LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

TESTU AUTOMĀTISKA ĢENERĒŠANA

MAĢISTRA DARBS

Autors: **Artūrs Ivanovs**
Stud. apl. Nr. ai10040
Darba vadītājs: Dr.dat., prof. Jānis Bičevskis

RĪGA 2016

Anotācija

Maģistra darbā ir apskatīti dažādi rīki un to pieejas testpiemēru ģenerēšanai ar ieejas datu kopu. Darbā tiek apskatīts reāli praksē pielietojams rīks – IntelliTest, kā tas ģenerē testpiemērus dažādām funkcionāli vienkāršām programmām, kas ir iekļauts Visual Studio programmatūras izstrādes vidē. Līdzīgi ir apskatīts arī EvoSuite rīks, kas ir atvērtā koda testpiemēru ģenerēšanas rīks. Tiek apskatīti rīku ierobežojumi.

Autors piedāvā idejas šo rīku uzlabojumiem, ņemot vērā to ģenerētās testpiemēru kopas kvalitāti, iegūtos rezultātus.

Atslēgas vārdi: testēšana, testpiemēri, IntelliTest, EvoSuite.

Abstract

In this master's thesis – "Automatic test data generation" – the author examines different tools and techniques used by them for test generation with an input value set. A real, used in practice tool – IntelliTest is examined, how it generates tests with an input value set for some functionally simple programs, which is included in the Visual Studio application development suite. Similarly, the EvoSuite tool is examined, it is a standalone, open-source tool.

The author proposes some ideas for the examined tool improvement, based on the quality of their generated test set and results.

Keywords: testing, test, IntelliTest, EvoSuite.

Autoreferāts

Maģistra darba izstrādes gaitā autors ir aplūkojis testpiemēru ģenerēšanas sistēmu *SMOTL,* ar to iegūtos rezultātus, ir apskatījis pilnu testu sistēmu algoritmu darba 1. nodaļā. Sistēmas apraksts, iegūtie rezultāti un algoritma apraksts ir iegūts no norādītajām publikācijām un disertācijām.

Darba 2. nodaļā autors ir veicis testpiemēru ģenerēšanas pieeju apskatu. Autors šajā nodaļā veic *IntelliTest* un *EvoSuite* rīku apskatu, kuri ģenerē testpiemērus programmām valodās *C#* un *Java*. Tiek analizētas rīku izmantotās dažādās pieejas testpiemēru ģenerēšanai. Tiek apskatīts to darbības princips, dažādi to iespējamie ierobežojumi, iespējas. Darba autors šajā nodaļā ir izveidojis vairākas nelielas datorprogrammas, kurām ir uzģenerēti dažādi testpiemēri ar *IntelliTest* un *EvoSuite* rīkiem. Autors veic iegūto rezultātu analīzi, tie atbilst izmantotajos resursos aprakstītajām rīku iespējām. No avotiem internetā ir iegūti rīku vispārēji apraksti, vispārēji algoritmi, ierobežojumi.

Darba 3. nodaļā autors turpina iegūto rezultātu analīzi, salīdzina testpiemēru ģenerēšanas rīku rezultātus. Tāpat arī darba autors spriež par iespējamiem uzlabojumiem šiem rīkiem. Tiek piedāvāts konkrēts algoritma uzlabojums IntelliTest rīkam un citi vispārīgi uzlabojumi.

SATURS

APZĪMĒJUMU SARAKSTS ... 7

IEVADS ... 8

1. PILNAS TESTU SISTĒMAS, SISTĒMA SMOTL ... 10

1.1 SMOTL sistēmas darbības princips ... 10

1.2 Rīka pielietojuma rezultāts ... 11

1.3 Nozīmīgums programmatūras izstrādē ... 11

2. MŪSDIENU TESTU ĢENERĒŠANAS RĪKI ... 13

2.1 IntelliTest ... 14

2.1.1 IntelliTest un Pex darbības princips ... 14

2.1.2 Darbības ierobežojumi ... 16

2.1.3 Programma ar zarošanos ... 17

2.1.4 Programma ar vienkāršu for ciklu ... 23

2.1.5 Programma ar objektiem ... 25

2.1.6 Programma ar pretrunu ... 26

2.1.7 Programmas ar matemātiski sarežģītu funkciju ... 28

2.1.8 Programma ar ieciklošanos ... 30

2.1.9 Darbības robežas ... 31

2.2 EvoSuite ... 32

2.2.1 EvoSuite darbības princips ... 33

2.2.2 Konfigurācijas iespējas ... 33

2.2.3 Programma ar zarošanos ... 34

2.2.4 Programma ar vienkāršu for ciklu ... 35

2.2.5 Programma ar objektiem ... 36

2.3 Citi rīki ... 38

3. APSKATĪTO RĪKU SALĪDZINĀJUMS UN IESPĒJAMIE UZLABOJUMI ... 39

3.1 IntelliTest un EvoSuite salīdzinājums, iegūto rezultātu salīdzinājums ... 39

3.2 Iespējamie uzlabojumi IntelliTest ... 40

3.2.1 IntelliTest vispārīgi, tehniski uzlabojumi ... 41

3.2.2 IntelliTest algoritma uzlabojumi ... 42

3.3 Iespējamie uzlabojumi EvoSuite 48
SECINĀJUMI 49
IZMANTOTĀ LITERATŪRA 50
PIELIKUMI 52
1. pielikums. Testpiemēru ģenerēšanas rīku saraksts 52
2. pielikums. EvoSuite ģenerētie testpiemēri 53

APZĪMĒJUMU SARAKSTS

Darbā ir izmantoti zemāk uzskaitītie apzīmējumi:

- *PTS* – pilna testu sistēma;
- *SMOTL* – testpiemēru ģenerēšanas sistēma, rīks;
- *SMOD* – programmēšanas valoda;
- *IEEE – Institute of Electrical and Electronics Engineers,* Elektronikas un Elektrotehnikas inženieru institūts, starptautiska organizācija;
- *.NET – Microsoft* izstrādāts ietvars, platforma lietojumprogrammu izveidei;
- *Pex* – testpiemēru ģenerēšanas rīks;
- *IntelliTest* – testpiemēru ģenerēšanas rīks;
- *CIL* – starpvaloda, kurā tiek pārvērsts kods, kad to kompilē;
- *EvoSuite* – testpiemēru ģenerēšanas rīks.

IEVADS

Testēšana ir būtiska programmatūras izstrādes sastāvdaļa. Tas ir laikietilpīgs process, kas ļauj izdarīt secinājumus par programmatūras kvalitāti, atbilstību prasībām. Tas ir arī radošs process, jo bieži kļūdas programmas darbībā tiek atklātas tieši ar nestandarta ievadēm vai to kombinācijām. Testēšana bieži tiek veikta nepilnīgi vai pat vispār tiek izlaista. No cilvēciskā viedokļa, visas kļūdas atrast ir grūti vai pat neiespējami. Tomēr praksē atrast visas kļūdas programmatūrā parasti netiek izvirzīts kā mērķis. Parasti tiek izvirzīts kāds pārklājuma kritērijs, kas ir jāsasniedz ar izveidotajiem testpiemēriem. Lai sasniegtu šo kritēriju, testētājam ir jāizveido atbilstoša testpiemēru kopa. Šo procesu var atvieglot, nepieciešams izmantot kādas programmas – palīgrīkus testēšanā. Tas ir īpaši aktuāli lielos projektos un sarežģītu sistēmu izstrādē. Ir izstrādātas dažādas testēšanas metodes un paņēmieni, to pielietojums ir atkarīgs no konkrētās situācijas, katrai metodei vai paņēmienam ir savas priekšrocības.

Kā īpašu testēšanas paveidu ir jāatzīmē vienībtestēšanu – mazākā funkcionālā vienuma korektuma pārbaudi. Vienībtestēšanu parasti automatizē, programmu izpilda ar vairākiem iepriekš sagatavotiem testpiemēriem ar konkrētiem ievadiem, to atkārto. Eksistē daudzi rīki, kas atvieglo šo darbu. Ar vienībtestēšanu saistīti rīki ir pievienojami kā papildinājumi un tiek piedāvāti arī jau integrēti populārākajās izstrādes vidēs. Attiecīgi tie ir pieejami populārākajām programmēšanas valodām.

Mazāk populāri ir rīki, kas veic arī datu kopas ģenerēšanu – vienībtestus ar konkrētiem ievadiem, parasti netiek integrēti izstrādes vidēs. Tie var risināt ar testēšanas procesu saistītas problēmas. Piemēram, atrast grūti pamanāmas kļūdas, izveidot testpiemērus, kas aptver visus programmas zarus. Šādi rīki eksistē, ir pieejami gan maksas, gan bezmaksas, un darbā tiek apskatīti turpmākajās nodaļās. Darbā tiek noskaidrots cik funkcionāli spēcīgi ir šie pieejamie testpiemēru ģenerēšanas rīki, kā tie strādā. Tiek noskaidrots vai ar tiem var veikt testēšanu programmām, kas satur tipiskas programmēšanas konstrukcijas – zarošanos, ciklus, objektu veidošanu, satur dažāda tipa mainīgos – *int*, *string*, peldošā punkta mainīgos. Darbā arī tiek spriests par testēšanas pilnīgu automatizāciju, vai var pilnīgi paļauties uz šāda veida rīkiem testēšanas procesā. Darbā tiek noteikti arī to darbības ierobežojumi.

Kopumā darbā tiek apskatīti 3 dažādi testpiemēru ģenerēšanas rīki, pirmais no tiem ir sistēma *SMOTL*, tiek apskatīts vispārīgi, lai iegūtu priekšstatu par to, tajā izmantotajām idejām. Tiek apskatīti mūsdienu risinājumi testpiemēru ģenerēšanai, tie ir rīki *IntelliTest* un *EvoSuite*.

Darba mērķis: Apskatīt un analizēt mūsdienu testpiemēru ģenerēšanas rīkus, to darbības algoritmus, pielietošanas ierobežojumus, piedāvāt uzlabojumus tiem.

Darba uzdevumi, kas jāveic, lai sasniegtu noteikto mērķi:

- iepazīties ar pilnām testu sistēmām, sistēmu *SMOTL*;
- apskatīt dažādus mūsdienu testu automātiskas ģenerēšanas rīkus;
- noteikt rīku ierobežojumus un izdarīt secinājumus par iegūtajiem rezultātiem;
- noteikt iespējamos uzlabojumus tiem, ieskaitot algoritma uzlabojumus.

Katrs no šiem uzdevumiem tiek veikts attiecīgajā darba nodaļā.

Darba 1. nodaļā tiek apskatīta *SMOTL* sistēma, pilnu testu sistēmu algoritms, sistēmas pielietojuma rezultāti, to nozīmīgums.

Darba 2. nodaļā tiek apskatītas testpiemēru ģenerēšanas pieejas, konkrēti rīki un to vispārēji algoritmi, kas veic testpiemēru ģenerēšanu. Tiek veikta testpiemēru ģenerēšana dažādām programmām un iegūto rezultātu analīze.

Darba 3. nodaļā tiek turpināta iegūto rezultātu analīze, rezultāti tiek salīdzināti. Tiek piedāvāti iespējamie uzlabojumi apskatītajiem testpiemēru ģenerēšanas rīkiem.

1. PILNAS TESTU SISTĒMAS, SISTĒMA SMOTL

Latvijas Universitātē 20. gadsimta 70. gadu beigās tika pētīts kā programmai, kuras kods ir pieejams, varētu uzģenerēt testpiemēru kopu ar ieejas datiem. Tika aprakstīta pilnu testu sistēmu (turpmāk tekstā – PTS) teorija un arī izstrādāts rīks, kas reālām datorprogrammām ģenerē PTS. PTS jēdziens definēts publikācijā [1], maģistra darba autora tulkojums: "Par PTS sauc tādu galīgu testpiemēru kopu *S*, ja tā sastāv no pieļaujamiem testiem (tādi testi, kur programma beidz darbu normāli) un katrs programmas zars, kas ir izpildāms, ir izpildāms uz kādā testpiemēra no kopas *S*." Tātad, programmai tiek izveidota testpiemēru kopa, kas sasniedz maksimālu zaru pārklājumu, bet netiek veidoti testpiemēri neizpildāmiem programmas zarojumiem.

SMOTL ir sistēma jeb rīks, kas ģenerē PTS reālai datorprogrammai, kas rakstīta valodā *SMOD*, kas ir līdzīga programmēšanas valodai *COBOL*. Rīks nodrošina arī atkārtotu programmas izpildi ar uzģenerētajiem testiem. Šī sistēma kā ieeju izmanto tikai dotās programmas kodu.

