KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Tomas Milašius

Simboliniu vykdymu grindžiamo mutacinio testavimo įrankio kūrimas ir tyrimas

Magistro darbas

Darbo vadovas
prof. E. Bareiša

Kaunas, 2011

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PRAKTINĖS INFORMATIKOS KATEDRA

Tomas Milašius

Simboliniu vykdymu grindžiamo mutacinio testavimo įrankio kūrimas ir tyrimas

Magistro darbas

Recenzentas
prof. Lina Nemuraitė
2011-05-30

Vadovas
prof. E. Bareiša
2011-05-30

Atliko
IFM-9/2 gr. stud.
Tomas Milašius
2011-05-30

Kaunas, 2011

TURINYS

PAVEIKSLĖLIŲ TURINYS

LENTELIŲ TURINYS

MUTATION TESTING BASED ON SYMBOLIC EXECUTION TOOL DEVELOPMENT AND RESEARCH

SUMMARY

This work consists of three major parts. The first (analytical) part is the review of software quality assurance activities - specifically the testing process. The main focus is on automated test generation.

The second (design) part describes the mutation testing based on symbolic execution test generation method. It also specifies the implementation details of the systems under development - described in the static and dynamic perspectives.

The third part (research and experimental) is devoted for analysis of developed method. Here wide range of characteristics and metrics are analyzed. Also, some improvements are implemented. This helped to reduce system's methods cyclomatic complexity and greatly increased speeds at witch tests generation are performed.

The method described is characterized by the fact that it can help generate tests that detect mutations in the software code and symbolic execution is used for test generation, rather than a random number generator.

1. ĮVADAS

1.1. Dokumento paskirtis

Šis dokumentas yra programų sistemų inžinerijos magistro baigiamasis darbas. Dokumente yra trumpai apžvelgiamos problemos su kuriomis yra susiduriama siekiant padidinti programinės įrangos kokybę. Analizuojama viena iš programinės įrangos kokybės užtikrinimo veiklų – testavimo procesas. Apžvelgiami skirtingi testavimo etapai, jų efektyvumas bei įrankiai, kurie palengvina programinės įrangos testavimą. Detaliau išanalizavus metodikas, kurios leidžia automatiškai sugeneruoti testinius atvejus tiriamai programinei įrangai, buvo pasiūlytas naujas testų generavimo metodas.

Siūlomo metodo įgyvendinimui buvo suprojektuota ir realizuota programinė įranga. Reikalavimų rinkimo bei architektūros sudarymo etapai yra aprašomi šio darbo projektinėje dalyje. Joje detalizuojamas testų generavimo algoritmas. Tyrimo dalyje yra analizuojama sukurtos programinės įrangos kokybė, pateikiami ir realizuojami jos patobulinimai, o eksperimentinėje dalyje pateikiamos svarbiausios atliktų bandymų metrikos.

1.2. Santrauka

Šiame dokumente aprašytas darbas susideda iš trijų pagrindinių dalių. Pirmojoje (analizės) dalyje yra apžvelgiamos programinės įrangos kokybės užtikrinimo veiklos – konkrečiai testavimo procesas. Didžiausias dėmesys yra skiriamas automatizuotam testų generavimui.

Antrojoje (projektinėje) dalyje aprašomas simboliniu vykdymu grindžiamas mutacinis testų generavimo metodas. Taip pat detalizuojamas jo realizavimas kuriamoje sistemoje – aprašomi statiniai ir dinaminiai vaizdai.

Trečiojoje dalyje (tyrimo ir eksperimentinėje) yra analizuojamas sukurtas metodas, vertinamos įvairiausios jo charakteristikos, metrikos ir realizuojami patobulinimai. Šie sistemos priežiūros darbai leido sumažinti ciklomatinį metodų sudėtingumą ir pagreitinti realizuoto testų generavimo metodo veikimą.

Aprašytas metodas pasižymi tuo, jog jo pagalba galima sugeneruoti testus aptinkančius programinio kodo mutacijas, o testų generavimui yra naudojamas simbolinis vykdymas, o ne atsitiktinių skaičių generatorius.

2. ANALITINĖ DALIS

2.1. Temos aktualumas ir perspektyvumas

Pasak JAV nacionalinio standartų ir technologijų instituto (angl. *National Institute of Standards and Technology*) atliktos studijos, programų klaidos JAV ekonomikai kainuoja $59.5 bilijono dolerių kasmet. Tai sudaro 0,6 procento JAV BVP [1]. Šioje studijoje taip pat yra pabrėžiama, jog ši suma turi augimo tendenciją, nes faktiškai visų įmonių veikla yra susijusi su programinės įrangos naudojimu. Taigi kuriant programas yra susiduriama su daugybe uždavinių ir vienas iš tokių privalomų spręsti uždavinių yra programos kokybės ir patikimumo įvertinimas.

Vienas iš būdų įvertinti ir pagerinti programų kokybę yra testavimo procesas. Nors testavimas niekada neidentifikuos visų programos klaidų, tačiau jis suteikia informacijos apie silpnas sistemų dalis ir leidžia lengviau bei greičiau identifikuoti klaidas[2]. Taip pat testavimo procesas suteikia galimybę programų kūrėjams numatyti galimus pirkėjus bei naudotojus, o pirkėjams įvertinti ar sukurta sistema atitinka jų lūkesčius.

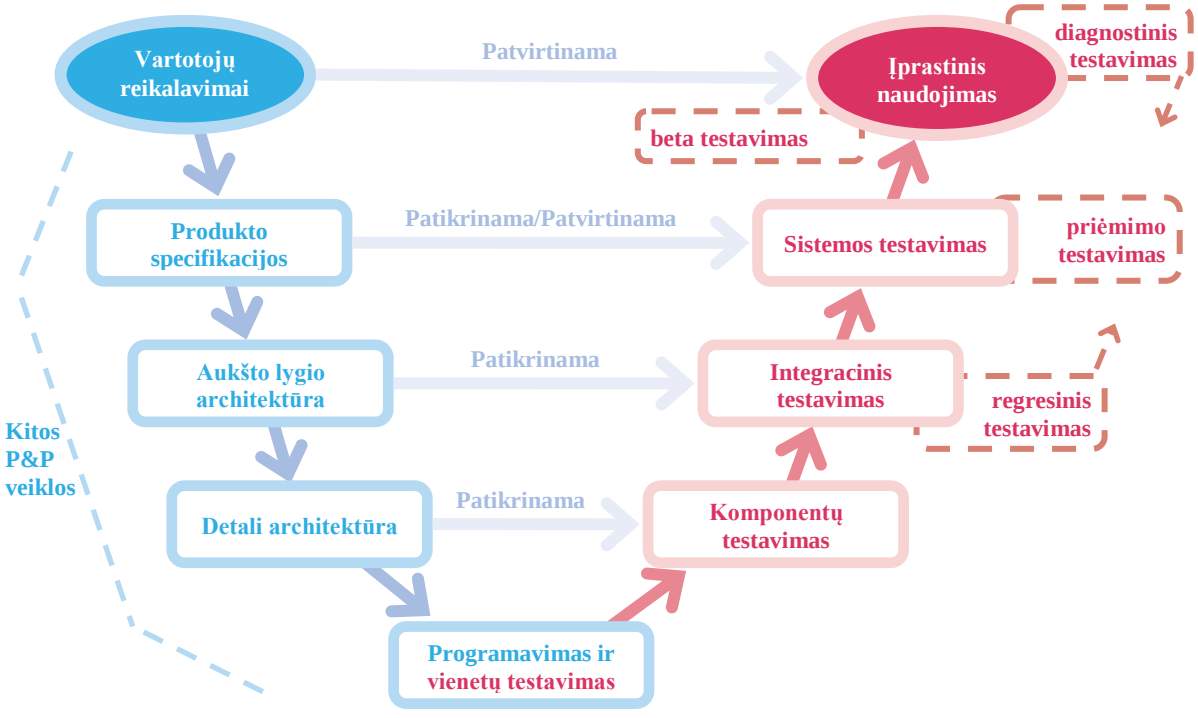
Nuo 1988 metų iki dabar testavime vyrauja prevenciniai (angl. *Prevention Oriented*) testai [3]. Tai testai, kurie parodo, jog sukurta sistema atitinka jos specifikaciją, sugeba išvengti ir yra apsaugota nuo gedimų [2]. Nors tai yra patikimas būdas užtikrinti programų kokybę, tačiau testų kūrimo metu yra susiduriama su daugybe sunkumų, tokių kaip: brangių resursų (laiko, rankų darbo) naudojimas. Taip pat reikia užtikrinti, jog testavimą vykdantys žmonės yra aukštos kvalifikacijos specialistai, gebantys numatyti silpnas programos vietas ir jas ištestuoti.

Siekiant palengvinti šių problemų sprendimą yra kuriami automatizuoto testavimo metodai ir įrankiai. Jie ne tik sutrumpina programų kūrimo laiką, bet ir padidina programų kokybę. Deja dažnai yra sunku automatiškai nustatyti ar programa duoda korektiškus rezultatus, t.y. sunku nustatyti ar programa ne tik skaičiuoja nesustodama, bet ar ji duoda tokius rezultatus kokių yra tikimasi [4]. Kad palengvinti tokių klaidų aptikimą yra naudojami įvairūs sprendimai: Testais paremtas kūrimas (angl. *Test-driven development*), automatizuotas vartotojo sąsajos testavimas (angl. *Automated GUI testing*), atsitiktinis automatizuotas testavimas (angl. *Monkey testing*), programos funkcijų vykdymo laikų gairių nustatymas ir matavimas (angl. *Benchmarks*) ir kt. Vienas iš tokių automatizuoto testavimo būdų yra pasinaudoti programos būsenų diagrama.

Būsenų diagrama tai yra diagrama naudojama programos elgsenos aprašymui. Jose yra pateikiama informacija apie būsenas, į kurias gali patekti programa [5]. Žinant visas programos būsenas ir įvykius, kurie iššaukia būsenų pasikeitimus (turint visas programos būsenų diagramas), būtų galima kurti automatizuotus testus, kurie patikrintų sistemą ir užtikrintų kokybę. Tokiems testams sukurti ne tik nereikėtų itin brangiai apmokamų specialistų, nes jie būtų kuriami automatiškai, bet ir jie padengtų visus galimus programos vykdymo kelius.

2.2. Testavimo etapai

Siekiant užtikrinti didelių sistemų kokybę dažniausiai yra taikomi keli kokybės užtikrinimo (testavimo) metodai. Šie metodai dažniausiai yra skirti patikrinti skirtingas sistemų charakteristikas, kurios gali būti tiek išorinės (teisingumas, panaudojamumas, efektyvumas, patikimumas ir kt.), tiek vidinės (palaikomumas, lankstumas, pernešamumas ir kt.). Kadangi šių charakteristikų yra daug ir jos skiriasi priklausomai nuo kiekvienos sistemos, tai ir jas patikrinantys metodai skiriasi. Šie metodai dažniausiai išsibarsto per visą sistemos kūrimo laikotarpį ir gali trukti nuo kelių savaičių iki kelių metų, todėl buvo sugalvota visą sistemų testavimo procesą sudalinti į etapus ir sudaryti modelius, kurie parodytų, kuris etapas po kurio seka bei leistų suvaldyti visą sistemų patikrą [6].



Pav. 1 - Testavimo etapai pagal V modelį

Vienas iš žinomiausių testavimo modelių yra V modelis. Pagrindinė šio modelio paskirtis – išskirti ir pažingsniui sudėlioti sistemų patikros ir patvirtinimo etapus (angl. *verification & validation* – būtent dėl *V&V* šis modelis yra vadinamas V modeliu). Žinoma V modelis nėra naujas ir bėgant laikui jis keitėsi (atsirado papildymai ir naujos jo interpretacijos). Paveikslėlyje „Pav. 1 - Testavimo etapai pagal V modelį“ yra pateiktas grafinis V modelio atvaizdavimas, kur:

- Rožine spalva - pažymėti testavimo etapai;
- Žydra spalva - programų konfigūravimo objektai iš kurių yra ruošiami testiniai atvejai kiekvieno testavimo etapo metu;
- Punktyrinėmis linijomis - originalaus V modelio papildymai ir pakeitimai.

2.2.1. Testavimo etapų efektyvumas

Skirtingi testavimo etapai yra skirti skirtingų klaidų paieškai ir turi skirtingą klaidų aptikimo bei pašalinimo laipsnį (efektyvumą), kuris yra bene pagrindinis rodiklis vykdant sistemų kokybės užtikrinimo darbus. Jis leidžia apsispręsti kokie testavimo etapai privalo būti vykdomi ir į kuriuos etapus reikia sutelkti daugiau dėmesio.

Etapų efektyvumas yra paskaičiuotas ilgų stebėjimų ir renkamos statistikos dėka bei išreikštas procentais. Šis procentas parodo kiek klaidų yra aptinkama kiekvieno etapo metu palyginus su visu klaidų skaičiumi (aptiktu visos sistemos gyvavimo metu).

Paveikslėlyje „Pav. 2 - Klaidų aptikimo dažniai kiekvieno kokybės patikros etapo metu“ yra pateikiama diagrama, kurioje yra surašytos pagrindinės programų kokybės užtikrinimo veiklos ir pateikiami rėžiai su *žemiausiu* klaidų aptikimo procentu, *dažniausiai pasitaikančiu* klaidų aptikimo procentu (pažymėta vertikalia linija) ir *didžiausiu* klaidų aptikimo procentu kiekvieno etapo metu. Diagramoje taip pat rožine spalva išskirti etapai (kokybės užtikrinimo veiklos), kurie yra įtraukti į V modelį.

Pav. 2 - Klaidų aptikimo dažniai kiekvieno kokybės patikros etapo metu

Bene įdomiausia informacija, kurią parodo ši diagrama yra tai, kad dažniausiai pasitaikantis klaidų aptikimo procentas vieno etapo metu nepakyla daugiau nei 75% ir visų etapų vidurkis yra apie 40%. O bene populiariausia ir dažniausiai minima testavimo veikla – vienetų testavimas – padeda aptikti tik maždaug 30% visų sistemos klaidų. Tipinės organizacijos kuriančios sistemas (programinę įrangą) remiasi kietu testavimu (angl. *test-heavy*) ir pasiekia apie 85% defektų pašalinimo efektyvumą. Tuo tarpu pirmaujančios organizacijos pasitelkia ne tik testavimo veiklas, bet ir peržiūros bei patikros veiklas ir taip pasiekia 95% ar didesnį klaidų pašalinimo efektyvumą prieš išleidžiant produktą [7].

2.2.2. Regresinis testavimas

Sukūrus programinę įrangą ir pateikus ją rinkai jos gyvavimo ciklas dažniausiai nenutrūksta. Ji gali būti atnaujinama kai yra siekiama ją pritaikyti prie besikeičiančios darbo aplinkos, išplečiama siekiant patekti į naujus rinkos segmentus arba taisoma, kai yra pastebimos klaidos. Tokiais atvejais yra naudojamas regresinis testavimas tam, kad būtų galima įsitikinti, jog įvedus pakeitimus (atnaujinus sistemą) išliko senas funkcionalumas [6].

Regresinis testavimas - tai bet kuri testavimo veikla kai tie patys testai yra vykdomi nebe pirmą kartą. Šis testavimo etapas (būdas) pasiskirsto po visus kitus testavimo etapus ir

yra naudojamas, kai testuojama nauja, jau ištestuotos sistemos versija ar yra testuojami panašūs produktai ir dalį testų (testinių atvejų) galima pritaikyti naujos sistemos testavimui. Kadangi pakeitus metodus ar sukūrus naujus komponentus dažniausiai pasikeičia ir jų sąsajos, seni testiniai atvejai pasidaro nebetinkami vienetų bei komponentų testavimui ir regresinis testavimas negali būti pritaikytas šių testavimo etapų metu. Tačiau kai nauji komponentai yra integruojami į seną sistemą regresinis testavimas gali būti panaudotas siekiant išsiaiškinti ar sistema išlaikė seną funkcionalumą t.y.: pradėjus integracinį testavimą galima pradėti ir regresinį.

Regresinis testavimas yra labiau susijęs su produktais, kurie turi ilgą gyvavimo laikotarpį ir yra išleisti keliomis versijomis. Esant didesniems sistemos pakeitimams regresinis testavimas dažniausiai yra naudojamas integracinio testavimo metu, kai nauji komponentai yra suintegruojami į sistemą. Esant mažesniems pakeitimams regresinis testavimas gali apimti didžiąją testavimo dalį.

2.2.3. Mutacinis testavimas

Mutacinis testavimas tai vienas iš defektais remto testavimo metodų. Tai kai testiniai atvejai yra kuriami ne pagal specifikacijas, realizavimo ypatybes ir sistemos naudojimą, bet pagal aptiktus defektus arba potencialius defektus. Pagrindinė šio testavimo paskirtis išsiaiškinti ar sukurti testai aptinka nekorektiškai veikiančią programą [6]. Šis testavimas paprastai yra atliekamas keliais etapais:

1) Iš pradžių yra kuriami programos mutantai. Mutantas – tai dalinai modifikuota originalios testuojamos programos versija, kuri nuo jos skiriasi mažu, sintaksiniu pakeitimu [2].
2) Kiekvienas testinis atvejis (sukurtas ankstesnių etapų metu arba sugeneruotas atsitiktinai) yra vykdomas su kiekvienu mutantu ir originalia programa, o gauti rezultatai yra kaupiami.
3) Vėliau kiekvieno mutanto grąžintas rezultatas yra palyginamas su originalios programos pateiktais rezultatais. Testas, kuris aptinka skirtumą tarp mutanto ir originalios programos yra vadinamas atspariu konkrečiam mutantui, o toks mutantas „nužudytu“ arba pagautu.

Nors pagrindinė mutacinio testavimo paskirtis – patikrinti ar testai yra atsparūs mutantams, šis metodas gali būti naudojamas kuriant testinius atvejus: testai gali būti

atsitiktinai generuojami, kol yra pasiekiamas tam tikras pagautų mutantų skaičius. Ši metrika yra svarbi vykdant regresinį testavimą, kai reikia įsitikinti, jog modifikuojant ar atnaujinant programą nebuvo atsitiktinai įvesti nepageidaujami sistemos pakeitimai.

2.2.4. Testų kokybės įvertinimo metrikos

Kodo padengimas testais – tai testavimo matas parodantis programinės įrangos ištestavimo laipsnį. Šis laipsnis yra naudojamas sprendžiant kada testavimo procesas gali būti baigtas (kada galima sakyti, jog programinė įranga yra ištestuota). Dažniausiai yra skiriami keli kodo padengimo testais kriterijai [8]:

- Sakinių padengimas – ši metrika parodo kiek kodo eilučių yra įvykdoma testuojant.
- Šakų padengimas – ši metrika parodo, ar kiekviena programos vykdymo šaka buvo bent kartą įvykdyta.
- Kelių padengimas – ši metrika parodo kiek programos vykdymo kelių yra padengiama testuojant.
- Įėjimų/išėjimų padengimas – parodo ar buvo įvykdytos visos funkcijos ir ar visi grąžinimo sakiniai (angl. *return*) buvo pasiekti.

2.3. Testavimo karkasai ir esami testavimo įrankiai

Pasak Boris Beizer pasitelkus neformalų testavimą ir nesinaudojant testavimo karkasais tipiškai yra pasiekiamas 50-60% kodo sakinių padengimas testais [7]. Tai yra žemas procentas parodantis, jog tokiu atveju lieka neištestuota 40% sukurtos programos. Todėl siekiant pagreitinti, padaryti kokybiškesniu ar automatizuoti testavimo vykdymą į pagalbą dažniausiai yra pasitelkiami testavimo karkasai. Pagal Swebok (Guide to the Software Engineering Body of Knowledge) testavimo įrankiai gali būti skirstomi į šias grupes [9]:

- Testų generatoriai. Šie įrankiai padeda sukurti testinius atvejus.
- Testų vykdymo karkasai. Šie įrankiai leidžia įvykdyti testus kontroliuojamoje aplinkoje, kurioje yra stebima testuojamų objektų veiksena.
- Testų įvertinimo įrankiai. Šie įrankiai padeda įvertinti testų vykdymo metu gautus rezultatus ir patikrinti ar jie atitinka lauktus rezultatus.
- Testų valdymo įrankiai. Šie įrankiai padeda palaikyti ir suvaldyti testavimo procesą.

- Našumo įvertinimo įrankiai. Šie įrankiai yra naudojami vertinant programinės įrangos našumą – tai specializuota testavimo forma, kur pagrindinis uždavinys yra įvertinti programinės įrangos našumą, o ne jos funkcines savybes (teisingumą).

Bene labiausiai paplitę ir dažniausiai naudojami yra xUnit šeimos testavimo karkasai. Tai testų vykdymo, testų įvertinimo ir dalinai testų valdymo karkasai pritaikyti kiekvienai programavimo platformai atskirai: nUnit - .Net aplinkai; jUnit – Java; dUnit – Delfi; fUnit – Fortran ir t.t. Šie karkasai leidžia testuoti skirtingus programos elementus (vienetus) tokius kaip funkcijos ar klasės. Pagrindinis šių platformų privalumas yra tas, jog jos suteikia automatizuotą testų rašymo galimybę: nereikia rašyti tų pačių testų daug kartų bei nereikia prisiminti kokius rezultatus testas turi grąžinti.

Siekiant paspartinti ir padaryti kokybiškesniu testavimo procesą yra automatizuojami ne tik testų vykdymo bei įvertinimo etapai, bet ir testų kūrimas. Tam yra kuriami testų generatoriai, kurių veikimas dažniausiai yra paremtas statine kodo arba modelių analize:

JTest[1]

JTest – tai komercinis *Parasoft* įrankis skirtas pagerinti programinės įrangos kokybę. Tai yra pasiekiama vykdant:

- Statinę analizę – statinę kodo analizę, statinę duomenų sekų analizę ir metrikų analizę.
- Kodo peržiūrų dalinį automatizavimą – yra dalinai automatizuojamas pasirengimas kodo peržiūroms, jų sekimas ir pranešimas apie artėjančias peržiūras.
- Vienetų testavimą – yra automatizuojamas vienetų testų kūrimo procesas, jų vykdymas, optimizavimas ir palaikymas.
- Vykdymo klaidų aptikimą – generuojant testus yra siekiama atsižvelgti į galimas gijų vykdymo sekas ir sukurti tokius testus, kurie iškviestų sistemos vykdymo klaidas.

[1] http://www.parasoft.com/jsp/products/jtest.jsp/

Reikia atkreipti dėmesį, jog nors *JTest* automatiškai generuoja vienetų testus tačiau tai yra vykdoma generuojant atsitiktinius duomenis ir analizuojant sistemos vykdymo kelius – taip ne visada yra pasiekiamas pilnas sistemos vykdymo kelių padengimas ir toks testų generavimas gali atimti daug laiko, o sugeneruoti testai gali būti ne visai tikslingi.

Visual Studio Test Professional 2010[2]

Tai *Microsoft* kompanijos įrankis apjungiantis visą testavimo procesą nuo testinių atvejų planavimo iki jų vykdymo ir sekimo. Kaip teigiama kompanijos svetainėje pagrindinis šio įrankio tikslas pagerinti bendravimą tarp testuotojų ir programuotojų. Šis įrankis pasižymi:

- Testavimo įrankiais, kurie leidžia suvaldyti testavimo atvejus, juos įvykdyti, įrašyti testavimo scenarijus ir juos atkartoti.
- Programinės įrangos gyvavimo ciklo valdymo galimybėmis.

Deja pasinaudojant šiuo įrankiu nėra galima visiškai automatizuoti testavimo atvejų kūrimo proceso – įrankis sugeneruoja tik testavimo atvejų šablonus kiekvienam testuojamam metodui, tačiau neparenka testinių duomenų ir neapskaičiuoja laukiamų rezultatų.

Java Path Finder[3]

Tai kiek nestandartinis testavimo įrankis ir jo veikimas priklauso nuo to kaip jis yra sukonfigūruotas. Bendru atveju – tai programinės įrangos modelio tikrinimo įrankis. Jis pasižymi tuo, kad iš esmės veikia kaip virtuali Java mašina – testuojamos programos yra analizuojamos jas vykdant, o ne analizuojant programinį kodą. Taip yra gaunami duomenys ne tik pavienių testinių atvejų kūrimui, bet ir surenkama informacija apie aptiktus defektus, suskaičiuojamos įvairiausios metrikos (kodo padengimo ir kt.), surenkama informacija apie galimas testų vykdymo sekas (pasiruošiama komponentų ir integracinio testavimui). Deja toks programinės įrangos tikrinimo būdas su JPF reikalauja nemažų laiko ir techninės aparatūros resursų, dėl to tokios programinės įrangos patikros nėra galima vykdyti itin dažnai.

