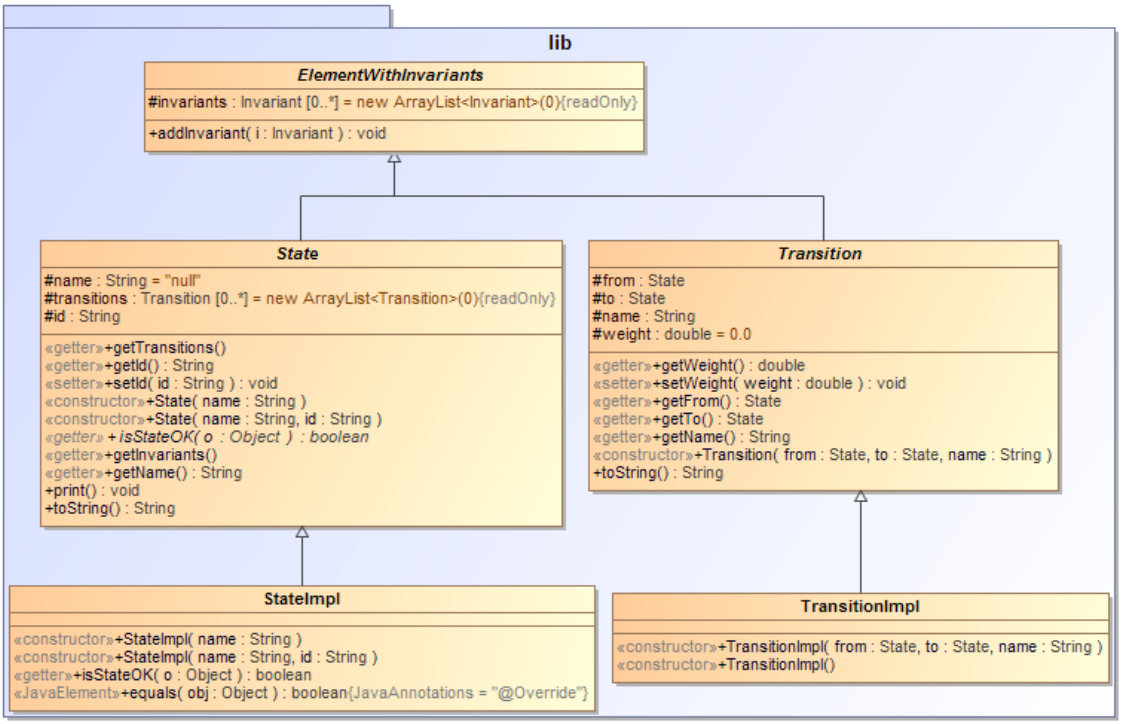
Prieš atliekant eksperimentą buvo tobulinamos šios dalys:

- Būsenų perėjimų nuskaitymas
- Testavimo kelių parinkimas
- Reikšmių naudojamų testų metodų generavimas

Toliau atlikti patobulinimai bus aptariami detaliau.

4.3.1. Būsenų perėjimų nuskaitymas

Buvo nuspręsta patobulinti testavimo kelių parinkimą, kad sekos būtų parenkamos automatiškai. Tam atlikti, reikėjo papildyti testavimo bibliotekos ir analizatoriaus paketus, kad būtų nuskaitomi ir išsaugomi perėjimai tarp būsenų. Analizatoriaus pakete buvo sukurta nauja klasė nuskaitanti būsenas ir perėjimus tarp jų, o testavimo bibliotekoje pridėtos naujos klasės aprašančios perėjimą tarp būsenų. Šis pakeitimas atvaizduotas žemiau esančiame paveiksliuke.



25 pav. Papildytas testavimo bibliotekos paketas

4.3.2. Testavimo kelių automatinis parinkimas

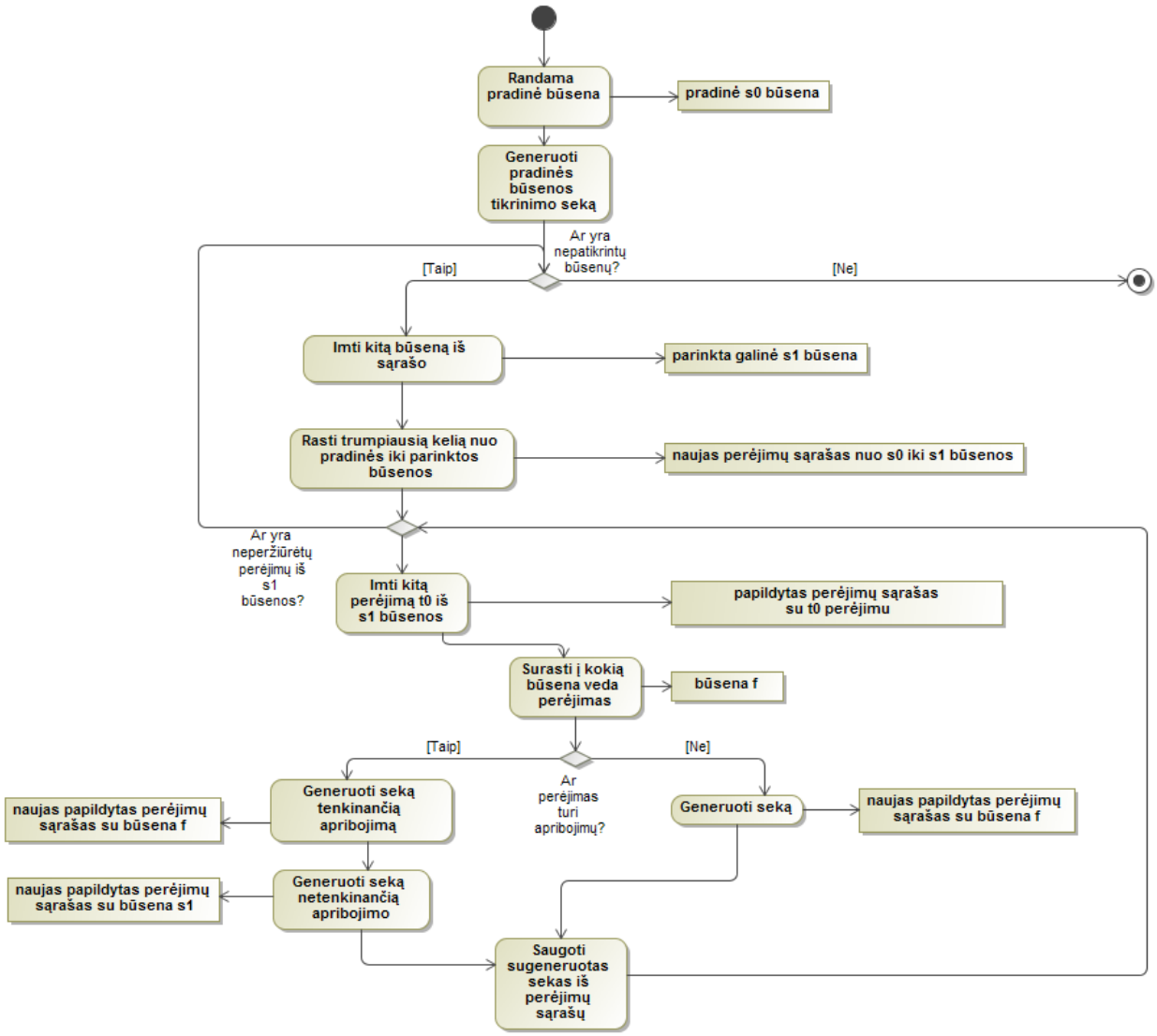
Iki šio pakeitimo, vartotojai turėdavo patys nurodyti norimas testavimo sekas tekstiniame faile, nurodant testuojamos klasės vardą, kviečiamus metodus ir kokių būsenų tikisi :

```
// nurodoma testuojama klasė
org.testing.demo.Lamp
// nurodomi kviečiami metodai ir galinės būsenos kurių tikimasi
on(),off(),on ON
off() OFF
```

Tačiau taikant tokį formatą galima netyčia įvelti klaidų – nurodoma ne ta klasė, praleidžiamas kablelis ir panašiai. Todėl buvo nuspręsta nuskaityti būsenų perėjimus iš diagramos ir parinkti testavimo kelius taikant grafų teorijos algoritmus, bei ribinę analizę. Tačiau galimybė nurodyti testavimo kelius buvo palikta.

Norint automatiškai parinkti kelius, buvo pritaikyta JGrapht biblioteka. Ji buvo pasirinkta, nes palaiko orientuotus, svorinius grafus, be to jau turi aprašytą Deikstros algoritmo realizaciją, su kuriuo bus atliekamas eksperimentas. Šis algoritmas randa trumpiausią kelią tarp dviejų viršūnių. Jo sudėtingumas yra $O(n^2)$.

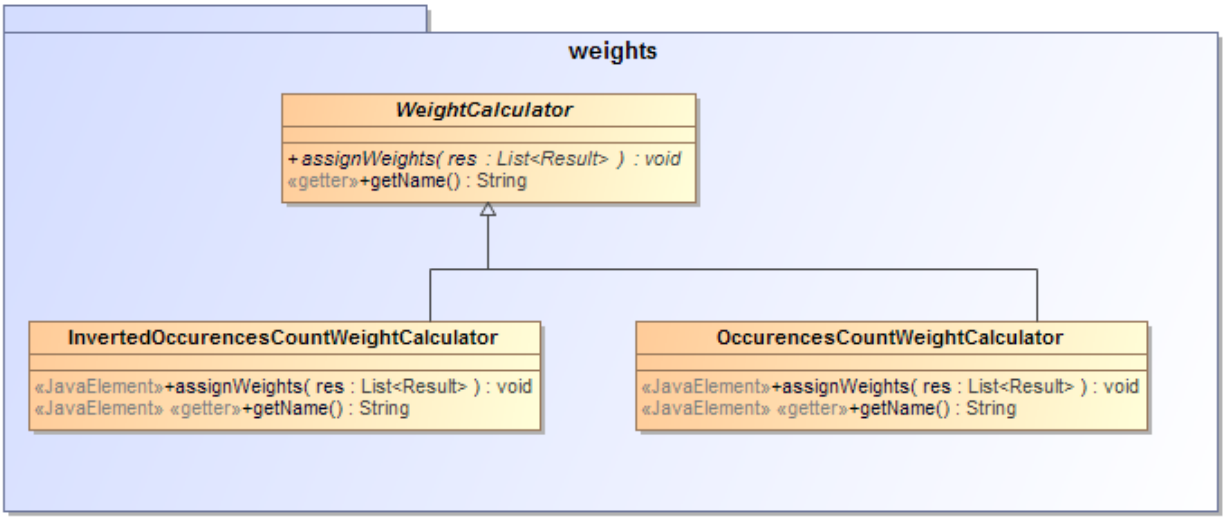
Testavimo sekoms generuoti buvo tikrinami perėjimai tarp jų. Realizuotas testavimo sekų generavimo algoritmas pavaizduotas žemiau esančiame paveiksliuke.



26 pav. Testavimo sekų parinkimo veiklos diagrama

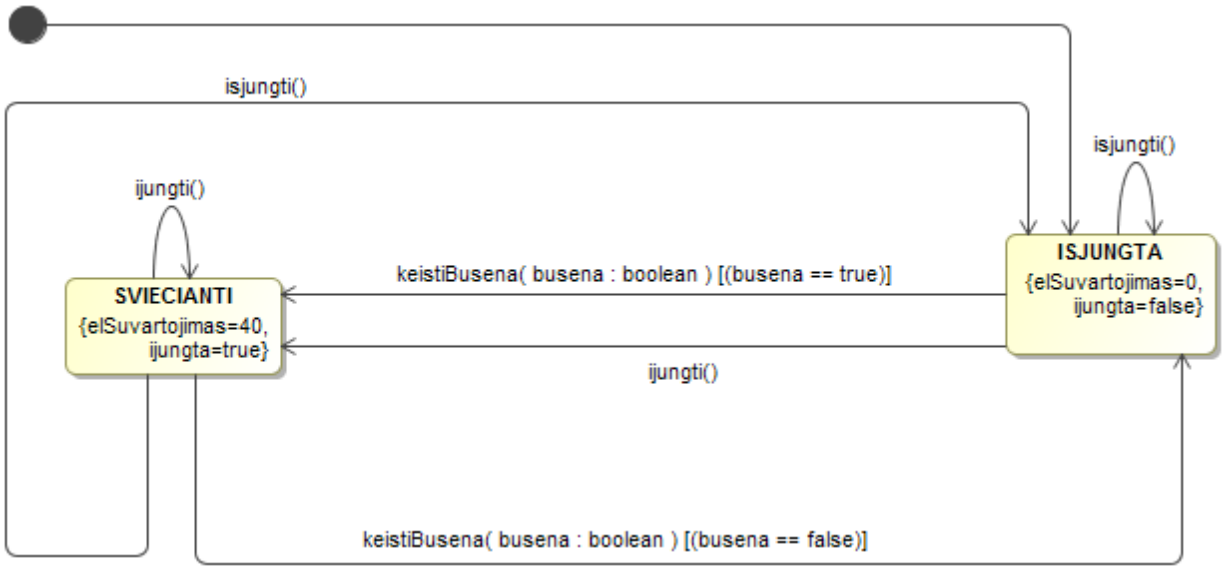
Visų pirma yra nustatoma pradinė būsena, tuomet imama kiekviena galima sistemos būsena ir generuojamos kelias iki parinktos būsenos. Tada tikrinama į kur galima pereiti iš šios būsenos ir kokias sąlygas reikia tenkinti norint pereiti į šią būseną. Papildomai generuojamos sekos, kur būtų tikrinama kas nutinka kai perėjimo sąlyga netenkinama. Taip padengiamos visos būsenos ir perėjimai tarp jų.

Taip pat norint kuo efektyviau taikyti Deikstros algoritmą, reikėtų suteikti perėjimams svorius. Norint atlikti šią užduotį, įskiepis buvo papildytas klasėmis, kurios randa kiek kartų metodas yra kviečiamas programoje ir šį skaičių priskiria perėjimui. Taip pat buvo realizuota klasė, kuri taip pat suranda, kiek kartų metodas kviečiamas projekte, tačiau priskiria šį svorį apskaičiuotą pagal formulę 1/(n+1), kur n yra kartų skaičius. Šioje formulėje yra pridedamas 1, nes metodas gali būti ir visai nekviečiamas, tuomet jo n reikšmė yra lygi 0, o dalyba iš 0 negalima. Šiomis klasėmis buvo papildytas analizatoriaus paketas. Realizuoto paketo struktūra pateikiama žemiau esančiame paveiksliuke.



27 pav. Analizatoriaus Weights paketo struktūra

Toliau bus pademonstruotas sekų generavimas. Įskiepiui buvo nurodyta diagrama pateikiama žemiau.



28 pav. Testavimo sekų generavimas

Nuskaičius būsenas ir perėjimus, skaičiuojant kiek kartų metodai kviečiami kode, perėjimams buvo priskirti svoriai pavaizduoti žemiau esančioje lentelėje.

Lentelė 25. Priskirti būsenų perėjimų svoriai

Iš būsenos:	Į būseną:	Metodas:	Perėjimo svoris:
ISJUNGTA	ISJUNGTA	isjungti	5.0
ISJUNGTA	SVIECIANTI	ijungti	12.0
ISJUNGTA	SVIECIANTI	keistiBusena	6.0
SVIECIANTI	SVIECIANTI	ijungti	12.0
SVIECIANTI	ISJUNGTA	isjungti	5.0
SVIECIANTI	ISJUNGTA	keistiBusena	6.0

Tuomet, tokiai sistemai bus sugeneruoti keliai pateikiami žemiau esančioje lentelėje.