1.1 SMOTL sistēmas darbības princips

Sistēmas *SMOTL* darbības loģiku var aprakstīt sešos soļos jeb fāzēs [1], [2]:

1. orientēta grafa konstruēšana, kur kā ieeju izmanto programmas kodu;
2. programma tiek analizēta, bet netiek darbināta; analīzē iegūtā informācija ļauj paātrināt turpmāko fāžu izpildi;
3. dažādu realizējamo programmas ceļu būvēšana; tiek veikta programmas simboliskā izpilde, nosacījumu sistēmu risināšana, lai noteiktu izvēlēto ceļu realizējamību;
4. no iepriekšējā fāzē izveidotajiem ceļiem izvēlas tādus ceļus, kas ietver visus programmas zarus (komandu pārejas); ceļu skaitu kopā optimizē, jo zaru pārklājumam visi nav vajadzīgi – daudzi var atkārtoties;
5. tiek noteiktas konkrētas ieejas vērtības, notiek nosacījumu sistēmu risināšana; iegūtās vērtības izpilda izveidotos ceļus, veido testpiemērus ar ieejas vērtībām;
6. testpiemēru izvade, saglabāšana, programmas izpilde un citas darbības, fāze nav tieši saistīta ar PTS ģenerēšanu.

Piektajā fāzē kā ieeja var būt vairāki ceļi. Lai iegūtu reālus testpiemērus, tiek veikta šo visu ceļu simboliskā izpilde – tiek noteikti ceļu izpildes nosacījumi. Šeit tiek iegūta simbolisko vērtību nosacījumu sistēma. Šie nosacījumi, kas var būt nevienādību sistēma, ir atrisināmi vai arī neatrisināmi. Ja atrisinājums eksistē, tad rezultātā tiek iegūtas konkrētas ieejas vērtības, kas izpilda aprakstīto ceļu. Sistēmas būtiskākā īpašība ir tā, ka tiek izmantoti stāvokļi. Tie tiek iegūti ceļa izpildes rezultātā, kas ir vispārināts iespējamo vērtību apraksts [1], [2].

1.2 Rīka pielietojuma rezultāts

PTS teorijas izstrādātāji pielietoja *SMOTL*, lai ģenerētu testpiemērus ar datiem reālām programmām valodā *SMOD*. Autori konstatēja, ka 25 programmām, kas nepārsniedz 300 komandas šajā valodā, PTS tika izveidots tikai 16 programmām ar dažādiem koda pārklājumiem. Katrā programmā vidēji bija 3 kļūdas, kuras tika pamanītas pielietojot šo rīku. Šīs kļūdas nebija nozīmīgas, nekorekta tukšu failu apstrāde, netipiski ievadi, taču labi parādīja to, ka rīks strādā un pamana problēmas, kuras ir grūti pamanīt programmētājām vai arī tiek aizmirsts apstrādāt [3].

SMOTL tika pielietots arī apjomīgām programmām – vairāk kā 300 komandu. Tikai 5 no 14 šādām programmām tika uzģenerētas PTS. Tas tika skaidrots ar sarežģītu programmu loģiku, ierobežotiem pieejamiem atmiņas resursiem, PTS būvēšanas algoritma nepilnībām. Šajās programmās ar *SMOTL* palīdzību tika atrastas vidēji 7 kļūdas katrā programmā, turklāt kļūdas bija daudz nozīmīgākas par izņēmuma gadījumiem [3].

Tika izdarīts secinājums, ka programmētājs nevar pārbaudīt manuāli visus iespējamos gadījumus lielās programmās [3]. Iegūtie rezultāti norāda uz to, ka testēšanas process nav pilnībā automatizējams, rezultāti ir atkarīgi no programmas apjoma un sarežģītības.

PTS teorijas, algoritma apraksts testu ģenerēšanai un *SMOTL* rīka pielietojuma rezultāti ir precīzi aprakstīti disertācijā [3] un daļa no tā ir publicēta IEEE žurnālā [1].

1.3 Nozīmīgums programmatūras izstrādē

Iespēja uzģenerēt testpiemērus programmatūras kodam ļauj palielināt programmas kvalitāti, jo automatizēts rīks var pamanīt iespējamās kļūdas, kuras cilvēks var arī nepamanīt. Šāda rīka lietošana var atvieglot arī testu bāzēto izstrādi, kuras pamātā ir vienībtestu rakstīšana. Testu bāzētais izstrādes paveids sastāv no cikliem, kas atkārtojas – sākotnēji tiek rakstīti vienībtesti, pēc tam, izmainot programmas kodu, tiek panākts pozitīvs šī testpiemēra

rezultāts. Izmantojot šādus ģenerēšanas rīkus var samazināt vai nosacīti izslēgt vienu no šādas izstrādes pamatelementiem – vienībtestu manuālu rakstīšanu.

Lai gan ir pagājis ilgs laiks kopš citētā darba publicēšanas, problēmas, ko tas mēģināja risināt ir aktuālas arī mūsdienās. Pārbaudīt visus iespējamos gadījumus, programmas zarojumus vai pat daļu no tiem ir grūti, jo datorprogrammas ir apjomīgas un sarežģītas, tas prasa būtiskus cilvēkresursus.

Mūsdienās ir izstrādāti rīki, kas atvieglo un nosacīti arī automatizē testēšanu, taču tikai retais piedāvā ko līdzīgu minētajam *SMOTL* rīkam, kas ģenerē testpiemērus ar ieejas datiem, turklāt spēj noteikt atsevišķu ceļu realizējamību. Šī rīka realizācijā izmantotās idejas, piemēram, nosacījumu sistēmas sastādīšana un risināšana, tiek izmantota arī mūsdienu šāda veida rīkos.

2. MŪSDIENU TESTU ĢENERĒŠANAS RĪKI

Šajā nodaļā ir apskatīti mūsdienu rīki, kas automātiski ģenerē testus, kā ieeju izmantojot tikai programmatūras kodu. Testu ģenerēšana tiek veikta dažādu uzdevumu un veida nelielām programmām, kas nosacīti atklāj rīku algoritmu. Tādā veidā tiek noteiktas šo rīku piedāvātās iespējas un to aptuvens darbības princips, algoritms pēc kā tie strādā. Tāpat arī tiek noteiktas iespējamās rīku nepilnības. Programmas ir rakstītas tādā programmēšanas valodā, kādu atbalsta konkrētais rīks. Ir izvēlēti tādi rīki, kas strādā ar populārām objektorientētām programmēšanas valodām. Autors pieņem, ka rīkiem nevajadzētu būt grūtībām uzģenerēt atbilstošu testpiemēru kopu, ja testējamās programmas nav funkcionāli sarežģītas.

Tiek nošķirtas šādas iespējamās metodes, kuras izmantojot ir iespējams veikt testpiemēru ģenerēšanu [4]:

- Patvaļīga atlase: nejauša ieejas parametru ģenerēšana, mēģina uzminēt vērtības ar kurām izpildās kāds ceļš programmā; vienkārša pieeja, taču sarežģītākām programmām ar vairākiem zarojumiem un cikliem, to kombinācijām, būs grūti sasniegt lielu koda pārklājumu; iespējama nosacīti patvaļīga atlase, ja iepriekš tiek noteikts kāds intervāls no kā izvēlēties ieejas parametrus;
- Koda anotācijas: ja kods ir veidots tā, ka ir noteikti dažādi "pirms" un "pēc" nosacījumi, šo informāciju arī var izmantot, lai ģenerētu attiecīgus ieejas parametrus; nosacījumi parasti attiecas uz kādu mainīgo, funkcijas sākumā tam ir jāatbilst vienam nosacījumam ("pirms" nosacījums) un pēc noteiktām darbībām jāatbilst citam nosacījumam ("pēc" nosacījums); *C#* valodā šos nosacījumus var uzstādīt ar kodu kontraktu (*Code Contract*) [5] palīdzību;
- Meklēšanas tehnikas: dažādi meklēšanas algoritmi, piemēram, ģenētiskie algoritmi, kur ieejas parametri tiek pakāpeniski attīstīti, lai sasniegtu vajadzīgo pārklājumu, izietu nepieciešamos ceļus; tiek izmantota atbilstības funkcija, šī funkcija apraksta kādu pārklājumu – programmas zaru, ceļu.
- Simboliskās izpildies tehnikas: šīs tehnikas ir dinamiskā simboliskā izpilde, simbolisko un konkrētu mainīgo tehniku apvienojums.

Jāatzīmē, ka iepriekšējās nodaļās apskatītais *SMOTL* rīks izmanto simbolisko izpildes tehniku, lai ģenerētu testpiemērus.

Šajā darbā tiek apskatīti un praktiska testpiemēru ģenerēšana tiek veikta ar *IntelliTest* rīku un *EvoSuite* rīku, tie izmanto kādu no augstāk uzskaitītajām tehnikām. Tie nav vienīgie

šāda veida rīki. Ir pieejami vēl daudzi citi rīki, to saraksts ir pieejams 1. pielikumā – "Testpiemēru ģenerēšanas rīku saraksts", to darbības principi atbilst kādai no metodēm, kas minēta augstāk esošajos pārskaitījuma punktos. Ir iespējams arī vairāku uzskaitīto tehniku apvienojums – hibrīda pieeja testpiemēru ģenerēšanai.

2.1 IntelliTest

Programmatūras izstrādes vide *Visual Studio* piedāvā rīku, kas var veikt vienībtestu ģenerēšanu ar uzģenerētu ieejas datu komplektu. Šī iespēja ir pieejama jaunākajā šīs izstrādes vides versijā – *Visual Studio 2015*, kas tiek izmantota arī šī darba izstrādes laikā ar pirmo atjauninājumu (*Update 1*). Rīks ir pieejams tikai pašā pilnīgākajā no tām – *Enterprise* apakšversijā. Šis rīks tika saukts arī par *Smart Unit Tests*, bet vēlāk tika pārsaukts par *IntelliTest*. Līdzīgs rīks, *Pex*, bija pieejams jau *Visual Studio 2010* versijā, kas ģenerēja parametrizētus vienībtestus. *IntelliTest* ir attīstīts uz *Pex* rīka bāzes.

Ar *IntelliTest* ir iespējams ģenerēt vienībtestu kopu kādai konkrētai klases metodei vai arī visām klases metodēm vienlaicīgi. Katram nosacījumam programmas kodā tiek ģenerēts testa ievads, kas izpildīs konkrēto nosacījumu. Rīks analizē katru nosacījumu zaru programmas kodā. Tiek analizēti *if* zarošanās nosacījumi, apgalvojumi un visas operācijas, kas var veidot izņēmumgadījumus. Analīzes rezultāta informācija tiek izmantota, lai katrai metodei izveidotu vairākus vienībtestus ar parametriem. *IntelliTest* veido vienībtestus tā, lai sasniegtu maksimālu koda pārklājumu. Uzģenerētajā testu kopā var redzēt, kuri testi ir izpildījušies, kādi ir bijuši ieejas dati un kļūdu paziņojumi. Lietotājs var izvēlēties testus no visas kopas, iegūtā rezultāta loga un saglabāt tos, lai vēlāk veiktu regresa testēšanu. Tiek apgalvots, ka rīks strādā tikai ar *C#* programmēšanas valodu [6]. Tas arī tika pārbaudīts – programmām valodā *Visual Basic* un funkcionālajā valodā *F#* nav iespējams veikt testpiemēru ģenerēšanu ar *IntelliTest*. Jāatzīmē, ka izstrādes vides *Visual Studio 2010* versijā pieejamais *Pex* to atbalstīja.

2.1.1 IntelliTest un Pex darbības princips

Pex rīka algoritms ir *IntelliTest* rīka pamatā, jo tas ir attīstīts no *Pex*. Lai saprastu kā tas strādā, pietiek saprast *Pex* rīka darbības principus. Rīks izmanto baltās kastes koda analīzes, lai izveidotu ievadus, ar kuriem tiktu izpildīti visi zarojumi kodā [7].

Vispārīgs rīka algoritms, kas tiek saukts arī par koda izpēti, aprakstīts *Microsoft* izstrādātāju tīkla (angl. MSDN) emuārā [8], darba autora tulkojums:

1. "Tiek instrumentēts programmas kods un izveidotas atsauces, kas ļauj testēšanas dzinējam pārraudzīt izpildi; Notiek programmas darbināšana ar vienkāršāko iespējamo ievades vērtību, kas ir atkarīga no parametra tipa; Tiek izveidots sākotnējais testpiemērs;
2. Testēšanas dzinējs pārrauga izpildi, aprēķina katram testam pārklājumu, seko līdzi tam, kas notiek ar ievades vērtību; Ja visi zari ir apstaigāti, tad process apstājas; Ja visi zari nav apstaigāti, tad tiek piemeklēts testpiemērs, kas aiziet līdz vietai, no kuras sākas neapstaigātais zars; Nosaka kā zarošanās nosacījums ir atkarīgs no ievades vērtības;
3. Testēšanas dzinējs izveido nosacījumu sistēmu, kas apraksta nosacījumu pie kura kontrole nonāk programmas vietā, kas aprakstīts 2. punktā, turpina neapstaigāto zaru; Nosacījumu risinātājs nosaka jaunu vērtību balstoties uz konkrēto nosacījumu;
4. Ja risinātājs spēj noteikt konkrētu ieejas vērtību, tad kods tiek darbināts ar jauno vērtību;
5. Ja testēšanas dzinējs konstatē, ka ir palielinājies testējamā koda pārklājums, tiek izveidots jauns testpiemērs iespējamam ceļam kodā.

Darbības, kas aprakstītas soļos 2 – 5 atkārto, lai sasniegtu visu zaru pārklājumu."