[2] http://www.microsoft.com/visualstudio/en-us/products/2010-editions/test-professional/overview
[3] http://babelfish.arc.nasa.gov/trac/jpf

2.4. Testų generavimo metodai

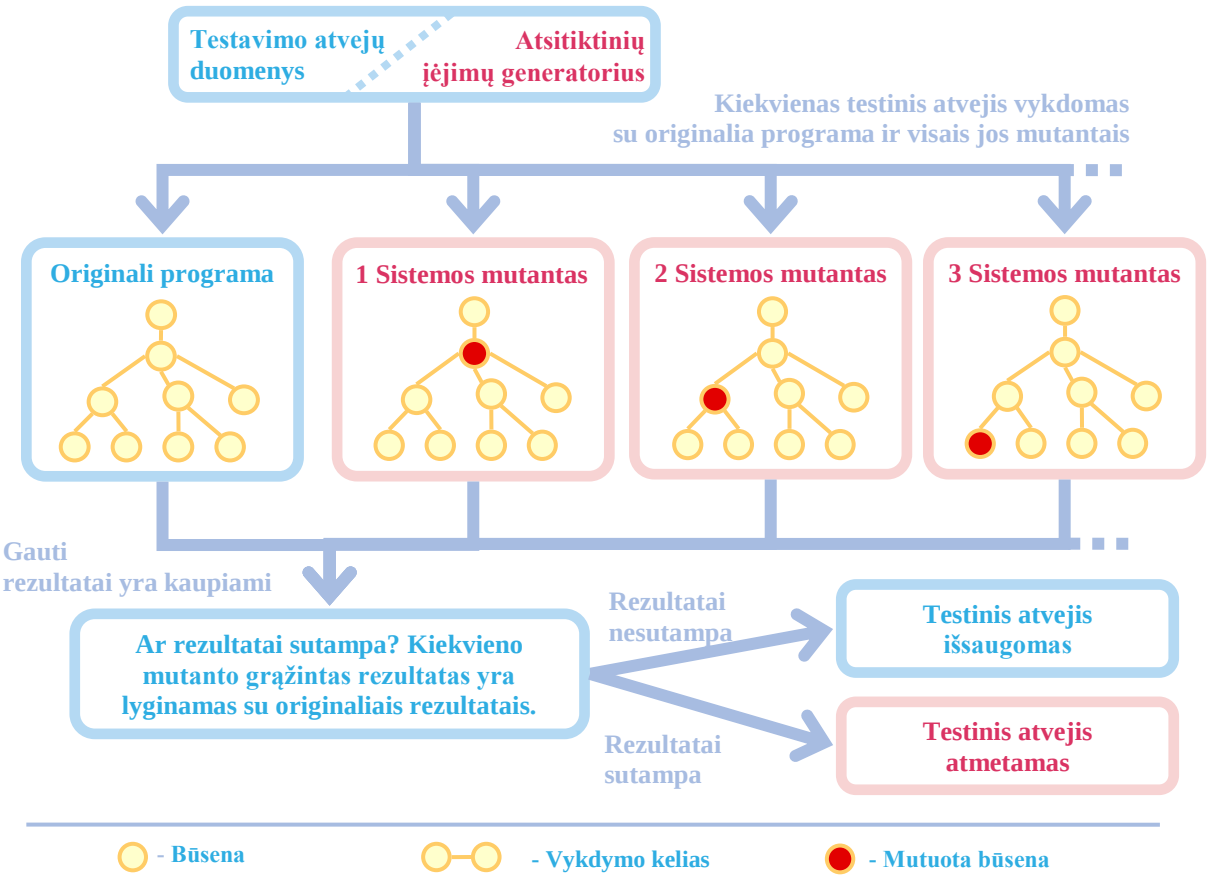
Bėgant laikui požiūris į testavimo procesą keitėsi. Dabar testavimas suprantamas ne vien kaip veikla, kuri prasideda po programinės įrangos suprogramavimo su ribotu tikslu – surasti defektus. Programinės įrangos testavimas dabar suvokiamas, kaip veikla, kuri apima visą sistemos kūrimo ir palaikymo procesą ir pati savaime yra svarbi produkto kūrimo dalis [2]. Dėl šios priežasties ir patys testų generavimo metodai skiriasi – vieni yra naudojami ankstyvosiose programinės įrangos kūrimo stadijose, kur testiniai atvejai yra sudaromi pasinaudojant reikalavimų specifikacijos ar aukšto lygio architektūros dokumentais; kiti yra naudojami vėlyvesnėse sistemų kūrimo stadijose – atlikus programavimo darbus, kaip pavyzdžiui:

2.4.1. Atsitiktinis testų generavimas

Tai turbūt pats paprasčiausias testinių atvejų kūrimo metodas, kurio metu yra pasitelkiamas atsitiktinių reikšmių generatorius. Testuojant programas šiuo metodu gali būti siekiama įgyvendinti tam tikras pasirinktas kodo padengimo metrikas, kaip: sugeneruoti tokius testinius atvejus, kad būtų bent kartą iškviečiamos visos programinio kodo eilutės ar įvykdomi visi grąžinimo sakiniai. Taip pat atsitiktinis įėjimų reikšmių generavimas yra naudojamas vykdant mutacinį testavimą, kai tikrinami du programos variantai – originali programa ir jos mutantas ir yra žiūrima ar gaunamos skirtingos programos reakcijos. Tačiau toks testų kūrimo metodas yra neefektyvus dėl kelių priežasčių:

1) Reikia sugeneruoti nemažą kiekį mutantų.
2) Kiekvienas testinis atvejis turi būti vykdomas su originalia programa ir kiekvienu jos mutantu.
3) Dažnai yra sunku sugeneruoti tokius testinius duomenis, kurie sugautų visus mutantus net ir vykdant kelias generavimo iteracijas.

Grafinis testų generavimo proceso panaudojant mutacinį testavimą atvaizdavimas yra pateikiamas „Pav. 3 - Atsitiktinis testų generavimas panaudojant mutacinį testavimą“ paveikslėlyje. Čia rodyklėmis yra pažymėti duomenų srautai, o rožine spalva yra išskirtos silpnos tokio testinių atvejų generavimo vietos.



Pav. 3 - Atsitiktinis testų generavimas panaudojant mutacinį testavimą

2.4.2. Testų generavimas panaudojant OCL apribojimus

Objektų Apribojimų Kalba (angl. OCL[4]) – tai IBM sukurta deklaratyvi kalba skirta aprašyti apribojimus skirtus UML kalbai (angl. *Unified Modeling Language*). Pasinaudojant šia kalba yra aprašomi apribojimai objektams, kurie gali būti [10]:

- Prieš sąlygos (angl. *pre-conditions*). Šios sąlygos nurodo apribojimus kviečiamų metodų parametrams.
- Po sąlygos (angl. *post-conditions*). Šių sąlygų pagalba yra nurodomi apribojimai, kuriuos turi tenkinti grąžinamos parametrų reikšmės.

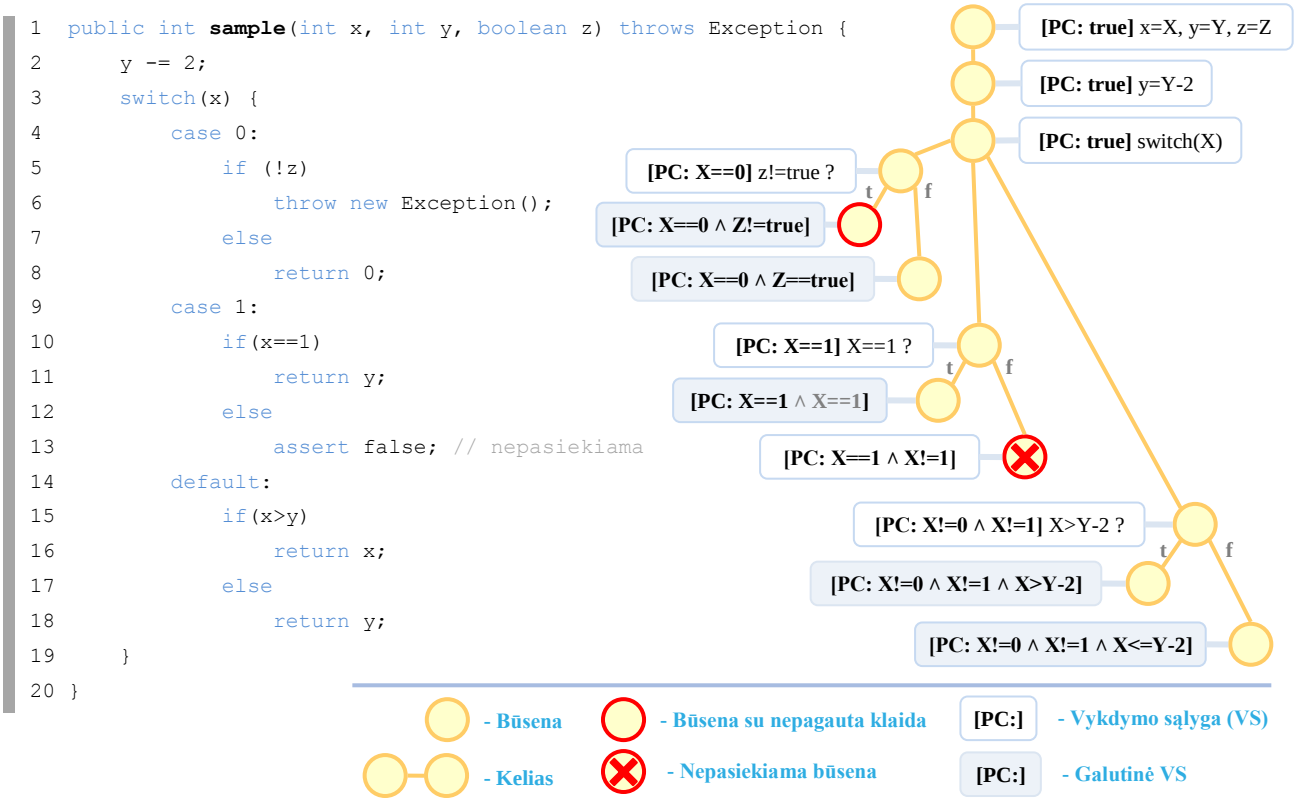
Šiuos apribojimus Šarūnas Packevičius savo darbe „Universal unit tests generator based on software models“ siūlo naudoti sprendžiant orakulo problemą generuojant testinius atvejus [11].

[4] Object Constraint Language

2.4.3. Testų generavimas panaudojant simbolinį vykdymą

Simbolinį vykdymą galima suvokti kaip įprasto programinės įrangos vykdymo praplėtimą, kur vykdant programą vietoj konkrečių kintamųjų reikšmių yra naudojamos simbolinės. Tai yra įgyvendinama praplečiant pagrindines programavimo kalbos skaičiavimo funkcijas (pvz. Java kalboje yra praplečiamas bytecode instrukcijų vykdymas), taip kad jos kaip argumentus priimtų simbolines reikšmes, o grąžintų simbolines formules (apribojimus) [12]. Taip vykdant programą yra sudaromas būsenų modelis, kur kiekviena būsena susideda iš:

- Vykdymo kelio sąlygos (PC – angl. *Path Condition*) – tai formulė apribojanti kintamųjų simbolines reikšmes ir parodanti ar būsena gali būti pasiekta (ar gali būti rastos konkrečios reikšmės simboliniams kintamiesiems).
- Kitos vykdomos operacijos (funkcijos).



Pav. 4 - Būsenų modelis sudarytas simbolinio vykdymo pagalba

Aukščiau yra pateiktas pavyzdinis programos kodas su šalia esančiu paveiksliuku „Pav. 4 - Būsenų modelis sudarytas simbolinio vykdymo pagalba“ iliustruojančiu būsenų modelio sudarymą. Šis modelis – tai vykdymo kelių medis parodantis, kokie programos keliai

buvo pereiti ją vykdant. Kiekviena modelio viršūnė atspindi programos būseną, o kraštinė – perėjimą tarp būsenų. Šis modelis pasižymi tuo, jog:

- Kiekviena būsena gali būti išspręsta ir gali būti rastos konkrečios kintamųjų reikšmės su kuriomis įvykdžius (įprastai) programą bus pereitas tas pats vykdymo kelias.
- Visos būsenos turi skirtingus vykdymo kelius – nėra nei vienos viršūnės, kuri turėtų tokius pačius vykdymo kelio apribojimus, kaip ir kita viršūnė.
- Vykdant simbolinį vykdymą galima pasiekti 100% kelių padengimą.
- Yra galimybė aptikti nepasiekiamas kodo vykdymo vietas (nepasiekiamas būsenas).

Pasinaudojant apribojimų sprendimo įrankiais (angl. *Constraint solving engines*) kiekvienos galutinės būsenos vykdymo sąlygas galima išspręsti taip gaunant duomenis reikalingus testinių atvejų generavimui, kurie užtikrintų aukštą (iki 100%) vykdymo kelių padengimą. Šis testinių atvejų generavimo būdas yra detaliau aprašytas Justino Prelgausko, E. Bareišos darbe „Symbolic Unit Testing Method Evaluation“ [13].

2.5. Analitinės dalies rezultatai

1. Vykdant analizės etapą buvo išsiaiškinta su kokiais sunkumais yra susiduriama norint pagerinti programinės įrangos kokybė ir kiek kainuoja nekokybiškos programinės įrangos išleidimas. Apžvelgtas visas testavimo procesas, išsiaiškinta iš kokių etapų jis susideda, kokios testavimo metodikos yra naudojamos, pamėgintas įvertinti jų efektyvumas, teigiamos savybės bei trūkumai (tobulintinos vietos). Apibrėžtos metrikos, kurių yra siekiama kuriant testinius atvejus.
2. Šio etapo metu didelis dėmesys buvo skiriamas esamų testavimo karkasų ir įrankių analizei. Išsiaiškinta, jog visi įrankiai pagal Swebok gali būti suskirstyti į pastarąsias grupes: testų generatoriai; testų vykdymo karkasai; testų įvertinimo įrankiai; testų valdymo įrankiai; našumo įvertinimo įrankiai. Šis įrankių suskirstymas leido objektyviau įvertinti dažniausiai naudojamas testavimo aplinkas. Buvo labiau struktūrizuotai atsižvelgta į įrankių teikiamas galimybes ir buvo vertinamas ne vien teikiamų funkcijų skaičius. Tačiau pastebėta, jog dauguma komercinių testavimo aplinkų naudoja pačius paprasčiausius testų generavimo metodus – dauguma sukuria tik testinių atvejų šablonus testuojamiems metodams, bet nesugeneruoja testinių duomenų, o tos kurios generuoja

testinius duomenis tai daro atsitiktinių skaičių generatoriaus pagalba. Toks testinių atvejų generavimas yra vertintinas, kaip itin neproduktyvus, netikslingas ir perteklinis.

3. Taip pat šio etapo metu pamėginta išanalizuoti ne tik testinių atvejų generavimo įrankius, bet ir testų generavimo metodus. Pastebėta, jog jų yra pačių įvairiausių ir jie yra naudojami skirtingais programinės įrangos kūrimo etapais – vieni surinkus reikalavimus, kiti realizavus programinį kodą ir pan. Daugiausiai dėmesio buvo skiriama atsitiktinei testų generavimo metodikai pasinaudojant mutaciniu testavimu ir testų generavimui pasinaudojant simboliniu vykdymu. Pastebėta, jog abi šios metodikos turi trūkumų, kurie gali būti išspręsti jas kombinuojant – tai nurodė kryptį kuria turi būti einama vykdant tolimesnius tyrimus ir ieškant konkretaus sprendimo leidžiančio sugeneruoti testus, kurie aptiktų mutantus, bet nereikalautų kurti daugybės mutantų bei atsitiktinai generuoti testinių duomenų. Sukurtas testų skirtų mutantų aptikimui generavimo metodas aprašytas projektinės dalies skyriuje.

3. PROJEKTINĖ DALIS

Projektinėje dalyje didžiausias dėmesys yra skiriamas sukurtam testų aptinkančių programinės įrangos mutantus generavimo metodui. Iš pradžių apibrėžiami pagrindiniai reikalavimai sistemai, panaudos atvejai, vėliau gilinamasi į testų generavimo metodo sukūrimą (esamo praplėtimą). Konkrečiai: analizuojamas testų generavimo panaudojant simbolinį vykdymą metodas, kadangi šis metodas yra dalis sistemoje realizuoto testų generavimo metodo. Apibrėžiami esamo testų generavimo metodo trūkumai ir pateikiami pasiūlymai, kaip jį patobulinti. Vėliau šis (patobulintas) metodas yra realizuojamas ir pateikiami su juo susiję esminiai sistemos architektūros aspektai.

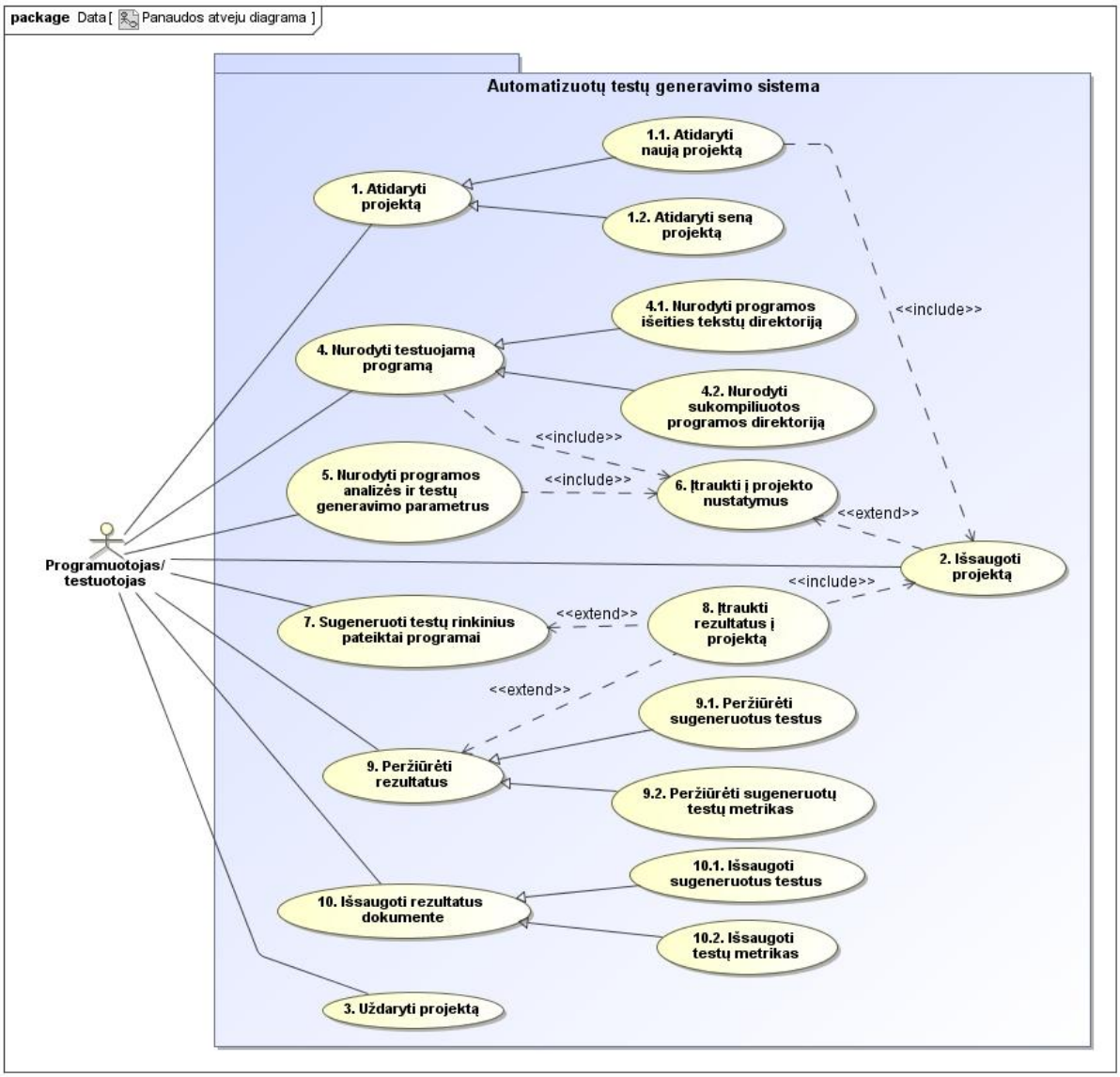
3.1. Sistemos paskirtis ir panaudos atvejai

Sistemos tikslas yra palengvinti testuotojų darbą tuo pačiu užtikrinant aukštą programų kokybę. Šio projekto metu yra siekiama sukurti metodiką, kaip įgyvendinti šiuos tikslus pasinaudojant automatizuoto testų generavimo teikiamais privalumais. Generuojami testai bus taikomi konkrečiai sričiai – programinės įrangos parašytos Java programavimo kalba veiklos patikrinimui ir patvirtinimui vykdant regresinį testavimą, todėl šis projektas nepadės visiškai atsisakyti testuotojų darbo, tačiau jį palengvins. Numatoma, jog sukurto įrankio pagrindiniai naudotojai būtų informacinių technologijų kompanijos, kurios kuria programinius produktus. Taip pat sukurta metodika, projekto įgyvendinimo metu surinktomis ir apibendrintomis žiniomis, galėtų pasinaudoti aukštosios mokyklos ar kitos kompanijos, kurios kuria panašius produktus.

Nors sistema turi vieną pagrindinį tikslą: pateikti vartotojams sugeneruotus testus, bet vykdant projektavimo darbus buvo pagalvota ir apie žmones, kurie šią sistemą naudos. Todėl apibrėžiant sistemos panaudojimo atvejus buvo išskirti ne vien tie, kurie yra tiesiogiai susiję su testų generavimu, bet ir tie, kurie padidina patogumą naudotis sistema. Iš viso buvo išskirta 18 panaudojimo atvejų, kurie gali būti suskirstyti į grupes:

- Panaudojimo atvejai susiję su testavimo aplinkos konfigūracijos valdymu. Tai testuojamos programinės įrangos nurodymas, testų generavimo parametrų nustatymas.
- Testų generavimo panaudos atvejis.
- Panaudojimo atvejai susiję su testų valdymu. Testų išsaugojimas, atidarymas, peržiūra.

„Pav. 5 - Sistemos panaudos atvejų diagrama“ paveikslėlyje yra pateikiama UML panaudos atvejų diagrama.



Pav. 5 - Sistemos panaudos atvejų diagrama

Čia yra išskiriamos dvi pagrindinės vartotojų grupės: programuotojai ir testuotojai. Kadangi abi vartotojų grupės naudosis tomis pačiomis sistemos funkcijomis, jos buvo apibendrintos į vieną grupę „Programuotojas/testuotojas“. Tačiau reikėtų paminėti, jog:

- Programuotojas – tai asmuo, kuris pasinaudodamas sistema nurodo, kokiai programinei įrangai reikia sugeneruoti testus – pateikia sukompiliuotą programą ir jos išeities tekstus. Nurodo kuriuos testuojamos sistemos komponentus (klases, metodus) reikia patikrinti ir paleidžia testų generavimą. Taip pat jei reikia gali pakoreguoti sugeneruotus testus.

- Testuotojas – tai asmuo, kurio pagrindinis tikslas yra pasinaudoti sugeneruotais testais, kuriuos jam pateikia programuotojas kartu su testuojama sistema. Tačiau jei sugeneruoti testai nėra pateikiami, jis gali pats atlikti programuotojų kategorijai priklausančių asmenų sprendžiamus uždavinius. Papildomai testuotojai analizuoja sugeneruotus testus bei priima su jais susijusius sprendimus (testų pašalinimas/ papildymas ir kt.).

3.2. Reikalavimai sistemai

Detalizuojant sistemos panaudos atvejus buvo apibrėžti 67 funkcinių ir 12 nefunkcinių reikalavimų. Dauguma funkcinių reikalavimų yra susiję su projekto valdymu sistemoje bei testavimo eiga ir testų generavimu bei jų pateikimu vartotojui. Nefunkciniai reikalavimai tai reikalavimai sistemos išvaizdai, atvaizduojamai informacijai, kuriamų testų struktūrai ir pan. Svarbiausi reikalavimai yra šie:

Reikalavimas #:	**33**	**Reikalavimo tipas:**	**10**	**Įvykis/ panaudojimo atvejis #:**	**5**
Aprašymas:	Nurodant testų generavimo parametrus turi būti pateikiamas testuojamos sistemos metodų ir klasių sąrašas, tam kad būtų galima pasirinkti, kuriems metodams ir klasėms turi būti generuojami testai.				
Pagrindimas:	Kadangi ne visada reikia generuoti testus visai sistemai, vartotojas turi turėti galimybę pasirinkti, kurioms dalims reikia generuoti testus. T.y. vartotojas turi turėti galimybę sužymėti tuos metodus ir klases, kurioms testų generavimo sistema turi sugeneruoti testus.				
Šaltinis:	Užsakovas				
Tikimo kriterijus:	Vartotojas gali pasirinkti metodus ir klases, kurioms turi būti sugeneruoti testai.				
Užsakovo tenkinimas:	3	**Užsakovo netenkinimas:**	3		
Priklausomybės:	R#31, R#32	**Konfliktai:**	-		
Papildoma medžiaga:	-				
Istorija:	Užregistruotas 2010 kovo 10 d.				

Reikalavimas #:	**35**	**Reikalavimo tipas:**	**10**	**Įvykis/ panaudojimo atvejis #:**	**5**
Aprašymas:	Vartotojas turi būti informuojamas apie tai kokius parametrus koks metodas turi, taip pat jis turi turėti galimybę nurodyti į kuriuos parametrus reikia atsižvelgti generuojant testus.				
Pagrindimas:	Testuojant ne visada yra būtina patikrinti sistemos reakciją į visus parametrus. Pvz.: tarkim yra metodas „suma(int a, int b, boolean c)“ skirtas sudėti du skaičius ir kintamasis c jame nėra naudojamas, tačiau programos išeities tekste yra įrašytas. Vartotojas turi turėti galimybę nurodyti, jog generuojant testus turi būti atsižvelgiama tik į a ir b reikšmes.				
Šaltinis:	Užsakovas				
Tikimo kriterijus:	Vartotojas gali nurodyti į kuriuos metodo parametrus turi būti atsižvelgta generuojant testus.				