Lentelė 26. Parinktos testavimo sekos

Seka	Galinė būsena
	ISJUNGTA
Išjungti	ISJUNGTA
Ijungti	SVIECIANTI
keistiBusena() // sąlyga turi būti tenkinama	SVIECIANTI
keistiBusena()// sąlyga turi būti netenkinama	ISJUNGTA
ijungti ijungti	SVIECIANTI
ijungti išjungti	ISJUNGTA
ijungti keistiBusena() //sąlyga turi būti tenkinama	ISJUNGTA
Ijungti keistiBusena()// sąlyga turi būti netenkinama	SVIECIANTI

4.3.3. Reikšmių generavimas

Testų generavimas yra sudėtingas uždavinys, ypač kai objektų kūrimui ir metodų kvietimui reikalingos reikšmės. Tokių duomenų generavimas gali būti daug laiko sunaudojantis ir sunkus uždavinys. Galimi įvairūs duomenų generavimo variantai – atsitiktinių reikšmių generavimas, konstantų naudojimas, ribinių reikšmių metodas ir panašiai, tačiau vis dėl to reikėtų atrinkti tik tas reikšmes, kurioms esant padengiama kuo daugiau kodo. Šiai problemai spręsti buvo pasirinkta ribinių reikšmių analizė, kai yra nuskaitomi perėjimų apribojimai.

Pavyzdžiui, 28 pav. pavaizduotoje diagramoje yra pateikiami apribojimai *keistiBusena()* metodams. Kaip galima matyti, kad esant būsenoje SVIECIANTI, norint pereiti į būseną ISJUNGTA ir kviečiant metodą *keistBusena()*, perėjimas įvyks tik tada, kai jo parametras *busena* bus lygus *true*. Tuomet generuojant reikšmes testų metodams, galima imti *true* ir *false* reikšmes, be to taip buvo papildytos testavimo sekos, kur tikrinama ar teisingai vyksta perėjimai tarp būsenų, kai šios sąlygos netenkinamos. Žemiau esančioje lentelėje pavaizduotos sugeneruotos sekos ir metodų reikšmės 28 pav. esančiai diagramai.

Lentelė 27. Testavimo sekos taikant ribines reikšmes

Seka	**Galinė būsena**
	ISJUNGTA
Išjungti()	ISJUNGTA
Ijungti()	SVIECIANTI
keistiBusena(true)	SVIECIANTI
keistiBusena(false)	ISJUNGTA
ijungti() ijungti()	SVIECIANTI
ijungti() išjungti()	ISJUNGTA
ijungti() keistiBusena(false)	ISJUNGTA
ijungti() keistiBusena(true)	SVIECIANTI

Norint nuskaityti metodų apribojimus ir generuoti sekas, buvo papildyti *XMLVisitorImpl* klasė, testavimo bibliotekos paketas (pakeitimas matomas 25 pav.) ir testų generavimo taisyklė *StateRule*. Metodų apribojimams galima tikrinti tokias reikšmes ir apribojimus:

- Skaičiams: >=, >, <, <=, ==, !=
- Loginio tipo kintamiesiems: ==, !=
- Tekstui: ==, !=
- Objektams: ==, != (tik null reikšmėms)

5. EKSPERIMENTINIS TYRIMAS

Šiame skyriuje pateikiami eksperimento rezultatai, kur buvo tiriami sugeneruoti testai, taikant UML būsenų diagramas.

5.1. Eksperimento planas

Eksperimentas buvo atliekamas su 5 klasėmis. Kiekvienai klasei buvo vykdomas toks scenarijus:

1. Generuojami testai, taikant Deikstros algoritmą, kai perėjimams priskiriamas svoris lygus metodų kvietimo kartų skaičiui ir yra taikomi perėjimų apribojimai.
2. Pamatuojami kiek testuojamos klasės padengia sugeneruoti testai
3. Sugeneruojami mutantai.
4. Tikrinama kiek klaidų sugeneruoti testai aptinka mutantuose.

5.2. Testuojamos klasės

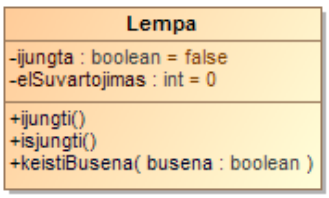
Testai buvo generuojami tokioms klasėms:

- Lempos klasei
- Bankomato klasei
- CheckLogic klasei
- Txt2Pdf klasei
- Kalkuliatoriaus klasei.

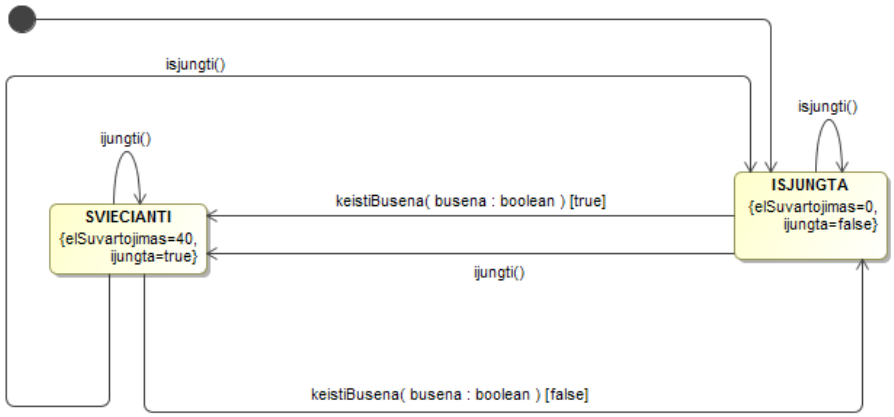
Toliau pateikiami jų aprašymai.

5.2.1. Lempos klasė

Ši klasė simuliuoja stalinės lempos veikimą. Jos klasių ir būsenų diagramos pateikiamos žemiau esančiuose paveiksliukuose.



29 pav. Klasės simuliuojančios lempos veikimą klasės diagrama



30 pav. Klasės simuliuojančios lempos veikimą būsenų diagrama

Šios klasės metrikos pateikiamos žemiau esančioje lentelėje.

Lentelė 28. Klasės simuliuojančios lempos veikimą metrikos

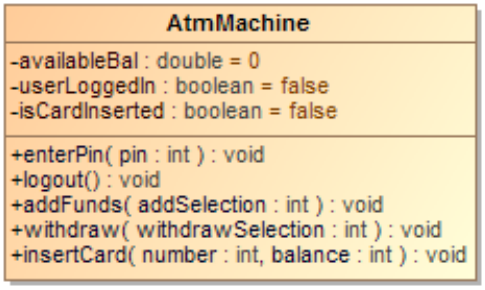
Metrika	**Reikšmė**
LOC (kodo eilučių kiekis)	19
MLOC (metodų kodo eilučių kiekis)	9
NOF (klasės laukų kiekis)	2
NOM (metodų kiekis)	3
WMC (ciklomatinis sudėtingumas)	4

Lentelė 29. Klasės simuliuojančios klasės veikimą būsenų apribojimai

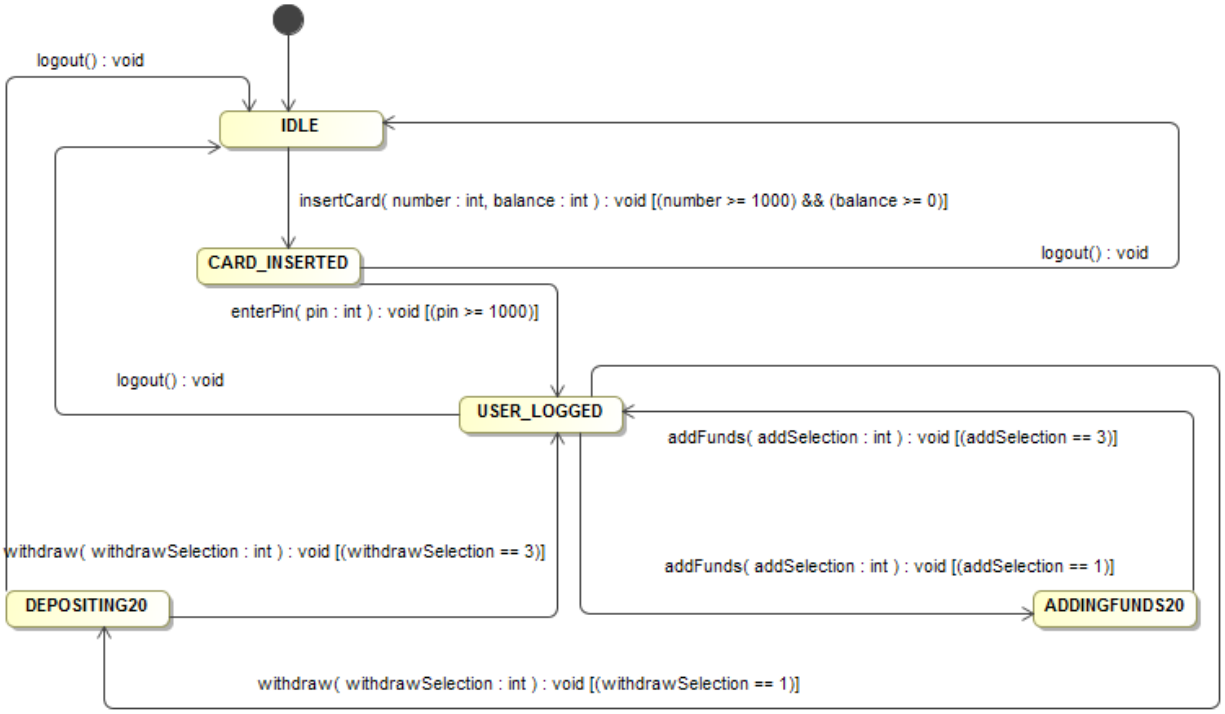
Būsena	**Apribojimai**
SVIECIANTI	elSuvartojimas = 40 ijungta = true
ISJUNGTA	elSuvartojimas = 0 ijungta = false

5.2.2. Bankomato klasė

Ši klasė simuliuoja bankomato veikimą.



31 pav. Bankomato klasės diagrama



32 pav. Bankomato klasės būsenų diagrama

Lentelė 30. Bankomato klasės metrikos

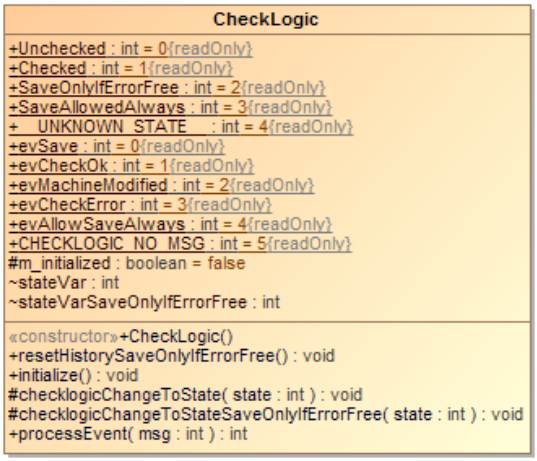
Metrika	**Reikšmė**
LOC (kodo eilučių kiekis)	48
MLOC (metodų kodo eilučių kiekis)	32
NOF (klasės laukų kiekis)	3
NOM (metodų kiekis)	5
WMC (ciklomatinis sudėtingumas)	20

Lentelė 31. Bankomato klasės būsenų apribojimai

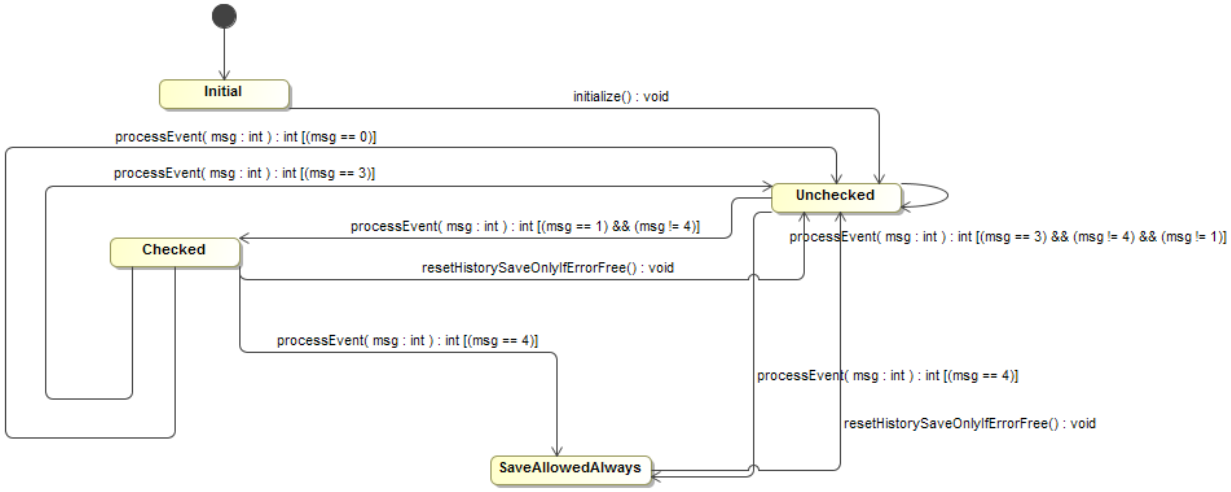
Būsena	**Apribojimai**
IDLE	userLoggedIn == false isCardInserted == false availableBal == 0
CARD_INSERTED	userLoggedIn == false isCardInserted == true availableBal >= 0
USER_LOGGED	userLoggedIn == true isCardInserted == true availableBal >= 0
ADDINGFUNDS20	userLoggedIn == true isCardInserted == true availableBal >= 0
DEPOSITING20	userLoggedIn == true isCardInserted == true availableBal >= 0

5.2.3. CheckLogic klasė

Ši klasė apdoroja įvykius ir tikrina ar galima atlikti failo saugojimą.



33 pav. CheckLogic klasės diagrama



34 pav. CheckLogic klasės būsenų diagrama

Lentelė 32. CheckLogic klasės metrikos

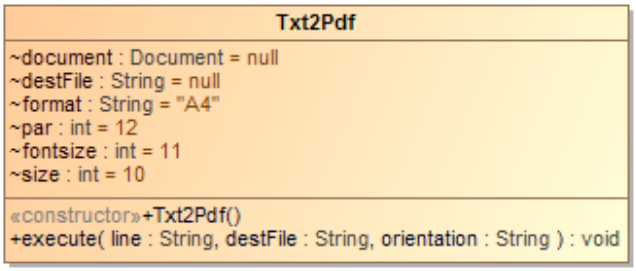
Metrika	Reikšmė
LOC (kodo eilučių kiekis)	82
MLOC (metodų kodo eilučių kiekis)	53
NOF (klasės laukų kiekis)	3
NOM (metodų kiekis)	6
WMC (ciklomatinis sudėtingumas)	19

Lentelė 33. CheckLogic klasės būsenų apribojimai

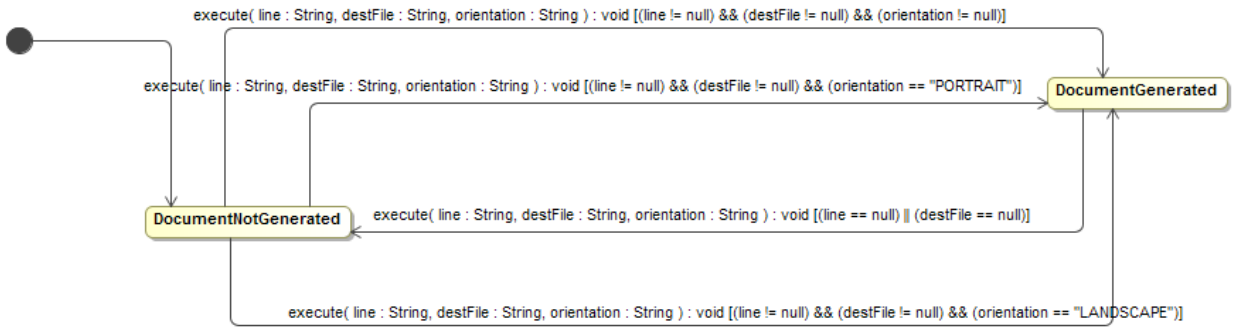
Būsena	Apribojimai
Initial	m_initialized == false
SaveAllowedAlways	stateVar == 3
Unchecked	stateVarSaveOnlyIfErrorFree == 0 stateVar == 2
Checked	stateVarSaveOnlyIfErrorFree == 1 stateVar == 2

5.2.4. Txt2Pdf

Ši klasė kuria PDF failą iš teksto.