Pex pielieto testēšanas veidu, kas izmanto nejaušu datu kopu un arī statiskās analīzes tehnikas. Pielietotā metode tiek saukta par dinamisku simbolisko izpildi [9]. Publikācijā [10] ir sniegts šīs metodes skaidrojumus, darba autora tulkojums: "Dinamiskā simboliskā izpilde tiek pielietota, lai ģenerētu testpiemērus ar lielu pārklājumu. Metode izpilda programmu ar nejauši ģenerētām ieejas vērtībām un paralēli veic arī tās simbolisko izpildi, lai apkopotu simboliskos nosacījumus no vietām, kur zarojas kods. Simbolisko nosacījumu kopums kādam ceļam tiek saukts par šī ceļa nosacījumu. Dinamiskā simboliskā izpilde notiek vairākas iterācijas, kas veicina koda pārklājuma palielināšanu. Apgriežot otrādi nosacījumus pie zarošanās, tiek būvēti jauni ceļi, saglabājot ceļu līdz šai vietai. Tiek novērtēts vai šāds ceļš ir iespējams, to nosaka būvējot nosacījumu sistēmu. Nosacījumu risinātājs risina nosacījumu sistēmu, ja tai ir atrisinājums, tad tiek ģenerēta jauna ievades vērtība, kura izpildīs konkrēto zaru."

Dinamiskās simboliskās izpildes metode, tehnika ir izvēlēta *Pex* rīka realizācijā, jo tai ir būtiskas priekšrocības salīdzinājumā ar citām, statiskām metodēm. Šīs priekšrocības un pamatojums izvēlētajai metodei rīka realizācijā ir dots rīka autoru izstrādātā tā lietošanas

pamācībā [9], darba autora tulkojums – "Šāda veida dinamiskajā pieejā tiek ņemta vērā informācija par izpildes ceļiem, datiem, kas tiek nodoti starp analizējamo programmu un izpildes vidi, kurā tiek izpildīta programma. Ja izmanto tikai statisku koda analīzi, šāda informācija netiek ņemta vērā, nav iespējams veikt pilnvērtīgu analīzi par tām koda daļām, kas piekļūst atmiņai caur patvaļīgām norādēm, kad programma komunicē ar izpildes vidi, kam nav pieejams kods. Testēšanas rīka dzinējs veido aptuvenu vides modeli no datiem, ko vide saņem un atgriež, tiek uzturēts aptuvens programmas darbības modelis, ar to pietiek, lai varētu veikt programmas testēšanu."

2.1.2 Darbības ierobežojumi

Lai arī rīks ir funkcionāli spēcīgs, tomēr tam ir noteikti zināmi ierobežojumi, tie ir aprakstīti *Microsoft* izstrādātāju tīkla emuārā [8], darba autora tulkojums:

- "Programmēšanas valoda – Testēšanas dzinējs spēj analizēt patvaļīgas *.NET* programmas, tomēr testpiemēru kodu spēj izveidot tikai valodā *C#*;
- Nedeterminētība – Testēšanas dzinējs pieņem, ka kods, kuram tiek veikta testpiemēru ģenerēšana ir determinēts. Gadījumā, ja tā nav, tad dzinējs nogriež attiecīgo nedeterminēto ceļu, var iecikloties līdz sasniedz noklusētās izpētes robežas, piemēram, laika, stāvokļu robežas;
- Pavedieni – Testēšanas dzinējs nespēj analizēt un notestēt daudzpavedienu programmas;
- Peldošā punkta aritmētika – Testēšanas dzinējs izmanto automātisku nosacījumu risinātāju, lai noteiktu būtiskas vērtības testpiemēram un testējamam kodam. Šis risinātājs ir ierobežots un nespēj precīzi spriest par peldoša punkta tipa mainīgajiem un to aritmētiku."

Šie ir tikai zināmie ierobežojumi, taču darba autors pieļauj, ka ir vēl arī citi ierobežojumi, kas ir mazāk acīmredzami.

Jāpiebilst, ka programmas valodā *C#* un citās *.NET* saistītās valodās veidotās programmas tiek pārvērstas uz vienotu starpvalodu *CIL*, kuru izpilda virtuālā mašīna. Tas notiek kompilācijas brīdī. Ar to var skaidrot rīka spēju strādāt ar visām *.NET* valodām, neatkarību no izmantotās programmēšanas valodas.

2.1.3 Programma ar zarošanos

Ar šo programmu tiek pārbaudīts kā tiek ģenerēti testpiemēri programmai ar vienkāršiem zarošanās nosacījumiem, iegūtie rezultāti – uzģenerētā testpiemēru kopa, brīdinājumi un cita informācija lietotājam ļauj darba autoram novērtēt un izdarīt secinājumus par *IntelliTest* rīka iespējām un dažādiem ierobežojumiem. Līdzīga pieeja tiek izmantota arī turpmākajos 2.1 apakšnodaļas punktos, tiek veidotas programmas, kurām tiek ģenerēti testpiemēri. Šī pieeja tiek pamatota ar to, ka *IntelliTest* kods nav pieejams, pieejamie algoritma apraksti ir vispārīgi.

Tika izveidota neliela programma, kas nosaka trijstūra veidu pēc padotajiem malu garumiem. Programma sastāv no vairākiem zarojumiem. Zarojumu nosacījumos tiek veikti aprēķini un padoto malu garumu salīdzinājumi. Programma no padotajiem malu garumiem spēj noteikt vai trijstūris ir iespējams.

Programmas funkcija, kas nosaka trijstūra veidu, ir parādīta 2.1. att.

```
public static TriangleType triangle(int[] maluGarumi)

        {
            int a = maluGarumi[0];
            int b = maluGarumi[1];
            int c = maluGarumi[2];

            if (a <= 0 || b <= 0 || c <= 0)

            {
                // ja kāds no malu garumiem negatīvs lielums vai nulle
                return TriangleType.Kluda;
            }

            else if ((a + b <= c) || (b + c <= a) || (a + c <= b))

            {
                // neiespējams trijstūris, jo
                // jebkuru divu malu summai ir jābūt lielākai par trešo malu
                return TriangleType.Kluda;
            }

            else if ((a == b) && (b == c))

            {
                // ja pirmā mala vienāda ar otro un otrā vienāda ar trešo
               return TriangleType.Vienadmalu;
            }

            else if ((a == b) || (b == c) || (a == c))

            {
                // ja tieši divas malas vienādas
               return TriangleType.Vienadsanu;
            }

            else

            {
                // šeit nonāk tikai tad, kad pārbaudīti visi gadījumi,
                // atliek tikai dažādmalu trijstūris
                return TriangleType.Dazadmalu;
            }
        }
```

2.1. att. **Trijstūra veida noteikšanas funkcija valodā C#**

Atkarībā no padotajā masīvā esošajiem elementiem, funkcija atgriež trijstūra tipu vai paziņo par kļūdu, ja trijstūris nav iespējams no dotajiem malu garumiem. Funkcijai kā atgriežamais tips tiek izmantots numurēts (*enumeration*) tips *TriangleType*, kas sastāv no trijstūru veidu apzīmējumiem un kļūdas apzīmējuma.

Šai funkcijai var ģenerēt vienībtestus ar rīku *IntelliTest*. Uzģenerētie iespējamie vienībtesti ir parādīti 2.2. att.

IntelliTest Exploration Results - stopped

ProgramTest.triangleTest(Int32[] n ▸ Run 0 Warnings

9 4 20/20 blocks, 0/0 asserts, 14 runs

	maluGarumi	result	Summary / Exception	Error Message
1	null		NullReferenceException	Object reference not set to an instance of an object.
2	{}		IndexOutOfRangeException	Index was outside the bounds of the array.
3	{0}		IndexOutOfRangeException	Index was outside the bounds of the array.
4	{0, 0}		IndexOutOfRangeException	Index was outside the bounds of the array.
5	{0, 0, 0}	Kluda		
6	{1, 0, 0}	Kluda		
7	{1, 1, 0}	Kluda		
8	{1, 1, 1}	Vienadmalu		
9	{678, 678, 331}	Vienadsanu		
10	{800, 223, 514}	Kluda		
11	{38, 136, 174}	Kluda		
12	{196, 609, 802}	Dazadmalu		
13	{771, 402, 402}	Vienadsanu		

Error List IntelliTest Exploration Results

2.2. att. **IntelliTest darbināšanas rezultāts funkcijai, kas nosaka trijstūra veidu**

Šajā attēlā var redzēt, ka rīks ir uzģenerējis 13 testpiemērus ar dažādiem ieejas datiem, 9 testpiemēri ir bijuši veiksmīgi un 4 testpiemēri ir izraisījuši izņēmuma gadījuma iestāšanos (1. līdz 4. testpiemērs). Izsaucot šo funkciju ir iespējami *NullReferenceException* un *IndexOutOfRangeException* izņēmumgadījumi, ja padots *null* masīvs vai masīvs ar nepareizu elementu skaitu – trijstūra malām. Uzģenerētie testpiemēri parāda, ka šie īpašie gadījumi kodā nav apstrādāti. Programmētājs var arī norādīt rīkam, lai tas šāda veida vai kāda cita veida kļūdas gadījumus neuzskatītu par kļūdām. Lai to izdarītu ir nepieciešams papildināt parametrizētā testpiemēra kodu ar attiecīgām pieņēmuma (*assumption*) norādēm, tam jāizmanto klases *PexAssume* metodes. Lai labāk saprastu, kas ir parametrizēts vienībtests, kā veido pieņēmumus, tā kods ir parādīts 2.3. att.

```
namespace triangle.Tests
{
    [TestClass]
    [PexClass(typeof(Program))]
    [PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException), AcceptExceptionSubtypes = true)]
    [PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
    8 references
    public partial class ProgramTest
    {
        [PexMethod]
        7 references | 0/2 passing
        public Program.TriangleType triangle(int[] maluGarumi)
        {
            PexAssert.AreEqual(maluGarumi[0], maluGarumi[1]);
            PexAssert.AreEqual(maluGarumi[1], maluGarumi[2]);

            PexAssume.IsNotNull(maluGarumi);
            PexAssume.IsTrue(maluGarumi.Length == 3);
            Program.TriangleType result = Program.triangle(maluGarumi);
            return result;
            // TODO: add assertions to method ProgramTest.triangle(Int32[])
        }
    }
}
```

2.3. att. **Parametrizēta vienībtesta kods ar pieņēmumiem, novērtējumiem zarošanās programmai**

Pievienojot attēlā redzamos pieņēmumus un darbinot rīku vēlreiz tiks izlaisti attiecīgie testpiemēri, jo rīkam ir norādīts, ka malu garumu masīvs nekad nebūs *null* un vienmēr būs padots masīvs ar tieši 3 malu garumiem. Līdzīgi var veidot arī citus pieņēmumus, taču programmētājam jāatceras par iespējamām sekām, iespējamā kļūda netiek labota. Šajā vietā var izveidot arī novērtējumus ar klases *PexAssert* metožu palīdzību. Šajā parametrizētajā vienībtestā ir pievienoti arī divi novērtējumi *PexAssert.AreEqual*, kuros ir noteikts, ka visi malu garumi ir vienādi. Šie pievienotie novērtējumi un pieņēmumi ļauj iegūt interesantus izpētes rezultātus, skatīt 2.4. att.