Užsakovo tenkinimas:	3	**Užsakovo netenkinimas:**	2
Priklausomybės:	R#31, R#33	**Konfliktai:**	-
Papildoma medžiaga:	-		
Istorija:	Užregistruotas 2010 kovo 10 d.		

Reikalavimas #:	**43**	**Reikalavimo tipas:**	**10**	**Įvykis/ panaudojimo atvejis #:**	**7**
Aprašymas:	Testai turi būti generuojami Java programavimo kalba.				
Pagrindimas:	Kadangi testuojama programinė įranga bus parašyta Java programavimo kalba, tai ir generuojami testai turi būti generuojami naudojant Java programavimo kalbos sintaksę.				
Šaltinis:	Užsakovas				
Tikimo kriterijus:	Sugeneruoti testai yra parašyti Java programavimo kalba.				
Užsakovo tenkinimas:	4	**Užsakovo netenkinimas:**	4		
Priklausomybės:	R#41	**Konfliktai:**	-		
Papildoma medžiaga:	-				
Istorija:	Užregistruotas 2010 kovo 10 d.				

3.3. Testų generavimo algoritmas

Šiame skyriuje yra aprašomas testų generavimo metodas sukurtas Tomo Milašiaus ir Dominyko Bariso. Šis metodas ne tik leidžia atsikratyti atsitiktinio testų generavimo metodo paremto mutaciniu testavimu, aprašyto analizės dalyje, trūkumų, bet ir pasiekti 100% programinės įrangos vykdymo kelių padengimą. Sukurtas testų generavimo metodas remiasi simboliniu vykdymu, kuris taip pat yra aprašytas analizės dalyje. Tačiau čia analizuojamas ne tik simbolinis vykdymas, bet ir kaip jis yra panaudojamas generuojant testus.

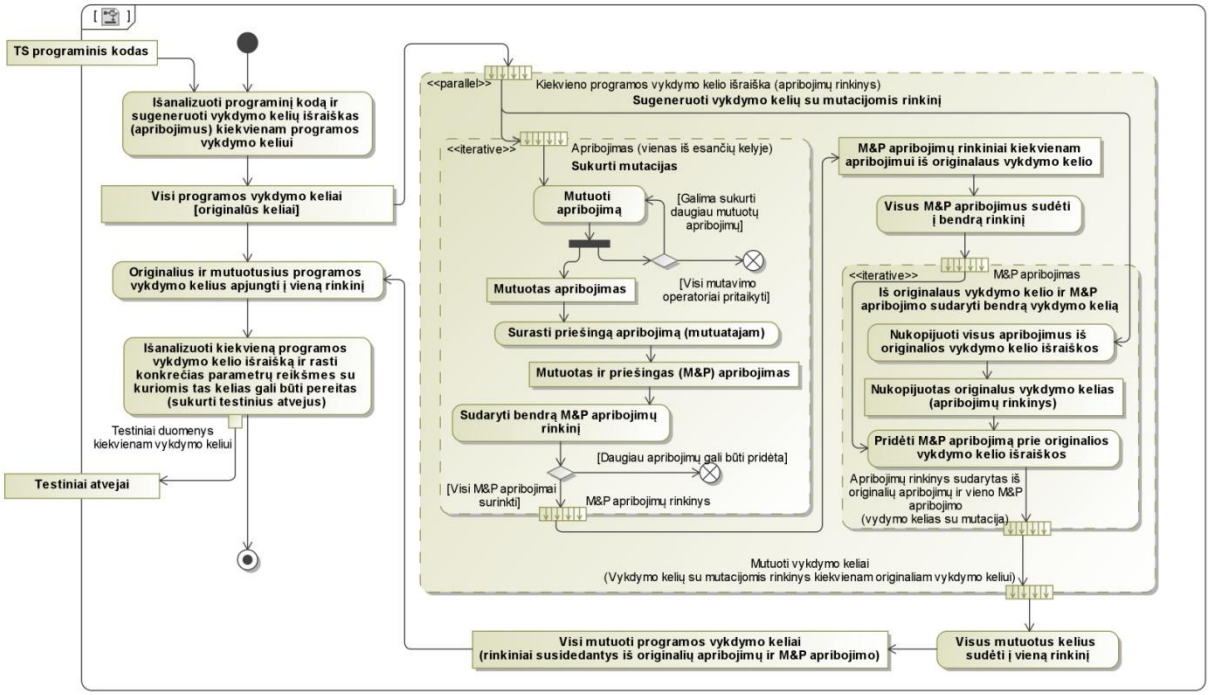
3.3.1. Testavimo technologijos pasiūlymas

Visas testavimo procesas, kuriame naudojamas sukurtas metodas, gali būti sudalintas į šias dalis:

- Vykdymo kelių išraiškų (sąlygų) sudarymas iš programinio kodo;
- Testinių duomenų generavimas analizuojant vykdymo kelių išraiškas. Šiame etape yra vykdomas patobulintas testų generavimo metodas, kuris leidžia aptikti programinės įrangos mutacijas.
- Sugeneruotų testinių atvejų vykdymas ir gautų rezultatų palyginimas su lauktais rezultatais (apskaičiuotais generuojant testus). Šiame etape testinis

atvejis yra laikomas nepavykusiu, jei gauti rezultatai nesutampa su lauktais rezultatais, o testuota programa, kurioje rezultatai nesutampa – mutuota.

Pagrindinis tikslas yra sukurti vienetų karkasui (jUnit) skirtus testus, kadangi šie testai gali būti paleidžiami gan dažnai ir yra vykdomi pakankamai greitai. „Pav. 6 - Siūlomo testų generavimo metodo veiklų diagrama“ paveikslėlyje yra pateikiama detalizuota siūlomo testavimo metodo UML veiklų diagrama. Čia: *TS* – testuojama sistema; *M&P* – mutuota(-as) ir priešinga(-as); *Vykdymo kelias* – tai apribojimų rinkinys (vykdymo kelio išraiška), kurį išsprendus galima rasti konkrečias parametrų reikšmes, su kuriomis vykdant programą bus pereitas būtent tas kelias (pvz.: vykdymo kelias gali būti apibrėžiamas apribojimais a>0 && b+a<2, ir parinkus a = 1, b = 0 bus pereinamas būtent šis vykdymo kelias). Taip pat reikia atkreipti dėmesį į tai, jog vykdymo kelio apribojimai yra gaunami pasinaudojant simboliniu vykdymu.



Pav. 6 - Siūlomo testų generavimo metodo veiklų diagrama

Šiuo metodu sugeneruoti testiniai atvejai turėtų aptikti programinės įrangos klaidas, kurios gali atsirasti dėl:

- Programinio kodo modifikavimo.
- Programinės įrangos priežiūros metu atnaujintų trečiųjų šalių paketų.

- Pasikeitusios programinės įrangos vykdymo platformos. Tai kai pasikeičia Java virtuali mašina (atnaujinama jos versija ar panaudojama kitos kompanijos kurta JVM) arba pasikeičia operacinė sistema, kurioje veikia testuojama programinė įranga.

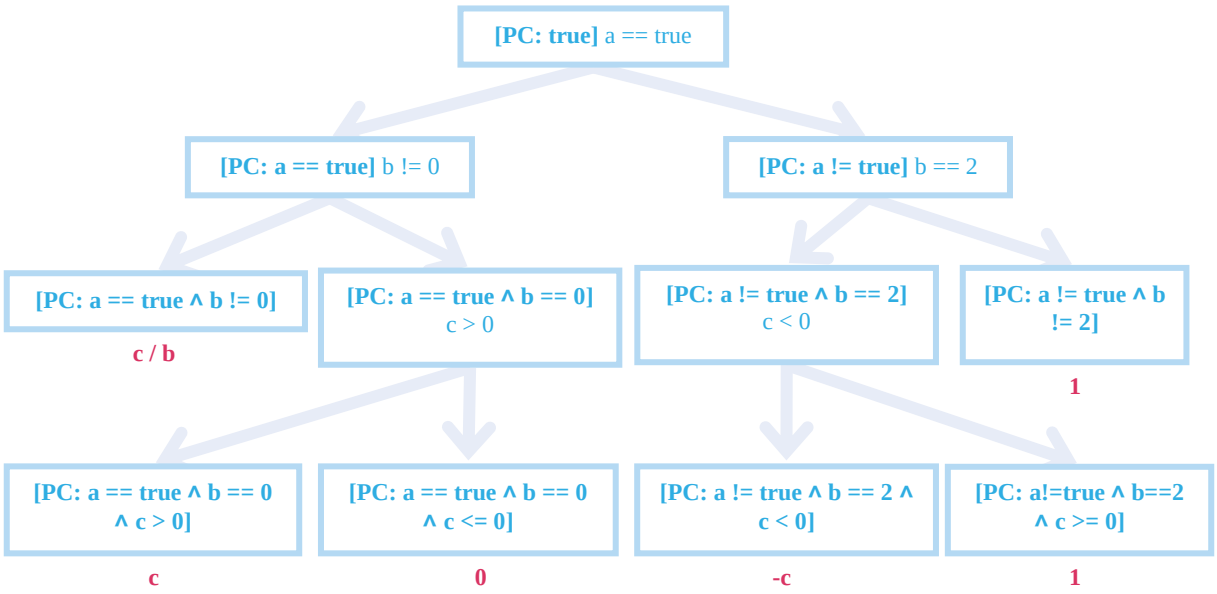
3.3.2. Būsenų modelio sudarymas ir vykdymo kelių išraiškų išskyrimas

Sistemos būsenų (vykdymo kelių) modelis yra sudaromas simbolinio vykdymo pagalba, taip kaip aprašyta analitinės dalies 2.4.3 „Testų generavimas panaudojant simbolinį vykdymą“ skyriuje. Šioje dalyje yra pateikiamas konkretus programos pavyzdys ir aprašomas visas būsenų modelio sudarymo procesas.

Tarkim turim tokį programinio kodo dalį (metodą), kuriam reikia sudaryti testinius atvejus:

```
 1: public int metodas(boolean a, int b, int c) {
 2:   if(a) {
 3:         if(b != 0)
 4:               return c/b;
 5:         if(c > 0)
 6:               return c;
 7:         return 0;
 8:   } else if (b == 2) {
 9:         if(c < 0)
10:               return -c;
11:   }
12:   return 1;
13: }
```

Iš programinio kodo yra sudaromas programos būsenų modelis pavaizduotas „Pav. 7 - Būsenų modelis sudarytas simbolinio vykdymo pagalba“ paveikslėlyje, kuriame atsispindi visos programos vykdymo šakos (vykdymo keliai).



Pav. 7 - Būsenų modelis sudarytas simbolinio vykdymo pagalba

Pavyzdžiui iš *if* sąlygos sakinio esančio 2 programinio kodo eilutėje yra apibrėžiama pradinė sistemos būsena „[PC: true] a == true“ ir ji atvaizduojama modelio viršuje (kadangi tai pirma aptikta sistemos būsena). Kiekviena sistemos būsena yra asocijuojama su tam tikru programinio kodo gabalu ir aprašoma dvejomis dalimis:

- Laužtiniuose skliaustuose yra sudedami apribojimai, kurie turi būti tenkinami, kad atitinkama būsena būtų pasiekta. Pradiniu momentu yra įrašoma *true* reikšmė – tai reiškia, jog metodas gali būti vykdomas.
- Po laužtinių skliaustų yra aprašoma sąlyga, kuri yra tikrinama atitinkamoje būsenoje. Nuo šios sąlygos priklauso tolimesnis modelio šakojimasis (perėjimai į tolimesnes būsenas).

Kiekviena sistemos būsena priklausomai nuo ją sąlygojusio programinio kodo sakinio (*if, switch, for* ir t.t.) gali turėti kelis išsišakojimus (perėjimus) į kitas būsenas: šiuo atveju yra pereinama į dvi būsenas su atitinkamais vykdymo apribojimais „a==true” ir „a!=true”. Toliau yra tiriami tolimesni programinio kodo sakiniai kiekvienai šakai (programos vykdymo keliui) atskirai ir taip paeiliui yra sudarinėjimas programos būsenų modelis apimantis visus programos vykdymo kelius. T.y. šiuo atveju, kai *a==true,* toliau yra tiriamos 3-7 eilutės, o kai *a!=true* yra tiriamos 8-11 programinio kodo eilutės. Būsenų sudarinėjimas kiekviename kelyje yra tęsiamas tol kol yra pasiekiamas programos vykdymo galas (*return* sakinys ir pan.) arba yra sudaromi tokie apribojimai, kurie neturi sprendinio. Pvz.: jei 5 eilutėje būtų įdėtas tikrinimas *if(!a)* šio sakinio *true* šaka nebūtų tiriama, nes tokia būsena būtų nepasiekiama dėl

prieš tai atsiradusio apribojimo *a==true* iš 4 eilutėje esančio *if(a)* sakinio. Taip paeiliui pereinama nuo vienos būsenos prie kitos ir ištiriami visi programos vykdymo keliai.

Sudarius programos būsenų modelį yra analizuojamos tos būsenos, kurios neturi nei vieno tolimesnio perėjimo į kitą būseną. Šios būsenos, kiekviena atskirai, atspindi pilną programos vykdymo kelią. Taip gaunamos konkrečios programos vykdymo kelių išraiškos ir laukiamas rezultatas įvykdžius konkretų kelią. Iš pateikto pavyzdžio galima išskirti vykdymo kelius, kurie yra pateikti „Lentelė nr. 1 - Pavyzdinės programos vykdymo kelių sąlygos ir laukiami rezultatai“ lentelėje.

Lentelė nr. 1 - Pavyzdinės programos vykdymo kelių sąlygos ir laukiami rezultatai

Nr.	Kelio vykdymo sąlyga	Laukiamas rezultatas
1	`a == true && b != 0`	`c/b`
2	`a == true && b == 0 && c > 0`	`c`
3	`a == true && b == 0 && c <= 0`	`0`
4	`a != true && b == 2 && c < 0`	`-c`
5	`a != true && b == 2 && c >= 0`	`1`
6	`a != true && b != 2`	`1`

Šios sąlygos iš esmės labai gerai apibrėžia sistemos veikimą, tačiau yra netinkamos testavimui, dėl to, jog vykdant testavimą turi būti pasirinktos konkrečios kintamųjų reikšmės, o ne vykdymo kelių išraiškos. Taip pat buvo pastebėta, jog šios sąlygos iš esmės yra apribojimų programavimo (angl. *Constraint Programming*) paradigmos išraiškos, kurios gali būti išspręstos pasinaudojant tais pačiais metodais kaip ir šiame programavime. Jas išsprendus yra gaunamos konkrečios kintamųjų reikšmės, kurios jas tenkina – tai reiškia, jog iškvietus metodą su šiomis reikšmėmis bus pereitas atitinkamas programos vykdymo kelias. Pvz.: išsprendus pirmojo kelio apribojimus „`a==true && b!=0`“ gali būti sugeneruoti tokie duomenys: `a=true`; `b=1`. Su šiais duomenimis visada bus vykdomas tik pirmasis kelias (nesvarbu kokia *c* reikšmė – ši reikšmė parenkama atsitiktinai, kadangi neturi jokių apribojimų). Žinant šias reikšmes, grąžinamą rezultatą ir informaciją apie testuojamą metodą (metodo pavadinimą ir klasę, kurioje jis yra) yra galima sudaryti konkrečius testus (pvz. jUnit karkasui). Pvz.:

```
@Test
public void metodas_Test1() {
      Klase obj = new Klase();
      assertEquals( obj.metodas(true, 1, 1), 1);
}
```

Tačiau kaip parodė tolimesni tyrimai net ir sugeneravus testus visoms modelio šakoms (t.y. visiems programos vykdymo keliams) šie testai apsaugo tik nuo dalies programos mutantų. Pvz. aukščiau pateiktos programos `if(b!=0)` sąlygą pakeitus į sąlygą `if(b>0)` ir įvykdžius aukščiau pateiktą testą `metodas_Test1()` su parametrais `(true, 1, 1)` grąžinama reikšmė yra lygi 1, todėl toks programos mutantas nėra aptinkamas. T.y. su tokiais testiniais duomenimis nėra iškviečiamas kitos šakos vykdymas, o yra vykdoma ta pati šaka. Taip nutinka dėl to, jog šiuo atveju `b` ir `c` parametrų reikšmės yra pasirenkamos atsitiktinai, tačiau tik dalis reikšmių saugo nuo sistemos mutantų. `b` kintamojo reikšmė yra apribota `b != 0` sąlyga, tačiau tai tik viena negalima reikšmė iš begalybės. Vienintelio `a` kintamojo reikšmė yra griežtai apibrėžta.

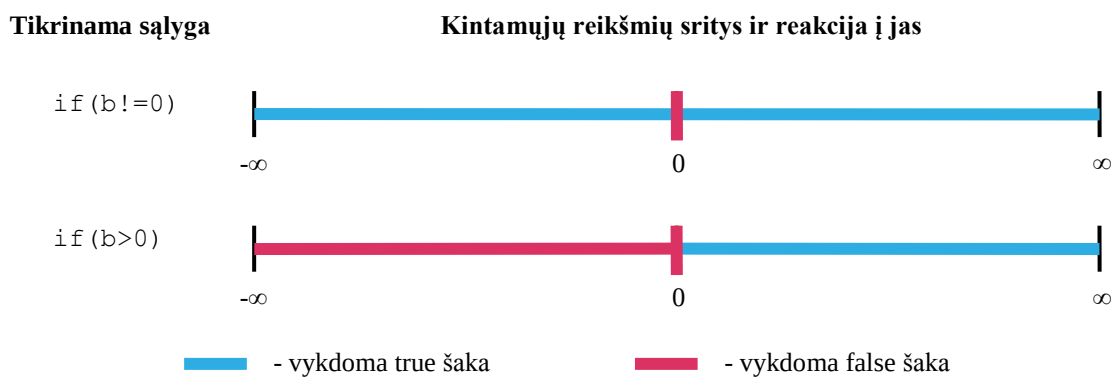
3.3.3. Testų skirtų mutantų aptikimui generavimas

Sprendžiant testų, kurie neaptinka mutantų problemą buvo pamėginta suprasti kas yra sistemos mutantas, su kokiomis kintamųjų reikšmėmis tą mutantą būtų galima aptikti, kaip jis būtų atvaizduojamas būsenų modelyje ir kaip būtų galima jo išvengti.

Buvo išskirtos trys mutantų kategorijos nuo kurių būtų galima apsisaugoti:

- Aritmetinių operacijų mutantai – tai mutantai, kai yra sukeičiamos aritmetinės operacijos („+“, „-“, „*“, „/“).
- Palyginimo operacijų mutantai – tai mutantai, kai yra sukeičiamos palyginimo operacijos („>“, „>=“, „==“, „!=“, „>=“, „>“).
- Loginiai mutantai – tai mutantai, kai yra sukeičiamos „&&“ ir „||“ (and, or, xor) operacijos.

Išanalizavus esamą mutacinio testavimo metodiką buvo prieita išvados, jog nuo mutanto galima apsisaugoti tuomet, kai yra sugeneruojami tokie duomenys, kurie iškviečia skirtingas sistemos reakcijas originalioje programoje ir jos mutante. T.y. kai su tais pačiais duomenimis yra vykdomi skirtingi programos keliai. Tai pasiekti galima analizuojant su kokiomis reikšmėmis prie kokių sąlygų yra vykdomi programos true ir false keliai. Pvz.: esant sąlygai `if(b!=0)`, `true` šaka bus vykdoma tada, kai b reikšmė bus rėžiuose $[-\infty;0)$ ir $(0;\infty]$, o `false` šaka bus vykdoma, kai `b` bus lygu 0. Tačiau jei šią sąlygą pakeisime į `if(b>0)`, `true` šaka bus vykdoma, kai `b` bus rėžyje $(0;\infty]$, o `false` šaka, kai `b` bus rėžyje $[-\infty;0]$. Grafinis šių teiginių atvaizdavimas yra pateiktas „Pav. 8 - Reakcijų į kintamųjų reikšmes palyginimas“ paveikslėlyje.



Pav. 8 - Reakcijų į kintamųjų reikšmes palyginimas

Iš pateikto paveikslėlio galima nesunkiai pastebėti, jog sąlygų reakcija į kintamojo `b` reikšmes skiriasi tik tuomet, kai jis yra rėžyje [-∞;0). Vadinasi norint apsisaugoti nuo mutavimo iš `if(b!=0)` į `if(b>0)` ir atvirkščiai reikia kintamųjų reikšmes generuoti būtent šituose rėžiuose – kur skiriasi sistemos reakcija (vykdymo keliai). Tačiau bendrą atvejį tinkantį visoms mutacijoms nėra taip paprasta apibrėžti. Todėl buvo pamėginta įsivaizduoti kaip atrodytų mutuotos programos vykdymo kelių išraiškos.

Jei anksčiau nagrinėtoje programoje pritaikytume tą pačia mutaciją (`if(b!=0)` į `if(b>0)`) gautume tokius programos vykdymo kelius pateiktus „Lentelė nr. 2 - Pavyzdinės mutuotos programos vykdymo kelių sąlygos ir laukiami rezultatai“ lentelėje.

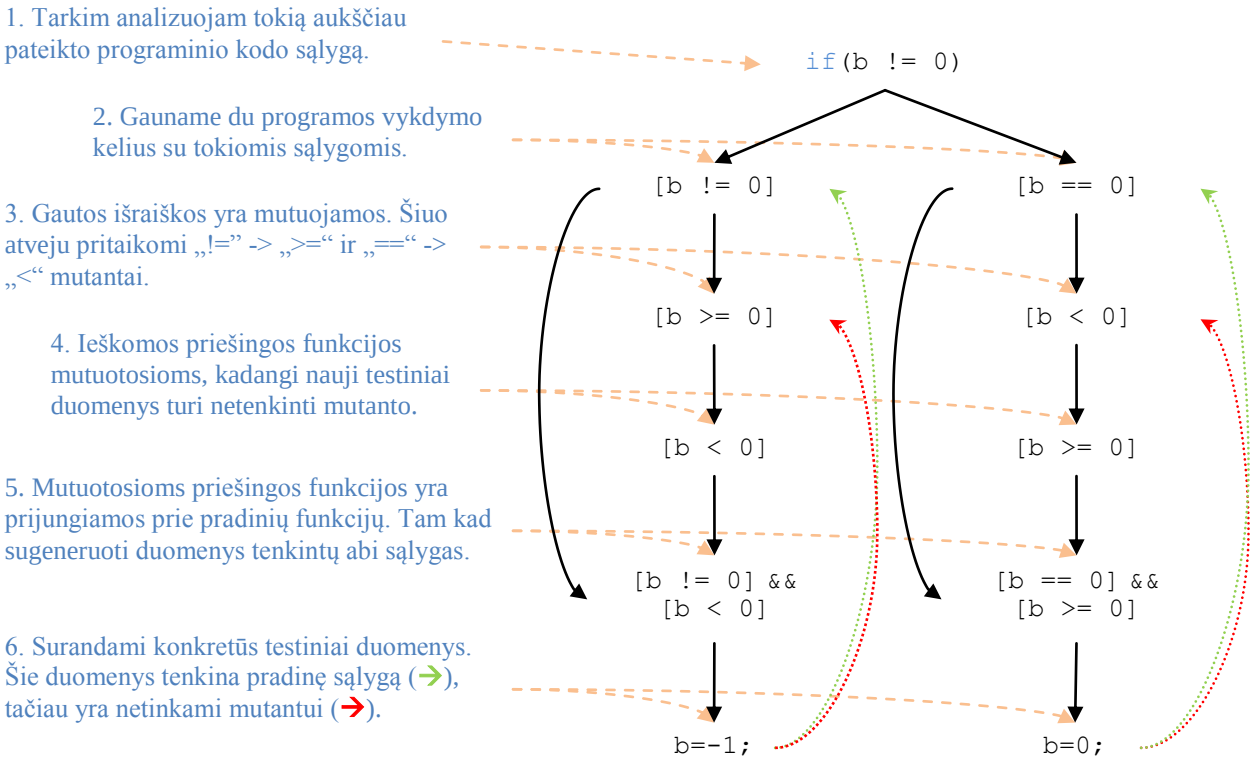
Lentelė nr. 2 - Pavyzdinės mutuotos programos vykdymo kelių sąlygos ir laukiami rezultatai

Nr.	Kelio vykdymo sąlyga	Laukiamas rezultatas
1	`a == true && b > 0`	`c/b`
2	`a == true && b <= 0 && c > 0`	`c`
3	`a == true && b <= 0 && c <= 0`	`0`
4	`a != true && b == 2 && c < 0`	`-c`
5	`a != true && b == 2 && c >= 0`	`1`
6	`a != true && b != 2`	`1`

Reikia pastebėti, kad šios 1-3 kelių vykdymo išraiškos nuo pradinių pateiktų „Lentelė nr. 1 - Pavyzdinės programos vykdymo kelių sąlygos ir laukiami rezultatai“ lentelėje skiriasi tik vienu apribojimu ir taip pat gali būti išspręstos. Tačiau jas išsprendę mes nelabai ką sužinotume, nes vis tiek negautume konkrečių reikšmių prie kurių programos veikimas yra skirtingas. Ir čia iš naujo apibendrinus visą iki šiol surinktą informaciją buvo prieiti esminiai sprendimai:

- Mutantą galima pagauti tada, kai skiriasi sistemos reakcija į pateiktas kintamųjų reikšmes, t.y. tada kai pateikus tas pačias reikšmes vienoje programoje yra vykdoma tarkim true šaka, o kitoje false.
- Tokią reakciją galima iššaukti generuojant kintamųjų reikšmes, kurios tiktų pradiniam programos keliui, bet netiktų tam pačiam keliui programos mutante. O kito kelio veikimą mutante galima iššaukti įvedus priešingą apribojimą mutantui.