35 pav. Txt2Pdf klasės diagrama



36 pav. Txt2Pdf klasės būsenų diagrama

Lentelė 34. Txt2Pdf klasės metrikos

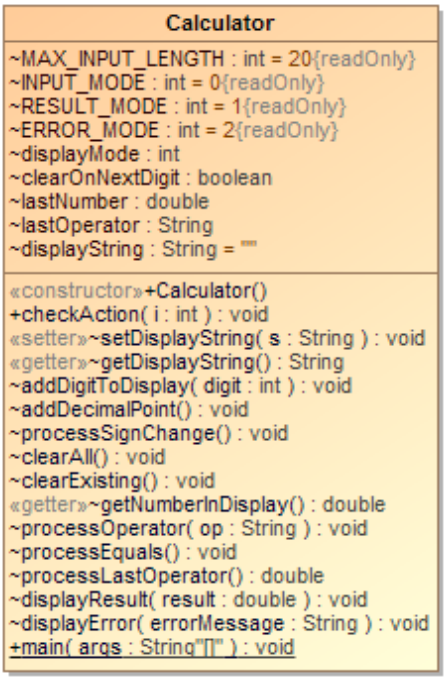
Metrika	Reikšmė
LOC (kodo eilučių kiekis)	75
MLOC (metodų kodo eilučių kiekis)	39
NOF (klasės laukų kiekis)	4
NOM (metodų kiekis)	6
WMC (ciklomatinis sudėtingumas)	10

Lentelė 35. Txt2Pdf klasės būsenų apribojimai

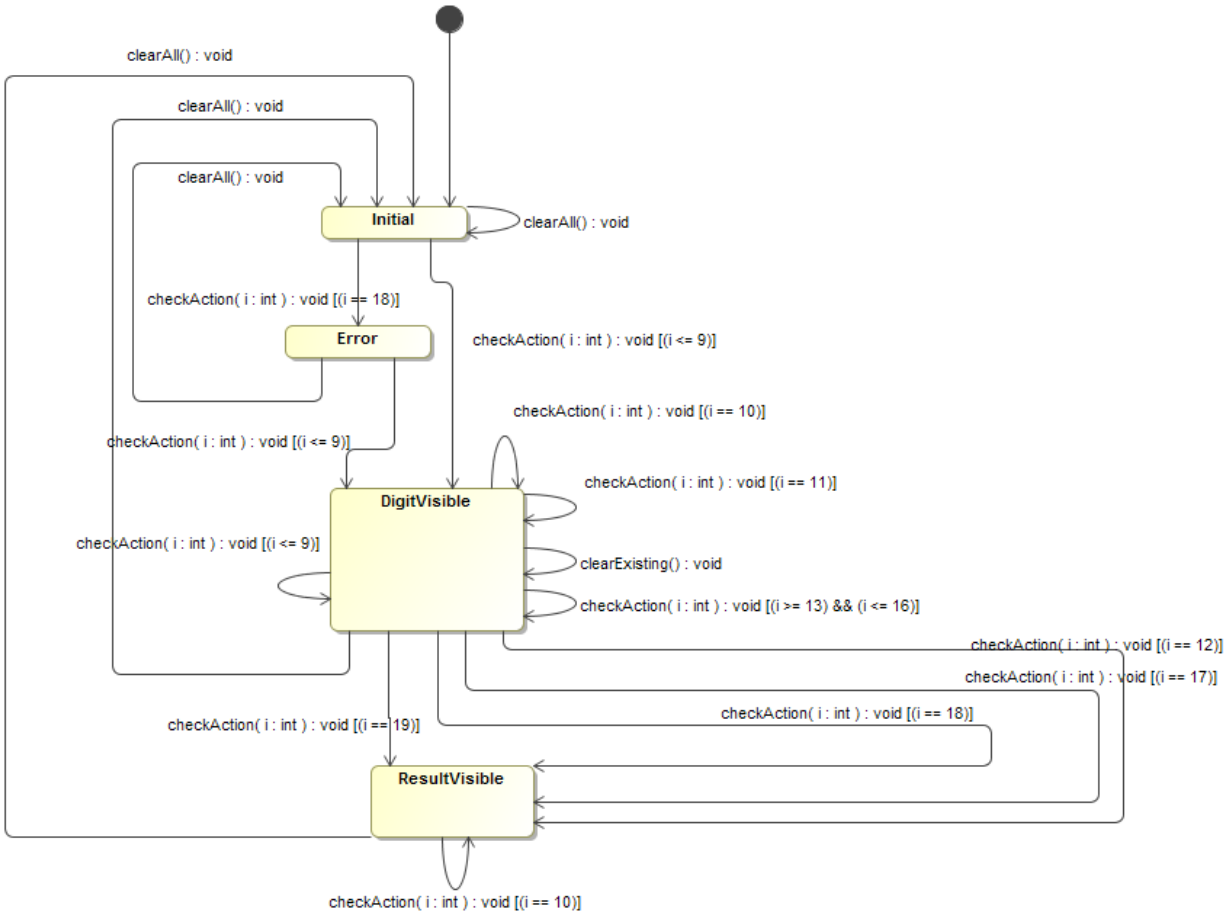
Būsena	Apribojimai
DocumentGenerated	format == „A4“ document != null destFile != null size == 10 fontsize == 11 par == 12
DocumentNotGenerated	format == „A4“ document == null destFile == null size == 10 fontsize == 11 par == 12

5.2.5. Kalkuliatorius

Ši klasė simuliuoja kalkuliatorių ir leidžia vartotojui atlikti matematinius veiksmus ir pateikia rezultatus.



37 pav. Kalkuliatoriaus klasės diagrama



38 pav. Kalkuliatoriaus būsenų diagrama

Lentelė 36. Kalkuliatoriaus klasės metrikos

Metrika	**Reikšmė**
LOC (kodo eilučių kiekis)	258
MLOC (metodų kodo eilučių kiekis)	207
NOF (klasės laukų kiekis)	10
NOM (metodų kiekis)	18
WMC (ciklomatinis sudėtingumas)	75

Lentelė 37. Kalkuliatoriaus klasės būsenų apribojimai

Būsena	**Apribojimai**
DigitVisible	displayString != null lastOperator != null displayMode != 2
ResultVisible	displayString != null lastOperator != null displayMode != 2 clearOnNextDigit == true
Initial	displayString == „0“ lastOperator != null displayMode == 0

	clearOnNextDigit == true
Error	displayMode == 2 displayString != null

5.3. Mutacinis testavimas

Mutavimas – yra klaidomis paremta strategija, kuri matuoja testavimo kokybę/pakankamumą tikrinant ar testas gali aptikti tam tikras klaidas. Pavyzdžiui, testuojamoje sistemoje sudėties operatorius pakeičiamas į atimties ir pan. [12]. Tad kiekvienas mutantas yra neteisinga programos realizacija, kurią testai turėtų aptikti.

Toks testų testavimas buvo atliekamas naudojant Jumble įrankį. Šis įrankis mutuoja klases ir vykdo testus. Vienu metu yra įvykdoma viena mutacija. Tokiu būdu yra tikrinamas testų efektyvumas. Jei mutantai neaptinkami, vadinasi testai nėra efektyvūs arba pakankami. Šis įrankis yra gali mutuoti šias operacijas:

- Sąlygos sakinius – pavyzdžiui, sąlyga x > y, pakeičiama į !(x > y)
- Dvejetainius aritmetinius operatorius:
  - + į – ir atvirkščiai
  - * į / ir atvirkščiai
  - % į *
  - & į | ir atvirkščiai
  - ^ į &
  - << į >> ir atvirkščiai
  - >>> į <<
- Padidinimo/Pamažinimo vienetu operatorius – pavyzdžiui i++ pakeičiama į i--
- Konstantas – jos yra pakeičiamos į reikšmę „__jumble__“ arba „___jumble___“
- Grąžinamas reikšmes – jei pvz.: metodas grąžina paprasto tipo reikšmes kaip int, double, boolean ir pan. šias reikšmes galima pakeisti t.y. vietoj ne nulinės reikšmės grąžinti 0 ir pan.
- Šakojimosi sakinius – šiuos sakinius Jumble įrankis gali mutuoti sukeičiant šakas vietomis [13].

5.4. Eksperimento rezultatai

1. Klasės simuliuojančios lempos veikimą rezultatai testavimo rezultatai pateikiami žemiau esančioje lentelėje, kurioje galima matyti, kad buvo padengta 100 % kodo ir aptikti visi 7 mutantai.

Lentelė 38. Lempos klasės testavimo rezultatai

Metrika	Reikšmė
LOC (kodo eilučių kiekis)	82
MLOC (metodų kodo eilučių kiekis)	28
NOM (metodų kiekis)	10
WMC (testo klasės ciklomatinis sudėtingumas)	10
Kodo padengimas, %	100
Sugeneruota mutantų	7
Aptikta mutantų, %	100

2. Lentelė 39 atvaizduoja bankomato klasės testavimo rezultatus. Buvo aptikta tik 45 % mutantų.

Lentelė 39. Bankomato klasės testavimo rezultatai

Metrika	Reikšmė
LOC (Sugeneruota kodo eilučių)	174
MLOC (metodų kodo eilučių kiekis)	72
NOM (metodų kiekis)	18
WMC (Testo klasės ciklomatinis sudėtingumas)	18
Kodo padengimas, %	85,6
Sugeneruota mutantų	32
Aptikta mutantų, %	45

3. CheckLogic klasės testavimo rezultatai yra pateikiami žemiau esančioje lentelėje. Joje galima matyti, kad buvo aptikta 75 % mutantų.

Lentelė 40. CheckLogic klasės testavimo rezultatai

Metrika	Reikšmė
LOC (Sugeneruota kodo eilučių)	155
MLOC (metodų kodo eilučių kiekis)	66
NOM (metodų kiekis)	18
WMC (Testo klasės ciklomatinis sudėtingumas)	18
Kodo padengimas, %	83,8
Sugeneruota mutantų	40
Aptikta mutantų, %	75

4. PDF klasės testavimo rezultatai yra pateikiami žemiau esančioje lentelėje. Joje galima matyti, kad buvo aptikta 42 % mutantų.

Lentelė 41. Txt2Pdf klasės testavimo rezultatai

Metrika	**Reikšmė**
LOC (Sugeneruota kodo eilučių)	89
MLOC (metodų kodo eilučių kiekis)	28
NOM (metodų kiekis)	11
WMC (Testo klasės ciklomatinis sudėtingumas)	11
Kodo padengimas, %	96,4
Sugeneruota mutantų	14
Aptikta mutantų, %	42

5. Kalkuliatoriaus klasės testavimo rezultatai yra pateikiami žemiau esančioje lentelėje. Joje galima matyti, kad buvo aptikta 32 % mutantų.

Lentelė 42. Kalkuliatoriaus klasės testavimo rezultatai

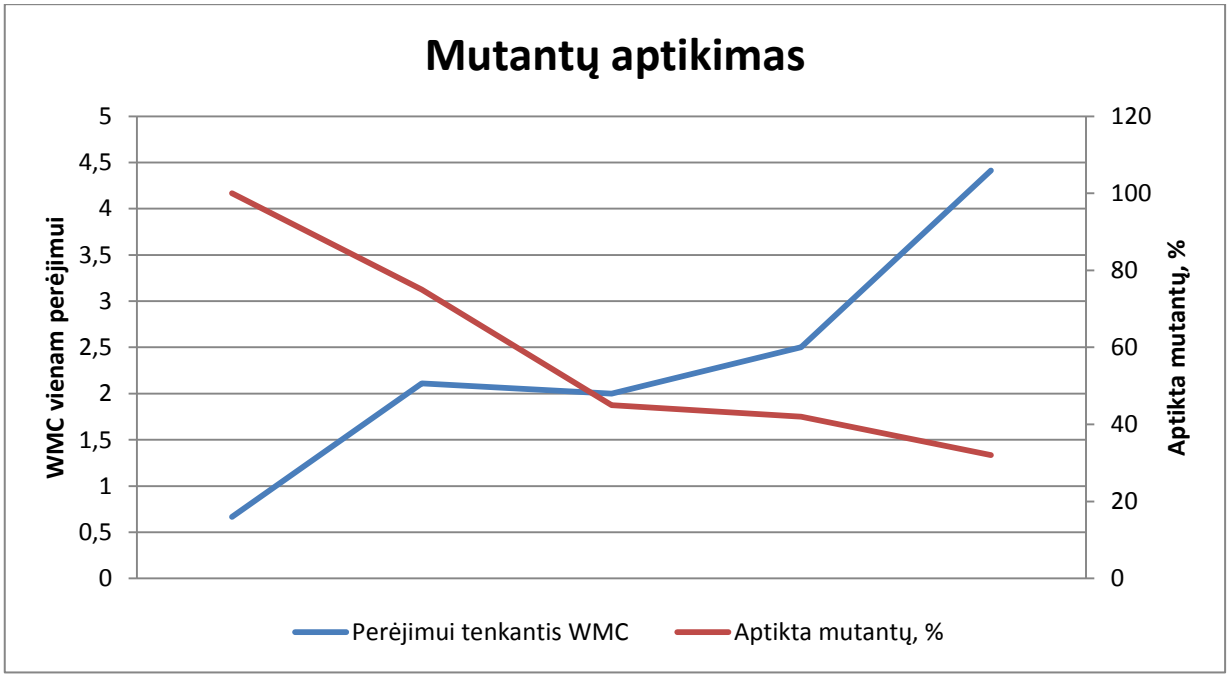
Metrika	**Reikšmė**
LOC (Sugeneruota kodo eilučių)	216
MLOC (metodų kodo eilučių kiekis)	95
NOM (metodų kiekis)	30
WMC (Testo klasės ciklomatinis sudėtingumas)	29
Kodo padengimas, %	65,5
Sugeneruota mutantų	113
Aptikta mutantų, %	32

Gavus aukščiau esančius eksperimento rezultatus, buvo paskaičiuoti papildomai tokie išvestiniai parametrai:

- 1 būsenai tenkantis perėjimų kiekis
- 1 būsenai tenkantis sąlygų kiekis
- 1 būsenos sąlygai tenkantis klasės laukų kiekis
- 1 perėjimui tenkantis klasės sudėtingumas
- Vidutinis būsenai tenkantis perėjimų kiekis