IntelliTest Exploration Results - stopped

ProgramTest.triangle(Int32[] malu ▾ Run 0 Warnings

3 ✓ 4 ✗ 13/20 blocks, 0/0 asserts, 19 runs

	▲	maluGarumi	result	Summary / Exception	Error Message
✗	1	{771, 402, 402}		PexAssertFailedException	Expected '771', got '402'
✗	2	{678, 678, 331}		PexAssertFailedException	Expected '678', got '331'
✓	3	{0, 0, 0}	Kluda		
✓	4	{1, 1, 1}	Vienadmalu		
✓	5	{1073741825,...	Kluda		
✗	6	{771}		IndexOutOfRangeException	Index was outside the bounds of the array.
✗	7	null		NullReferenceException	Object reference not set to an instance of a

2.4. att. **Atkārtots IntelliTest darbināšanas rezultāts funkcijai, kas nosaka trijstūra veidu**

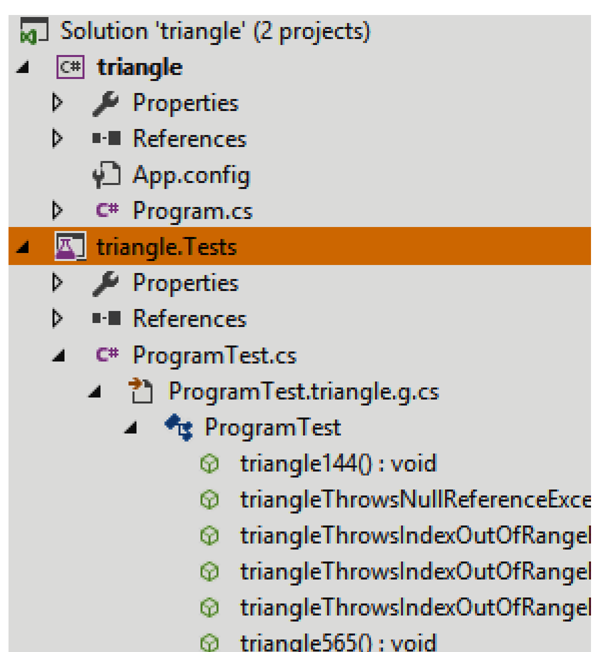
Atkārtoti darbināts, rīks atgriež citādus rezultātus. Pirmajā un otrajā testpiemērā ir redzams, ka tika sagaidīts masīvs ar vienādiem malu garumiem, testpiemērs ir kļūdains. Tiek ziņots par izņēmumgadījuma *PexAssertFailedException* iestāšanos, jo novērtējums nav izpildījies – padotajā parametra masīvā visu malu garumi nav vienādi, ko rīks arī parāda attiecīgajā kļūdas ziņojumā. Piektais testpiemērs ir atgriezis vērtību *Kluda*. Kā ieeja ir masīvs ar trīs vienādām malām, kuru katras garums 1073741825. Ir acīmredzams, ka rīks ir centies pārbaudīt kādu robežgadījumu. Šeit funkcijai vajadzētu atgriezt rezultātu, paziņot, ka tas ir vienādmalu trijstūris. Rezultāts *Kluda* tiek atgriezts, jo šajā valodā un šajā gadījumā saskaitot divas lielas *int* tipa vērtības tiek iegūts negatīvs skaitlis. Divu šādu malu summa ir 2147483650, taču maksimālā vērtība *int* tipa mainīgajam ir 2147483647, par 3 mazāka. Tādā veidā ir atklāta būtiska kļūda programmas darbībā, kura praksē visticamāk būtu palikusi nepamanīta, turklāt rīks ziņo, ka testpiemērs ir veiksmīgi iziets. Programmētājam testējot ir tāpat jāpārliecinās par testu korektumu un jāizvērtē visi testpiemēri. Interesanti, ka 6. testpiemērā ir padots masīvs ar vienu malu garumu un 7. testpiemērā – masīva vietā padota vērtība *null*. Autors uzskata, ka šiem testpiemēriem nevajadzēja parādīties iegūtajā rezultātā, jo tika noteikti attiecīgi pieņēmumi parametrizētajā vienībtestā. Tāpat ir redzams, ka uzģenerētie testpiemēri nesasniedz visus blokus, kas ir korekti, jo bija uzstādīti attiecīgi novērtējumi.

Visi *IntelliTest* ģenerētie testpiemēri, izmanto šāda veida (2.3. att.) parametrizēto vienībtestu, kas ir kā pamats uzģenerētajiem testiem ar konkrētām ievades vērtībām, katrs šāds testpiemērs ir daļa no daļējās klases *ProgramTest*. Parametrizētajā vienībtestā ir atribūts *[PexMethod]*, kas apzīmē parametrizētā vienībtesta sākumu, ir arī citu atribūtu anotācijas. Tās norāda uz klasi, kurā būs ģenerētie testpiemēri (klase *ProgramTest* ir *[TestClass]*), testējamās klases nosaukumu *([PexClass(typeof(Program))])* un pieļaujamos izņēmumgadījumus (*[PexAllowedExceptionFromTypeUnderTest…]*), kas netiks iekļauti ģenerētajos testpiemēros.

Rīks 2.2. att. ir izveidojis testpiemērus visiem trīs iespējamiem trijstūru veidiem un arī neiespējamiem trijstūriem – testpiemēri ar rezultātu *Kluda*. Apskatot šos testpiemērus var secināt, ka rīka algoritms ir izveidojis tādu ieejas datu kopu, kas ļauj izpildīt katru zarošanās nosacījumu vismaz vienu reizi funkcijas kodā 2.1. att. Tādā veidā apstiprinās 2.1 nodaļas sākotnējos punktos minētais, algoritms ir orientēts uz maksimāla pārklājuma sasniegšanu, šajā piemērā tiek apstaigāti visi iespējamie zarošanās gadījumi. Vēl var secināt, ka rīks pamana neapstrādātus izņēmumgadījumus un piemeklē atbilstošas vērtības, kas var izraisīt kļūdu programmas darbībā. Šāda veida nepilnības kodā ir mazāk pamanāmas, jo funkcija var strādāt korekti, taču tikai uz sagaidāmajiem, tipiskajiem ievadiem. Rīks spēj analizēt arī robežgadījumus. Praksē bieži netiek testēti ievadu speciālgadījumi, dažādi robežgadījumi.

Darba autors pārbaudīja arī rīka darbību ar *float* vērtībām, lai pārliecinātos par iespējamām problēmām, kas aprakstītas pie rīka darbības ierobežojumiem iepriekšējā darba punktā. Dotajā gadījumā ieejas parametri funkcijai, trijstūra malu garumi tika pārmainīti – no *int* uz *float* tipa mainīgajiem. Tika iegūts aptuveni tāds pats rezultāts, kāds ir parādīts 2.2. att., acīmredzamas *float* tipa vērtības rezultātos neparadījās. Taču rīks izveidoja brīdinājuma paziņojumus – paziņojums par peldošā punkta pārvēršanu un vienādību. Rīks ziņo, ka nevar ģenerēt ievadus un sasniegt attiecīgo kodu, kas ir aiz nosacījuma, kur tiek pārbaudīta skaitļu vienādība. Uzģenerētie testpiemēru ieejas parametri sastāv tikai no veseliem skaitļiem, taču tie iekšēji testpiemēros tiek konvertēti uz *float*.

Testpiemērus ir iespējams izvēlēties un saglabāt. Saglabājot šos testpiemērus tiek izveidots atsevišķs testēšanas projekts konkrētajā risinājumā (*Solution*), kur ir pieejams katra vienībtesta kods, skatīt 2.5. att.



2.5. att. **Testu projekts ar uzģenerētajiem testpiemēriem**

Katrs testpiemērs tiek veidots kā atsevišķa funkcija ar atgriežamo tipu *void*, kas nekādu informācija neatgriež. Savukārt funkcija, kuru izsauc katrs testpiemērs atgriež trijstūra veidu, skatīt 2.3. att. Funkciju nosaukumi tiek veidoti no projekta nosaukuma un nejauša skaitļa. Nosaukumu veidošanā tiek izmantots arī izņēmuma gadījuma nosaukums, kuru izraisa konkrētais testpiemērs.

2.1.4 *Programma ar vienkāršu for ciklu*

Lai noskaidrotu vai rīks spēj uzģenerēt datus arī programmām ar cikliem, tika izveidota neliela programma, kas nosaka vai simbols pieder simbolu virknei, funkcijas kods 2.6. att. Lai mazliet sarežģītu algoritma darbu un panāktu interesantākus rezultātus, tika pievienots papildus nosacījums ciklā, atgriezt vērtību "3", ja izpildās īpašs nosacījums, kas ir paslēpts dziļi ciklā.

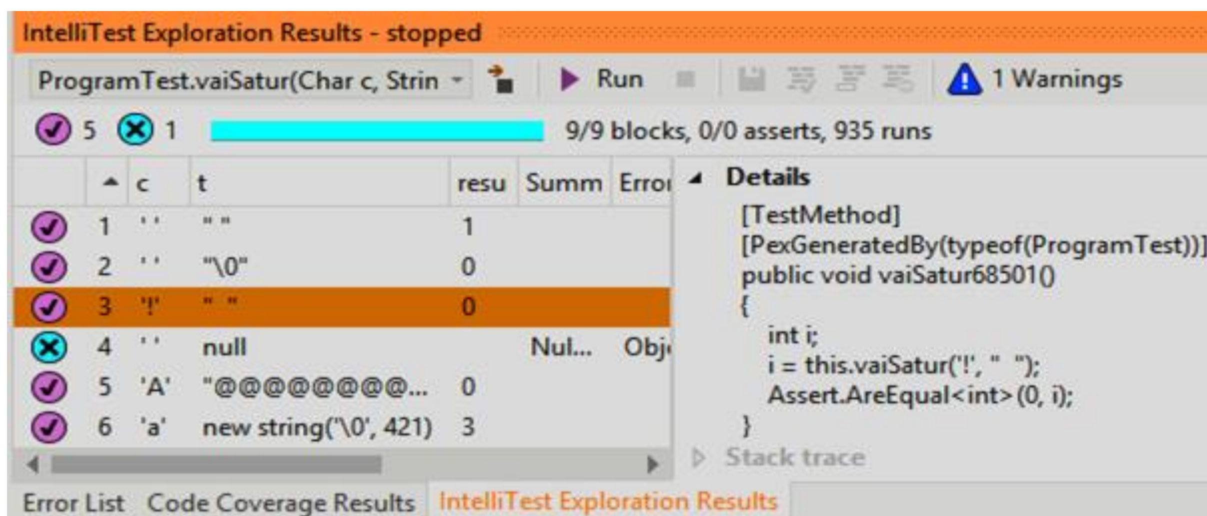
```
    public static int vaiSatur(char c, string t)
        {
            // cikls līdz virknes beigām
             for (int i=0; i<t.Length; i++)
             {
            // ja virknes i-tais elements sakrīt ar norādīto simbolu c
                 if (c == t[i])
                     return 1;
            // īpašais nosacījums dziļi ciklā
                 if ((i == 420) && (c=='a')) return 3;

             }
             return 0;
        }
```

2.6. att. **Simbola piederības virknei noteikšanas funkcija valodā C#**

Funkcija atgriež "1", līdzko padotais parametrs *c* sakrīt ar kādu no virknes elementiem, secīgi apstaigājot virkni un salīdzinot ar katru virknes elementu.

Funkcijai tiek ģenerēti vienībtestu testpiemēri, rezultāts parādīts 2.7. att.



2.7. att. **IntelliTest darbināšanas rezultāts funkcijai, kas nosaka simbola piederību virknei**

Šajā attēlā var redzēt, ka rīks ir uzģenerējis 6 testpiemērus ar dažādiem ieejas datiem, 5 testpiemēri ir bijuši veiksmīgi. Viens no testpiemēriem ir izraisījis izņēmuma gadījuma iestāšanos, kā virkne padots *null*, kas norāda uz to, ka speciālgadījums nav kodā apstrādāts.

Attēlā var arī redzēt uzģenerēto koda fragmentu 3. testpiemēram. Testpiemēra kodā ir redzams, ka ir ieviests *int* tipa mainīgais *i*, kurā glabā funkcijas rezultātu. Tālāk tiek izmantots novērtējums (*assertion*), klases *Assert* metode *AreEqual*. Tā sastāv no sagaidāmās vērtības, šajā gadījumā "0" un faktiskās vērtības, kas ir mainīgajā *i*.

IntelliTest darbināšanas rezultātā var novērot, ka ar īpašo nosacījumu ir saistīti 5. un 6. testpiemēri. 5. testpiemērā tiek izpildīta tikai daļa no nosacījuma, tiek padota 423 simbolu gara virkne, kas sastāv no daudziem simboliem "@", taču padotais uzģenerētais parametrs *c* ir vienāds ar "A", tāpēc atgriezts tiek "0" nevis "3". Pēdējais uzģenerētais testpiemērs izpilda šo īpašo nosacījumu, jo rīkam izdevās uzģenerēt attiecīgus ieejas parametrus – pietiekoši garu simbolu virkni un pareizu parametru *c*, kura vērtība "a".

Tāpat arī ir redzams viens brīdinājums. Dotajā gadījumā tas ir saistīts ar izpētes robežas sasniegšanu – pagājušas divas minūtes (noklusētā vērtība *Timeout=120*), jo bija jāizpēta apjomīgs cikls. Darbinot šo rīku pirmoreiz tika iegūts cits brīdinājums - noklusētās vērtības *MaxRunsWithoutNewTests=100* sasniegšana, programma darbināta 100 reizes, taču jauni testpiemēri netika izveidoti. Pirmajā rīka darbināšanas reizē tika izveidoti tikai pirmie četri testpiemēri, rīks arī ziņoja, ka ir nesasniegti programmas bloki (īpašais nosacījums). Lai izietu zaru, kur atrodas īpašais nosacījums, noklusētās robežas vērtību ir jānomaina uz *MaxRunsWithoutNewTests=1000* un jāpalaiž testu ģenerēšana, izpētes rīks atkārtoti. Ja to palielina tikai līdz 200, 400 reizēm, tad jauni testpiemēri netiks izveidoti, jo visi vienkāršie zari (atgriež "0" vai "1") jau ir apskatīti, ar to nepietiek, lai apskatītu īpašo nosacījumu (atgriež "3"). Kad ir veiktas aprakstītās izmaiņas, visi funkcijas bloki (9/9) ir sasniegti. Tāpat arī jāatzīmē, ka konkrētajam koda fragmentam pielietojot *IntelliTest* visi testpiemēri netiek uzģenerēti vienlaicīgi. Paiet noteikts laiks, lai uzģenerētu 5. un 6. īpašo testpiemēru, jo ir nepieciešamas daudzas izpildes, 1.-4. testpiemērs tiek izveidoti vienlaicīgi un salīdzinoši ātri pēc rīka palaišanas. No tā var secināt, ka rīks simulē cikla izpildi, katrā iterācijā piemeklē jaunus ievades parametrus, palielina virknes garumu un pārbauda atbilstošo nosacījumu.