Taip buvo sukurtas metodas kurio pagrindinė mintis yra ta, kad įvykdžius testuojamos programos modelio sudarymą ir gavus programos vykdymo kelių išraiškas jos turi būti mutuojamos, surandamos joms priešingos sąlygos ir prijungiamos prie pradinių išraiškų. Taip yra gaunami duomenys, kurie tinka pradiniam programos vykdymo keliui, bet netinkami tam pačiam mutanto keliui (pridėjus priešingą mutantą yra sumažinama duomenų aibė iš kurios pasirenkami konkretūs testiniai duomenys). Metodo veikimo principas (etapai) su konkrečiu pavyzdžiu yra pateiktas „Pav. 9 - Sukurto metodo veikimo principas su pavyzdžiu“ paveikslėlyje.



Pav. 9 - Sukurto metodo veikimo principas su pavyzdžiu

Tokiu būdu yra gaunami testiniai atvejai aptinkantys konkrečius mutantus. Pasinaudojant šiuo metodu yra gaunamas toks testinis atvejis:

```
@Test
public void metodas_Test2() {
      Klase obj = new Klase();
      assertEquals( obj.metodas(true, -1, 1), -1);
}
```

Šis testas yra ne tik tinkamas originaliai programai, bet ir saugo nuo `if(b>0)` mutanto. O sukurtas testų generavimo metodas yra tinkamas ir bendru atveju ne tik šiam, konkrečiam, pavyzdžiui. Tai leidžia sugeneruoti testus, kurie yra atsparūs mutantams ir aptinka programos veikimo pasikeitimus (galimas programines klaidas).

Testų generavimo algoritmas buvo aprašytas pseudokodu:

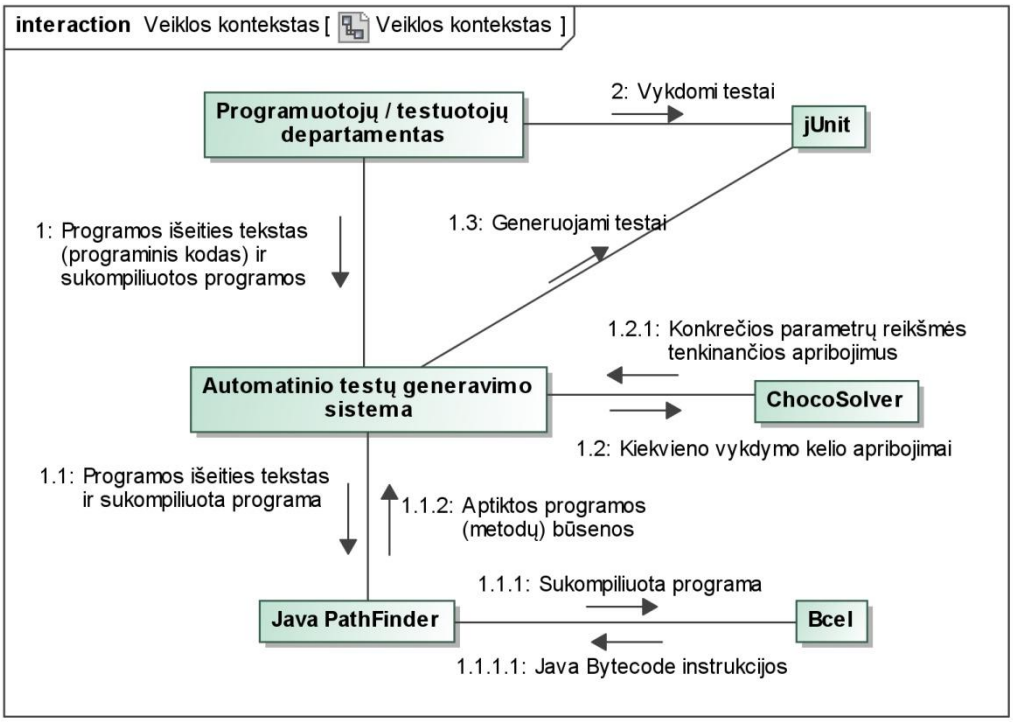
```
MethodsToBeTested : List of methods which should be tested
MethodsInfoList : List of collected information about methods
TestSuite : A set of returned testcases

1.    MethodsInfoList ::= []
2.     TestSuite ::= []
3.    for each method in MethodsToBeTested
4.          MethodInfo = new MethodInfo;
5.          MethodInfo.method = method;
6.          MethodInfo.pathConditions = JPF.findPathConditions(method);
7.          MethodsInfoList.append(MethodInfo);
8.    end for
9.    for each MethodInfo in MethodsInfoList
10.         for each pathCondition in MethodInfo.pathConditions
11.               mutatedPathConditions = mutate(PathCondition);
12.               mutatedPathConditions = invert(mutatedPathConditions);
13.               for each mPC in mutatedPathConditions
14.                     mPC = pathCondition && mPC;
15.                     MethodInfo.mutatedPathConditions.append(mPC);
16.               end for
17.         end for
18.   end for
19.   for each MethodInfo in MethodsInfoList
20.         for each pathCondition in MethodInfo.pathConditions
21.               TestCase = generateTestCase(pathCondition.solve());
22.               TestSuite.append(TestCase);
23.         end for
24.         for each pathCondition in MethodInfo.mutatedPathConditions
25.               TestCase = generateTestCase(pathCondition.solve());
26.               TestSuite.append(TestCase);
27.         end for
28.   end for
29.   return TestSuite;
```

3.4. Sistemos architektūra

Šiame skyriuje yra pateikiami sukurtos sistemos esminiai architektūriniai sprendimai. Sistemos projektavimo proceso metu buvo naudojama UML projektavimo kalba (*NoMagic MagicDraw UML* įrankis).

3.4.1. Sistemos kontekstas

Kadangi sistema yra gan didelės apimties, siekiant sumažinti jos kūrimo sąnaudas, buvo nuspręsta pasinaudoti jau sukurtais įrankiais. Paveikslėlyje „Pav. 10 - Sistemos sąveikos su kitais įrankiais UML diagrama“ yra pateikiamas sistemos veikimo kontekstas.



Pav. 10 - Sistemos sąveikos su kitais įrankiais UML diagrama

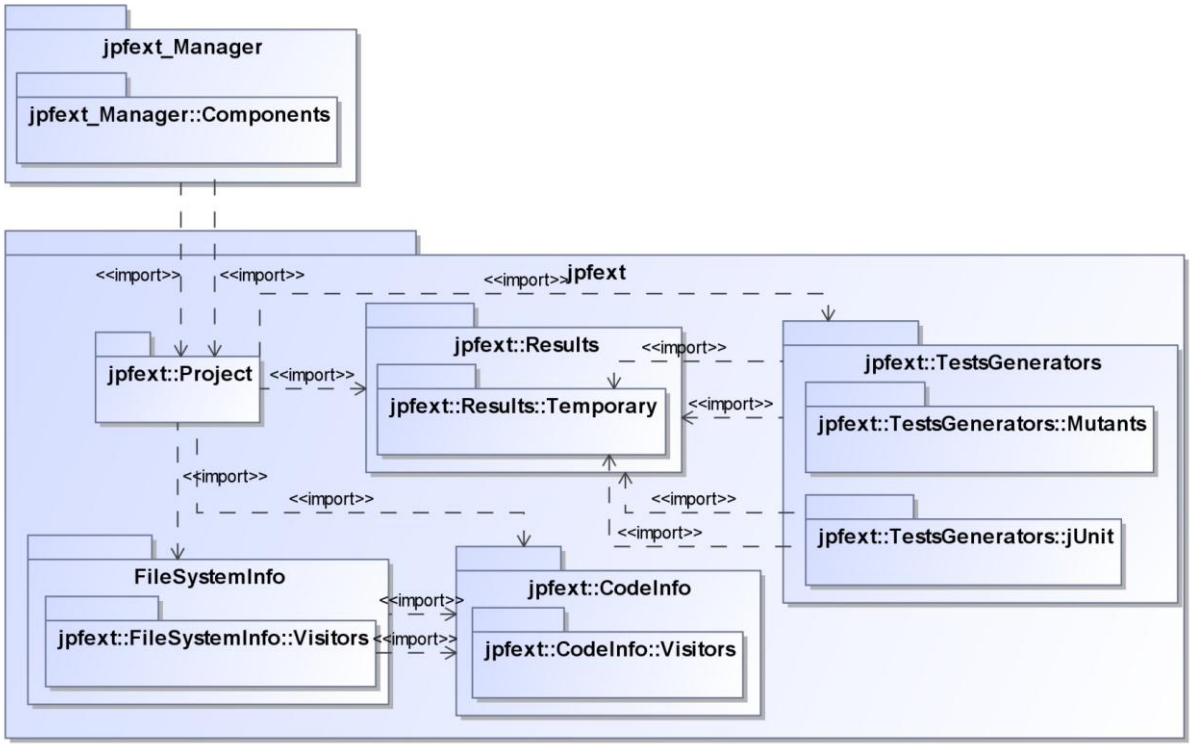
Pati sistema yra pažymėta „Automatinio testų generavimo sistema“. Ja naudojasi programuotojų/testuotojų departamento personalas, pateikdamas sistemai testuojamos programinės įrangos išeities tekstus ir pačią sukompiliuotą programą. Sistema šią programą paduoda Java PathFinder įrankiui (tiksliau Symbolic PathFinder[5] „jpf-symbc“ subprojektui), kuris atlieka simbolinį vykdymą ir grąžina sistemai aptiktas programos būsenas (vykdymo kelius). Tuo tarpu Java PathFinder naudoja eilę kitų projektų, tokių kaip Bcel[6] projektą - Java bytecode instrukcijų aptikimui ir pan. Vėliau, gavus sistemos būsenų modelį, sistemoje yra atliekamos jo mutacijos ir pasinaudojant Java PathFinder duomenų struktūromis, šis modelis yra išsprendžiamas ChocoSolver[7] įrankio pagalba. Iš gautų konkrečių parametrų reikšmių yra

[5] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
[6] http://jakarta.apache.org/bcel/manual.html
[7] http://www.emn.fr/z-info/choco-solver/

sudaromi testiniai atvejai, kurie realizuojami jUnit[8] karkase. Vėliau šiuos testus gali įvykdyti programuotojų/ testuotojų departamente dirbantis personalas.

3.4.2. Statinis sistemos vaizdas

Šis skyrius aprašo esminius sistemos loginės struktūros aspektus. Pateikia sistemos išskaidymą į paketus ir juos sudarančias klases. Paveikslėlyje „Pav. 11 - Sistemos paketų diagrama“ yra pateikiamas sistemos išskaidymas į paketus.



Pav. 11 - Sistemos paketų diagrama

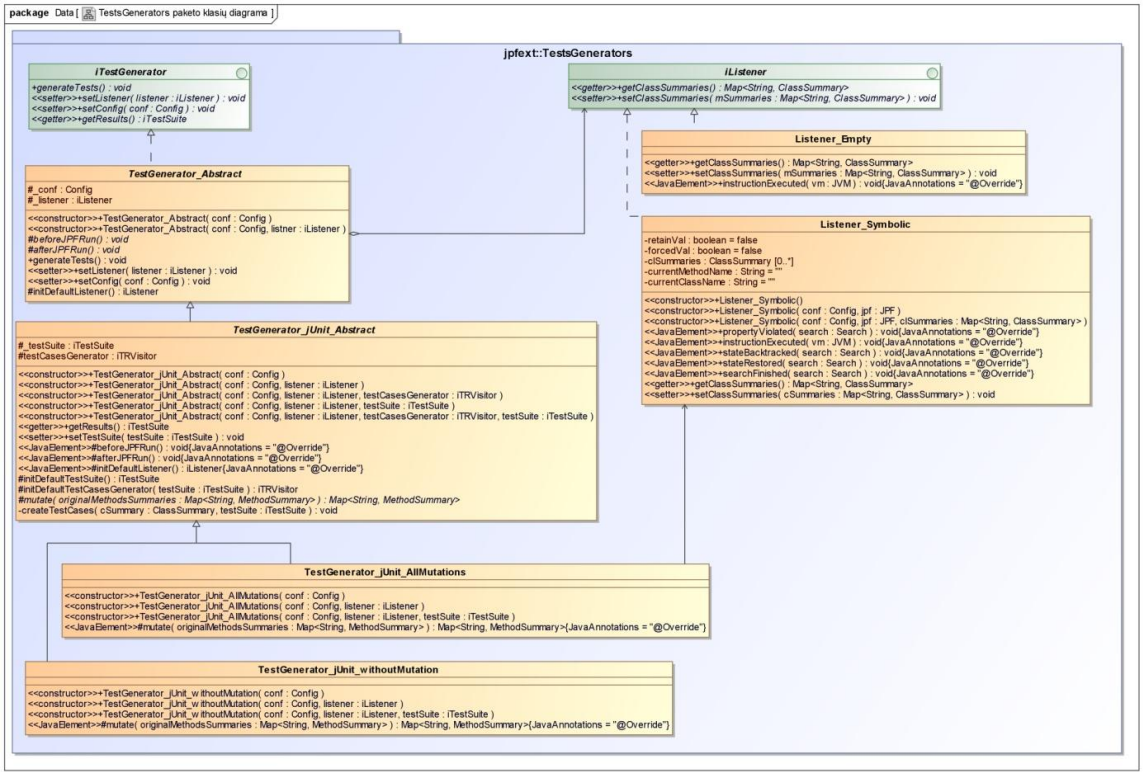
Visa sistema yra sudalinta į 13 paketų:

- jpext_Manager – šiame pakete yra aprašomi kuriamoje sistemoje naudojami dialogai (vartoto sąsajos objektai).
- jpfext_Manager.Components - Šiame pakete yra aprašomi sistemoje naudojami informacijos atvaizdavimo komponentai.
- jpfext.Project – šiame pakete yra realizuojamas pagrindinis darbas su sistema – realizuojami projektai, duomenų nuskaitymo, apdorojimo komandos, testų generavimo komandos ir kt.

[8] http://junit.org/

- jpfext.FileSystemInfo - pakete realizuojamos klasės, kurios leidžia nuskaityti direktorijas ir surinkti informaciją apie ten esančius dokumentus.
- jpfext.CodeInfo - paketas skirtas kaupti informacijai apie aptiktus paketus, klases, metodus bei atributus ir leidžia vartotojui nustatyti, kurie iš jų turi būti testuojami (generuojami testai), o kurie ne.
- jpfext.TestsGenerators – šiame pakete realizuojamos svarbiausios sistemos funkcijos – testų generavimo algoritmai.
- jpfext.TestsGenerators.jUnit – šiame pakete realizuojami jUnit testinių atvejų sukūrimo algoritmai.
- jpfext.TestsGenerators.Mutants – šiame pakete realizuojami apribojimų mutavimo ir invertavimo algoritmai.
- jpfext.Results – paketas yra skirtas kaupti informacijai apie sugeneruotus testus ir jų metrikas.
- jpfext.Results.Temporary – paketas yra skirtas kaupti informacijai apie metodus, kuriems yra generuojami testai.

jpfext.TestsGenerators paketas



Pav. 12 - TestGenerators paketo klasių diagrama

Paveikslėlyje „Pav. 12 - TestGenerators paketo klasių diagrama“ yra pateikiama jpfext.TestsGenerators paketo klasių diagrama. Šiame pakete yra realizuotos pastarosios klasės bei interfeisai:

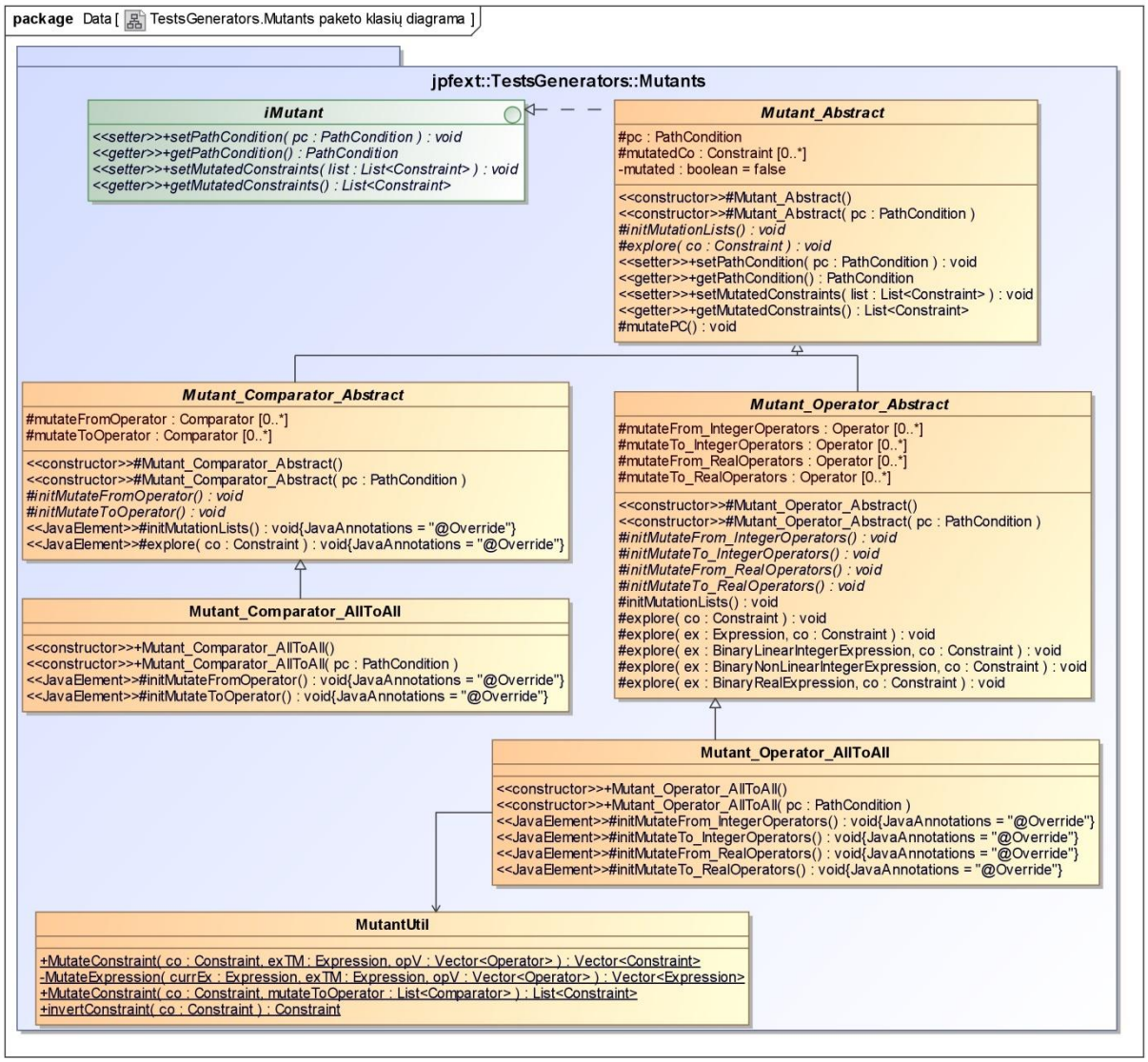
- *iListener, Listener_Empty, Listener_Symbolic* – šie komponentai yra naudojami surinkti reikiamai informacijai apie testuojamus metodus (jų vykdymo kelius, laukiamus rezultatus kiekviename vykdymo kelyje ir pan.).
- *iTestGenerator, TestGenerator_Abstract, TestGenerator_jUnit_Abstract* – šie komponentai yra realizuoti Template projektavimo šablono principu (anlg. *Template Method Desing Pattern*). Čia pagrindinis metodas yra *generateTests*(), kuriame yra apibrėžta testų generavimo algoritmo eiga bei realizuoti kvietimai į būsenų modelio sudarymo algoritmus esančius JPF projekte.
- *TestGenerator_jUnit_AllMutations, TestGenerator_jUnit_withoutMutations* – šios klasės realizuoja vykdymo kelių išraiškų mutavimo algoritmus – viena mutuoja visus kelius pritaikydama visus sistemoje sukurtus mutantus, kita – neatlieka jokio mutavimo. Mutavimo algoritmų įvairovė gali būti nesunkiai padidinta praplečiant *TestGenerator_jUnit_Abstract* klasę.

jpfext.TestsGenerators.Mutants paketas

Paveikslėlyje „Pav. 13 - Mutants paketo klasių diagrama“ yra pateikiama jpfext.TestsGenerators.Mutants paketo klasių diagrama. Šiame pakete yra realizuotos pastarosios klasės ir interfeisai susiję su vykdymo kelių išraiškų (apribojimų) mutavimu:

- *iMutant, Mutant_Abstrast* – šie komponentai realizuoja bendrą elgseną visiems mutantams tai: mutuojamo kelio nurodymas, mutuotų išraiškų rinkinio grąžinimas.
- *Mutant_Comparator_Abstract, Mutant_Comparator_AllToAll* – šios klasės realizuoja palyginimo operatorių mutavimą – pirmojoje yra apibrėžiama palyginimo operatorių mutavimo eiga, o antrojoje yra nurodoma kokie palyginimo operatoriai į kokius turi būti mutuojami.
- *Mutant_Operator_Abstract, Mutant_Operator_AllToAll* – šios klasės realizuoja aritmetinių operatorių mutavimą – pirmojoje yra apibrėžiama aritmetinių operatorių mutavimo eiga, o antrojoje yra nurodoma kokie aritmetiniai operatoriai į kokius turi būti mutuojami.

- *MutantUtil* – šioje klasėje yra realizuoti *static* tipo metodai skirti mutuotinų sąlygų paieškai, kiekvienos sąlygos atskirai mutavimas ir priešingų sąlygų mutuotoms sudarymas.



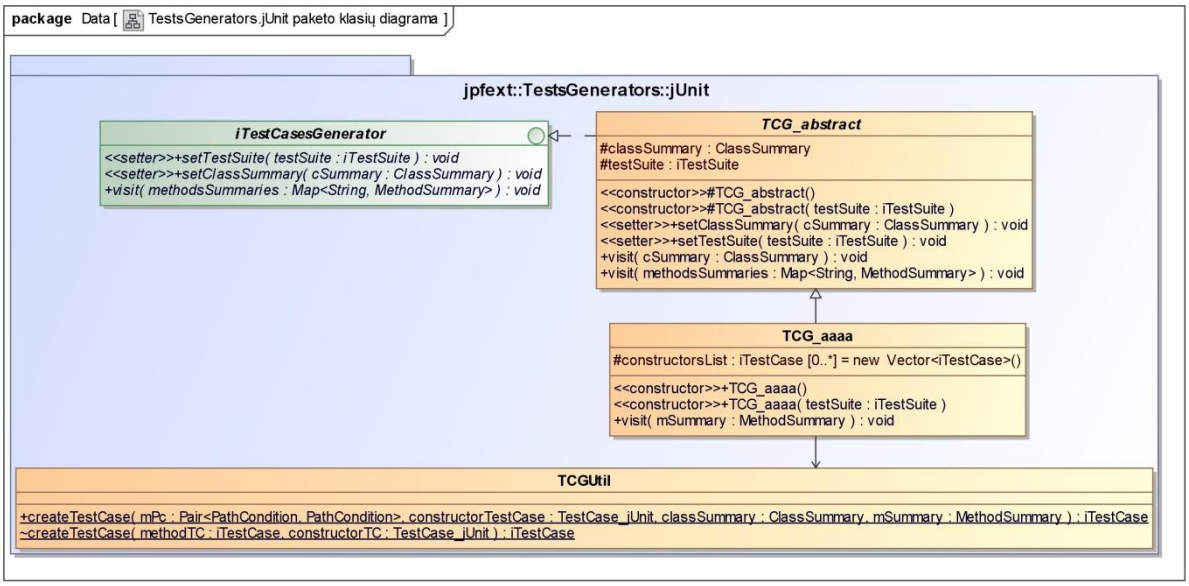
Pav. 13 - Mutants paketo klasių diagrama

jpfext.TestsGenerators.jUnit paketas

Paveikslėlyje „Pav. 14 - jUnit paketo klasių diagrama“ yra pateikiama jpfext.TestsGenerators.jUnit paketo klasių diagrama. Šiame pakete yra realizuotos pastarosios klasės ir interfeisai susiję su sugeneruotų testinių atvejų pritaikymu jUnit karkasui:

- *iTestCasesGenerator, TCG_abstract* – šie komponentai realizuoja bendrą funkcionalumą reikalingą sudarant jUnit testus.