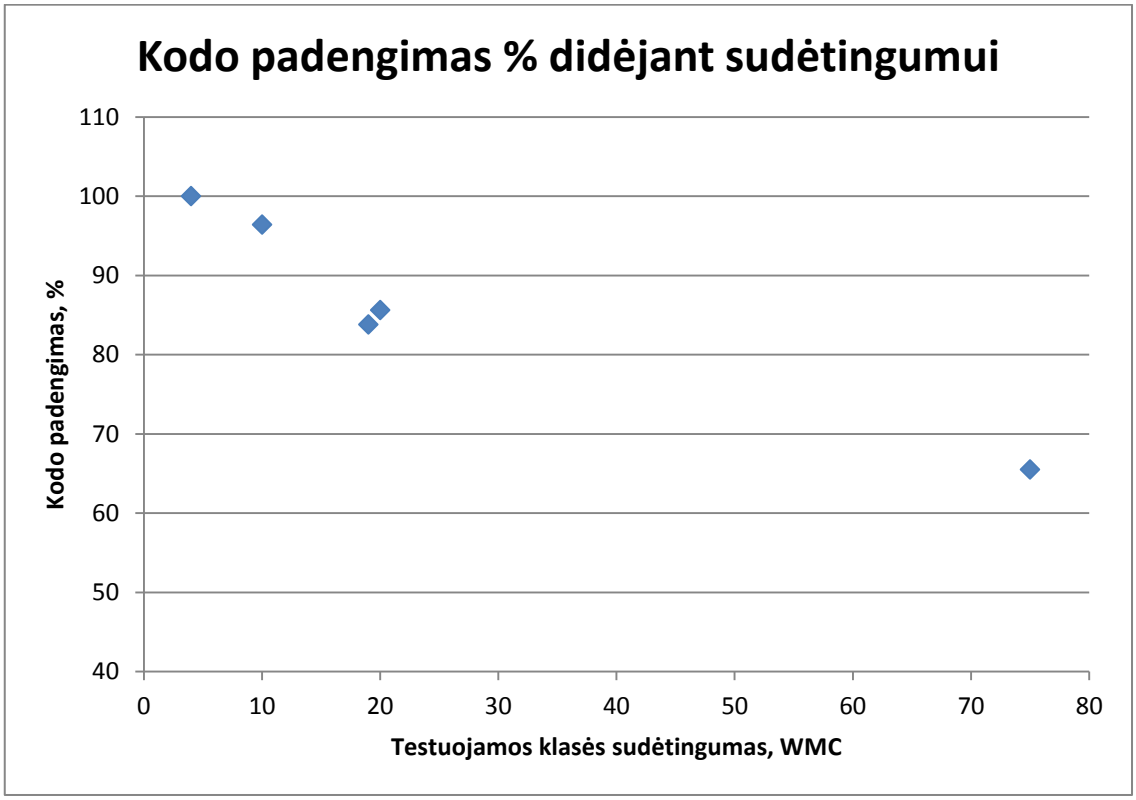
Tuomet, paskaičiavus šiuos parametrus buvo paskaičiuota parametrų koreliacija, norint nustatyti kurie parametrai labiausiai susiję su aptiktų mutantų ir padengtomis kodo eilutėmis. Koreliacijai paskaičiuoti naudotus duomenis galima rasti pirmame priede, o koreliacijos matricą galima rasti antrame priede.

Atlikus išmatuotų parametrų koreliacijos analizę, buvo nustatyta, kad aptiktų mutantų procentinį kiekį labiausiai įtakoja vienam būsenų diagramos perėjimui tenkantis klasės sudėtingumas. Tai atvaizduota žemiau esančiame paveiksliuke. Taigi, norint, kad generuojami testai būtų kuo efektyvesni, reikėtų, kad diagramoje būtų kuo daugiau perėjimų, kurie aprašytų testuojamos klasės galimus kelius, t.y. vienas perėjimas nusakytų vieną galimą sąlygą.



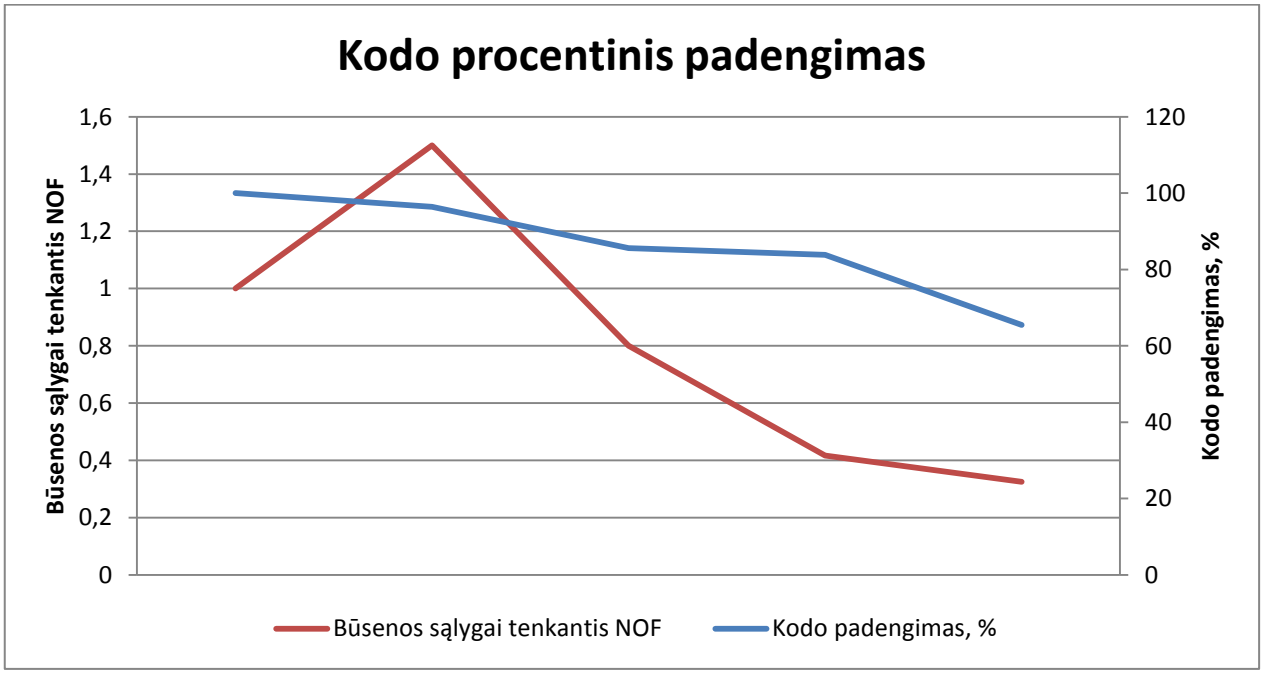
39 pav. Aptiktų mutantų procentinis kiekis

Taip pat buvo ištirta kas labiausiai įtakoja kodo procentinį padengimą. Iš koreliacijos matricos buvo nustatyta, kad labiausiai procentinis kodo padengimas teigiamai koreliuoja su būsenos sąlygai tenkančiu klasės laukų skaičiumi ir testuojamos klasės sudėtingumu.



40 pav. Kodo padengimo % priklausomybė nuo testuojamos klasės sudėtingumo

Taip pat, kaip galima matyti žemiau esančiame paveiksliuke, generuojant testus taikant būsenų modelį ir norint kuo didesnio kodo padengimo, reikėtų, kad kiekviena būsena nusakytų visus galimus klasės laukus tuo metu.



41 pav. Kodo procentinis padengimas

5.5. Sprendimo taikymo rekomendacijos

UML būsenų diagramas galima taikyti testuoti toms klasėms, kurių būsenas galima nusakyti klasės laukais, tai yra, kur klasės laukų kiekis > 0. Taip pat, norint, kad testai aptiktų kuo daugiau klaidų, jei testuojamoje klasėje yra metodų, kuriems reikia konkrečių reikšmių kvietimui, reikia nurodyti kviečiamiems metodams kuo daugiau apribojimų.

6. REZULTATŲ APIBENDRINIMAS IR IŠVADOS

Šiame darbe buvo aptarta kaip galima generuoti vienetų testus klasėms taikant UML būsenų diagramas. Buvo pasiūlytas ir ištirtas metodas leidžiantis generuoti būsenų testus, kur testavimo sekų generavimui naudojama UML būsenų modelio perėjimai, o šių sekų rezultatų teisingumo tikrinimui – būsenose nurodytos sąlygos, kurias turi tenkinti testuojamas objektas esant šioje būsenoje.

Tokio metodo pagrindinis privalumas tas, kad norint generuoti testus nereikia atskleisti realizuotų klasių laukų, nes būsenų sąlygų tikrinimui naudojama refleksija. Be to testavimo sekos yra parenkamos automatiškai taikant Deikstros algoritmą ir ribinių reikšmių analizę, kas leidžia pasiekti didesnį kodo padengimą, nes yra aplankomos visos būsenos ir visi perėjimai tarp jų.

Taip pat buvo ištirtas ir patobulintas suprojektuotas įrankis, gebantis generuoti automatinius testus taikant UML būsenų modelį. Eksperimento metu buvo tiriamos 5 klasės, kurios skyrėsi savo sudėtingumu ir būsenų diagramų aprašymo pilnumu. Joms buvo generuojami testai ir tikrinama, kiek kodo yra padengiama ir kiek mutantų šie testai aptinka. Buvo pasiektas net 100% klasių kodo ir mutantų aptikimas, tačiau eksperimento metu nustatyta, kad norint padidinti kodo padengimą ir mutantų aptikimą, reikia, kad UML būsenų diagramose butų nusakytos visos sistemos būsenos taikant visus galimus testuojamos klasės laukus, o perėjimams taip pat reikia nurodyti kiek įmanoma tikslesnius apribojimus.

7. LITERATŪROS SĄRAŠAS

[1] B. Hughe ir M. Cotterell, Software Project Management, London: Mcgraw Hill, 1999.

[2] P. Ammann ir J. Offutt, Introduction to Software Testing, Cambridge: Cambridge University Press, 2008.

[3] Guide to the Software Engineering Body of Knowledge, 2005.

[4] K. E.-F. Ibrahim ir J. A. Whittaker, Model-based Software Testing, Willey, 2001.

[5] OMG, OMG Unified Modeling Language Superstructure, 2011.

[6] T. Pocius, „InfoBitas," 19 03 2011. [Tinkle]. Adresas: http://www.infobitas.lt/index.php?option=com_content&view=article&id=114:uml&catid=119&Itemid=484. [Kreiptasi 15 05 2013].

[7] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley Professional, 2004.

[8] T. J. Grose, G. C. Doney ir S. A. Brodsky, Mastering XMI: Java Programming with XMI, XML, and UML, Willey, 2002.

[9] K. Plukas, E. Mačikėnas, B. Jarašiūnaitė ir I. Mikuckienė, Taikomoji diskrečioji matematika, Kaunas: KTU, 2004.

[10] D. H. Nam, E. C. Mousset ir D. C. Levy, „Automating the Testing of Object Behaviour: A Statechart-Driven Approach," *World Academy of Science, Engineering and Technology,* nr. 11, pp. 827 - 831, 2007.

[11] L. Briand, Y. Labiche ir J. Cui, Towards Automated Support for Deriving Test Data from UML Statecharts, 2004.

[12] S. Kim, J. A. Clark ir J. A. McDermid, Class Mutation: Mutation Testing for Object-Oriented Programs, 2000.

[13] „Jumble," [Tinkle]. Tinkle: http://jumble.sourceforge.net/mutations.html. [Kreiptasi 19 05 2013].

8. PRIEDAI

8.1. priedas. Eksperimento rezultatų duomenys taikyti koreliacijai skaičiuoti

	Lempa	CheckLogic	ATM	PDF	Kalkuliatorius
Aptikta mutantų, %	100	75	45	42	32
Aptikta mutantų, kiekis	7	30	14,4	5,88	36,16
Kodo padengimas, eilutėmis	19	68,716	41,088	72,3	168,99
Kodo padengimas, %	100	83,8	85,6	96,4	65,5
LOC (kodo eilučių kiekis)	19	82	48	75	258
MLOC (metodų kodo eilučių kiekis)	9	53	32	39	207
NOF (klasės laukų kiekis)	2	3	3	4	10
NOM (metodų kiekis)	3	6	5	6	18
WMC (ciklomatinis sudėtingumas)	4	19	20	10	75
LOC (testų kodo eilučių kiekis)	82	155	174	89	216
MLOC (testų metodų kodo eilučių kiekis)	28	66	72	28	95
NOM (testų metodų kiekis)	10	18	18	11	30
WMC (testo klasės ciklomatinis sudėtingumas)	10	18	18	11	29
Sugeneruota mutantų	7	40	32	14	113
Skirtingų metodų naudojamų diagramoje kiekis	3	3	5	1	3
Perėjimų kiekis	6	9	10	4	17
Perėjimų kiekis su sąlygomis	2	6	6	4	12
Būsenų kiekis	2	4	5	2	4
Būsenų sąlygų kiekis	4	5	12	12	13
Būsenai tenkantis sąlygų kiekis	2	1,25	2,4	6	3,25
Būsenos sąlygai tenkantis NOF	1	0,416667	0,8	1,5	0,325
Perėjimui tenkantis WMC	0,666667	2,111111	2	2,5	4,411764706
Vid. būsenai tenkantis perėjimų kiekis	3	2,25	2	2	4,25

8.2. priedas. Paskaičiuota koreliacijos matrica

	Aptikta mutantų, %	*Aptikta mutantų, kiekis*	*Kodo padengimas, eilutėmis*	*Kodo padengimas, %*	*LOC (kodo eilučių kiekis)*	*MLOC (metodų kodo eilučių kiekis)*	*NOF (klasės laukų kiekis)*	*NOM (metodų kiekis)*	*WMC (ciklomatinis sudėtingumas)*	*LOC (testų kodo eilučių kiekis)*	*MLOC (testų metodų kodo eilučių kiekis)*	*NOM (testų metodų kiekis)*	*WMC (testo klasės ciklomatinis sudėtingumas)*	*Sugeneruota mutantų*	*Skirtingų metodų naudojamų diagramoje kiekis*	*Perėjimų kiekis*	*Perėjimų kiekis su sąlygomis*	*Būsenų kiekis*	*Būsenų sąlygų kiekis*	*Būsenai tenkantis sąlygų kiekis*	*Būsenos sąlygai tenkantis NOF*	*Perėjimui tenkantis WMC*	*Vid. būsenų per. Kiekis*
Aptikta mutantų, %	1,000																						
Aptikta mutantų, kiekis	-0,320	1,000																					
Kodo padengimas, eilutėmis	-0,686	0,765	1,000																				
Kodo padengimas, %	0,608	-0,916	-0,875	1,000																			
LOC (kodo eilučių kiekis)	-0,639	0,787	0,991	-0,903	1,000																		
MLOC (metodų kodo eilučių kiekis)	-0,608	0,799	0,976	-0,917	0,996	1,000																	
NOF (klasės laukų kiekis)	-0,682	0,688	0,975	-0,859	0,987	0,985	1,000																
NOM (metodų kiekis)	-0,647	0,764	0,981	-0,903	0,997	0,998	0,994	1,000															