Rīka algoritms ir izveidojis tādu ieejas parametru kopu ar simboliem *c* un virknēm *t*, kur rezultāts ir gan "0", gan "1", attiecīgi apskatīti abi iespējamie gadījumi simbola piederībai virknei. Var secināt, ka arī vienkāršām funkcijām ar cikliem rīks spēj uzģenerēt testpiemērus ar datiem. Šis piemērs demonstrē nepieciešamību pamainīt noklusētās izpētes robežas – laiku, darbināšanas reižu skaitu un citas vērtības, lai sasniegtu un uzģenerētu testpiemērus īpašiem nosacījumiem un zariem. Ar noklusētajām robežu vērtībām var neizdoties sasniegt visus programmas blokus. Nosacījuma atrašanās dziļi ciklā (salīdzinoši liels cikla skaitītājs) būtiski paildzina testpiemēru kopas izveidošanas laiku.

2.1.5 Programma ar objektiem

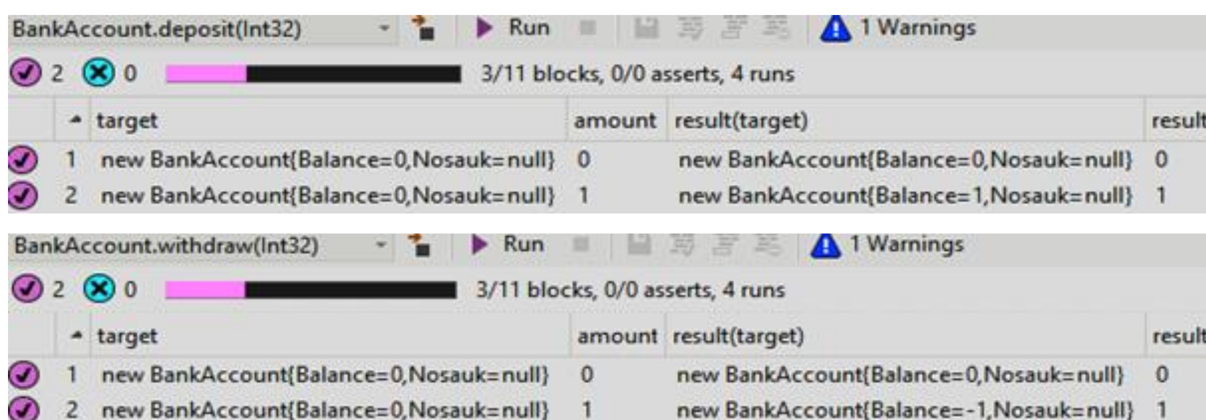
Tika izveidota programma, kas veido jaunus objektus – bankas kontus, kuru atribūti ir naudas atlikums un nosaukums. Bankas konta klasei ir divas būtiskas funkcijas – papildināt kontu un izņemt naudu no konta. Funkciju kods attēlots 2.8. att.

```
public int deposit(float amount)
        {
        // papildināt kontu var tikai ar daudzumu, kas lielāks par 0
            if (amount > 0)
            {
                balance = balance + amount;
                return 1;
            }
            return 0;
        }
public  int withdraw(float amount)
        {
        // no konta nevar izņemt negatīvu daudzumu un nulli
            if (amount > 0)
            {

                balance = balance - amount;
       // ja darbība ar kontu izdevās atgriež 1
                return  1;
            }
            else  return 0;
        }
```

2.8. att. **Bankas konta klases funkcijas konta papildināšanai un naudas izņemšanai valodā C#**

Ir ieviestas atgriežamās vērtības – “1”, ja darbība notikusi veiksmīgi un “0”, ja tika padots nekorekts naudas daudzums funkcijām. Katrai funkcijai tika uzģenerēti testpiemēri, skatīt 2.9. att.

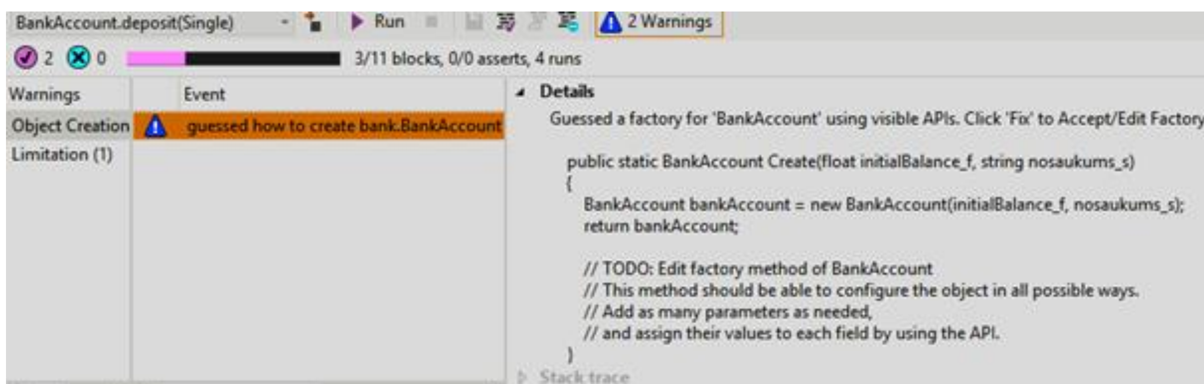


2.9. att. **IntelliTest darbināšanas rezultāts funkcijām, kuras veic darbības ar kontu**

Katrai funkcijai ir uzģenerēti divi veiksmīgi testpiemēri. Konta papildināšana ar naudas daudzumu, kas vienāds ar nulli, atgriež rezultātu “0”, kas ir korekti. Ja papildina kontu ar

vienu naudas vienību, tad konts veiksmīgi tiek papildināts un funkcija atgriež “1”. Līdzīgi testpiemēri ir uzģenerēti arī naudas izņemšanas funkcijai. Attiecīgā darbība ar kontu notiek tikai tad, kad ievades summa ir lielāka par nulli. Šie testpiemēri demonstrē minimālo testpiemēru kopu, kas apskata abus iespējamos gadījumus. Turklāt, šie testi parāda to, ka rīks spēj veidot vienkāršus objektus un testēt attiecīgās objektu klases metodes. Ja šķiet, ka rīks ģenerē objektus nekorekti, tad to var testu kodā pielabot – izmantot klašu rūpnīcu (*Factories*), kas tiks izveidota testēšanas projektā.

Rīks brīdina lietotāju, ka tas ir veicis pieņēmumus veidojot objektus, skatīt 2.10. att.



2.10. att. **IntelliTest brīdinājumi lietotājam par objektu veidošanu, ierobežojumiem**

Attēlā ir redzams ar kādiem parametriem un kā rīks ir veidojis klases *BankAccount* objektu, to var rediģēt, tad tiks izveidota jau minētā klašu rūpnīca. Tāpat arī rīks parāda brīdinājumus par ierobežojumiem, dotajā gadījumā tas ir saistīts ar peldošā punkta vērtību apstrādāšanu, jo šāda tipa mainīgais ir zarojuma nosacījumā – *amount* ir *float* tipa mainīgais. Apskatītajā piemērā parādās jau zināmie ierobežojumi, taču tie netraucē izveidot tādus testpiemērus, kurus izpildot tiek apstaigāti visi iespējamie zari konkrētajās funkcijās.

2.1.6 Programma ar pretrunu

Lai labāk izprastu rīka iespēju robežas un spriestu par tā algoritmu, tika izveidotas dažas papildus nelielas programmas, kuras demonstrē kādu īpašu gadījumu.

Tika izveidota programma, kurā ir pretruna zarošanās nosacījumos. Šīs funkcijas kods ir attēlots 2.11. att.

```
public static int function1(int a)
        {
            int y;
            if (a<5)
            {
                y = a;
          //pretruna nosacījumā, kods aiz šī nosacījuma nav sasniedzams
                if (y >= 5) return 3;
                return 1;
            }
            return 0;
        }
```

2.11. att. **Programmas funkcija ar pretrunīgiem zarošanās nosacījumiem**

Funkcijas kodā ir redzams, ka ieejas parametram, mainīgajam *a* ir jābūt mazākam par pieci, lai izpildītu pirmo zarošanās nosacījumu. Šī vērtība tiek piešķirta mainīgajam *y*, kuru tālāk izmanto iekšējā zarojuma nosacījumā. Iekšējā zarojuma nosacījumu apmierināt un izpildīt kodu, kas ir aiz tā, nav iespējams, mainīgā *y* vērtība jau sākotnēji ir mazāka par pieci un nevar būt vienlaicīgi vienāda vai lielāka par pieci. Programma var atgriezt rezultātu "1" vai "0", atgriezt "3" nav iespējams. Programmai tiek uzģenerēti testpiemēri, skatīt 2.12. att.

IntelliTest Exploration Results - stopped

ProgramTest.function1(Int32 a) Run 0 Warnings

2 0 4/5 blocks, 0/0 asserts, 2 runs

	a	result	Summary / Exception	Error Message
1	5	0		
2	0	1		

2.12. att. **IntelliTest darbināšanas rezultāts funkcijai ar pretrunu**

Divi izveidotie testpiemēri pārbauda visus reāli izpildāmos ceļus programmā. Nav izveidots testpiemērs, kas izpildītu pretrunīgo nosacījumu un atgrieztu vērtību "3", ja šāds testpiemērs tiktu uzģenerēts, tad tas liecinātu par būtiskām nepilnībām rīka darbībā. Lai gan rezultātā ir redzams, ka izpildīti tikai 4 no 5 programmas blokiem, tomēr rīks nepaziņo par to, ka funkcijā ir nesasniedzams kods pretrunīgu zarojuma nosacījumu dēļ. Šādā nelielā programmā tas ir viegli pamanāms, taču apjomīgākās programmās tas var būt grūtāk pamanāms. Ir iespējams arī apskatīt testpiemēru koda pārklājumu, taču tas jādara ar citu iebūvēto rīku, testpiemēru pārklājumu nosacīti norāda arī apskatītie bloki.

Rīks ir pietiekoši spēcīgs, lai izveidotu augstāk aprakstītos testpiemērus un neizveidotu pretrunīgus piemērus, taču konkrētajā gadījumā nespēj brīdināt testētāju un norādīt iemeslu tam, ka visi programmas bloki nav apskatīti.

2.1.7 *Programmas ar matemātiski sarežģītu funkciju*

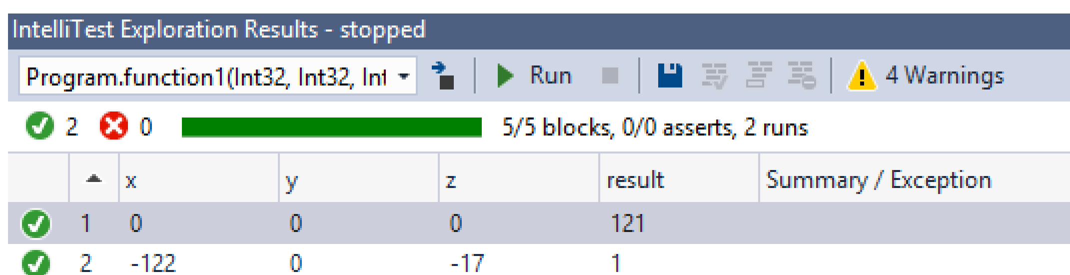
Tika izveidotas programmas, kas aprēķina kādas matemātiskas funkcijas, izteiksmes vērtību, dažādi varianti. Pirmajā programmas variantā tiek padoti trīs *int* tipa parametri, tiem tiek pieskaitīta, atņemta vērtība, mainīgais *y* tiek dalīts. Notiek vērtību kāpināšana ar to pašu vērtību. Funkcijas kods ir redzams attēlā 2.13. att.

```
public static int function1(int x, int y, int z)
        {
            x = x + 123;
            y = y / 10;
            z = z - 2;
            if (x == (Math.Pow(y,y) + Math.Pow(z, z))) return 1;
            else return x + y + z;

        }
```

2.13. att. **Matemātisku aprēķinu funkcijas pirmais variants**

Kāpinājumu summa tiek salīdzināta ar *x* vērtību, ja tā sakrīt tiek atgriezta vērtība "1". Testpiemēru ģenerēšanas rīks uzģenerē testpiemērus, skatīt 2.14. att.



2.14. att. **IntelliTest darbināšanas rezultāts 1. var. funkcijai, kas veic aprēķinus**

Ir acīmredzams, ka pirmais testpiemērs ir korekts. Otrajā testpiemērā *z* vērtība (-19) tiek kāpināta pati ar sevi. Lai izietu 2. testpiemēru, pēc ieejas parametriem var secināt, ka *Math.Pow(y,y) = Math.Pow(0,0) = 1* un *Math.Pow(z,z) = Math.Pow(-19,-19) = 0*. Pēdējais no tiem nav īsti korekti, taču šāds rezultāts ir saistīts ar pielietotajiem *int* tipa mainīgajiem, notiek apaļošana. Jāatzīmē, ka rīks brīdina par *Math.Pow* funkcijas izmantošanu, peldošā punkta konversiju, reizināšanu, vienādību. Ja šo programmu izpilda bez *Math.Pow* kvadrātsaknes aprēķināšanas funkcijas, nosacījumu aizstāj ar *if (x == (y + z))*, tad rīks arī izveido divus korektus testpiemērus, taču tad brīdinājumu nav.