- *TCG_aaaa* – ši klasė realizuoja testų sudarymo algoritmą, kur kiekvienas klasės konstruktoriaus vykdymo kelias yra kombinuojamas su kiekvienu testuojamo metodo vykdymo keliu.
- *TCGUtil* – ši klasė realizuoja jUnit testų kodo sudėliojimo seką.



Pav. 14 - jUnit paketo klasių diagrama

Programinio kodo metrikos

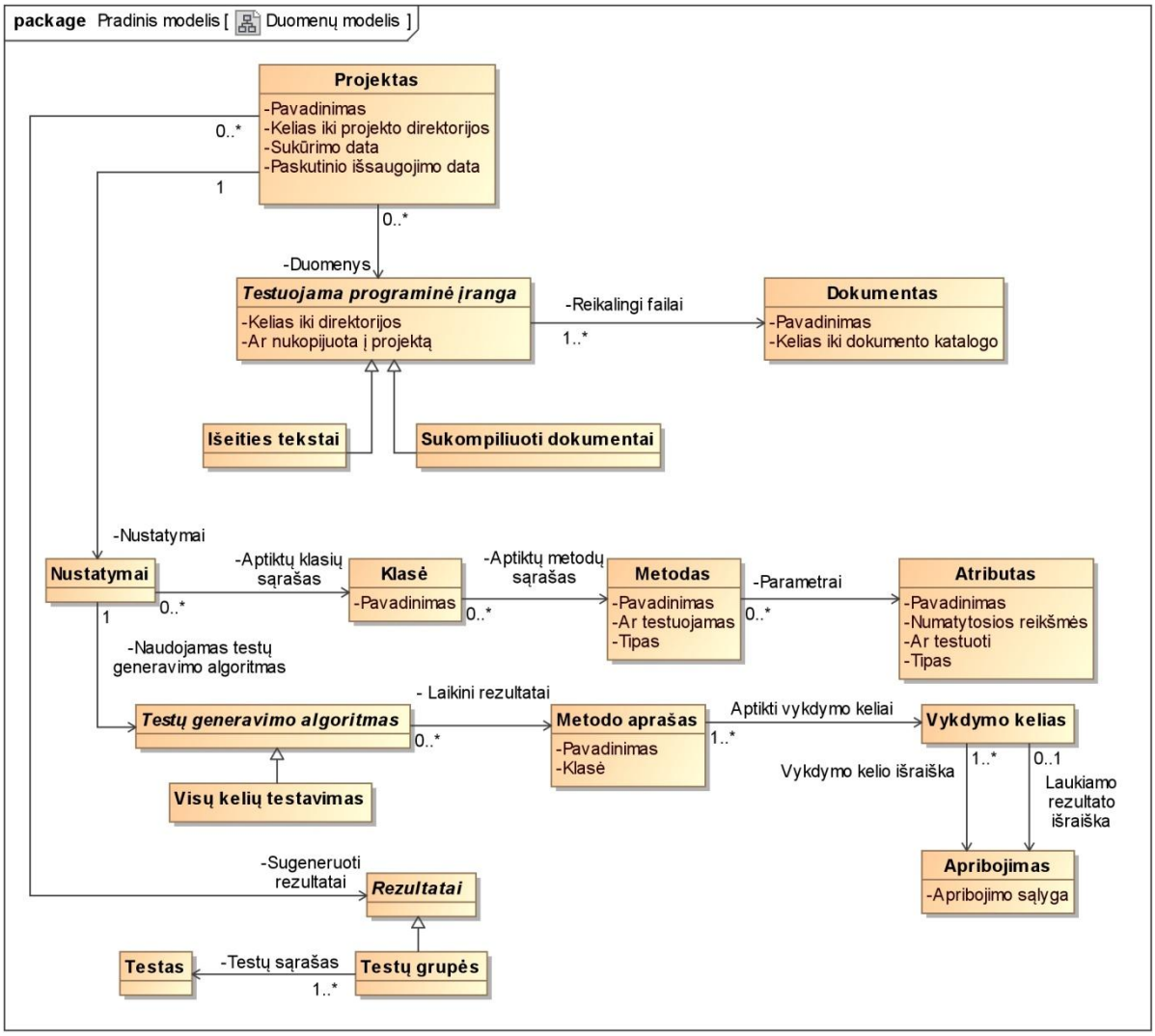
Iš viso programuojant sistemą buvo parašytos 7640 programinio kodo eilutės, kurių pasidalinimas po paketus yra pateiktas „Lentelė nr. 3 - Programinio kodo metrikos pagal paketus“ lentelėje. Taip pat šioje lentelėje yra pateikiamas klasių skaičius (KS) kiekviename pakete, metodų suma (MS), bibliotekų importavimo sakinių skaičius (BISS). Paketų metrikos yra skaičiuojamos įtraukiant ir sub-paketų metrikas.

Lentelė nr. 3 - Programinio kodo metrikos pagal paketus

Paketo pavadinimas	Kodo eilučių skaičius	KS	MS	BISS
jpfext	5231	66	440	335
jpfext.Project	321	7	19	30
jpfext.FileSystemInfo	468	8	34	29
jpfext.FileSystemInfo.Visitors	170	3	9	15
jpfext.CodeInfo	1055	12	83	44
jpfext.CodeInfo.Visitors	311	4	23	30
jpfext.TestsGenerators	2255	24	89	213
jpfext.TestsGenerators.Mutants	831	10	41	59
jpfext.TestsGenerators.jUnit	376	4	10	37
jpfext.Results	1081	14	106	16
jpfext.Results.Temporary	202	4	26	4
jpfext_Manager	2409	17	97	129
jpfext_Manager.Components	1875	14	75	107

3.4.3. Duomenų vaizdas

Sistemos naudojamus duomenis ir pateikiamus rezultatus galima suskirstyti į tam tikras grupes. Tai buvo padaryta sudarant duomenų vaizdo UML diagramą, kuri yra pateikta „Pav. 15 - Sistemos duomenų vaizdas“ paveikslėlyje. Išanalizavus šią diagramą galima lengviau suprasti duomenų struktūras naudojamas sistemoje. Taip pagal šią diagramą būtų galima kurti kitas automatizuotas testų saugojimo ar duomenų pateikimo sistemas.

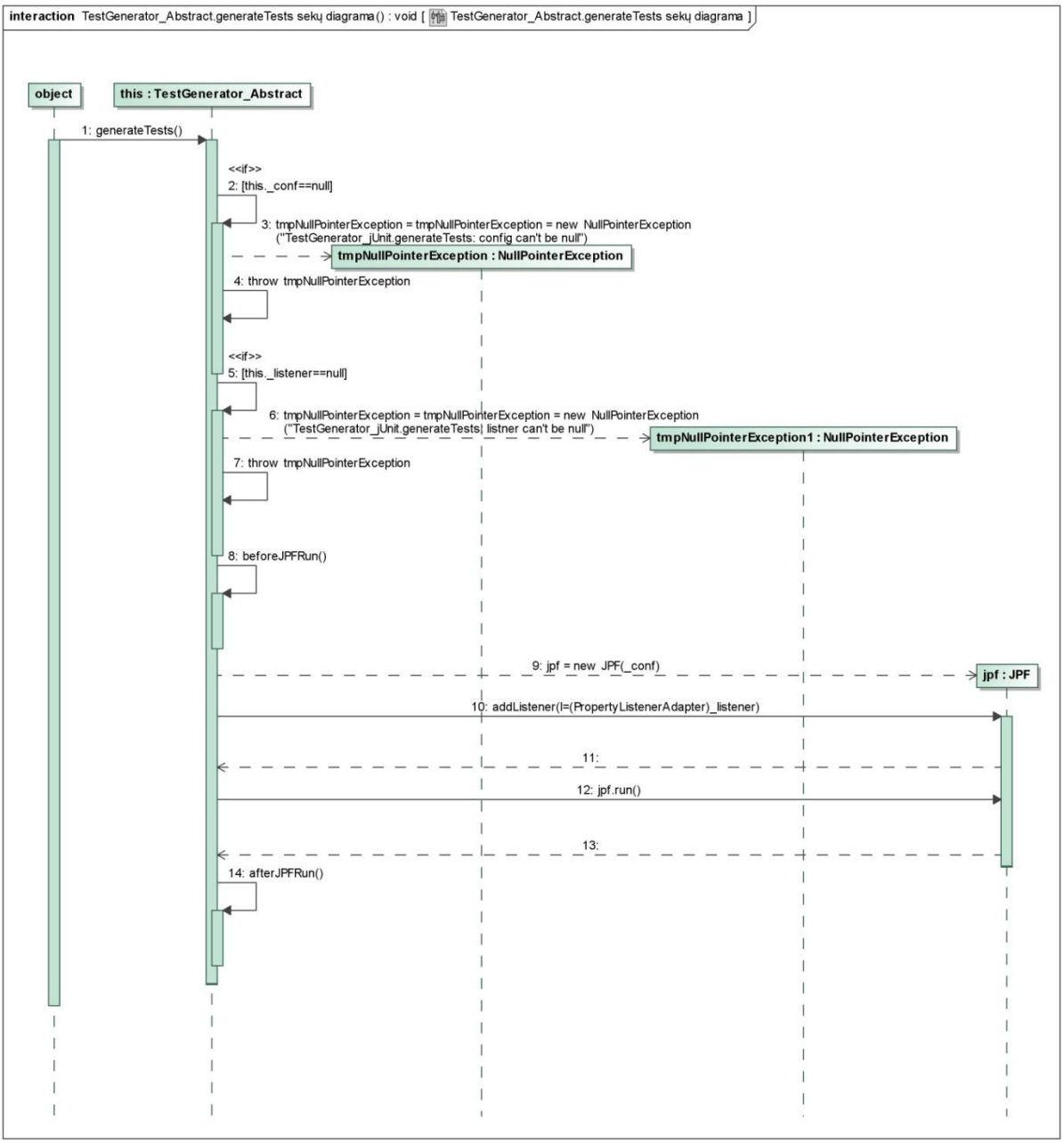


Pav. 15 - Sistemos duomenų vaizdas

3.4.4. Dinaminis sistemos vaizdas

Šioje dalyje yra pateikiamos esminės sistemos objektų sekų diagramos, kurios detaliai paaiškina testų sudarymo eigą. Sudarant testus visada yra kviečiamas generateTests() metodas realizuotas TestGenerator_Abstract klasėje šio metodo sekų diagrama yra pateikta „Pav. 16 -

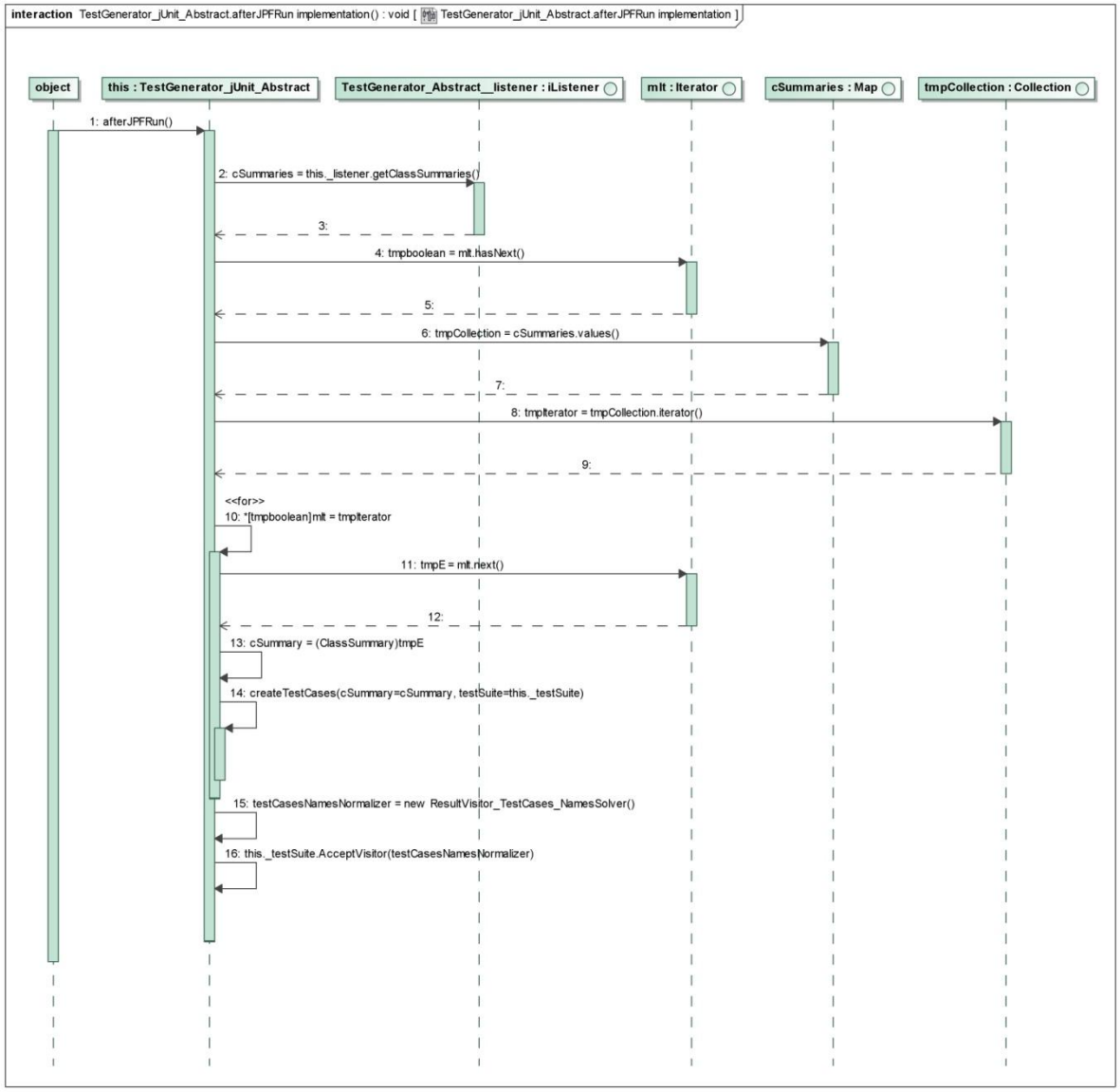
TestGenerator_Abstract klasės generateTests() metodo vykdymo sekų diagrama“ paveikslėlyje. Čia matyti: patikrinimai ar visi reikalingi duomenys yra pateikti; būsenų modelio sudarymo JPF įrankyje iškvietimas.



Pav. 16 - TestGenerator_Abstract klasės generateTests() metodo vykdymo sekų diagrama

Įvykdžius testuojamos sistemos būsenų modelio sudarymą jis yra gaunamas Listener_Symbolic klasės pagalba ir toliau apdorojamas TestGenerator_jUnit_Abstract klasės afterJPFRun() metode. Šio metodo veikimas yra pateiktas „Pav. 17 - TestGenerator_jUnit_Abstract klasės afterJPFRun() metodo vykdymo sekų diagrama“ paveikslėlyje. Čia yra iteruojama per visus metodus, kuriems buvo sudarytas būsenų modelis ir kviečiamas testų sudarymo metodas. Galiausiai sudarius testus jų pavadinimai yra

suformuojami ResultVisitor_TestCases_NamesSolver klasės pagalba taip, kad būtų kuo informatyvesni.



Pav. 17 - TestGenerator_jUnit_Abstract klasės afterJPFRun() metodo vykdymo sekų diagrama

Testų sudarymo metodas createTestCases() klasėje TestGenerator_jUnit_Abstract yra labai paprastas, todėl vykdymo sekų diagrama nebus pateikta. Tačiau galima paminėti, jog šis metodas yra skirtas testų sudarymui konkrečiam testuojamam metodui. Iš pradžių yra kviečiamas testų generavimas kiekvienam originaliam vykdymo keliui testuojamame metode, o vėliau vykdomas kelių mutavimas (metodas mutate()) ir testų generavimas iš mutuotų vykdymo kelių.

3.4.5. Apribojimai sistemai

Dėl sistemoje naudojamų trečių šalių simbolinio vykdymo ir apribojimų sprendimo įrankių galimybių apribojimų, pasiūlytas testų generavimo metodas negali būti pritaikytas visų realių sistemų testavimui. Visų pirma šitam sprendime eina apribojimai iš simbolinio vykdymo:

1) Kol kas nėra tokio sprendimo, kuris pilnai įgyvendintų visus simbolinio vykdymo principus. Dabar yra naudojamas JPF-symbc, kuris dar tik plėtojamas ir šiuo metu iš esmės palaiko tik skaitinius kintamųjų tipus. Taip pat jame yra daugybė tokių apribojimų kaip: begalinis būsenų skaičius susidarantis iš for ciklo; rekursijos nepalaikymas; kelių kintamųjų tipų toje pačioje programos vykdymo šakoje nepalaikymas; masyvų nepalaikymas ir pan. Tačiau šitame darbe nėra stengiamasi išspręsti simbolinio vykdymo problemas. Čia labiau koncentruojamės į simbolinio vykdymo grąžinamų būsenų sprendimą, o ne jų sudarymą.
2) Dalis apribojimų atsiranda dėl naudojamo apribojimų sprendimo varikliuko (angl. library for constraint satisfaction problems (CSP) and constraint programming (CP)). Realizuotoje sistemoje yra naudojama ChocoSolver biblioteka (varikliukas). Šioje bibliotekoje nėra sprendžiami tokie apribojimai, kuriuose yra perstūmimo ">>" operatoriai, taip pat jei yra dalybos veiksmas iš kintamojo, o ne konkrečios reikšmės ir kt. Taip pat rėžiai kuriuose gali būti kintamieji yra nuo -1 000 000 iki 1 000 000 - kitu atveju yra tiesiog išmetama klaida, kad tokio apribojimo nėra įmanoma išspręsti (pvz. apribojimas x < - 2 000 000 iškvies klaidos pranešimą). Šios problemos šiame darbe taip pat nėra sprendžiamos.

3.5. Projektavimo etapo rezultatai

1. Vykdant projektavimo etapą buvo stengiamasi kuo labiau įsigilinti į testų generavimo problemą. Iš pradžių buvo išskirti ir aprašyti kuriamos sistemos galimi vartotojai bei panaudos atvejai. Vėliau šiems panaudos atvejams buvo surinkti, surūšiuoti ir apibendrinti funkciniai bei nefunkciniai reikalavimai.
2. Surinkus reikalavimus sistemai ir žinant kokių rezultatų yra tikimasi iš jos buvo pradėta ieškoti galimo sprendimo, kuris leistų automatiškai sugeneruoti testinius atvejus atsparius mutantams. Šio etapo metu buvo gilinamasi į mutacinio testavimo apribojimus (problemas) bei analizuojamas simbolinio vykdymo metodikos panaudojimas generuojant testinius atvejus. Po ilgų bandymų ir turimos informacijos

analizės, padedant KTU informatikos fakulteto studentams ir darbuotojams, buvo sukurtas naujas (patobulintas senasis) testų generavimo metodas. Šis metodas ne tik leidžia atsisakyti įprastinio mutacinio testavimo etapų, bet tuo pačiu užtikrina, jog sugeneruoti testiniai atvejai padengs iki 100% testuojamos programinės įrangos vykdymo kelių ir bus atsparūs apibrėžtai aibei mutantų.

3. Sukūrus testų generavimo metodą, jis buvo detaliai aprašytas ir realizuotas sistemoje. Šis metodas įtakojo visą sistemos architektūrą nuo naudojamų trečiųjų šalių įrankių iki realizuotų klasių ir ryšių tarp klasių. Nepaisant, jog visas testų generavimo algoritmas turi aukštą sudėtingumo laipsnį (vykdomos kelių lygių iteracijos), sistemos architektūrą pavyko gan detaliai struktūrizuoti. Buvo panaudoti gerai žinomi projektavimo šablonai, kurie tūrėtų palengvinti sukurtos sistemos priežiūros procesą.
4. Aprašant sistemos architektūrą buvo naudojamos UML diagramos, kurios leidžia aprašyti ne tik statinį sistemos vaizdą, bet ir dinaminį (sistemos elgesį). Taip buvo dar labiau įsigilinta į sprendžiamą problemą ir daugelis sistemos kūrimo rizikų buvo pašalintos, arba bent identifikuotos, ankstyvosiose sistemos kūrimo stadijose.
5. Pati sistema buvo kuriama prototipų pagalba. Tai sutrumpino sistemos kūrimo laiką, nes sistemos projektavimo, programavimo bei testavimo darbai buvo atliekami iteratyviai – iškylančios problemos buvo sprendžiamos tuoj pat – kol dar neįsipynė į visą sistemą.

4. TYRIMO DALIS

Pirmojoje tyrimo dalyje yra analizuojama sukurtos sistemos kokybė pasinaudojant kodo statistika. Įvertinamos tokios metrikos kaip ciklomatinis sudėtingumas ir metodų rišlumo stoka, kurios parodo koreguotinas sistemos vietas. Antrojoje tyrimo dalyje yra aprašoma konkreti problema, kuri buvo pastebėta vykdant testų generavimą testuojamoms programoms – laiko sąnaudos buvo netikėtai didelės. Išanalizuojamos šios problemos priežastys, pasiūlomas galimas sprendimas, jis realizuojamas ir apibendrinami pastebėti pasikeitimai.

4.1. Sistemos išeities kodo analizė

Vykdant sistemos išeities kodo analizę buvo tiriami visi sistemos paketai. Tam atlikti buvo panaudotas automatizuotas Netbeans IDE [9] įskiepis Simple Code Metrics [10]. Šis produktas atlieka matavimus Java platformos projektuose ir gali išmatuoti tokias metrikas, kaip:

- Loginis kodo eilučių kiekis (LOC) – tai sistemos dydžio metrika parodanti programinės įrangos dydį kodo eilutėmis. Skirtingai nuo įprasto kodo eilučių kiekio, loginis kodo eilučių kiekis parodo sekų taškų kiekį, o ne fizinį kodo eilučių kiekį. Šis įvertis yra tikslesnis, nes yra nepriklausomas nuo programavimo stiliaus.
- Klasių kiekis (KK) – tyrime yra skaičiuojamas klasių kiekis kiekviename pakete įtraukiant ir vaikinius paketus. Klasių kiekio metrika tai viena iš objektinio sudėtingumo metrikų. Pagal šias metrikas yra vertinamas bendras sistemos sudėtingumas.
- Metodų kiekis (MK) – tyrime yra skaičiuojamas metodų kiekis kiekvienoje klasėje. Didelis metodų skaičius vienoje klasėje gali rodyti, jog klasė atlieka ne vieną funkciją, todėl yra didesnė tikimybė, jog ji ateityje turės keistis – kas pasunkina bendrą sistemos prižiūrimumą. Klases su dideliu metodų skaičiumi yra rekomenduojama skaldyti.

[9] http://netbeans.org/
[10] http://plugins.netbeans.org/plugin/9494/simple-code-metrics

- Ciklomatinis sudėtingumas (CC) – tai kiekvieno metodo atskirai vykdymo kelių skaičius. Kuo daugiau vykdymo kelių yra metode tuo daugiau testinių atvejų reikia jam patikrinti – taip sudėtingėja ne tik sistemos kūrimas, bet ir jos priežiūra. Yra rekomenduojama, jog ciklomatinis sudėtingumas metodui nebūtų didesnis nei 20, o didesnius metodus sudalinti į mažesnius – lengviau patikrinamus.
- Metodų rišlumo stoka (LCOM) – įvertina metodų nesusietumą klasėje. Šios metrikos reikšmė svyruoja nuo 0 iki 1, kur: 0 – reiškia, jog vertinamos klasės metodai yra labai glaudžiai susiję tarpusavyje (kviečia vieni kitus ir naudoja bendrus klasės atributus); 1 – jog metodai yra visiškai nesusiję. Pastarasis variantas yra vertinamas kaip problematiškas, nes nesant susietumui tarp metodų (ar atributų) yra didesnė tikimybė, jog dalis metodų yra realizuoti netinkamoje klasėje.
- Eilutės su bibliotekų importavimo sakiniais (BI) - ši metrika parodo kiek sudėtinga yra klasė (kiek ji naudoja kitų klasių). Kuo didesnis naudojamų bibliotekų skaičius tuo didesnė klaidų atsiradimo ir klasės keitimosi tikimybė.

4.1.1. Paketų metrikos

Lentelėje „Lentelė nr. 4 - Sistemos paketų metrikos“ yra pateikiamos pagrindinės paketų metrikos. Šioje lentelėje skaičiuojant paketų susidedančių iš mažesnių paketų statistiką yra įtraukiami duomenys ir iš vaikinių paketų. *LCOM vid.* - tai visų klasių vidutinė metodų rišlumo stoka.