WMC (ciklomatinis sudėtingumas)	-0,622	0,812	0,940	-0,952	0,973	0,987	0,965	0,983	1,000													
LOC (testų kodo eilučių kiekis)	-0,591	0,856	0,709	-0,956	0,747	0,773	0,704	0,756	0,849	1,000												
MLOC (testų metodų kodo eilučių kiekis)	-0,544	0,879	0,701	-0,957	0,743	0,771	0,693	0,750	0,846	0,998	1,000											
NOM (testų metodų kiekis)	-0,595	0,897	0,869	-0,997	0,905	0,925	0,870	0,910	0,964	0,954	0,955	1,000										
WMC (testo klasės ciklomatinis sudėtingumas)	-0,594	0,901	0,859	-0,998	0,895	0,915	0,857	0,900	0,957	0,961	0,962	1,000	1,000									
Sugeneruota mutantų	-0,583	0,875	0,934	-0,975	0,965	0,979	0,939	0,969	0,993	0,879	0,881	0,981	0,976	1,000								
Skirtingų metodų naudojamų diagramoje kiekis	0,038	0,220	-0,192	-0,283	-0,102	-0,031	-0,110	-0,060	0,125	0,526	0,531	0,310	0,326	0,151	1,000							
Perėjimų kiekis	-0,480	0,860	0,790	-0,964	0,850	0,887	0,824	0,867	0,943	0,940	0,945	0,980	0,979	0,956	0,427	1,000						
Perėjimų kiekis su sąlygomis	-0,695	0,863	0,924	-0,989	0,943	0,950	0,916	0,945	0,974	0,923	0,917	0,987	0,985	0,983	0,189	0,941	1,000					
Būsenų kiekis	-0,416	0,607	0,262	-0,668	0,297	0,329	0,244	0,307	0,451	0,853	0,849	0,658	0,677	0,501	0,791	0,660	0,598	1,000				

Būsenų sąlygų kiekis	-0,968	0,134	0,577	-0,486	0,544	0,522	0,623	0,568	0,546	0,485	0,431	0,489	0,486	0,484	0,000	0,405	0,587	0,327	1,000			
Būsenai tenkantis sąlygų kiekis	-0,574	-0,393	0,244	0,163	0,157	0,094	0,250	0,155	0,010	-0,291	-0,340	-0,172	-0,185	-0,070	-0,693	-0,290	-0,018	-0,473	0,637	1,000		
Būsenos sąlygai tenkantis NOF	0,056	-0,919	-0,507	0,805	-0,571	-0,613	-0,473	-0,564	-0,671	-0,822	-0,856	-0,802	-0,810	-0,741	-0,521	-0,838	-0,710	-0,696	0,088	0,703	1,000	
Perėjimui tenkantis WMC	-0,833	0,710	0,972	-0,873	0,950	0,929	0,944	0,943	0,909	0,750	0,731	0,861	0,855	0,898	-0,131	0,762	0,931	0,370	0,730	0,330	-0,436	1,000
Vid. būsenų per. Kiekis	-0,140	0,579	0,715	-0,649	0,787	0,822	0,800	0,808	0,807	0,482	0,498	0,689	0,671	0,784	0,000	0,750	0,662	0,019	0,139	-0,114	-0,546	0,568

8.3. priedas. Straipsnis pateiktas „Information Technologies and control“ žurnalui

TEST DATA GENERATION FOR COMPLEX DATA TYPES USING IMPRECISE MODEL CONSTRAINTS AND CONSTRAINT SOLVING TECHNIQUES

Šarūnas Packevičius[1], Greta Krivickaitė[1], Evaldas Guogis[2],

Dominykas Barisas[1], Robertas Jasaitis[1], Tomas Blažauskas[1]

[1] Kaunas University of Technology, Department of Software Engineering,

Studentų st. 50-406, LT-5136, Kaunas, Lithuania

[2] Singleton Labs, Studentų st. 65-307, LT-3000, Kaunas, Lithuania

Abstract. Number of software applications is growing rapidly, as well as their importance and complexity. The need of quality assurance of these applications is increasing. Testing is one of the key processes to ensure the quality of software in general and web services or object-oriented applications in particularly. In order to test large and complex systems, test automation methods are needed, which evaluate whether the software is working properly. The main goal is to improve effectiveness of object-oriented applications testing by creating an automated test data generation method for complex data structures.

This paper presents a test data generation method by adhering to software under test static model and its model constraints. The method provides an algorithm that allows generating test data for complex data structures, by analysing software under test model, its constraints and using constraint solving techniques for building corresponding test data objects and theirs hierarchies. The presented method is exemplified by simple case studies as well as a large I++ protocol implementing web service project.

1. Introduction

In a typical software development organization, the cost of providing assurance that the program will perform satisfactorily within the expected deployment environments via appropriate debugging, testing, and verification activities can take from 50 to 75 percent of the total development cost [1][?]. Many of these activities can be automated, reducing their cost, increasing system quality, reducing the time to market and the maintenance costs.

The aim of this paper is to present the automated test data generation method for complex data structures. Object-oriented applications manipulate with various object hierarchies. Those applications in most cases operate with complex data structures, which in most cases are presented as objects aggregating or composing other objects. Traditional test data generation methods mainly generate tests for software unit under test (or parts of it) that use simple data structures, such as integers, floats, strings, arrays and pointers. However, these methods are unable generating meaningful or at least usable test data for unit testing of object oriented programs.

Most of real-world applications are composed of units working with complex data types as arguments. A test data generator generates input values for the selected method depending on the parameter type. For example, if the parameter is of type *signed short int* (in C++), the tests generator will generate a value from the interval starting with –32,768 and ending with 32,767. If the parameter is of type *signed short int*, the generation algorithm is straightforward – the generated value has to be selected from the allowed range (the range is determined by software under test implementation programming language and parameter type). However, there are more complicated types and one of them is a *string*. The parameter of type string can be any length, but usually is limited by software under test programming implementation language, addressable memory space. The generation based on the values selection from allowed intervals algorithm is suitable for generating values for types which are built-in into programming language. The usual types are: *integer*, *float*, *long*, *double*, *short*, *byte*, *char* and *string*. Real-world software implementations use

complex types in most cases. The complex types are: arrays, lists, classes, interfaces, and structures.
The usual value generation algorithm for the parameter of type array or list is trivial. Generation for class or structure types is more complicated and requires a data flattening technique [2]. The parameter of a complex type is converted into a set of parameters which are of simple types.
The challenge comes when test data generation is applied for complex types and hierarchical class structures, as illustrated in the example in Figure 1.

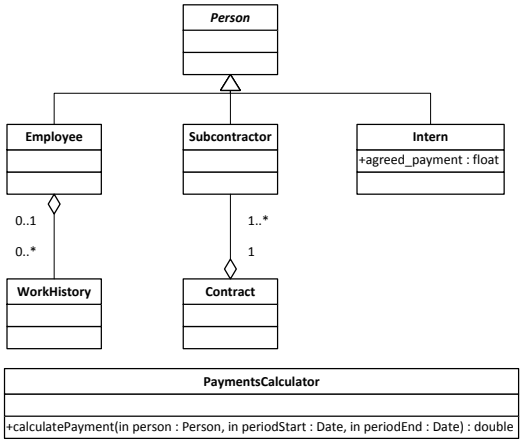


Figure 1. Class structure with various relations.

The question is how to generate test data for method *calculatePayment*. Random values will not always fit since the implicit condition *periodStart* < *periodEnd* needs to be satisfied. Moreover, most of generators would use *null* value for *person* object as its type is an abstract class. Another case is the *Subcontractor* object, which wouldn't make sense without the relation to the *Contract* class.
This paper proposes solution to these problems by applying type flattening strategy, generating values for primitive types and filtering out those values, which do not match model constraints. Besides, it provides test data generation algorithm, evaluation, describes data generation end condition and the means for test data quantity reduction.

2. Problem statement

Figure 2 gives an example of the class Triangle, which is a part of the 3D renderer software. The class Triangle contains three attributes: a, b and c. Each attribute represents the triangle edge length and is of a float type. There also is the class Rasterizer with the method *render*. The method accepts two input parameters: the triangle and the texture and returns the array of Pixel objects. The triangle class is a complex type.

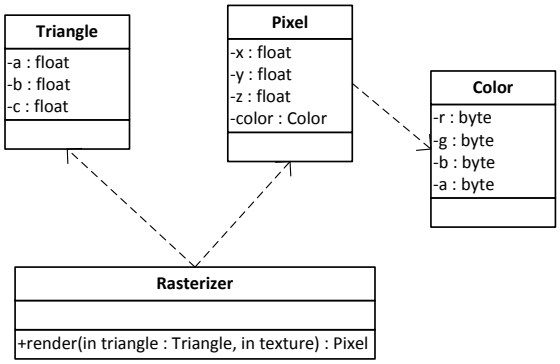


Figure 2. The 3d renderer software class diagram.

The tests generator cannot generate input values for the parameter triangle. To overcome this, the type flattening could be performed, method Render could be transformed into the one which accepts 4 parameters: *triangle_a*, *triangle_b*, *triangle_c* and *tex*. But this approach could have removed the meaning of a triangle.
The generator can generate invalid triangle objects. If the generator can understand the triangle object, it could adhere to some constraints, like: triangle edges have to be positive values, the sum of two edges lengths cannot be larger than the length of the remaining edge. In this case, the test generator just sees three float parameters and can generate any values, thus making some invalid triangles. This is similar to a possibility to generate an invalid float value and pass it as a parameter value to a method under test. But this situation is not possible with simple types; the programming language syntax will not allow this to happen. But there is no means in programming language syntax to overcome a similar situation with complex types.
The main issues addressed in this paper are the following:

1. How to deal with complex data types and generate valid values at the same time. After the type flattening, the test generator could generate a larger amount of invalid objects than an amount of valid ones. For example, if the random generator [3] is used for generating test data for testing Render method, it will generate a large amount of invalid triangles and just a few valid ones.
2. How to solve such cases, when the type of an input value is defined by interface or abstract class. In most cases test data generators can provide empty (null) value [4].

3. Test data generation driven by static model constraints

If the test generator is aware of model constraints associated with complex types it can generate only the valid test data. The test generator can differentiate between correct and invalid objects of complex types (for performing boundary value testing) and a required

amount of valid objects of complex types. In this case the generator could create test data for parameters of complex types which is the same as test data for parameters of simple types.
For example the class Triangle has the OCL [5] constraints presented in the Figure 3.

```
1. context Triangle
2.    inv: a > 0
3.    inv: b > 0
4.    inv: c > 0
5.    inv: a + b > c
6.    inv: a + c > b
7.    inv: b + c > a
```

Figure 3. OCL constraints for the complex type Triangle in the Renderer software.

Table 1. OCL constraints for the complex type Triangle fields in the Renderer software.

No.	a	b	c	Valid	Constraint
1.	3	4	5	true	
2.	1	1	1	true	
3.	1	2	3	false	Invalidates the 5th OCL line.
4.	0	1	2	false	Invalidates the 2nd OCL line.
5.	34	30	3	false	Invalidates the 7th OCL line.
6.	10	10	14	true	
7.	12	12	20	true	
8	12	24	12	false	Invalidates the 6th OCL line.
9.	1	-1	1	false	Invalidates the 3rd OCL line.
10.	-1	0	-5	false	Invalidates the 2nd, 3rd and 4th OCL lines.

The OCL constraints for the Triangle class define that attributes *a*, *b* and *c* have to be positive values (2-4 lines). The OCL constraints also define what a correct rectangle is. Each edge length of the triangle has to be smaller than the sum of the other remaining two edge lengths (5-7 lines). The generated values for Render method are presented in the Table 1.
If the constraints are not used during tests generation, all ten test data sets could be suitable for testing the Render method. But only the 4 test data sets are valid out of 10. In this case too many values are testing the boundary conditions. The constraints for triangle class reduce an available set of input values: the constraint on the 2nd line reduces the set by half, the constraint on the 3rd line reduces the remaining set by other half, and the constraint on the 4th line reduced the remaining set by other half as well. All those reduced halves could be replaced by 1-2 boundary values only. This could reduce tests generation time, memory consumption and storage requirements for storing generated tests.

3.1. Generation algorithm

The proposed test data generation algorithm uses a complex type flattening strategy, generates values for simple types, assembles complex types from simple types and filters out unnecessary generated values, which do not match model constraints.
The test data generation algorithm for generating test data for single class method is presented in the Figure 4.

```
Input:  The Method M which accepts a set
        of parameters
        Pi (i=0;n).
        Parameter types Ti (i=0;n) for
        each parameter Pi.
        OCL constraints PC for the method M
        (pre-conditions)
Output: A set of generated values Vi (i=0;n)
        for each parameter Pi.

1. While there are methods parameters left do
2.   Get  parameter type Ti
3.   value = determine the parameter type
4.        and generate the value for Pi, Ti.
5.   If (Vi do not match PC conditions) then
6.      Return to the step 3.
7.   else
8.      add value to the set V.
9.   Select next parameter Pi
     (go to the step 1.)
```

Figure 4. The test data generation algorithm for a class method.

For clarity, the test data generation algorithm is separated into several parts. The test data algorithm takes a method description (extracted from a code or a model) as an input. The generator extracts all method parameters and their types and generates a test value for each parameter. After the test value is generated for a parameter, the generator checks if the value matches OCL pre-condition and if not, it is rejected and a new one is generated. The process continues until a generated value matches OCL pre-conditions or the time limit for generating value is exceeded or generated invalid values count is exceeded, which is a safeguard for terminating infinitive generation. The generation is repeated for all method parameters. The generated test values for each parameter are assembled into a test data set, which in turn is stored for later usage or executed on software under test.
The test data generator uses several different test data generation algorithms. The algorithm has to be selected based on the parameter type. The algorithm selection is presented in the Figure 5.

```
Input:  The parameter P
        The parameter P type T
Output: The generated value V for the
        parameter P of the type T

 1. If the T is a simple type then
 2.   V = generate simple value (T)
 3. If the T is an array then
 4.   V = generate an array of values of
          the type T
 5. If T is complex type then
 6.   V = generate the value for the complex
          type T
 7. If P is pointer of type T then
```

```
 8.   If generation number is even then
 9.     V = generate the array of values
            of type T
10.   Else
11.     V = generate the value of type T and
12.         create pointer to it
```

Figure 5. The Test generation algorithm selection based on the parameter type.

The test generator checks parameter type and selects the suitable generation algorithm. The algorithm is applied for such types as a simple type, a complex type and an array. When the parameter is of the pointer type in one case the generation algorithm for an array is used, in other case the generation algorithm for a parameter of the type, which the pointer refers to is used. Selected algorithm is different each time in order to generate data for both cases.
The test data generation for a parameter of a simple type is trivial and is presented in the Figure 6.

```
Input:  The parameter type T
Output: The generated value V

1. Select the float value from the
   interval [0;1]
2. Scale the selected value to the interval
   allowed by the T range.
3. V = scaled value.
```

Figure 6. Test data generation for a parameter of a simple type.

The test generator has to generate a value from the interval [0; 1]. The normal distribution generation can be used. Then the generator determines the allowed values range for parameter type and scales the generated value to match it. For example if parameter is of the integer type, the allowed values range is [-32768; 32767]. For example, generator will select a value 0.1. The value 0.1 is scaled and it becomes -26214 (-26214 = (32767 – (-32768)) * 0.1 + -32768). If parameter is of the char type (Unicode, 16-bit), it is actually an integer value with-in a range [0; 65535]. The value generation is the same as for the integer type parameter.
The test data generation for a parameter of an array type is presented in the Figure 7. The generator selects a length for an array. Then it executes the test data generation algorithm for each array value. The generated value is checked if it matches OCL constraints. If the generated value does not match, it is generated again. The value generation algorithm is executed until all array elements are generated.

```
Input:  The parameter type T
Output: The generated array V of elements
        of the type T

1. Select array length L
2. i = 0
3. While i < L do
4.   generate the value for the type T
     add generated value to the array V
5.   check if element match OCL
     constraints for it.
6.   If does not match then
7.     go to the step 3.
8.   i = i + 1
```

Figure 7. Test data generation for a parameter of an array type.