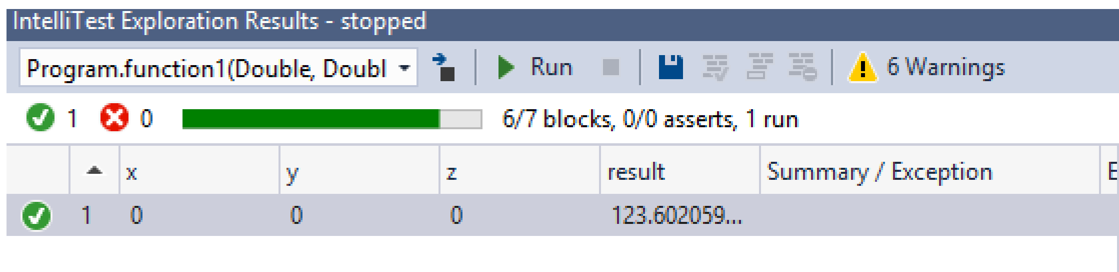
Cits funkcijas variants, tiek veikti sarežģītāki aprēķini, funkcija pārveidota un pievienotas iebūvētās funkcijas, kas aprēķina tangensu un decimālo logaritmu. Šīs funkcijas kods ir attēlā 2.15. att. Šajā funkcijā visi parametri ir *double* tipa.

```
public static double function1(double x, double y, double z)
        {
            x = x + 123;
            y = Math.Tan(y);
            z = Math.Log10(4);
            if (x == (Math.Pow(y,y) + Math.Pow(z, z))) return 1;
            else return x + y + z;

        }
```

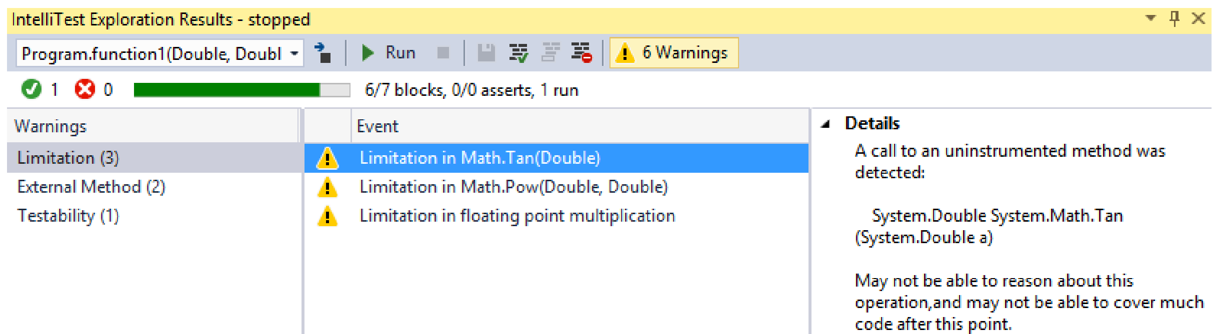
2.15. att. **Matemātisku aprēķinu funkcijas otrais variants**

Šādai programmai *IntelliTest* uzģenerē tikai vienu testpiemēru, darbības rezultāts redzams 2.16. att.



2.16. att. **IntelliTest darbināšanas rezultāts 2. var. funkcijai, kas veic aprēķinus**

Kā redzams nav apskatīti visi programmas bloki un ir vairāki brīdinājumi. Var redzēt, ka tangensa un logaritma vērtība ar ieejas parametru nulle ir aprēķināta korekti, rezultāts atbilst sagaidāmajam. Daži no brīdinājumiem lietotājam ir redzami attēlā 2.17. att.



2.17. att. **IntelliTest brīdinājumi par ierobežojumiem, ārēju metožu izsaukšanu, peldošo punktu**

Šie brīdinājumi kārtējo reizi norāda uz to, ka rīkam ir ierobežojumi ar peldošā punkta aritmētiku. Problēmas apstrādāt, instrumentēt *Math* iebūvētās funkcijas.

Var secināt, ka rīks uzģenerē testpiemēru kopu arī programmām, kur tiek izmantotas *Math* iebūvētās funkcijas. Taču iegūtie rezultāti nav pārliecinoši, ne visos gadījumos tiek uzģenerēta pilna testpiemēru kopa, ko demonstrē augstāk aprakstītie piemēri. Tāpat arī ir

jāņem vērā uzskaitītie brīdinājumi. Darba autors uzskata, ka šādās matemātiski sarežģītās funkcijās nevar paļauties uz šo testpiemēru ģenerēšanas rīku augstāk minēto iemeslu dēļ.

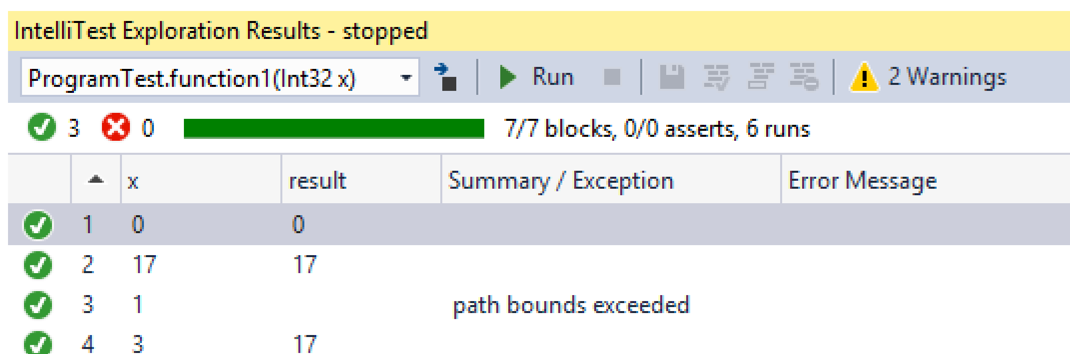
2.1.8 Programma ar iecikošanos

Ar doto programmu tiek pārbaudīts kā rīks reaģē, ja kodā ir iecikošanās. Apskatāmās funkcijas kods 2.18. att.

```
public static int function1(int x)
        {
        Outer:
            int value = x;
            switch (value)
            {
                case 1:
                    {
                        goto Outer;
                    }
                case 17:
                    {
                        return 17;
                    }
                case 3:
                    {
                        x = 17;
                        goto Outer;
                    }
                default:
                    return 0;
        }  }
```

2.18. att. **Programmas funkcija ar iecikošanos**

Funkcija kā parametru saņem *int* tipa mainīgo, kas tiek piešķirts mainīgajam *value*. Atkarībā no tā vērtības, izpildās kāds no *case* zarojumiem. Ja šī vērtība ir "1" notiek iecikošanās. Programmai tika ģenerēti testpiemēri, rezultātu skatīt 2.19. att.



2.19. att. **IntelliTest darbināšanas rezultāts funkcijai ar iecikošanos**

Ir iegūti 4 testpiemēri un apskatīti visi programmas bloki. Pirmais no tiem pārbauda gadījumu, kad vērtība neatbilst nevienam no *case* gadījumiem un nostrādā *default* nosacījums, kas atgriež nulli. Otrais un ceturtais testpiemērs ir saistīti, jo pirmajā gadījumā uzreiz izpildās nosacījums, kur tiek atgriezts skaitlis "17", bet otrajā vispirms tiek izpildīts *case 3* nosacījums, kur uzstāda *x* vērtību un tikai nākošajā iterācijā izpildās nosacījums, kas atgriež vērtību "17". Interesantākais no šiem testpiemēriem – 3. testpiemērs, kur notiek ieciklošanās. Rīks ziņo, ka ceļa robežas ir pārsniegtas, ir arī brīdinājumi. Pie paziņojuma ir paskaidrots, ka programmas analīze tika pārtraukta, jo konkrētais testpiemērs izpildījās pārāk ilgi. Zars tika izpildīts 5000 reižu, paziņojumā testētājs tiek aicināts pārliecināties vai kodā nav ieviesies bezgalīgais cikls. Tiek piedāvāts arī palielināt izpētes robežu iekšējo nosacījumu *MaxBranches*, piemēram, līdz 20000 reizēm, taču tas situāciju nemaina. Līdzīgs saturs ir arī brīdinājumiem – izpēte ir apstādināta, jo sasniegts maksimālais noteiktais zarojuma izpildes skaits. Var secināt, ka rīks nav pilnīgi pārliecināts par bezgalīgā cikla esamību programmā, jo piedāvā palielināt izpildes reižu skaitu, kas neko konkrētajā gadījumā nemainīs. Taču, ņemot vērā, ka ir pārsniegtas izpildes robežas, tas spēj pamanīt ieciklošanos un beigt darbu.

Šoreiz testētājam tiek paziņots, ka programmā ir iespējama problēma – ieciklošanās. Līdzīgs paziņojums par iespējamu problēmu netika novērots pretrunu programmā, kas varētu palīdzēt izlabot kļūdaino programmas daļu ātrāk.

2.1.9 Darbības robežas

Izpētes rīks beidz darbu, ja ir sasniegts kāds nosacījums, piemēram, sasniegti visi programmas bloki vai zari. Lai tas neieciklotos un vienmēr beigtu darbu, rīkam ir dažādas darbības robežas, piemēram, veiktais izpētes laiks sekundēs vai darbināšanas reižu skaits, funkciju izsaukumu skaits. Visas uzskaitītās robežas tiek izmantotas tad, kad tiek veikta testējamās programmas izpēte, lai ģenerētu ieejas datu kopu. Vienai un tai pašai programmai var sasniegt dažādas izpētes robežas. Piemēram, apskatītajā programmā ar vienkāršu ciklu tiek sasniegta izpildes reižu *MaxRunsWithoutNewTests* robeža un vēlāk arī *Timeout* robeža.

Šiem lielumiem ir noklusētās vērtības, taču lietotājs tās var rediģēt un mainīt. Vēl viens iemesls šādai nepieciešamībai – nepietiekama testpiemēru kopas izveide, kādu zarojumu, programmas bloku neizpildīšana.

Visas darbības robežas tiek definētas parametrizētā vienībtesta koda sākumā, iespējamās uzstādāmās atribūtu vērtības skatīt 2.20. att.

```
[PexMethod(MaxRunsWithoutNewTests = 1001, MaxConditions = 1000, )]
{
}
PexMethodAttribute(Properties: [Categories = string], [DisableObservableAssertions = bool],
            [DisablePostAnalysis = bool], [IncludeNonFinalSegmentCoverage = bool], [MaxBranches = int], [MaxCalls = int],
            [MaxConditions = int], [MaxConstraintSolverMemory = int], [MaxConstraintSolverTime = int], [MaxExceptions = int],
            [MaxExecutionTreeNodes = int], [MaxRuns = int], [MaxRunsWithoutNewTests = int], [MaxRunsWithUniquePaths = int],
            [MaxSequenceLength = int], [MaxStack = int], [MaxWorkingSet = int], [NoSoftSubstitutions = bool], [NotReproducible = bool],
            [ObserveChoices = bool], [Owner = string], [Priority = int], [SupportedPlatform = PexPlatform], [TestClassName = string],
            [TestEmissionBranchHits = int], [TestEmissionFilter = Microsoft.Pex.Framework.Settings.PexTestEmissionFilter],
            [TestExcludePathBoundsExceeded = bool], [ThreadApartmentState = Microsoft.Pex.Framework.Settings.PexApartmentState], [Timeout = int])
DisablePostAnalysis: A named parameter that specifies whether to disable the post analysis.
```

2.20. att. **Uzstādāmās izpētes robežas un citi atribūti parametrizētam vienībtestam**

Šiem robežu atribūtiem parasti ir nosakāmas *int* tipa vērtības. Šādi atribūti, kā redzams attēlā, ir vairāki, tas ļauj testētājam precizēt un uzlabot uzģenerēto testpiemēru kopu. Vairums no apskatītajām programmām iepriekšējos punktos pietiek ar noklusētajām vērtībām, kas katram atribūtam ir noteikta.

Daži no šiem atribūtiem tieši ietekmē nosacījumu risinātāja (vienādību un nevienādību risinātāja) darbību, piemēram, ar atribūtiem *MaxConstrainSolverTime* un *MaxConstraintSolverMemory*, attiecīgi tiek palielināta tikai nosacījumu risināšanai atvēlētais laiks un atmiņa.

2.2 EvoSuite

EvoSuite [11] ir bezmaksas rīks, kas ģenerē testpiemērus programmām valodā *Java*. Rīkam ir vairākas versijas – pieejams kā spraudnis populārām izstrādes vidēm, darbināms arī no komandrindas. Darba izstrādes gaitā tika izmēģināta no komandrindas darbināmā rīka versija, *IntelliJ IDEA* izstrādes vides spraudnis un *Eclipse* izstrādes vides spraudnis. Darba autors novēroja problēmas *IntelliJ IDEA* spraudņa darbībā, tika atgriezti kļūdas ziņojumi un netika ģenerēti testpiemēri. Komandrindas versija un *Eclipse* spraudnis darbojās veiksmīgi, taču bija nepieciešams laiks un dažādas darbības, lai to iedarbinātu. Rīka mājas lapā esošā dokumentācija ir nepilnīga.