Lentelė nr. 4 - Sistemos paketų metrikos

Paketo pavadinimas	LOC	KK	MK	LCOM Vid.	BI
jpfext	5231	66	440	0,414879	335
jpfext.Project	321	7	19	0,557143	30
jpfext.FileSystemInfo	468	8	34	0,403571	29
jpfext.FileSystemInfo.Visitors	170	3	9	0,5375	15
jpfext.CodeInfo	1055	12	83	0,60339	44
jpfext.CodeInfo.Visitors	311	4	23	0,5	30
jpfext.TestsGenerators	2255	24	89	0,286524	213
jpfext.TestsGenerators.Mutants	831	10	41	0,270238	59
jpfext.TestsGenerators.jUnit	376	4	10	0,21875	37
jpfext.Results	1081	14	106	0,314149	16
jpfext.Results.Temporary	202	4	26	0,441468	4
jpfext_Manager	2409	17	97	0,394362	129
jpfext_Manager.Components	1875	14	75	0,416667	107
Vidutiniškai:	**1276**	**14,4**	**81**	**0,412203**	**81**

Apibendrinus „Lentelė nr. 4 - Sistemos paketų metrikos“ lentelėje pateiktus duomenis galima pastebėti, jog:

- Sistemoje yra realizuotos 83 klasės (68 klasės,15 interfeisų), o visa sistema realizuota 7640 programinio kodo eilutėmis. Vidutiniškai (atmetus interfeisus) vienai klasei tenka truputi daugiau nei 100 programinio kodo eilučių. Tai rodo, jog visa sistema yra gan smulkiai struktūrizuota ir skubinai atlikti, jos priežiūros darbų (sudalinimo į mažesnes dalis) nereikia.
- Vienai klasei vidutiniškai tenka 6,5 metodo – tai taip pat rodo aukštą sistemos struktūrizavimo laipsnį. Daugiausiai metodų vienai klasei tenka `jpfext.Results` pakete – 7,6. Tačiau tai taip pat yra žemas metodų skaičius klasėje.
- Nei viename pakete nėra fiksuojama aukšta metodų rišlumo stoka. Todėl apibendrinus galima pasakyti, jog sistema yra gan smulkiai sudalinta į paketus bei klases, tačiau visi metodai yra gan rišliai tarpusavyje susiję. Tai leidžia teigti, jog sistema yra gan gerai struktūrizuota ir vykdyti, jos palaikymo darbus neturėtų būti sudėtinga.

4.1.2. Klasių metrikos

Analizuojant klases buvo paimta 10 klasių, kurios turi daugiausiai metodų – tai klasės, kurios atlieka ne vieną funkciją ir turi didžiausią tikimybę keistis ateityje. Rezultatai pateikti „Lentelė nr. 5 - Klasių metrikos“ lentelėje.

Lentelė nr. 5 - Klasių metrikos

Klasės pavadinimas	MK	LOC	LCOM	BI
jpfext.Results.TestCase_jUnit	27	270	0,848148	0
jpfext.Results .TestSuite_Abstract	26	236	0,834066	5
jpfext_Manager .NewJFrame	19	484	0,770813	17
jpfext_Manager.Components.ProjectModel	15	454	0,766667	18
jpfext.Results.Temporary.MethodSummary	14	91	0,888095	2
jpfext.CodeInfo.MethodInfo	13	119	0,842308	4
jpfext.FileSystemInfo.FileInfo_Catalog	11	101	0,822	3
jpfext.CodeInfo.FieldInfo	11	90	0,809091	1
jpfext.CodeInfo.Visitors.CodeInfoVisitor_ClassLoader	10	144	0,8	10
jpfext.TestsGenerators.Mutants.Mutant_Operator_Abstract	10	140	0,75	12

Išanalizavus pateiktas metrikas buvo priimtas sprendimas peržiūrėti klases, kurių vidutinė metodų rišlumo stoka (LCOM) viršija 0,8. Peržiūrėjus šias klases paaiškėjo, jog jos priklauso rezultatų `jpfext.Results` ir informacijos apie programinį kodą `jpfext.CodeInfo` saugojimo paketams. Tai paketai, kuriuose realizuotos klasės yra skirtos informacijos saugojimui, pvz.: jpfext.Results.Temporary.MethodSummary klasė su vidutine metodų rišlumo stoka 0,888095 yra skirta saugoti informaciją apie testuotus metodus – jų pavadinimą, klasės pavadinimą, atributų seką ir pavadinimus, vykdymo kelius. Visos šios klasės pasižymi tuo, jog jų atributai tarpusavyje yra nelabai susiję (pvz. metodo pavadinimas, yra visiškai nepriklausomas nuo jo klasės pavadinimo), dėl to ir šiose klasėse realizuoti metodai yra tarpusavyje nesusiję. Dėl šios priežasties buvo nuspręsta nekreipti dėmesio į aukštą metodų rišlumo stokos rodiklį ir šių klasių nekoreguoti, nes tokie darbai būtų tik apsunkinę (padarę sudėtingesne) visą sistemą. Dėl šių klasių keitimo turėtų keistis ir metodų analizės (būsenų modelio), ir testų generavimo algoritmai.

4.1.3. Metodų metrikos

Išanalizavus paketus ir jas sudarančias klases buvo nuspręsta dar ištirti pavienių metodų veikimą. Šio etapo metu buvo skaičiuojama prieš tai neskaičiuota metrika – ciklomatinis sudėtingumas. 20 metodų su aukščiausiu ciklomatiniu sudėtingumu yra pateikiami „Lentelė nr. 6 - Metodų ciklomatinis sudėtingumas“ lentelėje.

Lentelė nr. 6 - Metodų ciklomatinis sudėtingumas

Metodo klasė ir pavadinimas	Metodo ciklomatinis sudėtingumas
Listener_Symbolic::instructionExecuted	33
ProjectModel::getChild	26
ProjectModel::getIndexOfChild	25
ProjectTree::convertValueToText	23
MutantUtil::MutateExpression	21
MutantUtil::MutateConstraint	20
ProjectModel::getChildCount	20
TCGUtil::createTestCase	18
MutantUtil::MutateConstraint	18
ProjectModel::isLeaf	17
ResultVisitor_jUnitContentBuilder::visit	10
TCG_aaaa::visit	10
CodeInfoVisitor_ConfigBuilder::visit	9
NewJFrame::projectTreeMouseClicked	9
ResultVisitor_jUnitDescriptionBuilder::visit	8
NewJFrame::findTab	8

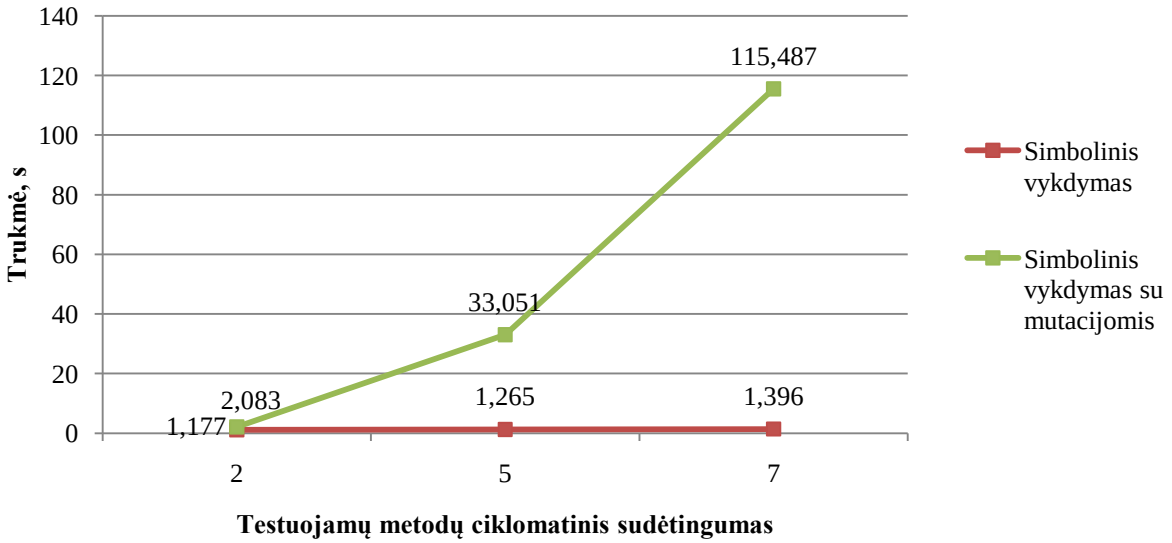
FileInfoVisitor_JavaFileExplorer::exploreCode	7
Project::generateNewTests	7
CodeInfoUtil::seachPackage	7
CodeInfoVisitor_ClassLoader::isTheSame	7

Įvertinus pateiktus rezultatus buvo nuspręsta išanalizuoti metodus, kurių ciklomatinis sudėtingumas yra didesnis ar lygus 20. Peržiūrėjus visus metodus buvo pastebėta bendra tendencija – visuose juose aptikta daug duomenų tipo tikrinimo salygų (pvz. `if(obj instanceof Project) { /* ... */ } else if(obj instanceof ProjectData) { /* ... */ }`). Būtent šie sakiniai ir sąlygojo visų metodų didelį vykdymo kelių skaičių (ciklomatinį sudėtingumą). Buvo nuspręsta, jog šią problemą galima nesudėtingai išspręsti pasinaudojant vizitatoriaus projektavimo šablonu.

Vizitatoriaus projektavimo šablonas buvo pritaikytas visiems metodams, kurių ciklomatinis sudėtingumas buvo didesnis nei 20 išskyrus `Listener_Symbolic::instructionExecuted`. Pastarojo metodo buvo nuspręsta nemodifikuoti, dėl to, kad jis suprogramuotas pagal Java PathFinder įrankio naudojimo šabloną. Šis šablonas laikas nuo laiko keičiasi ir norint lengviau atsekti pasikeitimus yra rekomenduojama jį palikti toks koks yra. Modifikavus metodus buvo gauti papildomi 3 interfeisai ir 6 klasės turinčios po 5 – 8 metodus, kurių ciklomatinis sudėtingumas neviršija 5.

4.2. Sistemos veikimo analizė

Antrasis tyrimų etapas prasidėjo įvykdžius pirmuosius eksperimentus aprašytus eksperimentinėje dalyje. Buvo pastebėta, jog vykdant testų generavimą yra sugaištama netikėtai daug laiko: priklausomai nuo testuojamų metodų ciklomatinio sudėtingumo testų generavimas truko nuo 2,083 sekundžių iki beveik 2 minučių. Tuo tarpu kuriant sistemos prototipą ir vykdant tik simbolį vykdymą, be mutacijų įvedimo testų generavimas truko 1,177 – 1,396 s. Diagrama iliustruojanti testų generavimo vykdymo laikus yra pateikta „Pav. 18 - Testų generavimo trukmė prieš sistemos patobulinimą“ paveikslėlyje.



Pav. 18 - Testų generavimo trukmė prieš sistemos patobulinimą

Buvo išsiaiškinta, jog daugiausiai laiko yra sugaištama sprendžiant mutuotas kelių išraiškas. Taip pat buvo atkreiptas dėmesys, jog esant neišsprendžiamoms kelių sąlygoms yra išmetama `TimeOutException` tipo klaida. Šią klaidą metė ChocoSolver apribojimų sprendimo įrankis.

Neišsprendžiamos vykdymo kelių sąlygos susidaro tada kai yra pritaikomi netikslingi mutantai, pvz.: kai kelio sąlyga `b>0` yra mutuojama į `b>=0` ir priešinga pastarajai sąlyga yra prijungiama prie pradinės, gaunama `[b>0 && b<0]` mutuoto vykdymo kelio išraiška, kurios neįmanoma išspręsti. Taip buvo prieita išvados, jog ne visos mutacijos yra tikslingos ir atlikta detalesnė analizė, kurios metu išsiaiškinta kokios mutacijos turi būti atliekamos.

4.3. Palyginimo operacijų mutavimo optimizavimas

Analizuojant, kokias palyginimo operacijas yra tikslinga mutuoti buvo paeiliui sprendžiami šie klausimai:

1. Ar pradinis apribojimas ir mutuoto apribojimo priešinga sąlyga turi bendrą sprendinį. Pvz.: iš `[b>0]` apribojimo galima sudaryti kelis mutantus `[b>=0]`, `[b!=0]`, `[b==0]`, `[b<0]` ir `[b<=0]`. Šiems mutantams priešingos sąlygos atitinkamai bus: `[b<0]`, `[b==0]`, `[b!=0]`, `[b>=0]` ir `[b>0]`. Tačiau kiekvieną mutanto priešingą sąlygą prijungus prie pradinės ir gavus mutuotus vykdymo kelius ne visi jie gali būti išspręsti:
    - Neišsprendžiami keliai: `[b>0 && b<0]` ir `[b>0 && b==0]`.

- Išsprendžiami keliai: `[b>0 && b!=0]`, `[b>0 && b>=0]` ir `[b>0 && b>0]`.

Taip buvo sudaryta visa lentelė „Lentelė nr. 7 - Ar pradinė sąlyga ir mutanto priešinga sąlyga turi bendrą sprendinį?“ parodanti ar pradinis apribojimas ir mutanto priešinga sąlyga turi bendrą sprendimą („+” – turi, „-“ – ne).

Lentelė nr. 7 - Ar pradinė sąlyga ir mutanto priešinga sąlyga turi bendrą sprendinį?

	Mutantas	==	!=	<	<=	>	>=
	Priešinga m. op.	!=	==	>=	>	<=	<
Pradinė op.	==	-	+	+	-	+	-
	!=	+	-	+	+	+	+
	<	+	-	-	-	+	+
	<=	+	+	+	-	+	+
	>	+	-	+	+	-	-
	>=	+	+	+	+	+	-

2. Ar mutanto priešinga sąlyga sumažina sprendinių aibę? Pastebėta, jog mutuojant pradinį apribojimą ir suradus mutantui priešingą apribojimą kartais yra gaunamas tas pats pradinis apribojimas. Pvz. mutavus `[b>0]` į `[b<=0]` ir suradus mutantui priešingą apribojimą yra gaunamas `[b>0]` apribojimas identiškas pradiniam. Buvo nuspręsta atsisakyti tokių mutacijų ir papildyta mutacijų lentelė „Lentelė nr. 8 - Ar mutanto priešinga sąlyga sumažina sprendinių aibę?“.

Lentelė nr. 8 - Ar mutanto priešinga sąlyga sumažina sprendinių aibę?

	Mutantas	==	!=	<	<=	>	>=
	Priešinga m. op.	!=	==	>=	>	<=	<
Pradinė op.	==	-	-	-	-	-	-
	!=	-	-	+	+	+	+
	<	-	-	-	-	-	-
	<=	+	+	+	-	-	+
	>	-	-	-	-	-	-
	>=	+	+	-	+	+	-

3. Buvo sudaryta lentelė „Lentelė nr. 9 - Pradinės operacijos ir mutanto priešingos sąlygos bendras sprendinys“, kurioje yra surašyti pradinių operacijų ir mutantų priešingų sąlygų bendri sprendiniai. Pvz. apribojimo `[b>=0]` ir mutanto `[b<=0]` priešingos sąlygos `[b>0]` bendras sprendinys yra `[b>0]`.

Lentelė nr. 9 - Pradinės operacijos ir mutanto priešingos sąlygos bendras sprendinys

Pradinė op.	Mutantas	==	!=	<	<=	>	>=
	Priešinga m. op.	!=	==	>=	>	<=	<
==		-	-	-	-	-	-
!=		-	-	>	>	<	<
<		-	-	-	-	-	-
<=		<	==	==	-	-	<
>		-	-	-	-	-	-
>=		>	==	-	>	==	-

4. Pastebėta, jog net jei ir pradinė operacija bei jos mutanto priešinga sąlyga turi bendrą sprendinį, tai dar nereiškia, kad mutanto atvirkštinė sąlyga sumažina pradinės sąlygos sprendinių aibę. Pvz.: apribojimo `[b>=0]` ir mutanto `[b<=0]` priešingos sąlygos `[b>0]` bendras sprendinys yra `[b>0]` – t.y. `[b>0]` apribojimas sumažina pradinio apribojimo `[b>=0]` galimų sprendinių aibę iki `[b>0]`. Tačiau jei pradinis apribojimas yra `[b>0]`, jo sprendinių aibė negali būti sumažinta įvedant mutantus ir tokio apribojimo nereikia mutuoti. Taip buvo sudaryta „Lentelė nr. 10 - Mutanto atvikštinė sąlyga tik sumažina pradinės operacijos sprendinių aibę“ lentelė.

Lentelė nr. 10 - Mutanto atvikštinė sąlyga tik sumažina pradinės operacijos sprendinių aibę

Pradinė op.	Mutantas	==	!=	<	<=	>	>=
	Priešinga m. op.	!=	==	>=	>	<=	<
==		-	-	-	-	-	-
!=		-	-	-	>	-	<
<		-	-	-	-	-	-
<=		-	==	-	-	-	<
>		-	-	-	-	-	-
>=		-	==	-	>	-	-

Taip buvo sudarytos palyginimo operacijų mutavimo taisyklės aprašytos „Lentelė nr. 11 - Palyginimo operacijų mutavimo taisyklės“ lentelėje.

Lentelė nr. 11 - Palyginimo operacijų mutavimo taisyklės

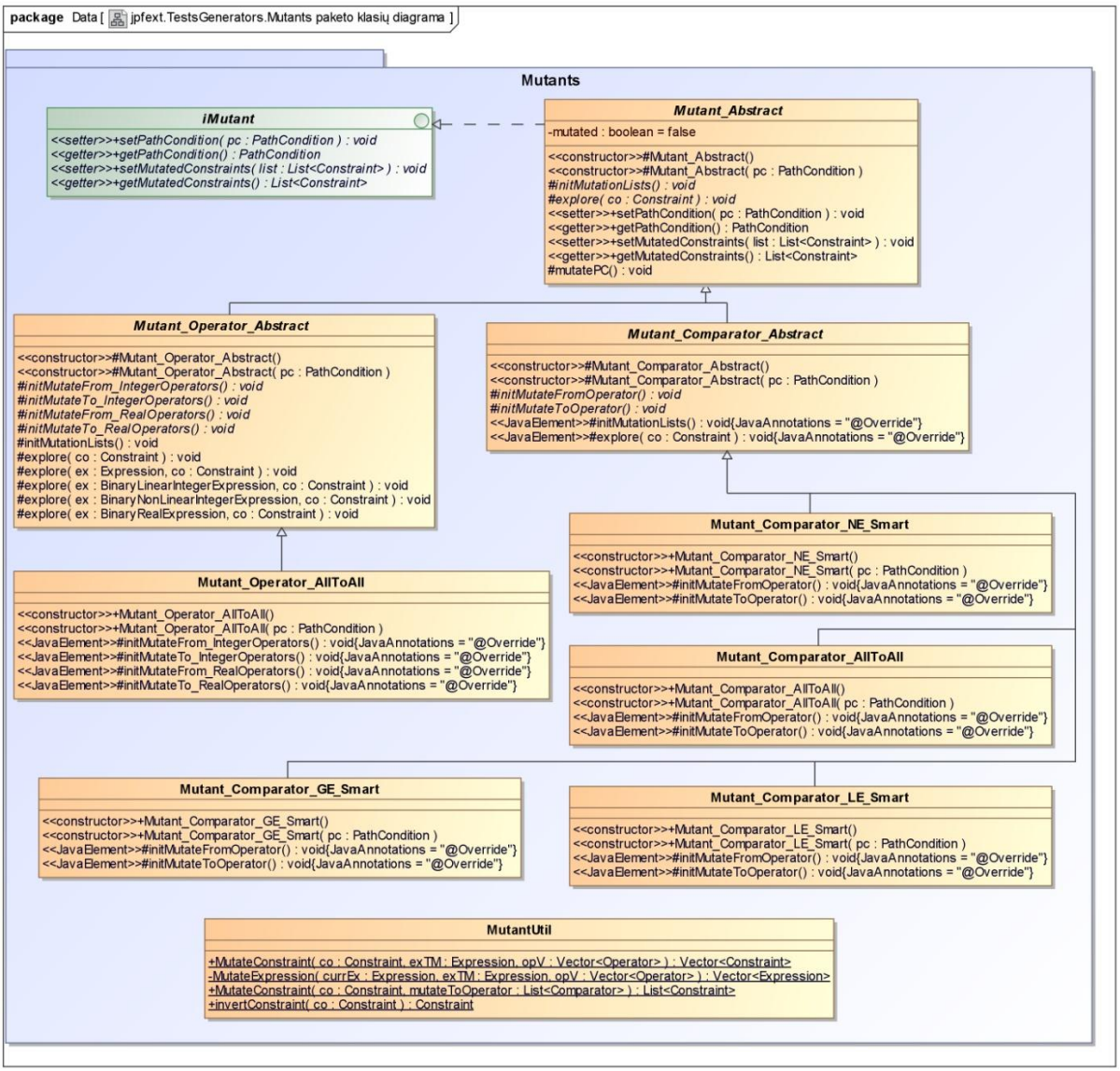
Palyginimo operatorius	Mutuojamas į
!=	<=
	>=
<=	!=
	>=
>=	!=
	<=

Gali pasirodyti, jog esant tam tikroms palyginimo operacijų mutacijoms (programiniame kode) jos negali būti aptinkamos, tačiau tai nėra tiesa. Reikia neužmiršti, jog

čia apibrėžtos taisyklės yra taikomos ne programiniam kodui o iš jo sudarytam būsenų modeliui. Šis modelis pasižymi tuo, jog iš kiekvienos palyginimo operacijos išeina dvi šakos (true ir false), kurios yra priešingos viena kitai. Dėl šios priežasties bent viena šaka bus mutuota ir taip bus aptinkamos visos palyginimo operacijų mutacijos.

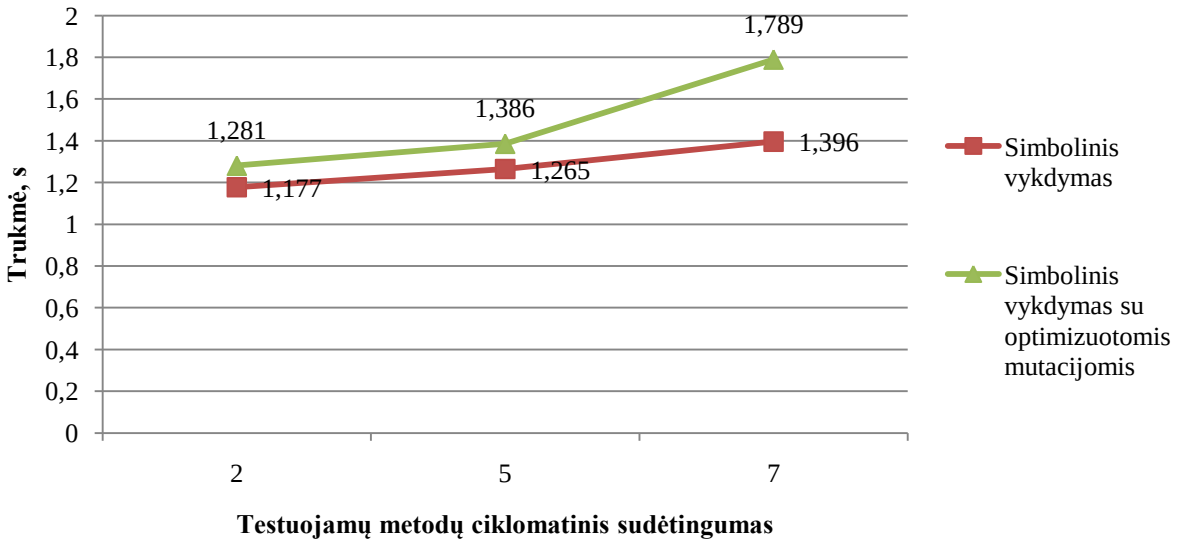
4.4. Palyginimo operacijų mutavimo optimizavimo realizavimas ir rezultatai

Palyginimo operacijų mutavimo optimizavimas buvo realizuotas gan nesudėtingai. Jau kuriant sistemą buvo numatyta jog gali prireikti keisti mutavimo taisykles, todėl užteko realizuoti tik tris naujas `Mutant_Comparator_GE_Smart`, `Mutant_Comparator_LE_Smart`, `Mutant_Comparator_NE_Smart` klases `jpfext.TestsGenerators.Mutants` pakete ir pamodifikuoti vieną klasę, kad naujas funkcionalumas būtų realizuotas.



Pav. 19 - Atnaujinto jpfext.TestsGenerators.Mutants paketo klasių diagrama

Atnaujinto paketo klasių diagrama yra pateikiama „Pav. 19 - Atnaujinto jpfext.TestsGenerators.Mutants paketo klasių diagrama“ paveikslėlyje. Dėl atnaujintų mutavimo taisyklių sutrumpėjo mutacijų įvedimo į modelį laikas, sumažėjo mutuotų kelių skaičius ir nebeliko ilgo testų generavimo laiko problemos. Testų generavimo laikų diagrama su optimizuotu palyginimo operacijų mutavimu yra pateikiama „Pav. 20 - Testų generavimo trukmės po sistemos patobulinimo“ paveikslėlyje.



Pav. 20 - Testų generavimo trukmės po sistemos patobulinimo

Sugeneruojami testiniai atvejai nepakito, o tai leidžia daryti išvadą, jog mutacijų taisyklės buvo apibrėžtos teisingai ir eksperimentiškai patvirtintos.