The test data generation algorithm is used for generating values for parameters of a string type. Due to the nature of the string, which is just an abstraction of an array of character elements, the array generation algorithm is used. By execution the value generation algorithm for each array element, the generator can generate test data for parameters which are of the array type, where the array is composed of complex type elements.
When the generation algorithms for simple types, array types, and pointer types are present, the test data generation algorithm can be defined for complex types. The test data generation for the parameter of the complex type is presented in the Figure 8.
The test data generation algorithm for complex types reuses the algorithms for selecting generation algorithm, generating test data for simple, pointer and array type parameters. The test data generation algorithm for complex types takes as an input parameter type and OCL invariant constraints for that type. The OCL constraints are used to ensure that the correct object is constructed. The algorithm performs type flattening. At the first step, the empty object of parameter type is constructed. At this step the OCL constraints are not checked, because they obviously would be invalidated.

```
Input:  The parameter type T (class name)
        The list of the class T invariants
        INVi (i=0;n)
Output: The generated object V of the type T

1. Select implementing type for the class V.
2. Construct the empty class object V
3. For each class field Fi (i=0;n) do
4.   Get field type Ti
5.   Generate the value for the field Fi
     of the type T
6.   For each the class invariant INVj
     (j=0;n) do
7.      Check if the generated value
        satisfies INVj
8.      If the value does not match then
9.         go to the step 4.
10.   assign the generated value to
      the class object V field Fi
11. For each class invariant INVj (j=0;n)
12. which defines relations between
    attributes do
13.    Check if the generated value satisfies
       INVj
14.      If value does not match then
15.         go to the step 2.
```

Figure 8. Test data generation for parameter of class/structure type.

The generator extracts all attributes defined in the class and executes the test data generation algorithm selection and value generation algorithm for each attribute. If the attribute is of the complex type, it executes the same algorithm for value generation of

the complex type, but in this case the other type is passed as an input to the algorithm. Then the value for the attribute is generated, it is checked against OCL invariants. If the value satisfies all OCL invariants (defined for a class type and relevant to the attribute) the value is assigned to the object attribute. At this step only invariants are checked which do not define relations between attributes (for example, inv: a < b + c is not checked at this step, only inv: a > 0 is checked). If value does not match OCL invariants, the new value is generated. The generation is repeated until the correct value is selected or time out is reached (to avoid deadlock conditions). This generation process is repeated for all object attributes. When all attributes are generated, OCL invariants are once again evaluated. In this case, the invariants which define relations between attributes are evaluated (these invariants cannot be checked in the first phase, because not all attributes have defined values). If the OCL invariants are not satisfied the whole generated object is discarded and a new one is generated. Test data generation algorithm is provided in the Figure 9.

3.2. Inheritance and interface handling

Test data generation for complex object types gets more complicated when the argument has a type of its parent. The object-oriented inheritance principle prevents generator from directly selecting required class and building its object. The software under test model could contain a set of classes that extend the argument class for which generator has to generate an object. If the type is a regular class (not an interface or an abstract class), the simplest case would be just to create object of required class and assign values for its fields.

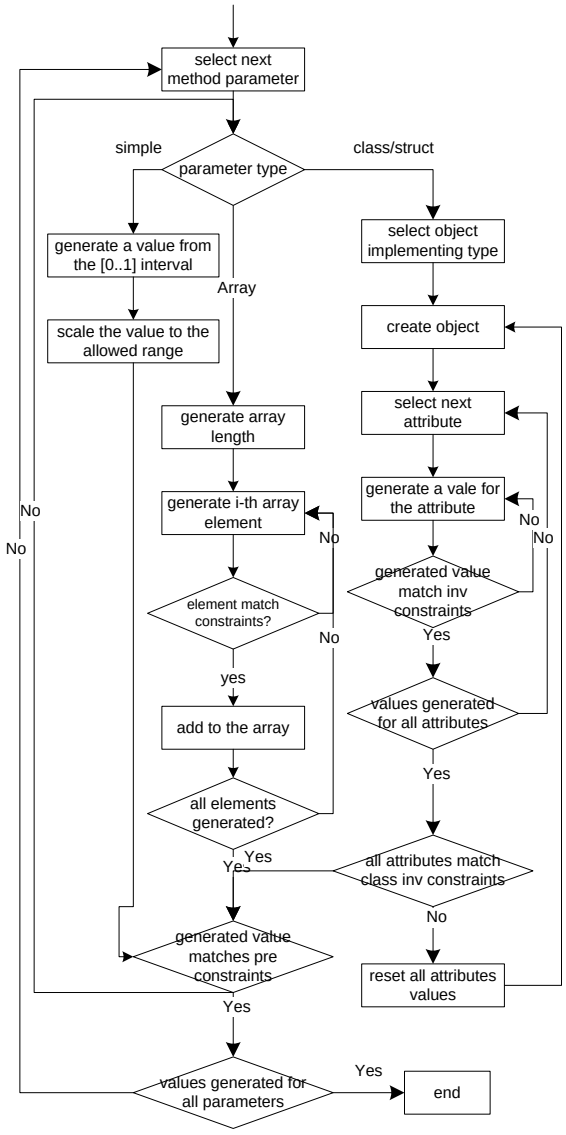


Figure 9. Test data generation algorithm for method parameters of complex types.

More general solution is to find all classes in the model that directly or indirectly inherits class T (some parameter type). Then all classes that inherit type T are identified, the generator can choose any of them and instantiate the object. The next test case could take the same or the other class. The coverage criteria could also be extended by measuring what subset of possible classes was used for tests. In some cases, the list of classes, that inherit T class, could be empty. When such situation is encountered, test generator can build a stub class adhering to model constraints. Constraints, that are valid for the base class, should be valid and for the stub class now.

The situation with interface could be handled in the same manner: finding implementing classes, selecting one of them or building a stub, as it is illustrated in Figure 10. Adhering to the model constraints it is possible to build the more intelligent stub.

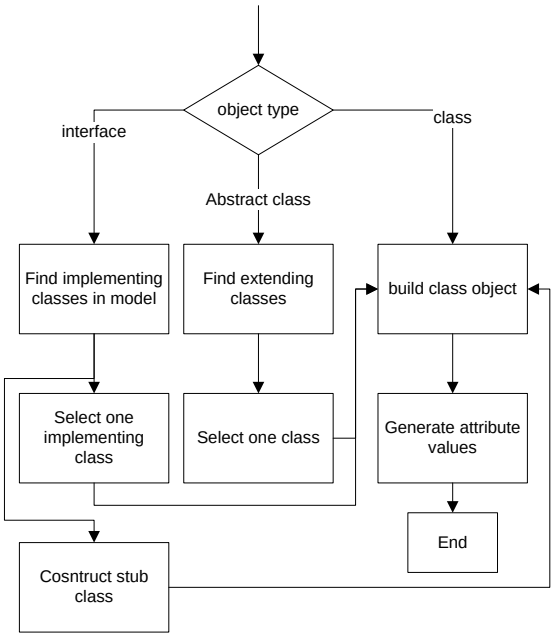


Figure 10. Object type selection and its generation.

The stub could randomly extend any of classes that extend class T or implements interface. The generator implements missing methods by providing empty method bodies, and after such object is stubbed and generated, the OCL constraints are checked to verify if the object is correct, if it's not the object and its stub are discarded and the algorithm is repeated again – a new abstract class or a new class created as a stub with an implemented interface.

Inheritance and interface handling for test generation includes the following steps:

1. Find objects having a type of abstract class or interface;
2. Detect a set of extending or implementing classes;
3. Select a class from inheritance hierarchy to use in object creation using random selection algorithm and having a higher priority for classes, which were not used in previous test data sets. This would increase the generated tests code coverage;
4. Create complex type object and attribute values.

3.3. Test data generation end condition

The test generator has to overcome one critical problem – the object generation has to be completed in a finite amount of time. For deciding when to terminate test generation, test generation algorithm has to adhere to two problems:

- Generation of one test data set (object hierarchy for one parameter is fully generated);
- All test data sets generation.

Objects hierarchies can contain circular references, for example, when object A points to B, and B points to A (Such structure could be created adhering the composite design pattern [6]). The generator can endlessly generate object hierarchy in this case, as it follows the composition, and aggregation links in the model. To overcome this issue, the generator constructs the class composition graph and searches for loops in it. First of all, the loop is traversed only once, and for the next loop iteration the terminating values (null) are selected instead of building another set of objects.

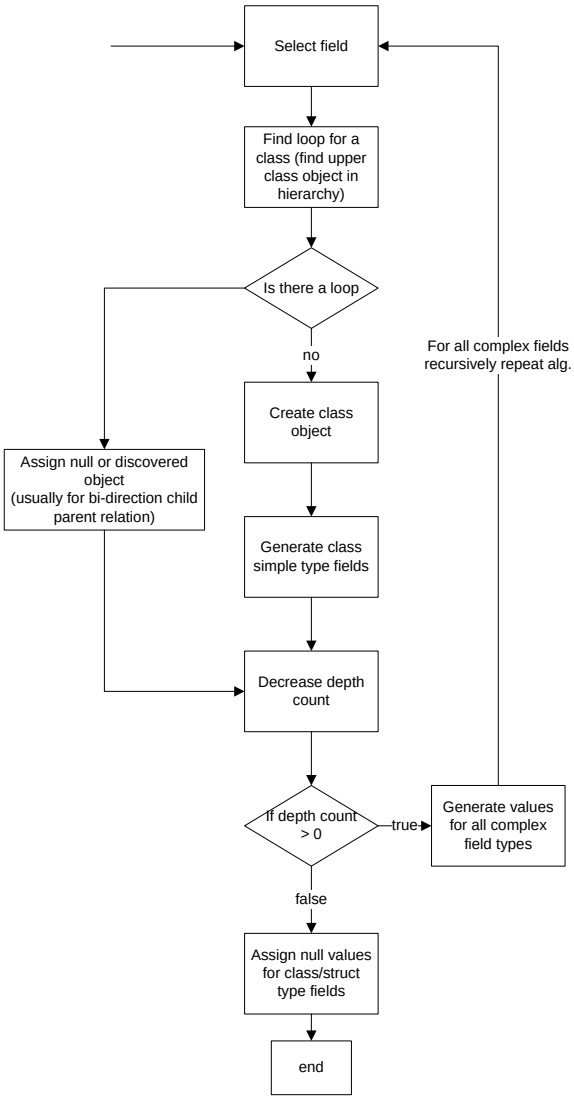


Figure 11. Object hierarchy generation and loops handling.

In other cases, the object hierarchy could be too deep. To handle this situation the generator can be parameterized with maximum depth value, thus limiting generation time. For example generation depth of 1 means that values for a class with fields of simple type would be generated and fields of complex types would have assigned null values. The depth of 2 would mean that all values of class fields would be generated, and recursively algorithm would be invoked for all fields that are of complex types, but for those fields the objects would be constructed as for depth 1 – simple field would get values, complex ones would get terminating (null) value. The summarized

hierarchy depth and loops handling in test data generation algorithm is presented in the Figure 11.

The whole tests generation usually stops when the generator has generated a specific amount of test data. The tester specifies the needed amount of tests. The generator produces the required amount of tests and terminates. After the generation the usual testing procedure occurs: the generated tests are executed, the code coverage is measured, and bugs are detected or not. When the tester has generated a certain amount of tests and bugs have not been found, the tester has to end testing assuming that the software under test is defect free or has to device that another set of tests has to be generated. The tester has to make choice: end testing or generate more tests and continue testing. The tester usually uses code coverage metric as a source for decision making on ending or continuing testing procedure. Based on the code coverage, the tests running time or the project schedule constraints, the tester decides when to end testing. The automatic test generator has no any influence on deciding when to end the tests generation.

The proposed test cases generation method uses the feedback driven test cases generation strategy. The test generator generates a test data set, executes the software unit under test with the generated test data set, evaluates the testing result and testing metrics and decides if an additional set of test data is needed. The tests generator uses the testing oracle (OCL constraints) as a means to evaluate if more tests are needed. The test generator has two different cases to evaluate:

- When the bug is found;
- When bug is not found.

When the bug is found, there is no point in generating more unit tests and executing them. The bug has been already found in the class method under test, and the tests generator can work on testing other class methods and other classes.

When there are no bugs detected the test generation could continue indefinitely. The proposed test generator uses the test pass accuracy coefficient and coverage as a stop condition. The test generation ends when one of the following conditions is met:

- The defined coverage is achieved;
- The coverage does not change after the subsequent tests generation.

For example the tests generator has generated 1000 test cases and executed them. After the execution the code coverage is 78%, and the required coverage is 90%. Additional 1000 tests cases were generated and executed, but the coverage still remains at 78%. This situation could mean that there are unreachable branches in the software under test. If after some more tests cases generation and execution iterations, the coverage does not increase, the testing is stopped. These cases are reported as warnings for possible unreachable parts of code. The tests ending condition decision is presented in the Figure 12.

The tests generator generates some test cases and executes them. If the bug has been found, the testing ends. If the bug has not been found, the code coverage is measured. If the selected coverage level has been achieved, the tests generation and execution ends. If the coverage has not been achieved, the time out value is checked. In this context, time out value means, if the code coverage level has not been changed after additional tests generation and execution, the testing has to be terminated.

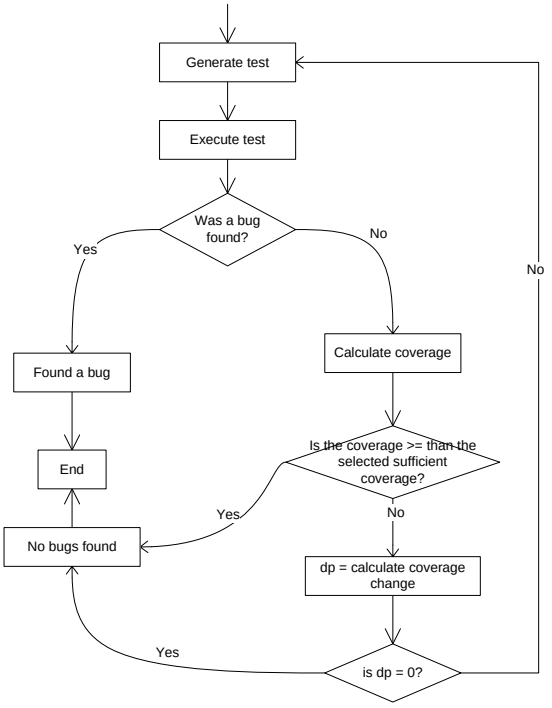


Figure 12. The tests generation end decision algorithm.

The coverage criterion can be selected freely by tester and could be one of the following [7, 8]:

- All-branches;
- All-operators;
- All-code lines;
- The percentage of possible input values exercised;
- Model coverage.

When using OCL constraints for filtering input values, the possible input values amount can be reduced. The tester can select an input values generation function (for example, uniform distribution). Based on the generation function, the software under test could be tested with the most important values from the filtered interval. The tests generation will continue until the selected percentage of possible input values are exercised. For example, if test input values are generated using normal distributions, all possible input values are generated with the same probability. If the tester chooses to generated values based on normal distribution, the majority of input values will be generated closely to average (most typical) value.

The tester can also select after what amount of subsequent test cases generation and execution testing should be stopped when the coverage level is not increasing anymore. For example, the tester can select that after 10000 additional test cases were generated and executed and the coverage has not increased, the testing should stop then.