Darba autors pielieto šo rīku, lai iegūtu testpiemēru kopu dažādām programmām. Tas tiek pielietots tām pašām programmām, kuras ir aprakstītas 2.1 *IntelliTest* apakšnodaļā. Lai to paveiktu, visas programmas ir pārveidotas valodā *Java*. Tiek saglabāts programmas algoritms, atšķirības kodā ir nelielas, sintaktiskas.

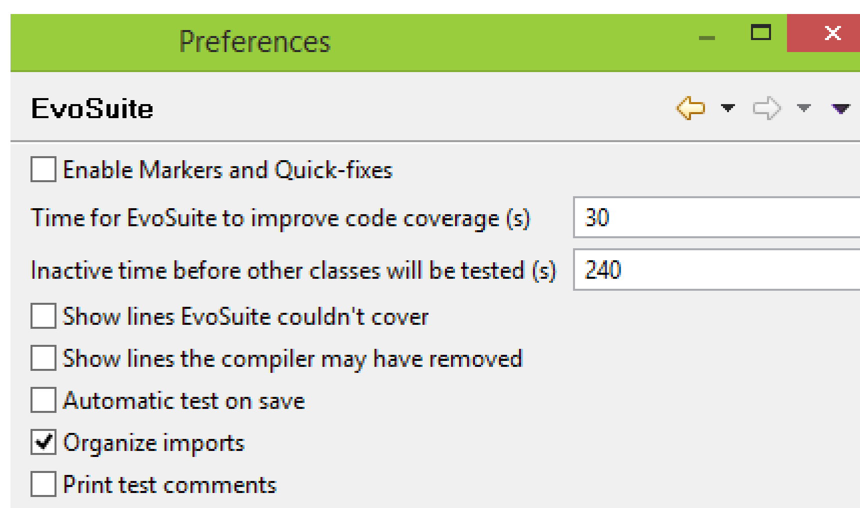
Šīs apakšnodaļas turpmākajos punktos tiek aprakstīti un analizēti iegūtie rīka pielietojuma rezultāti katrai programmai.

2.2.1 *EvoSuite darbības princips*

Rīks izmanto meklēšanas bāzētu tehniku testpiemēru ģenerēšanai, tiek pielietots ģenētiskais algoritms [12]. Tā pamatā ir programmas darbināšana ar nejauši ģenerētiem sākotnējiem ieejas parametriem, testpiemēru komplektu. Vairāku iterāciju rezultātā notiek šo testpiemēru attīstība. Šo procesu vada noteikta atbilstības funkcija, kas nosaka cik tuvu risinājumam – kādam nosacījuma zaram programmā – ir konkrētie ieejas parametri. Testpiemēru kopa tiek pakāpeniski attīstīta, notiek labāko elementu atlase, tie tiek apvienoti. Tas tiek atkārtots, līdz tiek apstaigāti visi programmas zari.

2.2.2 *Konfigurācijas iespējas*

Rīku ir iespējams konfigurēt, konfigurēt var gan komandrindas rīka versiju norādot papildus parametrus, gan arī *Eclipse* spraudņa versiju, skatīt 2.21. att..



2.21. att. **Konfigurācijas iespējas Eclipse spraudņa versijā**

No komandrindas ir iespējams uzstādīt daudzus parametrus, rīks tiek piedāvāts kā *Java* arhīvs – *EvoSuite.jar*. Rīka *Eclipse* spraudņa versijā konfigurācijas iespēju ir ļoti maz.

Turpmākajos punktos, kur ir ģenerēti testpiemēri reālām programmām, tiek izmantoti konfigurācijas noklusētie iestatījumi, ja tas nav norādīts citādāk.

2.2.3 *Programma ar zarošanos*

Rīks *EvoSuite* tiek pielietots programmai, kas nosaka trijstūra veidu pēc to malu garumiem. Rīks izveido 16 testpiemēru kopu ar dažādiem trijstūra malu garumiem, visiem iespējamiem trijstūra veidiem. Pirmie trīs testpiemēri ir attēloti 2.22. att.

```
@Test
public void test00()  throws Throwable  {
    Triangle triangle0 = new Triangle(251, 3744, 3537);
    String string0 = triangle0.triangleType();
    assertEquals("dazadmalu", string0);
}
@Test
public void test01()  throws Throwable  {
    Triangle triangle0 = new Triangle(3689, 3689, 1);
    String string0 = triangle0.triangleType();
    assertEquals("vienadsanu", string0);
}
@Test
public void test02()  throws Throwable  {
    Triangle triangle0 = new Triangle(2375, 33, (-1654));
    String string0 = triangle0.triangleType();
    assertEquals("nav iespejams", string0); }
```

2.22. att. **EvoSuite ģenerētie testpiemēri programmai, kas nosaka trijstūra veidu**

Pirmajā testpiemērā rīks ir izveidojis jaunu trijstūri ar tādiem malu garumiem, kas atbilst dažādmalu trijstūrim – izpildās attiecīgais zarojums kodā. Novērtējums *assertEquals("dazadmalu", string0);* salīdzina sagaidāmo vērtību un funkcijas, kas nosaka trijstūra veidu, atgriezto vērtību. Līdzīgi tiek veidots arī otrais testpiemērs, taču tiek ģenerēti tādi malu garumu parametri, kas atbilst vienādsānu trijstūra definīcijai. Trešais testpiemērs apskata gadījumu, kad trijstūris nav iespējams. Tiek definēts trijstūris, kuram viens no malu garumiem ir negatīvs lielums. Tiek izpildīts attiecīgs novērtējums *assertEquals*.

Pārējo ģenerēto testpiemēru uzbūve ir vienāda ar uzskaitīto testpiemēru uzbūvi. No 16 testpiemēriem 9 pārbauda neiespējama trijstūra gadījumus, 4 – vienādsānu, 2 – dažādmalu. Tikai viens testpiemērs apskata vienādmalu trijstūra gadījumu. Šis rezultāts var atšķirties, katru reizi darbinot rīku no jauna. Nevienā testpiemērā netika pārbaudīti izņēmumgadījumi, piemēram, kad kāds no parametriem ir *null* vērtība. Ir uzģenerēti daži īpaši testpiemēri, piemēram, visi malu garumi vienādi, taču negatīvi. Tāpat arī visi malu garumi, kas ir garumā viena vienība.

Autors uzskata, ka ir nepieciešams arī robežas testpiemērs, kur visu malu garums ir nulle. Testpiemēros nav apskatīti arī robežu gadījumi pie maksimālajām un minimālajām vērtībām, kādi ir iespējami *int* tipa mainīgajiem.

Iegūtā testpiemēru kopa apskata visus trijstūru veidu zarojumus, taču neapskata dažus īpašus gadījumus, izņēmumgadījumus.

2.2.4 Programma ar vienkāršu *for* ciklu

Simbola piederības virknei noteikšanas programmai ar vienkāršu *for* ciklu ir iespējamas divas zarošanās, kas atrodas ciklā. Viens no zarojumiem atrodas dziļi ciklā un ir ar specifisku nosacījumu. Programma atgriež "1", ja simbola piederība virknei apstiprinās, atgriež "3", ja iestājies specifiskais nosacījums, kas atrodas dziļi ciklā (padota gara virkne un meklējamais simbols ir "a").

EvoSuite rīks programmai ar ciklu kopumā uzģenerē 5 testpiemērus. Pirmie trīs no tiem ir attēloti 2.23. att.

```
@Test
public void test0()  throws Throwable  {
    // Undeclared exception!
    try {
      cycle1.vaiSatur('}', (String) null);
      fail("Expecting exception: NullPointerException");

    } catch(NullPointerException e) {
       //
       // no message in exception (getMessage() returned null)
       //
       assertThrownBy("cycle1", e);
    }
}

@Test
public void test1()  throws Throwable  {
    int int0 = cycle1.vaiSatur('d', "@#+@IE:E':Td");
    assertEquals(1, int0);
}

@Test
public void test2()  throws Throwable  {
    int int0 = cycle1.vaiSatur('M', ",(p3 YiARa_N:1n+s");
    assertEquals(0, int0);
}
```

2.23. att. **EvoSuite ģenerētie testpiemēri programmai, kas nosaka simbola piederību virknei**

Pats pirmais no tiem norāda uz to, ka nav apstrādāts izņēmuma gadījums – simbolu virkne ir *null*. Uzģenerētajā testpiemēru kopā vajadzētu būt arī līdzīgam testpiemēram ar meklējamo simbolu, arī tas var būt *null*, taču dotajā gadījumā rīks tādu nav uzģenerējis.

Otrajā testpiemērā ir izsaukta funkcija *vaiSatur* ar ģenerētu simbolu virkni un simbolu, kas šai virknei pieder. Attiecīgi tas tiek novērtēts, sagaidāmā vērtība "1", jo izpildās

attiecīgais nosacījums, kuram ir jāatgriež vērtība “1”. Līdzīgi nākošajā piemērā tiek pārbaudīts pretējais gadījums, kad simbols virknei nepieder. Tiek sagaidīta vērtība “0”, kas dotajā gadījumā ir korekti. Pārējie ģenerētie testpiemēri, skatīt darba 2. pielikumu, neuzrādīja saturīgus rezultātus, jo neizsauca funkciju *vaiSatur*.

Būtiski ir tas, ka šoreiz netika izveidota testpiemēru kopa, kura sasniedz pilnu zaru pārklājumu, īpašais nosacījums, kas atrodas ciklā, netika ģenerētajos testpiemēros apskatīts. Autors pieļauj, ka rīkam pietrūkst resursu, nav uzstādīts kāds parametrs, lai uzģenerētu pietiekoši garu simbolu virkni. Dotajā gadījumā tai ir jābūt 420 simbolu garai, lai sasniegtu attiecīgo zaru. Iespējams, ka to var norādīt kādā no uzstādāmajiem parametriem komandas rindas rīka versijā, konfigurācijā *Eclipse* spraudnim tas nav iespējams. Līdzīga situācija tika novērota veicot šīs programmas testēšanu ar *IntelliTest* rīku, kur bija jāpalielina programmas darbināšanas reižu skaita parametrs, lai īpašā nosacījuma zars tiktu apskatīts testpiemēros.

2.2.5 Programma ar objektiem

Darbinot *EvoSuite* rīku programmai, kur tiek veidots bankas konta objekts, rīks sākotnēji uzģenerēja tikai divus testpiemērus, kuri bija nederīgi – izpildot šos testus netika pārbaudīta programmas galvenā funkcionalitāte, kas ir konta papildināšana un naudas izņemšana no konta. Jāatzīmē, ka iepriekš aprakstītajā piemērā, programmai par trijstūriem, rīks spēja veiksmīgi uzģenerēt testam nepieciešamos trijstūra objektus ar konkrētām vērtībām. Pirmie testpiemēri programmai ar bankas objektiem uzģenerētie ir redzami 2.24. att.

```
@Test
  public void test0() throws Throwable  {
    String[] stringArray0 = new String[9];
    Bank.main(stringArray0);
  }

@Test
  public void test1() throws Throwable  {
    Bank bank0 = new Bank();
  }
```

2.24. att. **Nesaturīgi EvoSuite ģenerētie testpiemēri programmai ar bankas objektiem**

Pirmajā testpiemērā ir redzams, ka tiek definēts *String* tipa masīvs *stringArray0* ar 9 elementiem, kas tiek padots *main* funkcijai. Dotais testpiemērs ir bezjēdzīgs, jo nav izsauktas funkcijas, kas veic darbības ar kontu – *deposit*, *withdraw*.

Otrajā testpiemērā var redzēt, ka rīks cenšas izveidot jaunu bankas klasi. Dotais testpiemērs arī ir bezjēdzīgs, jo nepārbauda klasēm definētās metodes darbībai ar kontu.

Uzģenerēt saturīgus piemērus tomēr izdodas – rīkam ir precīzi jānorāda, kurai klasei tieši ģenerēt testpiemērus. Dotajā gadījumā tā ir klase *BankAccount*, kas atrodas atsevišķā klases failā. Tika uzģenerēti 11 testpiemēri, pirmie no tiem ir redzami 2.25. att.

```
@Test
  public void test00()  throws Throwable  {
      BankAccount bankAccount0 = new BankAccount(0.0F, "");
      String string0 = bankAccount0.getNosauk();
      assertEquals("", string0);
  }

  @Test
  public void test01()  throws Throwable  {
      BankAccount bankAccount0 = new BankAccount((-1514.305F), "]/");
      int int0 = bankAccount0.withdraw((-1514.305F));
      assertEquals(0.0F, bankAccount0.getBalance(), 0.01F);
      assertEquals(0, int0);
  }

  @Test
  public void test02()  throws Throwable  {
      BankAccount bankAccount0 = new BankAccount(21.8268F, "E-,");
      int int0 = bankAccount0.deposit(0.0F);
      assertEquals(0, int0);
      assertEquals(21.8268F, bankAccount0.getBalance(), 0.01F);
  }
```

2.25. att. **Saturīgi EvoSuite ģenerētie testpiemēri programmai ar bankas objektiem**

Pirmajā testpiemērā *test00* ir redzams, ka tiek izveidots jauns, tukšs konts ar tukšu nosaukumu. Tiek izsaukta funkcija *getNosauk*, kas atgriež šī konta nosaukumu. Novērtējuma funkcija *assertEquals* pārbauda vai paredzamā vērtība sakrīt ar iegūto vērtību.