4.5. Tyrimo dalies rezultatai

1. Tyrimo etapo metu buvo įvertinta sistemos kokybė. Kokybės įvertinimui pasitelktos konkrečios metrikos: loginis kodo eilučių kiekis (LOC), klasių kiekis (KK), metodų kiekis (MK), ciklomatinis sudėtingumas (CC), metodų rišlumo stoka (LCOM), eilutės su bibliotekų importavimo sakiniais.
2. Apskaičiavus kiekvieną metriką buvo svarstoma priežiūros galimybės atnaujinant paketų struktūrą, realizuotas klases ir metodus. Tačiau buvo pasirinkta atnaujinti tik sudėtingiausius metodus, kurių ciklomatinis sudėtingumas buvo didesnis nei 20. Rezultate padaugėjo klasių, bet sistema tapo lengviau testuojama bei prižiūrima.
3. Antrojoje tyrimo dalyje buvo susidurta su konkrečia problema – ilga testų generavimo trukme. Ištyrus sistemos elgseną buvo išsiaiškinta, jog ši problema galėjo atsirasti dėl naudojamo apribojimų sprendimo įrankio ChocoSolver. Kadangi sistemos pritaikymas

kitam apribojimų sprendimų įrankiui reikalautų nemažai papildomo darbo ir nežinia ar rezultatai pagerėtų, buvo nuspręsta ieškoti kitų sprendimo būdų. Taip buvo atrasta, jog sistemoje neturi būti vykdomos visos galimos palyginimo operatorių mutacijos.

4. Buvo atlikta detalesnė palyginimo operacijų mutavimo taisyklių analizė ir jos metu sudarytos naujos palyginimo operacijų mutavimo taisyklės. Realizavus šias taisykles sistemoje buvo išspręsta ilgos testų generavimo trukmės problema. Taip pat buvo sumažinti ne tik reikalingi skaičiavimo (CPU), bet ir naudojamos atminties (RAM) resursai.
5. Tyrimo metu buvo įsitikinta, jog vykdant sistemos priežiūros darbus galioja II ir VI Lehmano dėsniai teigiantys:
    - II - Augančio sudėtingumo dėsnis. Sistemai evoliucionuojant jos sudėtingumas didėja, jei nėra atliekami jos priežiūros darbai. (Įvedus naujas mutavimo taisykles sistema pasudėtingėjo, o atlikus priežiūros darbus buvo sumažintas metodų ciklomatinis sudėtingumas)
    - VI - Nuolatinio augimo dėsnis. Sistemos funkcionalumas turi būti nuolat didinimas siekiant išlaikyti vartotojų pasitenkinimą produktu. (Naujos mutacijų taisyklės buvo įvestos būtent dėl to, kad buvo norima išlaikyti vartotojų pasitenkinimą)

5. EKSPERIMENTINĖ DALIS

Vykdant eksperimentus buvo išmėgintas sukurtas testų generavimo metodas ir jį realizuojanti sistema aprašyta projektinėje dalyje. Šis etapas susidėjo iš kelių dalių:

1. Buvo pasirinktos 5 eksperimentinės programos („`Test1`“, „`Test2`“, „`Test3`“, „`Test4`“, „`Test5`“), kurių testuojamų metodų (`testMe()`) programinis kodas yra pateikiamas prieduose.
2. Kiekvienai eksperimentinei programai buvo sukurta visa eilė mutantų. Buvo pritaikytos palyginimo operacijų ir aritmetinių operacijų mutavimo taisyklės.
3. Kiekvienai eksperimentinei programai (originaliai) buvo generuojami testai pagal:
    - Simboliniu vykdymu grindžiamą testų generavimo metodą.
    - Simboliniu vykdymu grindžiamo mutacinį testų generavimo metodą.

    Atsitiktinio testų generavimo panaudojant mutacinį testavimą buvo atsisakyta dėl to, jog šiuo metodu sugeneruotų testų kokybė priklauso nuo vykdomų iteracijų skaičiaus generuojant testus ir nėra galimybės pasakyti koks iteracijų skaičius yra pakankamas.

4. Sugeneruoti testai buvo vykdomi su mutuotomis programomis. Vėliau gauti rezultatai palyginti ir aprašyti.

5.1. Eksperimentų tikslai

- Įvertinti sugeneruotų testų efektyvumą aptinkant mutantus. Čia tikrinami testai, kurie buvo sugeneruoti pasinaudojant simboliniu vykdymu grindžiamo mutacinio testavimo įrankiu.
- Palyginti gautą efektyvumą su testų, sugeneruotų nenaudojant būsenų modelio mutacijų, efektyvumu.

5.2. Eksperimentinių programų metrikos

Lentelėje „Lentelė nr. 12 - Eksperimentinių programų kodo metrikos“ yra pateiktos pagrindinės eksperimentinių programų testuojamų metodų metrikos: kodo eilučių kiekis (LOC), ciklomatinis sudėtingumas (CC), aptiktų ir galinių būsenų skaičius, vykdomų Java ByteCode instrukcijų kiekis (JB), sugeneruotų mutantų skaičius (Mut).

Nors testuojami metodai yra nedideli ir neturi didelio eilučių kiekio, bet jie parinkti kuo įvairesni.

Lentelė nr. 12 - Eksperimentinių programų kodo metrikos

Programos pavadinimas	LOC	CC	Būsenos		JB	Mut
			Iš viso	Galinės		
Test1	9	3	6	3	31	7
Test2	3	1	10	5	46	10
Test3	6	2	10	5	61	43
Test4	20	7	29	10	141	63
Test5	15	6	12	6	53	29

5.3. Sugeneruotų testų metrikos

Lentelėje „Lentelė nr. 13 - Sugeneruotų testinių atvejų skaičius eksperimentinėms programoms“ yra pateikiami duomenys apie tai kiek testų kokiai programinei įrangai buvo sugeneruota pasinaudojant simboliniu vykdymu grindžiamo testų generavimo (SV) ir simboliniu vykdymu grindžiamo mutaciniu testų generavimo (SV+M) metodais.

Lentelė nr. 13 - Sugeneruotų testinių atvejų skaičius eksperimentinėms programoms

Programinė įranga, kuriai buvo generuojami testai	Test1	Test2	Test3	Test4	Test5
SV metodas	3	5	5	14	6
SV+M metodas	3	5	11	35	11

Galima pastebėti, jog nors „Test2“ ir „Test3“ programų vykdymo kelių (galinių būsenų) skaičius yra vienodas – sugeneruotų testų kiekis skiriasi. Taip nutinka, nes skiriasi testuojamų metodų parametrų tipai: boolean tipo parametrai gali įgyti tik dvi reikšmes – true arba false; sveikieji ar realieji skaičiai gali būti parinkti iš sąlygiškai begalinės reikšmių aibės.

Kiekvienas sugeneruotas testinis rinkinys buvo vykdomas su atitinkamos eksperimentinės programos mutantais. Leidžiant testinius rinkinius buvo matuojama, koks procentas testinių atvejų rinkinyje neaptinka mutuotos programos, t.y. praeina sėkmingai, neparodant klaidos pranešimo. Lentelėje „Lentelė nr. 14 - Sugeneruotų testinių atvejų metrikų palyginimas“ yra pateikiami apibendrinti gauti rezultatai (detalizuoti rezultatai yra pateikiami prieduose):

- Vidutinis testų efektyvumas (VTE) – tai procentas parodantis, kiek vidutiniškai testinių atvejų konkrečiame rinkinyje aptiko programinio kodo mutacijas.

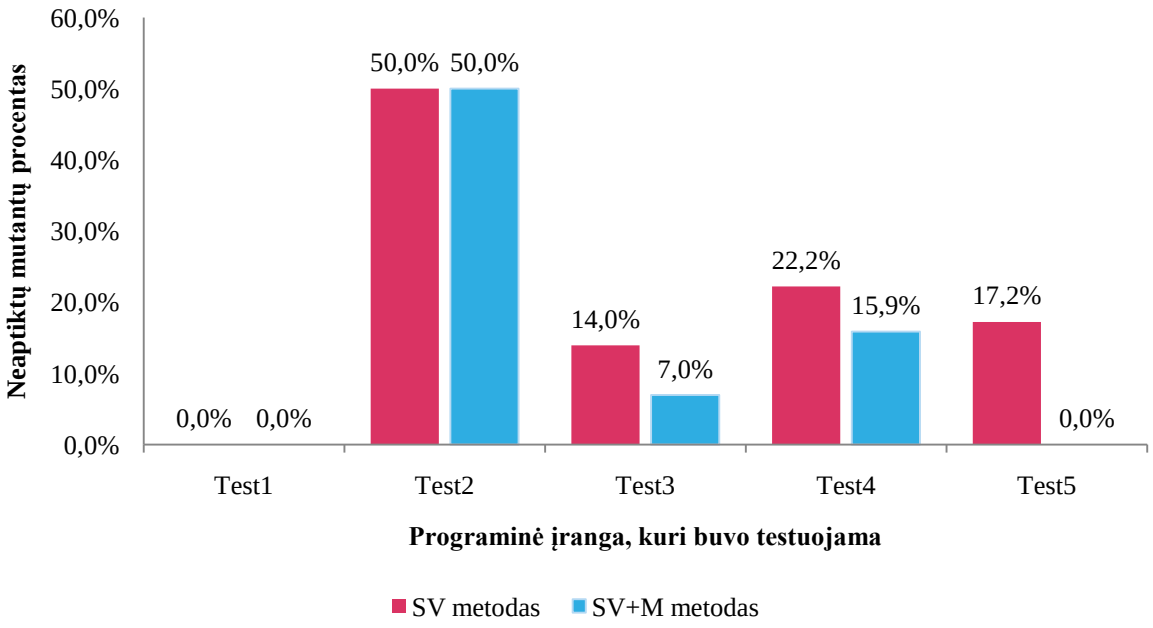
- Neaptiktų mutantų skaičius (NMS) – tai mutantų kiekis, kurio nepavyko aptikti pasinaudojant sugeneruotais testiniais rinkiniais.
- Neaptiktų mutantų procentas (NM%) – tai neaptiktų mutantų skaičiaus ir visų konkrečios PĮ (programinės įrangos) mutantų kiekio santykis padaugintas iš 100.

Lentelė nr. 14 - Sugeneruotų testinių atvejų metrikų palyginimas

PĮ, kuriai sugeneruoti testai / Testų generavimo metodas	Test1		Test2		Test3		Test4		Test5	
	SV	SV+M	SV	SV+M	SV	SV+M	SV	SV+M	SV	SV+M
VTE	47,6%	47,6%	16,0%	16,0%	30,2%	33,0%	15,4%	15,9%	20,1%	20,4%
NMS	0	0	5	5	6	3	14	10	5	0
NM%	0,0%	0,0%	50,0%	50,0%	14,0%	7,0%	22,2%	15,9%	17,2%	0,0%

Išanalizavus apibendrintus duomenis, galima pastebėti, jog nors vidutinis testų efektyvumas nėra didelis, bet daugiau nei pusė mutantų yra visada aptinkama sugeneruotais testų rinkiniais. Tai reiškia, jog galbūt tik vienas kitas testas iš viso testų rinkinio aptinka konkretų mutantą, tačiau didžioji dalis mutantų vis tiek yra aptinkama. Toks testų efektyvumas – sąlygoja teiginį, jog sugeneruoti testų rinkiniai negali būti sumažinti (negalima atsisakyti dalies testų dėl to, jog kitų testų efektyvumas yra mažas ir jie tiesiog neaptiks dalies mutantų) ir yra optimalūs norint aptikti PĮ mutacijas.

Kita metrika, kurią galima pastebėti pateiktuose rezultatuose (ir dėl, kurios buvo kuriamas naujas mutantų aptikimo metodas) – tai neaptiktų mutantų procento sumažėjimas. Kaip matyti iš pateiktų rezultatų, tam tikrais atvejais neaptiktų mutantų skaičius gali būti sumažintas iki 0. Tai reiškia, jog automatiškai sugeneravus testinius duomenis bus aptinkami visi PĮ mutantai. Grafinis šios metrikos atvaizdavimas yra pateiktas „Pav. 21 - Neaptiktų mutantų procentas“ paveikslėlyje.



Pav. 21 - Neaptiktų mutantų procentas

Nors stebimas neaptiktų mutantų procento sumažėjimas (t.y. generuojamų testų efektyvumo padidėjimas), naudojant simboliniu vykdymu grindžiamo mutacinį testų generavimo metodą, tačiau buvo tikėtasi, jog šis rodiklis bus dar žemesnis. Išanalizavus testavimo rezultatų lentelę pateiktą prieduose buvo apibrėžti mutantai, kurių nepavyko aptikti pasinaudojant sugeneruotais testais. Paaiškėjo, jog taip galėjo atsitikti dėl dviejų priežasčių:

1. Mutuojami yra tik vykdymo keliai, bet ne rezultatai. Atliekant eksperimentinių programų mutavimą buvo mutuojamas visas programinis kodas – neatsižvelgiant į tai ar mutuojama vykdomo kelio sąlyga ar grąžinamo rezultato reikšmė. Prie tam tikrų duomenų šios mutacijos nėra aptinkamos. Pvz.: jei rezultate yra grąžinama `a/b` išraiška ir `a` bei `b` reikšmės pasirenkamos lygios 1, tai tokie duomenys nesaugos nuo `a*b` mutanto. Šią problemą galima nesunkiai išspręsti – tereikia mutuoti grąžinamų rezultatų išraiškų aritmetinius operatorius ir lygybę pakeisti į nelygybę. Prijungus pastarąją nelygybę prie pradinės reikšmės bus gaunami rezultatai tenkinantys pradinę išraišką, bet netinkami mutuotajai. Pvz.: jei rezultatas yra apibrėžiamas lygybe `[rez=a/b]`, prie jos prijungus mutuotą nelygybę `[rez!=a*b]` bus gaunami rezultatai `[a=1]` bei `[b=2]` tenkinantys sąlygą `[rez=a/b && rez!=a*b]`.
2. Realiose programose dažnai pasitaiko vykdymo kelių, kuriuose yra grąžinamas toks pats rezultatas, kaip ir kituose keliuose. Ir nors duomenys būna

sugeneruoti tokie, kad vykdymo keliai pasikeistų esant mutacijoms – to neužtenka, kad būtų pateiktas kitoks rezultatas. Šiai problemai išspręsti yra siūlomi du variantai:

a. Stengtis sugeneruoti tokius duomenis, su kuriais būtų galima papulti į konkretų vykdymo kelią grąžinantį kitokį rezultatą.
b. Stengtis sugeneruoti tokius duomenis, su kuriais būtų galima išvengti vykdymo kelių grąžinančių tą patį rezultatą.

Abu šie variantai turi savų apribojimų ir norint juos detalizuoti bei įgyvendinti reikia įdėti daugiau darbo.

5.4. Eksperimentinės dalies rezultatai

1. Apibendrinant gautus rezultatus, reikia pastebėti, jog tikslas, nors ir ne visiškai toks kokio tikėtasi, bet buvo pasiektas. Vykdant eksperimentus yra stebimas 6 – 17 % padidėjęs mutantų aptikimas testais. Nors nėra aptinkami visi PĮ mutantai, bet yra pasiekiamas aukštas jų aptikimo laipsnis. Galima teigti, jog sukurtas testų aptinkančių mutantus metodas pasiteisino.
2. Didžiausias privalumas yra tas, jog buvo atsisakyta atsitiktinio parametrų reikšmių generavimo vykdant mutacinį testavimą – dabar šios reikšmės yra suskaičiuojamos, vietoj to, kad būtų spėliojamos. Be to yra gaunami optimizuoti testinių atvejų rinkiniai.
3. Testų generavimo trukmės įvedus būsenų modelio mutavimus neišaugo taip stipriai, kaip iš pradžių buvo tikėtasi. Čia labai didelę įtaką padarė palyginimo operacijų mutavimo taisyklių optimizavimas.

6. IŠVADOS

Darbe apibrėžti mutaciniu testavimu ir simboliniu vykdymu grindžiamų testų generavimo metodų trūkumai. Išanalizuota galimybė juos apjungti ir sukurtas bendras testų skirtų mutantų aptikimui generavimo metodas: simboliniu vykdymu grindžiamas mutacinio testų generavimo metodas.

Sukūrus testų generavimo metodą jis buvo detaliai aprašytas projektinėje dalyje ir realizuotas sistemoje. Vykdant sistemos tyrimo etapą buvo vertinama sistemos kokybė pasitelkiant konkrečias metrikas. Jos leido aptikti silpnas sistemos vietas ir jas pakoreguoti. Rezultate padaugėjo klasių, bet sistema tapo lengviau testuojama bei prižiūrima. Antrojoje tyrimo dalyje buvo optimizuotas palyginimo operacijų mutavimas, todėl testų generavimo laikas sutrumpėjo net kelis kartus.

Eksperimentų vykdymo metu buvo įsitikinta, jog sukurtas metodas pasiteisino ir yra stebimas 6 – 17 % padidėjęs mutantų aptikimas testais. Be to šio metodo pagalba buvo atsikratyta būtinybės kurti mutantus, atsitiktinai generuoti testinius duomenis bei išvengiama perteklinio testinių atvejų skaičiaus. Testiniai duomenys aptinkantys mutantus yra apskaičiuojami, o ne atsitiktinai spėliojami.

7. PADĖKOS ŽODIS

Naujo testų generavimo metodo sukūrimas pareikalavo nemažų pastangų. Visą kūrimo laikotarpį buvo susiduriama su daugybę problemų, kurias ne visada pavykdavo iš karto išspręsti. Toks darbas pareikalavo labai gerų žinių, išmanymo bei suvokimo ne vien tik informacinių technologijų srityje. Todėl norėčiau padėkoti visiems KTU darbuotojams, kurie prisidėjo prie šios sistemos sukūrimo. Ypač prof. Eduardui Bareišai, doktorantams Dominykui Barisui ir Tomui Neverdauskui už kantrybę, įdėtas pastangas, suteiktas žinias ir nuoširdų atsidavimą darbui. Ačiū!

8. LITERATŪRA

[1] Newman M., The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology, 2002. Prieiga per internetą: http://www.nist.gov/public_affairs/releases/n02-10.htm [žiūrėta: 2009.11.11]

[2] Swebok, Software Testing [interaktyvus]. IEEE computer society, 2004. Prieiga per internetą: http://www.computer.org/portal/web/swebok/html/ch5 [žiūrėta: 2009.11.03]

[3] Gelperin, D.; B. Hetzel. The Growth of Software Testing [interaktyvus]. Communications of the ACM (CACM), Volume 31, Number 6, June 1988. Prieiga per internetą: http://www.clearspecs.com/downloads/ClearSpecs16V01_GrowthOfSoftwareTest.pdf

[4] Dustin E., Garrett T., Gauf. B., Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality. Addison-Wesley Professional; 1 edition (March 14, 2009).

[5] OMG Unified Modeling Language (OMG UML) [interaktyvus], Superstructure, State Machines, Version 2.2. February 2009. Prieiga per internetą: http://www.omg.org/spec/UML/2.2/Superstructure/PDF/

[6] Tian J. Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement. John Wiley & Sons, Inc., Hoboken, New Jersey, 2005. p203-219.

[7] McConnell S. Code Complete: A Practical Handbook of Software Construction. Microsoft Press; 2nd edition, July 7, 2004. p463-477.

[8] Freeman H., Software Testing. IEEE Instrumentation & Measurement Magazine, September 2002. 48-50 p.

[9] Swebok, Software engineering tools and methods [interaktyvus]. IEEE computer society, 2004. Prieiga per internetą: http://www.computer.org/portal/web/swebok/html/ch10#Ref1.4

[10] Object Constraint Language[interaktyvus]. OMG, Version 2.2. February 2010. Prieiga per internetą: http://www.omg.org/spec/OCL/2.2/PDF/

[11] Packevičius Š., Ušaniov A., Bareiša E., Universal unit tests generator based on software models. // Information Technologies' 2009 : proceedings of the 15th

International Conference on Information and Software Technologies, IT 2009, Kaunas, Lithuania, April 23-24, 2009 / Kaunas University of Technology. Kaunas : Technologija. ISSN 2029-0020. 2009, p. 201-206.

[12] King J. C., „Symbolic Execution and Program Testing“ Commun. ACM, vol. 19, no. 7, pp.385-394, 1976.

[13] Prelgauskas J., Bareiša E., Symbolic Unit Testing Method Evaluation. Information Technologies’ 2010 : 16th International Conference on Information and Software Technologies, IT 2010, Kaunas, Lithuania, April 21-23, 2010 : research communications / Kaunas University p. 69-76.

9. TERMINŲ IR SANTRUMPŲ ŽODYNAS

BVP – bendras vidaus produktas.

DB – duomenų bazės.

Grafas – tai diagrama su viršūnėmis ir kraštinėmis.

JPF – (Java PathFinder) tai atviro kodo programinė įranga skirta Java programinės įrangos tikrinimui. Šis projektas yra sudarytas iš atskirų projektų, kurie yra (bus) naudojami (plečiami) kuriant šią sistemą.

JVM – (Java Virtual Machine) tai Javos virtuali mašina, kuri leidžia įvykdyti Java programavimo kalba parašytas programas skirtingose operacinėse sistemose.

OMG – The Object Management Group, tai organizacija kurianti standartus programų inžinerijos, vadybos ir kitose srityse.

OS – operacinė sistema.

PI – programų inžinerija.

PĮ – programinė įranga.

RAM – Random-access memory – kompiuterio darbinė atmintis.

Testavimas – procesas, kurio metu yra tikrinama programos kokybė.

Testas – tai programos kodas realizuojantis tam tikrą testinį atvejį.

Testinis atvejis – tai atsitiktinis sistemos būsenų perėjimas siekiant patikrinti programinės įrangos kokybę.

UML – Unified Modeling Language - Unifikuota modeliavimo kalba skirta programų sistemų projektavimui.

Virtuali mašina – tai programa leidžianti įvykdyti kitas, tam tikra kalba parašytas, programas. Virtualioje mašinoje veikiančios programos reikalauja daugiau sistemos resursų, tačiau tokias programas yra galima rašyti aukštesnėmis programavimo kalbomis.

XMI – XML Metadata Interchange – OMG organizacijos sukurtas standartas skirtas metaduomenų saugojimui ir apsikeitimui tarp programų.

XML – Extensible Markup Language – tai dokumentų standartas naudojamas duomenų saugojimui elektroniniame pavidale, kuris pasižymi tuo, jog naudoja žymas.

10. PRIEDAI

10.1. Publikacija

Pateikiamas magistrinio darbo metu paruoštas ir „Elektronika ir elektrotechnika.“ Kaunas: Technologija, 2011 Nr. 6 atspausdintas straipsnis:

Automated regression testing using symbolic execution

T. Milašius, D. Barisas, E. Bareiša

Software Engineering Department, Kaunas University of Technology
Studentų St. 50, LT−51368 Kaunas, Lithuania

1 Introduction

During development and support phases, software is modified to enhance its functionality, detect faults, and adapt it to different platforms. Regression testing is used to identify faults that were introduced when modifying code [1] or to assure that a change, for example a functional enhancement, bug fix, patches or configuration changes, did not introduce new faults. However, a lack of test inputs that exercise a changer behavior is a common issue in regression testing.

A large number of test inputs is generated in order to cover modified parts of the code. Then the tests are executed using generated test inputs on the old and new versions of the code, differences are identified and presented to developer with the details regarding the lines of changed code and the differences [2]. The proposed approach can provide developers with detailed information regarding code coverage and various statistics.

2 Related Work

A lot of research has been done in the area of automatic test case generation, for example an execution of various elements in the program [11] or detection of mutants [12, 19].

Test tools are used for test case execution (for example, Parasoft JTest [13]) and random test input generation. However, random test inputs may not be sufficient to detect different behavior of the new version of program.

Significant amount of research has been done in the area of regression testing in the past few years. Some of approaches [14, 15] rerun test case with the same test inputs and check the outputs of the test case against the captured outputs.

Another approach [16] generates test input set, executes them and collects the return values and object states after the execution of each method under test. The following executions retrieve the same information and check against the initially collected return values and states. Many approaches focus on testing the changed parts of two versions of a software application and takes into account changes related to method return values, object states, and program outputs.

In some cases, finding behavioral differences between two versions of program may not be sufficient and it can be expanded by predicting object state deviations of a changed program [17] or introducing mutation testing [18].

In some approaches, symbolic execution is used to improve test input quality. One of such tools for the Java language could be Java Pathfinder [4, 6] which is built on top of a custom Java Virtual Machine (JVM) and here is used for test input generation. Model checking is done via execution of Java byte-codes, an approach that allows different byte-code interpretations to be developed.

One of the model checking modes in JPF is symbolic execution [5]. Extended interpretation of byte-codes is used to work with symbolic values. Symbolic JPF checks the code for conditional branches incorporating symbolic values, then tries to find out if the branch condition is satisfied for true and false possibilities and identifies values for each branch.

There is a number of helper functions and classes available for JPF, that allow to annotate code, and develop extensions to change and monitor the execution of JPF. One of them is the ability to register Java listeners for various JPF events, for example monitor the execution of a byte-code instruction. Therefore, it allows extensions to access information used internally by JPF. The ability to annotate code and monitoring JPF's execution is helpful for test generation [7].

We aim to reach the following goals:

- ✓ Detect regressions faults in the program
- ✓ Reach as high code coverage as possible
- ✓ Improve test input quality by detecting mutants

3 Problem Statement

In general, mutation describes the modification of a program according to some fault model. Mutation testing is the process of deriving test cases that identify as many mutants as possible. One test input covers one path of the method which may change after the modification of the code and the path will not be executed. Therefore, there will be paths that are never tested and it will cause lower code coverage. Besides, a

lot of test inputs and mutants need to be randomly generated in order to cover all paths and catch the mutants. Classic mutation process and our proposed approach are illustrated in **Figure 1**.