Besides the regular coverage criteria, the test generator method uses the model coverage criteria – the test generator tries to use as much as possible data model classes while generating test data. For example, if software method under test has 10 classes implementing interface, the 100% coverage means that at least 10 test data sets were generated there each class was used and its object was generated.

3.4. Constraint solving technique

While generating values for a class fields the backtracking [9] constraints solving algorithm is used. Test generator orders fields by their dependences and generates initial value for least dependences having field and checks if it matches model constraints. If it does, the value is accepted and the algorithm proceeds to the next field. If the value does not match, generator backtracks and selects new value repeating the process again. The backtracking steps are presented in the Figure 9 (the back steps that follow on negative OCL validation and calls object values reset or just single value regeneration).

4. Related work

4.1. Random test data generation

The simplest test data generation is random test data generation. Test data are created by selecting input values randomly [3] for software under test methods and checking if generator has reached the defined coverage criterion. In this case test data contain mostly meaningless data (from software domain perspective).
The authors are proposing modifications to random generation technique by employing feedback analysis from tests execution [10, 11]. Tests are executed and the coverage is calculated after each test execution. Based on the coverage level the decision is made if next random values have to be generated.
The advantage of this approach is that the generation algorithm is quite simple and easy to implement. The drawback of this approach is that the generation is a time consuming process, especially when the software unit under test is quite complex.

4.2. Path based tests generation

During the software under test unit analysis, its control flow chart is created. Based on graph theory methods the test inputs are generated, selected inputs drive the software execution by some paths. The main part of a generator is an input data selection which would force the code to be executed into the selected code branch. To achieve this, the constraints solving techniques [12], the relaxation method are used [13]. These methods select initial values and based on the software execution feedback perform input values tuning. Unfortunately these approaches only work with values inside the unit under test which are of a simple type (float, integer, etc..) and are not capable to handle units which are calling other methods, functions and/or operates with variables of complex types (arrays, pointers, data structures, etc..). Authors have proposed some methods for tests generator when software uses pointers [14], calls procedures and/or functions [15, 16]. Tests generation, for software which uses complex data types, authors are proposing to employ data transformation into equivalent data types, for which existing generation methods are used [17, 18]. The advantage of path based tests generation is the possibility to generate the minimal needed tests data set which would satisfy the selected coverage criterion. Also during code analysis the unreachable paths of code could be detected and marked as failures [19]. The disadvantage of path based test generation algorithms is that they are quite complex and not always guarantee a full code coverage.
The tests generation is based on the data gathered during software unit execution instead of data collected during code static analysis. The software unit is executed with some input data and during its execution runtime parameters are observed: executed paths, executed branches, executed operators. Based on observations the new additional input data are generated in order to drive execution by selected control flow path [8]. Authors are proposing various methods for improving code coverage by tests, such as the chaining approach [8], the program slicing by diving software unit into separate branches [20]. The main drawback of these approaches is that the execution of software has to be performed, which requires the preparation of the whole software infrastructure (environment) - that could not be performed automatically. The advantage is the fast tests generation.

4.3. Search-based software testing

The genetic algorithms can be adapted for tests generation [21-24]. The initial set of tests cases is created, after that tests are executed and theirs efficiency is measured by adhering the selected coverage criterion. During the next iteration child tests are generated by selecting better performing tests and killing less successful tests. Using this approach tests are created which achieve the selected coverage criterion with less test data or with less testing time. The advantage of these approaches is that tests generation is fast, the drawback – there is no guarantee that the defined tests generation goal will be reached at all.

4.4. Model-based generation

Tests can be generated when the implementation of software under test is still not present. Tests are generated using software models. Formal and informal models can be used as a source for tests generation. It is also possible to generate tests directly from requirements specification, but that case the models are needed anyways, these models could be transformed from requirements specification, even from the textual ones [25]. During the tests generation using formal specifications the black box methods can be used, such as boundary values analysis and average values analysis. Model based tests generation is increasingly becoming more and more important due to the emergence of the model driven engineering [26]

and the model driven development [27] approaches based software development methods.

Formal specifications, expressed in the Z notation [28] and others [29], strictly define the software functionality. Based on the software formal specification it is possible to generate tests for that software. The formal specifications allow generating not only test data but can also provide an oracle which would be able to determine if software works correctly with given test data. Due to the fact that formal specifications are used for defining critical systems and real time systems, their testing can be alleviated by generated tests from formal specifications [30]. The disadvantage of such tests creation is that creating formal specifications is expensive and only a few projects are developed using such strategy.

The Unified Modelling Language (UML) [31] is semi-formal modelling language. These informal models have some features which could be handy during tests generation. These models are called tests-ready models [32]. They are usually extended to some extent in order to be suitable for tests generation, For example, UML has testing profile [33], or an Object Constraint Language (OCL) [5] model besides UML models could be used for tests generation. Informal models are actively used for testing software developed using product lines approach [32].

Tests generated using software models usually try to examine such cases as: missing action, incorrect data manipulation by overrunning buffers, incorrect data manipulation between class boundaries, incorrect code logic, incorrect timing and synchronization, incorrect program code sequence execution.

During software based on models testing, it is possible to transform models into graphs, such as state graphs. For example, UML diagrams, such as state or sequence can be used for tests generation by transforming them into graphs. For created graphs the usual test generation techniques can be used, the same techniques as for testing software when its code is available [34]. Authors are also proposing to transform models from one language into other ones. Target languages are more suitable for tests generation, for example, the UML models are transformed into SAL models and SAL models are used to generate tests [35].

Authors have proposed Jartege tool and a method for random generation of unit tests for Java classes defined in JML (Java Modelling Language) [36]. JML allows writing invariants for Java classes and pre and post-conditions for operations. JML specifications are used as a test oracle and for the elimination of irrelevant test cases. Test cases are generated randomly. Proposed method constructs test data using constructors and methods calls for setting state.

4.5. Combined techniques

It is possible to mix code based and model based tests generation methods together. It is not always possible to have the full specification of software under test. In order to test this software the mix of code based tests generation and model based test generation methods can be used. The authors have proposed the path finding tool [37]. Data structures are generated from a description of method preconditions. The generalized symbolic execution is applied to the code of the precondition. Test inputs are found by solving the constraints in the path condition. This method gives full coverage of the input structures in the preconditions. Then the code of a system under test is available it is executed symbolically. A number of paths are extracted from the method. An input structure and a path condition define a set of constraints that the input values should satisfy in order to execute the path. Infeasible structures are eliminated during input generation.

There are also methods for combining both techniques together [19]. Test data is generated based on code based generation techniques, software is executed with generated test data and it is checked if software has entered the undefined state in the model, or has exceeded restrictions for its variables values [38]. Based on code and specification it is possible to verify if code paths executed during testing are defined in the model and allow to check if software has not changed its state to the undefined in the model or has performed illegal transition from one state to another one, thus violating specification [19].

5. Test data quantity reduction

The possible input values count depends on the programming language used for the implemented software under test. But the usual possible amount values count is similar to many programming languages. If the class method takes only one parameter of integer type, there are already 4294967295 possible input values. If the method accepts two of integer input parameters the possible amount of input values has to be 4294967295 * 4294967295. If the class method takes, for example, a complex type parameter, the possible amount of inputs can be calculated by using the formula (1):

$$fpv(type) = \begin{cases} c_{type}, when\_simple\_type \\ \prod_{i=1}^{n} fpv(attr_i), otherwise \end{cases} \quad (1)$$

here, fpv(type) – the amount of possible input values for the type "type"; c_{type} – the amount of possible input values for the simple type "type", selected from the Table 2; $attr_i$ – the type of the attribute which is a part of the "type" type.

The formula flattens the complex type and multiplies all the possible input values count for each attribute type. For example, if there is the complex type Vector4D, which is composed of 4 attributes: x, y, z, and w (each attribute is of a float type), the possible input values count is calculated:

$$fpv(Vector\,4D) = (fpv(float))^4 = 3{,}4 \cdot 10^{34} \quad (2)$$

Table 2. The possible input values count for some basic types

No	Type	Possible input values count
1.	Integer	2^{32} = 4294967296
2.	Float	2^{32} = 4294967296
3.	Double	2^{64} = 18446744073709551616
4.	Long	2^{64} = 18446744073709551616
5.	Char	2^{8} = 256
6.	Byte	2^{8} = 256
7.	Vector3d	2^{32*3}= 79228162514264337593543950336
8.	Triangle	2^{32*3}= 79228162514264337593543950336

Thus, even for a method which accepts only one input parameter of quite not very complex type, the possible input values count is quite big. The amount of possible input values for a method which accepts n input parameters can be calculated using the 3rd formula:

$$Tpp = \prod_{i=1}^{n} fpv(p_i) \tag{3}$$

here, Tpp – the total possible input values count for a method of n input parameters; p_i – the i-th input parameter type; n – the number of input parameters for the method.

Using OCL constraints the possible input values count could be reduced. The reduction level depends on the OCL constraints precision levels. The possible input values could be calculated using the 4th formula when OCL constraints are available.

$$Tpp' = \prod_{i}^{n} fpv(p_i) \cdot cr_i \tag{4}$$

here, Tpp' – the total possible input values count for a method of n input parameters; p_i – the i-th input parameter type; n – the number of input parameters for the method; cr_i – the reduction level of input values for the i-th parameter.

The reduction level depends on the OCL constraints associated with the method parameter precision level. If the constraints are precise, only a few values are used (boundary values), if the constraints are not available – there is no reduction at all. The reduction level can be within bounds:

$$0 < cr_i <= 1 \tag{5}$$

The cr_i value of 1 means that there is no reduction at all (thus there is no OCL constraints restricting a method input parameter). The value 0 zero could mean that all possible input values are eliminated, but this case is not possible (because at least one boundary value has to be used). The cr_i value could be calculated using the 7-th formula, which uses the a_i value, calculated using the 6-th formula:

$$a = \sum_{i-1}^{n} \frac{0.5}{2^{i-1}} \tag{6}$$

here, a – the precision level for a class field/method result value; n – the number ">", "<", ">=", "<=" operators.

$$c_i = \begin{cases} 1, a_i \geq 1 - cb \\ 1 - a_i + cb, otherwise \end{cases} \tag{7}$$

here, c_i – the input values reduction level; a_i – the precision level calculated using the 6-th formula; cb – the percentage of input values allocated for boundary values (usually 0,01, and cb > 0).

The amount of how many times the possible input values count can be reduced can be calculated by dividing the amount of possible input values count calculated without using OCL constraints by the value calculated using the OCL constraints (the 8-th formula).

$$r = \frac{Tpp}{Tpp'} = \frac{\prod_{i=1}^{n} fpv(p_i)}{\prod_{i}^{n} fpv(p_i) \cdot cr_i} = \frac{1}{\prod_{i=1}^{n} cr_i} \tag{8}$$

The reduction level depends only on the available OCL constraints and their precision levels. If there is no OCL constraints the cr_i values are 1, and r = 1. The possible input values count is not reduced in this case. The reduction level can be calculated for the example system presented in the Figure 2, which also has OCL constraints presented in the Figure 3.

$$Tpp = 2^{32} \cdot 2^{32} \cdot 2^{32} = 7{,}9 \cdot 10^{28} \tag{10}$$

$$cr_1 = cr_2 = cr_3 = 1 - 0{,}75 + 0{,}01 = 0{,}26 \tag{11}$$

$$Tpp' = 2^{32} \cdot 0.26 \cdot 2^{32} \cdot 0.26 \cdot 2^{32} \cdot 0.26 \approx 1{,}3 \cdot 10^{27} \tag{12}$$

$$r = \frac{1}{0.26^3} \approx 56.8 \tag{13}$$

In the 7th formula the cb value was selected as 1%. The one percent of possible input values is allocated for boundary values. The OCL constraints precision level is 0.75. The reduction values are the same for all parameters – 0.26. After the calculation with all values included, the reduction is 56.8. This means that by adding 6 OCL constraints into the class Triangle, the possible input values count is reduced by 56.8 times.

6. Metrics

This paper presents the theoretical model of test data generation using model constraints. A model of InspectionPlan class is analysed, which is a part of I++ DMS data model. However, the algorithm does not use coverage calculation or other metrics at the moment. Further studies and the subsequent publication will include experimental method evaluation using the following metrics:

1. Code coverage;
2. Path coverage;
3. Model coverage and evaluation of the amount of existing classes and interfaces, which were used from the model.

The following section provides the method evaluation including an example model and OCL constraints.

7. Evaluation

The test generation method was assessed at testing a commercial application. The application under test was implemented using Java programming language as an xml web service. The application was an implementation of a quality parameters exchange protocol. The quality exchange protocol (I++) was developed by majority of European automakers. The protocol (and an application itself) is designed to allow transferring selected automobile quality parameters from CAD (Computer Aided Design) application to factory production software for measuring the selected quality parameters. The application was providing a simple interface: just a set of several methods in one class for loading, transforming and storing quality parameters data. The complexity lays in the methods parameters; each function takes as an input and returns a result of a complex data structure. The data structure is composed of about 129 different classes and several hundreds of objects of those types interconnected between each other. Figure 13 presents a fragment of I++ DMS [39] data model. The subset of measurement plan is presented in the figure, which models the whole measurement plan (InspectionPlan class) for one part of the car, and defines a set of possible features, that could be measured on the car body. The possible features have types (in this fragment the angle and surface point features are visible). The angle feature is presented as the IPE_Angle class and uses a composite design pattern [6], it aggregates any other two features (specifying the angle between the selected two features). The surface point feature (defined by IPE_SurfacePoint) only contains the coordinates in 3D space. Each feature contains nominal values (defined by IPE classes and their attributes) and allowed manufacturing deviations, tolerances that specify in what range the manufactured part features could fall. For example, a surface point feature position can vary in some interval by x, y, and z axis; the angle features' angle value can vary in three intervals parallel to x, y, and z axis. The nominal and tolerance values are joined by QC object (quality control) that connects IPE extending objects and relevant Tolerance extending objects. The InspectionPlan object in turn aggregates all the QC objects, thus containing the full measurement plan for one part of the car. The static model by itself does not contain information what objects could be aggregated by QC object. For example, if QC aggregates IPE_SurfacePoint object, it is not allowed for it to aggregate Tol_AxisAngle or its' child classes, at that would not make sense in measurement plan. To overcome this issue, the model is enhanced by using OCL model constraints. For this small mode fragment the OCL constraints are presented in the Figure 14

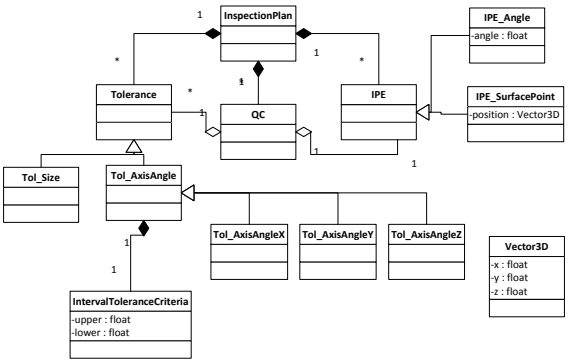


Figure 13. The fragment of I++ DMS data model

The model defines, that InspectionPlan object composes all the IPE, Tolerance, and QC objects, and each QC object only aggregates Tolerance and IPE objects from the InspectionPlan lists. This constraint is presented in 2-3 lines. This constraint states that every object referenced by QC object, must be contained in InspectionPlan objects lists. The other complex relation is modelled in the 3rd line. That constraint states that if QC aggregates IPE_Angle object, it also is required to aggregate 3 Tolerances objects: Tol_AxisAngleX, Tol_AxisAngleY, and Tol_AxisAngleZ. As well IPE_angle is not allowed to aggregate itself (line 5), it cannot measure angle between itself and some other features, and as well it has to reference two different features. The similar situation is for IPE_SurfacePoint object: if QC aggregates this object, it also have to aggregate Tol_AxisPosX, Tol_AxisPosY, Tol_AxisPosZ (not visible on a diagram snippet) objects (line 7). Another constraint defines that for angle features the nominal value has to be within tolerance bounds (lines 15-21). The final constraint (lines 23 - 25) defines that features name (IPE.name) has to match certain regular expression – name is four number strings that does not start with 0.

```
1.  context InspectionPlan
2.   inv invariant_InspectionPlan1:
3.    qc->forAll(qc : QC |
      qc.ipe.oclIsTypeOf(IPE_Angle) = true
      implies qc.tolerances.length = 3 and
qc.tolerances[0].oclIsTypeOf(Tol_AxisAngleX)
= true and
qc.tolerances[1].oclIsTypeOf(Tol_AxisAngleY)
= true and
qc.tolerances[2].oclIsTypeOf(Tol_AxisAngleZ)
= true)
4.   inv invariant_InspectionPlan2:
5.    qc->select(qc : QC |
      qc.ipe.oclIsTypeOf(IPE_Angle) = false)
6.   inv invariant_InspectionPlan3:
7.    qc->forAll(qc : QC |
qc.ipe.oclIsTypeOf(IPE_SurfacePoint) = true
implies qc.tolerances.length = 3 and
qc.tolerances[0].oclIsTypeOf(Tol_AxisPosX) =
true and
qc.tolerances[1].oclIsTypeOf(Tol_AxisPosY) =
true and
qc.tolerances[2].oclIsTypeOf(Tol_AxisPosZ) =
true)
8.
9.  context QC
10.  inv invariant_QC1:
11.   ipe->forAll(ipe: IPE |
      parent.ipe.contains(ipe))
```

```
12.  inv invariant_QCCheckTolerances:
13.   tolerances->forAll(tolerances:
      Tolerance |
      parent.tolerances.contains(tolerances))
14.
15. context TolAngle
16.  inv invariant_TolAngle1:
17.   value.upper > value.lower
18.  inv invariant_TolAngle2:
19.   value.upper <= parent.ipe.angle
20.  inv invariant_TolAngle3:
21.   value.lower >= parent.ipe.angle
22.
23. context IPE
24.  inv invariant_IPE:
25.   name.regExpMatch('[1-9][0-9]{3}')
```

Figure 14. OCL constraints fragment for I++ DMS data model

The tests data were generated for one single method "storeProductStructure" in the class IppWebService. This method allows exchanging quality data between the car manufactured and their suppliers. This method accepts just several parameters: inspectionPlan, and sender.

The sender parameter is a simple one: just the application name and the user of calling web service method. The inspection plan is more complex, it is the root object (presented in the Figure 13), that aggregates all other objects, visible in the diagram and presents measurement plan for one car part. The Figure 14 shows related model constraints for I++ DMS model, presented as OCL. OCL constraints for all objects are written manually using Eclipse with OCL plugin and placed in a separate single file.

The test generator has analysed the model and constructed several test data sets, presented in the Figure 15 as an object diagram.

The generator starts from the root object and constructs InspectionPlan object. This object is supposed to compose several QC objects. To satisfy this requirement generator chooses random QC list size (2), and generates QC objects in the loop. The first QC objected is supposed to aggregate one IPE object. To satisfy this requirement test data generator randomly picks one of the classes that extend IPE class (in this case it is IPE_Angle).

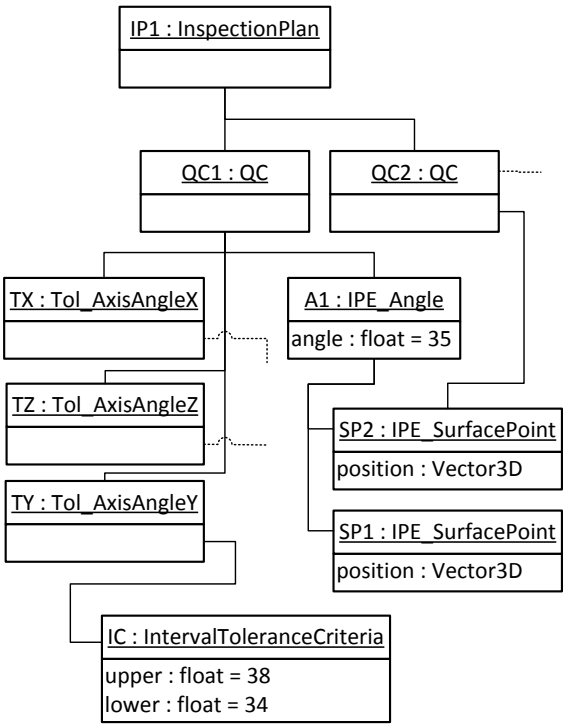


Figure 15. Generated inspection plan object structure for 1 test data set

Then test generator repeats test data generation algorithm and builds IPE_Angle. While generating the IPE_Angle object, the arbitrary value is selected for the angle field, as well as two additional objects has to be created that the IPE_Anlge aggregates. Those two object types were randomly selected from classes that extend the IPE class, and the test data generation algorithm is performed for them again. As the last IPE_SurfacePoint objects are terminating nodes, the test data algorithm returns back and generates remaining Tolerance objects that are required by QC object. As the QC now aggregates the IPE_Angle object, the OCL constraints allows only generating 3 tolerance objects of Tol_AxisAngleX. Tol_AxisAngleY, and Tol_AxisAngleZ types, thus again repeating test data generation algorithm for those objects. As each Tol_AxisX,Y,Z objects aggregate InternalToleranceCriteria object, the generator has to generate those objects as well. As generator generates InternalToleranceCriteria objects, it builds random object and checks if upper and lower values match constraints (navigating through model till the IPE_Angle object and checking if upper and lower values match the IPE_Angle.angle field value). If upper and lower values are incorrect, generator discards them and selects new ones. As IntervalToleranceCriteria objects are build, the generator retreats back to inspectionPlan objects and generates remaining QC objects using the same approach, but in this case it can pick other random IPE_Types, for example IPE_SurfacePoint and build different objects sub-hierarchy. Then InspectionPlan object is generated, the test data for storeInspectionPlan method is completed and generated data structure creation is printed as JUnit

[40] statements and calls for the storeInspectionPlan method are generated. The test data generator can also classify the generated object hierarchies to the ones that are valid (matches all model constraints) and the ones that are invalid (do not match constraints). It's expected for unit test to fail with invalid data, and succeed with valid ones. Thus it serves as an approximate test oracle.

8. Conclusions and future work

The OCL constraints can be used for filtering test data. Using the OCL constraints a large amount of meaningless values is filtered out from the generated test data. The more meaningless test data is removed, the faster tests can be executed. The invariant and pre-condition OCL constraints can be used for filtering the generated test data. The pre-condition OCL constraints filter which test data could be passed to the software under test. The invariant constraints can be used to ensure that only the valid objects of complex types could be passed to the software under test.

Using the defined algorithm incorrectly constructed objects of complex types could be detected early and not used for testing. The constructed complex object is checked against invariant constraints, if it is not valid one, the tests generator could discard, or use as the boundary value in tests. By knowing which object is valid and which one is not, the generator can limit the amount of tests data, which use unnecessary boundary values. Even for trivial sample the test data amount can be reduced 50 times;

The use of a feedback driven tests generation technique prevents from test indefinite test generation and execution. When the bug is detected the tests generation and execution finishes. When bugs are not found, the tests generation and execution continues until the selected coverage a criterion is reached. After generation and execution of some tests, the coverage change is measured, if it is not changing the testing ends. This happens when the code contains unreachable branches;

We have presented the theoretical model of test data generation using model constraints. The future work includes improving and evaluating test data generation method by using other constraints solving algorithms instead of simple backtracking one, as well as evaluation method fitness by using testing full I++ DMS service and measuring selected coverage achievement fitness.

Acknowledgments

The work described in this paper has been carried out within the framework the Eureka ITEA2 project ATAC: Advanced Test Automation for Complex Software-Intensive Systems (2011- 2014) , funded by the project "The 'Eureka' research and development projects - Eureka" VP1-3.1-ŠMM-06-V-01-003 of the European Social Fund (ESF) Human Resource Development Programme of Lithuania.

References

[1] Hailpern B, Santhanam P. Software debugging, testing, and verification. IBM Systems Journal 2002;41.

[2] Meyer B. Object-oriented software construction: Prentice-Hall, Inc.; 1997.

[3] Duran JW, Ntafos SC. An evaluation of random testing. IEEE transactions on software engineering 1984;10:438-44.

[4] Patrice G, Nils K, Koushik S. DART: directed automated random testing. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. Chicago, IL, USA: ACM Press; 2005.

[5] Clark T, Warmer J. Object Modeling with the OCL: The Rationale behind the Object Constraint Language (Lecture Notes in Computer Science): Springer; 2002.

[6] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns : elements of reusable object-oriented software. Boston: Addison-Wesley; 2003.

[7] Chilenski JJ, Miller SP. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal 1994;9:193-200.

[8] Ferguson R, Korel B. The chaining approach for software test data generation. ACM Trans Softw Eng Methodol 1996;5:63-86.

[9] Dechter R, Frost D. Backtracking algorithms for constraint satisfaction problems. Technical Report: University of California at Irvine; 1999. p. 45.

[10] Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-Directed Random Test Generation. Proceedings of the 29th international conference on Software Engineering: IEEE Computer Society; 2007. p. 75-84.

[11] Patrice G, Nils K, Koushik S. DART: directed automated random testing. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. Chicago, IL, USA: ACM Press; 2005. p. 213-23

[12] Gotlieb A, Botella B, Rueher M. Automatic test data generation using constraint solving techniques. Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis. Clearwater Beach, Florida, United States: ACM Press; 1998. p. 53-62.

[13] Gupta N, Mathur AP, Soffa ML. Automated test data generation using an iterative relaxation method. Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering. Lake Buena Vista, Florida, United States: ACM Press; 1998. p. 231-44.

[14] Gotlieb A, Denmat T, Botella B. Goal-oriented test data generation for programs with pointer variables. 29th Annual International Computer Software and Applications Conference (COMPSAC'05)2005. p. 449-54 Vol. 2.

[15] Sy NT, Deville Y. Consistency techniques for interprocedural test data generation. Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international

symposium on Foundations of software engineering. Helsinki, Finland: ACM Press; 2003. p. 108-17.
[16] Korel B. Automated test data generation for programs with procedures. Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis. San Diego, California, United States: ACM Press; 1996. p. 209-15.
[17] Korel B, Al-Yami AM. Assertion-oriented automated test data generation. Proceedings of the 18th international conference on Software engineering. Berlin, Germany: IEEE Computer Society; 1996. p. 71-80.
[18] Milicevic A, Misailovic S, Marinov D, Khurshid S. Korat: A Tool for Generating Structurally Complex Test Inputs. Proceedings of the 29th international conference on Software Engineering: IEEE Computer Society; 2007. p. 771-4.
[19] Beyer D, Chlipala AJ, Majumdar R. Generating tests from counterexamples. 26th International Conference on Software Engineering (ICSE'04) 2004. p. 326-35.
[20] Hierons R, Harman M, Danicic S. Using program slicing to assist in the detection of equivalent mutants. Software Testing, Verification and Reliability 1999;9:233-62.
[21] Corno F, Sanchez E, Reorda MS, Squillero G. Automatic test program generation: a case study. Design & Test of Computers, IEEE 2004;21:102-9.
[22] Seesing A, Gross H-G. A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software. International Transactions on System Science and Applications 2006;1:127-34.
[23] Harmanani H, Karablieh B. A hybrid distributed test generation method using deterministic and genetic algorithms. Fifth International Workshop on System-on-Chip for Real-Time Applications (IWSOC'05) 2005. p. 317-22.
[24] Kalpana P, Gunavathi K. A novel specification based test pattern generation using genetic algorithm and wavelets. 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05) 2005. p. 504-7.
[25] Gargantini A, Heitmeyer C. Using model checking to generate tests from requirements specifications. Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering. Toulouse, France: Springer-Verlag; 1999. p. 146-62.
[26] Uhl A. Model driven arcitecture is ready for prime time. Software, IEEE 2003;20:70, 2.
[27] Mellor SJ, Clark AN, Futagami T. Model-driven development - Guest editor's introduction. Software, IEEE 2003;20:14-8.
[28] Spivey JM. Understanding Z: A Specification Language and Its Formal Semantics: Cambridge University Press; 2008.
[29] Packevičius Š, Kazla A, Pranevičius H. Extension of PLA Specification for Dynamic System Formalization. Information Technology and Control 2006;3:235-42.
[30] Xin W, Zhi C, Shuhao LQ. An optimized method for automatic test oracle generation from real-time specification. 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05) 2005. p. 440-9.
[31] Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Third Edition ed. Boston: Addison-Wesley Professional; 2003.
[32] Olimpiew EM, Gomaa H. Model-based testing for applications derived from software product lines. Proceedings of the first international workshop on Advances in model-based testing. St. Louis, Missouri: ACM Press; 2005. p. 1-7.
[33] Baker P, Dai ZR, Grabowski J, Haugen O, Schieferdecker I, Williams C. Model-Driven Testing: Using the UML Testing Profile: Springer-Verlag Berlin and Heidelberg GmbH & Co. K; 2007.
[34] Paradkar A. Case studies on fault detection effectiveness of model based test generation techniques. Proceedings of the first international workshop on Advances in model-based testing. St. Louis, Missouri: ACM Press; 2005. p. 1-7.
[35] Kim SK, Wildman L, Duke R. A UML approach to the generation of test sequences for Java-based concurrent systems. 2005 Australian Software Engineering Conference (ASWEC'05) 2005. p. 100-9.
[36] Oriat C. Jartege: a tool for random generation of unit tests for java classes. First international conference on Quality of Software Architectures and Software Quality. Erfurt, Germany: Springer-Verlag; 2005. p. 242--56.
[37] Visser W, Pasareanu CS, Khurshid S. Test input generation with java PathFinder. Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. Boston, Massachusetts, USA: ACM Press; 2004. p. 97-107.
[38] Hessel A, Pettersson P. Model-Based Testing of a WAP Gateway: an Industrial Study. FMICS and PDMC2006.
[39] Zimmermann JU. Informational integration of product development software in the automotive industry : the ULEO approach. Enschede: University of Twente; 2005.
[40] Louridas P. JUnit: unit testing and coding in tandem. Software, IEEE 2005;22:12-5.

Programinės įrangos apimtis, svarba ir sudėtingumas sparčiai auga. Šių programų kokybės užtikrinimo poreikis didėja. Siekiant patikrinti didelių ir sudėtingų sistemų veikimą, reikalingi testavimo automatizavimo metodai, padedantys įvertinti ar programa veikia tinkamai ir atitinka specifikaciją. Pagrindinis tikslas yra sukurti efektyvų automatizuotą testinių duomenų generavimo metodą, naudojant sudėtingas duomenų struktūras.

Šiame straipsnyje pateikiamas testinių duomenų generavimo metodas sudėtingoms duomenų struktūroms, atsižvelgiant į testuojamos programinės

įrangos modelį, klasių ryšius ir pateiktus apribojimus, pritaikant apribojimų sprendimo metodus ir jų pagalba konstruojant atitinkamus testinių duomenų objektus ir jų hierarchijas. Pateiktas metodas iliustruojamas paprastu ir didelio projekto realizuojančio I++ protokolo žiniatinklio servisą pavyzdžiais.