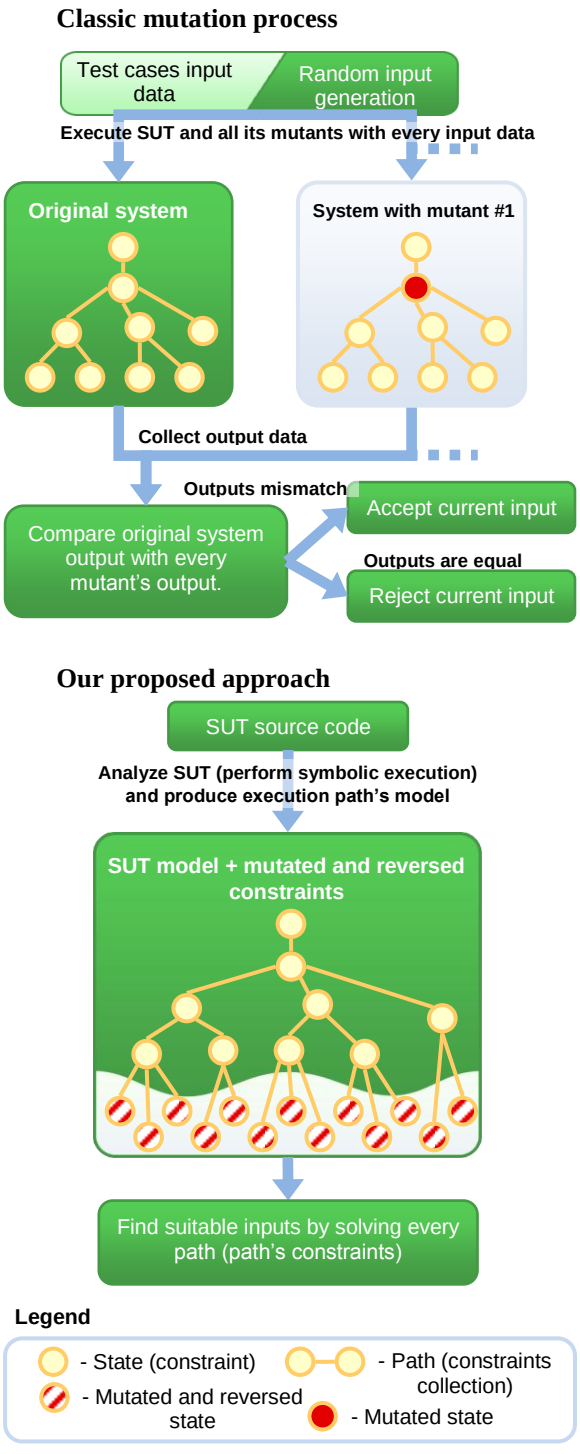


Figure 1. Comparison of classic mutation process and the proposed approach

The proposed approach uses symbolic execution which helps to improve code coverage and test input quality by detecting code mutation.

4 The Testing Technique Proposal

The process can be separated into these activities:

- ✓ Path condition generation from the source code.
- ✓ Test data generation from path conditions. An extension for symbolic execution will be developed to improve test data generation [3], which will detect mutation faults as well.
- ✓ Execution of generated test cases
- ✓ Stored result comparison with the expected results. The test case is considered to have failed in case the result does not match the expected result.

The aim is to produce unit tests because it may be run multiple times and relatively fast. **Figure 2** illustrates the described approach with more details.

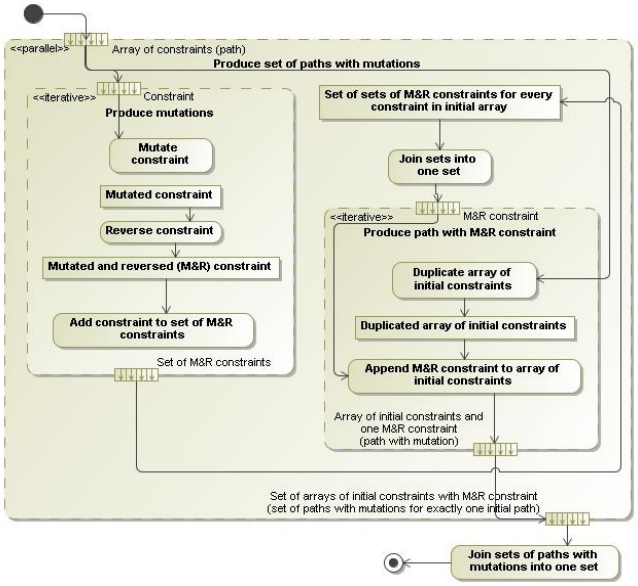


Figure 2. A concept of the software testing process

Proposed concept will address the following faults introduced because of:

- ✓ Modification of the application code
- ✓ Update of the packets that application is using. The functionality should remain unchanged.
- ✓ Changes of the platform

4.1 Symbolic Execution

Symbolic execution and software testing isn't the same. By proceeding from model checking to jUnit framework it was found that symbolic execution gives an interval of variable values in order to execute concrete path of the program. However there are cases when the infinite value set is returned and only one value needs to be chosen for the path. Only one choice doesn't always guarantee that regression faults will be detected. In order to solve this ambiguity, mutation testing will be introduced, which aims to help generating more precise test data [8, 9, 10]. This is explained in section 5.1 with more details and an example.

Main classes involved in test data retrieval and mutated test case generation are presented in **Figure 3**.

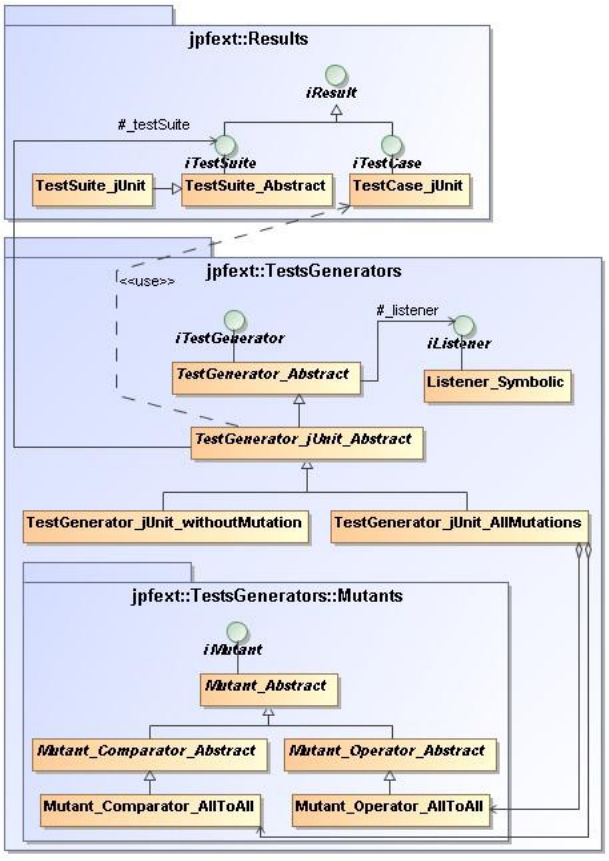


Figure 3. Symbolic execution extension diagram including mutant generation

5 Test Execution and Test Result Assessment

5.1 The Need For Mutation

After test data generation we are not sure that it detects changes in the program. Suppose we have this code:

```
public class TestPaths {
  public static void main(String[] args){
    testMe(1, 2, 3);
  }

 public static int testMe(int a, int b, int c){
    if (a + b > c) {
      return (a + b);
    } else {
      return c;
    }
  }
}
```

After symbolic execution two paths are found and returned:

- ✓ (**b**_2_SYMINT [0] + **a**_1_SYMINT [1])> **c**_3_SYMINT [0]
- ✓ (**b**_2_SYMINT [0] + **a**_1_SYMINT [0]) <= **c**_3_SYMINT [0]

These paths are used to generate corresponding test cases:

- ✓ testMe (1,0,0) -> Return value: (a_1_SYMINT + b_2_SYMINT)
- ✓ testMe (0,0,0) -> Return value: c_3_SYMINT

They are entirely correct test cases as all the program paths are executed at least once. However, after the modification of the program these test cases can be no longer adequate as they do not ensure that the faulty change of the program will be found. For example, suppose we had this code: "`if (a + b > c)`"; and it was changed to "`if (a - b> c)`" condition. Both the test with `testMe(1,0,0)` and `testMe(0,0,0)` will return a successful test execution value "Passed", although at least one of them should return "Failed" value. Both of these tests will not detect changes in the program and the possible fault.

For these reasons, we introduce mutation testing and trying to predict possible changes in the program. The main idea is that the generated test cases should fit the initial version of the program, but may not be suitable for the mutants (changed versions of the program). In other words, generated test cases will successfully pass using the initial application and fail using mutated application. In order to generate needed test cases, we do not mutate the program itself, but the expressions of execution paths. This approach has the following advantages:

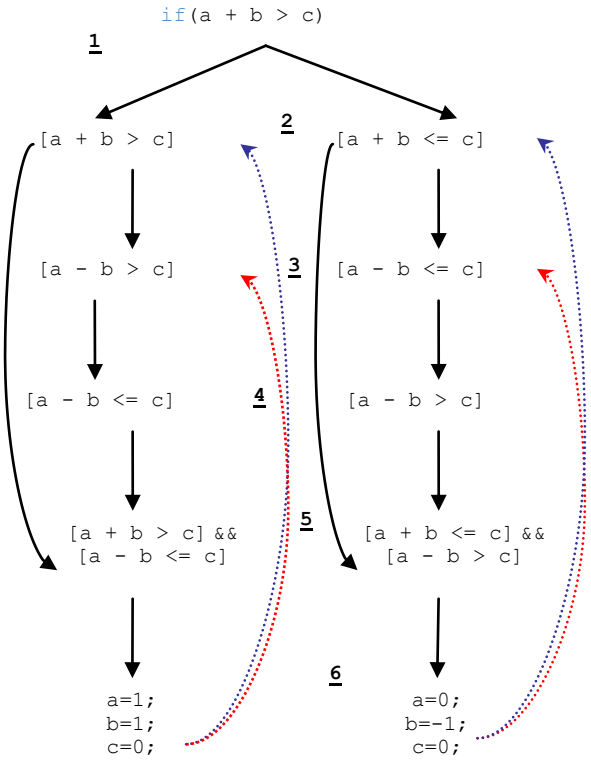
- ✓ The process of mutation is simplified because we do not try to replace the original byte-code instructions with mutated instructions. There is no need to modify the software code, compile it and execute a full analysis of the model in order to get the program execution paths and new test cases.
- ✓ There is no need to compare execution paths (the initial program and the mutant), so we can combine them and get those test cases that meet the initial version of the program and would not be appropriate to mutants.

Disadvantages of the proposed technique are the following:

- ✓ We do not know what the mutant returns. The execution path is mutated, and not the program itself, therefore it may be difficult to determine what values the mutated method will return. However, this is not needed for test case generation and test execution.
- ✓ With more complex paths, especially when there are unreachable states in the initial program, it is not possible to have 100% code coverage. One suggestion for the future work could be the extension to detect unreachable code and report it.

5.2 The Concept of Mutation Process

Once the analysis of the model of SUT is finished and the expressions of program execution paths obtained, it can be mutated and connected to the initial expressions, as illustrated in **Figure 4**.



1. Let's assume we analyze this condition of the application code.
2. We obtain two program execution paths with such conditions.
3. Obtained expressions are mutated. In this case the mutant „+" -> „-" is applied.
4. Looking for a reverse functions of the mutated, because the new test data should not fit to mutant
5. Reverse functions are connected with the initial functions so that the generated data satisfies both conditions.
6. Concrete test data is found. It satisfies the initial condition (→), but does not satisfy the mutant (→).

Figure 4. A process of test data generation

This explained how the test cases are obtained which take the mutants into account and the program changes (possible errors) are detected.

A test case construction algorithm is defined as follows:

```
MethodsToBeTested : List of methods which should be tested
MethodsInfoList : List of collected information about
methods
TestSuite : A set of returned testcases

1.  MethodsInfoList ::= []
2.  TestSuite ::= []
3.  for each method in MethodsToBeTested
4.          MethodInfo = new MethodInfo;
5.          MethodInfo.method = method;
6.          MethodInfo.pathConditions =
            JPF.findPathConditions(method);
7.          MethodsInfoList.append(MethodInfo);
8.  end for
9.  for each MethodInfo in MethodsInfoList
10.         for each pathCondition in
            MethodInfo.pathConditions
11.                 mutatedPathConditions =
                    mutate(PathCondition);
12.                 mutatedPathConditions =
```

5.3 Experimental Results

Tests were executed using the following code snippet:

```
public int testMe(int x, int y, int z, boolean k)
throws Exception {
        int res = 0;
        if((15 > y) && (x + 10 < y) && (y > 10) && (y
        > -x + 5)) {
                switch(z) {
                        case 0: res = 0; break;
                        case 1: res = x; break;
                        default: res = y; break;
                }
        } else {
                if (k) {
                        y *= 10;
                        if (x > y) {
                                res = y + 3;
                        }
                } else {
                throw new Exception();
                }
        }
        return res;
}
```

The application was tested three times: first with random test input generation (JTest), second using symbolic execution (JPF) which gives full code coverage and the third with symbolic execution and the extension enabled which takes mutants into account (`>`, `<`, `<=`, `>=`, `==`, `!=`, `&&`, `||`, `^`, `+`, `-`, `*`, `/`). There are six conditions and five mutants for each of them, three ampersand and five mathematical operator replacements (6*5+3*2+5*3=51 mutants). The number of detected faults is showed in **Figure 5**.

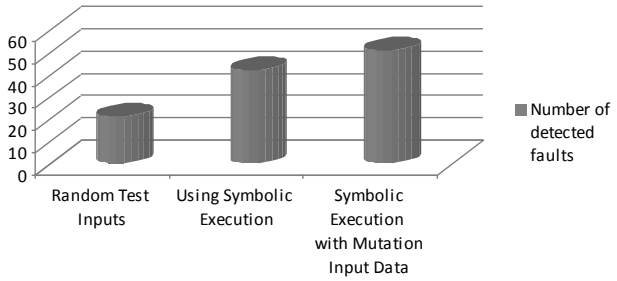


Figure 5. Test result assessment using different test inputs

The number of detected faults increased from 9 to 13 in the experiment. Test results show that symbolic execution with our extension increases the number of detected possible faults using the same number of test inputs.

6 Conclusions and Future Work

This paper presented a formal technique to the regression testing process satisfying structural code coverage with a higher quality of test data.

Experimental results showed that test data generated with symbolic execution gives a full structural code coverage which increases a number of detected faults in the program comparing to randomly generated test inputs. However, some of mutation faults still remain. This is solved using symbolic execution extension and improved test data generation which increases test case quality and detect more mutants using the same number of test inputs.

Tasks that could be accomplished in the future:

- ✓ Combine a number of test cases derived from the different mutants into one test case.
- ✓ Create and integrate jUnit extension in test code which keeps track of how many lines of code were executed using the generated tests.
- ✓ Add extension that supports complex data structures.
- ✓ Add extension that verifies the correctness of code not only according to the returned values, but also based on the inner states of objects or functions.

References

[1] **Orso A., Xie T.** BERT: BEhavioral Regression Testing. *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, 2008

[2] **Oezbek C.** Introducing Automated Regression Testing in Open Source Projects. *TECHNICAL REPORT SERIES B - B-10-01, OSS 2010*, 2010

[3] **Visser W., Pasareanu C., Khurshid S.** Test Input Generation with Java PathFinder. *Proceedings of ISSTA 2004. Boston, MA*, 2004.

[4] **Khurshid S., Pasareanu C. S., Visser W.** Generalized Symbolic Execution for Model Checking and Testing. *Proceedings of TACAS 2003. Warsaw, Poland*, 2003

[5] **Pasareanu C. S., Visser W.** Symbolic Execution and Model Checking for Testing. *Verification Conference 2007, LNCS 4899: 17-18 (IBM HVC Award)*, 2007

[6] **Artho C., Drusinsky D., Goldberg A., Havelund K., Lowry M., Pasareanu C., Rosu G., Visser W.** Experiments with Test Case Generation and Runtime Analysis. *Proceedings of Abstract State Machines 2003 . Taormina, Italy, March 2003. LNCS 2589*, 2003

[7] **Walkinshaw N., Bogdanov K., Ali S., Holcombe M.** Automated discovery of state transitions and their functions in source code. *Software Testing, Verification and Reliability. Wiley InterScience.* 2007.

[8] **Bybro M, Arnborg S.** A Mutation Testing Tool for Java Programs. *Thesis, Department of Numerical Analysis and Computer Science, Nada at the Royal Institute of Technology, KTH, Sweden*, 2003.

[9] **Offutt J., Untch R. H.** Mutation 2000: Uniting the orthogonal. In Mutation2000 Conference: Mutation Testing in the Twentieth and the Twenty First Centuries, *San Jose, CA, pages 45-55*, 2000.

[10] **Kim S., Clark J. A., McDermid J. A.** Class Mutation: Mutation Testing for Object-oriented Programs. *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, 2000

[11] **Gupta N., Mathur, A.P., Soffa, M.L.** Generating Test Data for Branch Coverage. *ASE'00, 219-227*, 2000.

[12] **DeMillo R., Offutt, J.** Experimental results from an automatic test case generator. *ACM TOSEM, 2(2), 109-127*, 1993.

[13] Parasoft Jtest manuals version 4.5. *Online manual, http://www.parasoft.com/*, 2003

[14] **Saff D., Artzi S., Perkins J. H., Ernst M. D.** Automatic test factoring for Java. *In Proc. IEEE International Conference on Automated Software Engineering (ASE 2005), pages 114–123*, 2005.

[15] **Orso A., Kennedy B.** Selective capture and replay of program executions. *In Proc. International ICSE Workshop on Dynamic Analysis (WODA 2005), pages 29–35*, 2005.

[16] **Xie T.** Augmenting automatically generated unit-test suites with regression oracle checking. *In Proc. European Conference on Object-Oriented Programming (ECOOP 2006), pages 380–403*, 2006.

[17] **DeMillo R. A., Offutt A. J.** Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering, 17(9):900–910*, 1991.

[18] **DeMillo R. A., Lipton R. J., Sayward F. G.** Hints on test data selection: Help for the practicing programmer. *IEEE Computer, 11(4):34–41*, 1978.

[19] **Fraser G.** Automated Software Testing with Model Checkers. *Dissertation, IST - Institute for Softwaretechnology Graz University of Technology*, 2007

10.2. Eksperimentinių programų testuotų metodų kodo išeities tekstai

Šiame priede yra pateikiami eksperimentinių programų testuoti metodai tam, kad būtų aiškiau koks programinis kodas buvo analizuojamas ir mutuojamas.

Test1 programos metodas

```
public int testMe(boolean a, boolean b, int c) {
    if(a) {
        if(b) {
            return c;
        }
        return -c;
    }
    return c*4;
}
```

Test2 programos metodas

```
public boolean testMe(boolean a, boolean b, boolean c, boolean d) {
    return a && b && c && d;
}
```

Test3 programos metodas

```
public int testMe(int a, int b, int c, int d) {
    if(a > 20 && b < c && c >= 3 && d != a+b) {
        return d-a;
    }
    return a-d;
}
```

Test4 programos metodas

```
public int testMe(int x, int y, int z, boolean k) throws Exception{
    int res = 0;

    if(15>y && x+10<y && y>10 && y>-x+5) {
        switch(z){
            case 0: res = 0; break;
            case 1: res = x; break;
            default: res = y; break;
        }
    } else {
        if(k) {
            y*=10;
            if(x>y){
                res = y+3;
            }
        } else {
            throw new Exception();
        }
    }

    return  res;
}
```

Test5 programos metodas

```
public boolean testMe(boolean a, int b, int c){
    if(a){
        if(b > 0)
            return true;
        if(c > 0)
            return true;
        return false;
    } else {
        if(b == 2){
            if(c > 0)
                return true;
        }
        return false;
    }
}
```

10.3. Eksperimentinių programų mutantų detalūs testavimo rezultatai

Reikia atkreipti dėmesį į tai, jog lentelėje mutantai nėra išskiriami – yra sužymėti tik jų sekos numeriai. Tačiau kiekvienai eksperimentinei programai mutantai skiriasi ir visi jie yra unikalūs.

Lentelė nr. 15 - Detalūs testavimo rezultatai

PĮ, kuriai sugeneruoti testai	Test1		Test2		Test3		Test4		Test5	
Testų generavimo metodas / Mutantai	SV	SV+M	SV	SV+M	SV	SV+M	SV	SV+M	SV	SV+M
Mutantas 1	0,0%	0,0%	**100%**	**100%**	**100%**	90,9%	92,9%	97,1%	33,3%	36,4%
Mutantas 2	33,3%	33,3%	**100%**	**100%**	80,0%	54,5%	85,7%	85,7%	83,3%	81,8%
Mutantas 3	66,7%	66,7%	80,0%	80,0%	80,0%	90,9%	**100%**	**100%**	**100%**	90,9%
Mutantas 4	66,7%	66,7%	40,0%	40,0%	60,0%	54,5%	71,4%	60,0%	**100%**	90,9%
Mutantas 5	66,7%	66,7%	60,0%	60,0%	60,0%	45,5%	92,9%	91,4%	83,3%	81,8%
Mutantas 6	66,7%	66,7%	80,0%	80,0%	80,0%	81,8%	92,9%	94,3%	83,3%	81,8%
Mutantas 7	66,7%	66,7%	**100%**	**100%**	**100%**	**100%**	92,9%	91,4%	83,3%	72,7%
Mutantas 8			**100%**	**100%**	80,0%	63,6%	**100%**	**100%**	66,7%	63,6%
Mutantas 9			**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	90,9%
Mutantas 10			80,0%	80,0%	80,0%	63,6%	78,6%	82,9%	83,3%	72,7%
Mutantas 11					80,0%	63,6%	**100%**	97,1%	83,3%	81,8%
Mutantas 12					80,0%	90,9%	85,7%	82,9%	66,7%	63,6%
Mutantas 13					80,0%	72,7%	78,6%	80,0%	66,7%	54,5%
Mutantas 14					**100%**	90,9%	92,9%	94,3%	83,3%	90,9%
Mutantas 15					60,0%	63,6%	85,7%	91,4%	66,7%	72,7%
Mutantas 16					60,0%	54,5%	**100%**	**100%**	83,3%	90,9%
Mutantas 17					80,0%	81,8%	**100%**	**100%**	66,7%	81,8%
Mutantas 18					60,0%	72,7%	**100%**	94,3%	**100%**	90,9%
Mutantas 19					60,0%	45,5%	**100%**	**100%**	83,3%	81,8%
Mutantas 20					80,0%	81,8%	**100%**	**100%**	83,3%	90,9%
Mutantas 21					60,0%	63,6%	85,7%	85,7%	66,7%	81,8%
Mutantas 22					80,0%	63,6%	85,7%	85,7%	**100%**	90,9%
Mutantas 23					**100%**	81,8%	85,7%	85,7%	83,3%	81,8%
Mutantas 24					80,0%	81,8%	85,7%	85,7%	83,3%	90,9%
Mutantas 25					80,0%	81,8%	**100%**	**100%**	66,7%	81,8%
Mutantas 26					80,0%	81,8%	85,7%	85,7%	66,7%	72,7%
Mutantas 27					20,0%	36,4%	**100%**	**100%**	83,3%	90,9%
Mutantas 28					40,0%	54,5%	85,7%	85,7%	83,3%	81,8%
Mutantas 29					20,0%	36,4%	85,7%	85,7%	83,3%	72,7%
Mutantas 30					80,0%	63,6%	85,7%	85,7%		
Mutantas 31					80,0%	63,6%	**100%**	94,3%		
Mutantas 32					80,0%	63,6%	85,7%	80,0%		
Mutantas 33					80,0%	63,6%	85,7%	88,6%		
Mutantas 34					60,0%	54,5%	71,4%	74,3%		

Mutantas 35	80,0%	63,6%	71,4%	68,6%
Mutantas 36	**100%**	**100%**	85,7%	82,9%
Mutantas 37	60,0%	54,5%	**100%**	88,6%
Mutantas 38	80,0%	81,8%	35,7%	40,0%
Mutantas 39	80,0%	72,7%	50,0%	45,7%
Mutantas 40	20,0%	36,4%	92,9%	94,3%
Mutantas 41	20,0%	36,4%	**100%**	**100%**
Mutantas 42	40,0%	45,5%	85,7%	82,9%
Mutantas 43	20,0%	36,4%	92,9%	94,3%
Mutantas 44			92,9%	94,3%
Mutantas 45			92,9%	91,4%
Mutantas 46			21,4%	20,0%
Mutantas 47			78,6%	71,4%
Mutantas 48			78,6%	77,1%
Mutantas 49			78,6%	74,3%
Mutantas 50			71,4%	77,1%
Mutantas 51			78,6%	68,6%
Mutantas 52			57,1%	54,3%
Mutantas 53			50,0%	45,7%
Mutantas 54			64,3%	71,4%
Mutantas 55			85,7%	74,3%
Mutantas 56			78,6%	85,7%
Mutantas 57			78,6%	85,7%
Mutantas 58			78,6%	85,7%
Mutantas 59			78,6%	85,7%
Mutantas 60			92,9%	94,3%
Mutantas 61			92,9%	91,4%
Mutantas 62			92,9%	94,3%
Mutantas 63			92,9%	91,4%