Testpiemērā *test01* ir izveidots konts ar konkrētām vērtībām – negatīvu konta atlikumu un konta nosaukumu “]/”. Testpiemērs apraksta gadījumu, kad tiek mēģināts izņemt no konta negatīvu lielumu. Tiek veikti novērtējumi. Bankas konta atlikumam pēc minētās darbības ir paredzēts būt nullei, naudas izņemšanas operācijai nevajadzētu izpildīties, tiek sagaidīta atgriežamā vērtība – nulle. Pirmajā novērtējumā ir trīs parametri, taču īpaša uzmanība jāpievērš pēdējam no tiem – *0.01F*. Šis parametrs norāda uz pieļaujamo novirzi, par kādu var atšķirties paredzamā vērtība no sagaidāmās. Dotajā gadījumā šāda novirze nav pieļaujama, tai vajadzētu būt *0.00F*. Reālā sistēmā tas nozīmētu, ka kontā esošā naudas daudzums var būt par 1 eirocentu mazāks vai lielāks, kas nav pieļaujami.

Nākošajā testpiemērā tiek definēts konts ar pozitīvu naudas lielumu un konkrētu nosaukumu. Tiek palielināts konta atlikums, pieskaitīta nulle. Tiek novērtēts vai darbība ir izdevusies – sagaidāmā vērtība ir nulle, jo ar šādu lielumu kontu nav iespējams papildināt.

Tāpat arī tiek novērtēts konta atlikums pēc veiktās konta papildināšanas – "*assertEquals(21.8268F, bankAccount0.getBalance(), 0.01F).* Šeit atkal ir novērojams, ka rīks ir izvēlējies nekorektu novirzi trešajā parametrā, kas nav pieļaujami reālā sistēmā.

Var secināt, ka rīks ir sasniedzis maksimālu pārklājumu – ir izsauktas visas pamatdarbības ar kontu, funkcijas atlikuma un konta nosaukuma uzstādīšanai, arī citas funkcijas, kas atgriež konta atlikumu un nosaukumu, ir pārbaudītas.

Turpmākajos testpiemēros, kas pilnā apjomā pieejami darba 2. pielikumā, ir redzami līdzīgi testpiemēri, kas pārbauda konta definēšanu un darbības ar to. Nevienā no šiem testpiemēriem netiek pārbaudīta pamatoperāciju kombinācija, piemēram, konta papildināšana un naudas izņemšana vienā un tajā pašā testpiemērā. Tāpat jāuzsver, ka šos testus ir nepieciešams rediģēt un uz tiem nevar pilnībā paļauties, jo tika automātiski ģenerēts novirzes parametrs, par kuru jau tika aprakstīts augstāk.

2.3 Citi rīki

Darbā sīkāk netiek apskatīti citi, līdzīgi testpiemēru ģenerēšanas rīki, kas ir uzskaitīti darba 1. pielikumā. To ir daudz, tie ir paredzēti dažādām programmēšanas valodām. Autors šajā darbā liek uzsvaru un apskata rīkus, kas darbojas ar populārām objektorientētām programmēšanas valodām – *Java* un *C#*.

No uzskaitītajiem rīkiem ir jāatzīmē *AgitarOne* [13] vienībtestu ģenerēšanas rīks, kuru darba autoram neizdevās izmēģināt praksē, jo tas ir maksas, 30 dienu izmēģinājuma versija pieejama tikai uzņēmumiem un nav paredzēta studentiem. Šo iemeslu dēļ autors nolēma apskatīt alternatīvu bezmaksas rīku *EvoSuite* darba 2.2 apakšnodaļā. Autors pieļauj, ka *AgitarOne* rīks ir funkcionāli spēcīgāks un iegūtie rezultāti būtu interesantāki par *EvoSuite* iegūtajiem rezultātiem.

3. APSKATĪTO RĪKU SALĪDZINĀJUMS UN IESPĒJAMIE UZLABOJUMI

Nodaļā tiek salīdzināti un novērtēti apskatītie rīki, tas tiek darīts balstoties uz iegūtajiem rezultātiem, kas aprakstīti iepriekšējā nodaļā. Tāpat arī tiek spriests par iespējamiem uzlabojumiem tiem, to algoritmiem.

3.1 IntelliTest un EvoSuite salīdzinājums, iegūto rezultātu salīdzinājums

Lai gan rīki ir paredzēti dažādās programmēšanas valodās rakstītām programmām – *Java* un *C#*, tiem ir līdzības, taču galvenā atšķirība ir metode, kuru tie izmanto, lai ģenerētu testpiemērus. *IntelliTest* rīks izmanto dinamisko simbolisko izpildi, taču *EvoSuite* izmanto meklēšanas metodes.

Viens no iespējamiem kritērijiem pēc kura salīdzināt šos rīkus ir to ātrdarbība, taču autors uzskata to par nebūtisku kritēriju, tāpēc tas netiek veikts šajā darbā. Ir publikācijas, kur tas jau ir veikts, apskatīta dažādu testpiemēru ģenerēšanas rīku ātrdarbība. Daudz būtiskāks kritērijs ir iegūtās testpiemēru kopas kvalitāte, kas ir saistīta ar iegūto testpiemēru koda pārklājumu.

Ģenerētie testpiemēri trijstūra veida noteikšanas programmai abiem mūsdienu rīkiem ir līdzīgi, taču *EvoSuite* izveido testpiemērus, kur nav apskatīti daži būtiski izņēmumgadījumi. Nav testpiemēra ar ieejas parametru *null*, tāpat arī testpiemēra, kur ir padots nepietiekams skaits trijstūra malu parametru. Interesanti, ka *EvoSuite* izveido testpiemērus arī ar negatīviem malu garumiem, kombinē tos ar pozitīviem malu garumiem, šādus testpiemērus neizveido *IntelliTest*, tie varētu atklāt būtisku problēmu kodā. Jāatzīmē, ka abu rīku ģenerētie testpiemēri sasniedz zarojumu pārklājumu, jo testpiemēri apskata visus iespējamos trijstūru veidus.

Programmai ar vienkāršo ciklu rīki uzģenerē dažādus rezultātus. Šoreiz *EvoSuite* ģenerētie testpiemēri nesasniedz visus programmas zarus. Netiek sasniegts īpašais nosacījums dziļi ciklā. Lai to sasniegtu nepieciešams uzģenerēt garu simbolu virkni, jāizpilda cikls daudz reižu. Jāpiebilst, ka arī *IntelliTest* testpiemēri nesasniedza to pirmajā darbināšanas reizē – bija nepieciešams palielināt darbināšanas reižu skaitu. Tāpat arī tikai divi no četriem *EvoSuite* testpiemēriem ir saturīgi – pārbauda simbola piederību virknei. Šie testpiemēri nepārbauda dažādus īpašus gadījumus un izņēmumgadījumus, kas tiek pārbaudīts ar *IntelliTest* ģenerētajiem testpiemēriem – padota *null* virkne, tukša virkne, tukšs meklējamais simbols un līdzīgi testpiemēri.

Dažādi rezultāti ir iegūti ģenerējot testpiemēru komplektus programmai ar bankas objektiem. Rīks *IntelliTest* ir izveidojos tikai divus testpiemērus katrai darbībai (naudas izņemšana un konta papildināšana) ar kontu, *EvoSuite* ir izveidojis trīs testpiemērus katrai minētajai darbībai. Taču pats būtiskākais ir tas, ka visi *IntelliTest* testpiemēri izveido tukšu bankas kontu (*Balance=0*). Rīks *EvoSuite* gandrīz visos testpiemēros, kur tiek pārbaudītas darbības ar kontu, izveido netukšus kontus, arī izveido nejaušu virkni konta nosaukumam. Lai gan konta darbību funkcijas ir testpiemēros apskatītas abos rīkos, tomēr kvalitatīvāku un reāliem apstākļiem tuvinātāku testpiemēru kopu ir izveidojis *EvoSuite*. Arī uzģenerētās *float* tipa vērtības ir atbilstošākas šim datu tipam. Rīka ģenerētos testpiemērus ir jāpielāgo, jo testpiemēros ir izmantota nepieļaujama novirze.

Kopumā var secināt, ka *EvoSuite* tikai dažos gadījumos uzģenerē kvalitatīvākus testpiemērus. Tas nosacīti ir redzams programmai ar bankas objektiem, ir uzģenerēti arī atbilstošas *float* tipa vērtības. Šis rīks reti apskata izņēmumgadījumus – piemēram, ja padota *null* vērtība. Rīks *IntelliTest* salīdzinoši vairāk veido šādus testpiemērus, kur ieejas parametrs ir *null* vērtība. Abiem rīkiem problemātiski ir lieli cikli, nepieciešams pielāgot noklusētās izpētes robežas *IntelliTest* gadījumā. Iespējams, ka arī *EvoSuite* uzģenerētu testpiemērus visiem programmas ceļiem programmai ar vienkāršu *for* ciklu, ja to darba autoram būtu izdevies pielāgot.

Rīks *EvoSuite* tiek apskatīts darbībā tikai ar trīs programmām. Potenciālie uzlabojumi šim rīkam apakšnodaļā 3.3 tiek aprakstīti vispārīgāk par *IntelliTest* uzlabojumiem, jo darba autors, ņemot vērā iegūtos rezultātus, uzskata *IntelliTest* par pilnīgāku rīku.

3.2 Iespējamie uzlabojumi IntelliTest

Šajā apakšnodaļā tiek spriests par uzlabojumiem, kas varētu padarīt *IntelliTest* ģenerēto testpiemēru kopu kvalitatīvāku. Šo uzdevumu būtiski sarežģī tas, ka rīks nav atvērtā koda, ir pieejami dažādi materiāli, taču tie apraksta rīku vispārīgi, kas ir minēts arī kā pamatojums nepieciešamībai veikt testpiemēru ģenerēšanu dažādām programmām sākot ar 2.1.3 darba punktu.

3.2.1 IntelliTest vispārīgi, tehniski uzlabojumi

Testpiemēru ģenerēšanas rīks *IntelliTest* sastāv no vairākām komponentēm, to sadarbību var uzskatīt arī par šī rīka vispārēju algoritmu. Lai uzlabotu šo rīku ir nepieciešams uzlabot tā atsevišķās komponentes vai arī šo atsevišķo komponenšu sadarbību.

Viena no būtiskākajām komponentēm, kas palīdz izveidot jaunus testpiemērus, ir nosacījumu risinātājs, kas risina simboliskās izpildes rezultātā iegūtos ceļu nosacījumus. Rīks izmanto loģikas formulu izpildāmības (angl. *Satisfiability Modulo Theories Solver*) risinātāju *Z3* [14]. Ja uzlabo un pilnveido šo komponenti, tad varētu iegūt kvalitatīvāku un pilnīgāku testpiemēru kopu. Šī komponente ir atsevišķs rīks, izpētes projekts, kam iespējami arī citi pielietojumi, tas tiek attīstīts. Atšķirībā no *IntelliTest*, rīks *Z3* ir atvērtā koda. Eksistē arī citi, līdzīgi risinātāji, taču *Z3* ir *Microsoft* izstrādāts, tāpat kā *IntelliTest*. Autors pieļauj, ka *Z3* ir īpaši pielāgots un jau sākotnēji veidots, lai būtu izmantojams konkrētajā testpiemēru ģenerēšanas rīkā. Rīkam ir aprakstīti dažādi ierobežojumi, tas nozīmē to, ka arī *IntelliTest* rīkam kopumā ir arī tādi paši ierobežojumi. Būtiskākie no šiem ierobežojumiem ir redzami testu ģenerēšanas rezultātos dažādajām programmām 2.1 apakšnodaļā. Viens no tiem, kas ir saistīts tieši ar nosacījumu risinātāju, ir peldošā punkta aritmētika.

Ne mazāk svarīga komponente ir testēšanas dzinējs, kam ir vairāki uzdevumi, daži no tiem:

- veic simbolisko izpildi;
- veido nosacījumu sistēmu;
- padod nosacījumu sistēmu nosacījumu risinātājam *Z3*.

Dzinējs seko padotajai ieejas vērtībai kodā. Dzinējs arī veic testējamās programmas izpildi ar risinātāja iegūto rezultātu – konkrētu ievadu un veido jaunus testpiemērus. Dzinējs veic šo darbu vairākos ciklos – līdz ir sasniegts zaru pārklājums vai iestājies kāds noklusētais vai uzstādītais izpētes robežu nosacījums. Šī dzinēja būtiska īpašība ir robežu nosacījumu izmantošana. Šie iekšējie robežu nosacījumi ar noklusētajām vērtībām faktiski ir vienīgais veids pēc kā rīks konstatē, ka ir notikusi, piemēram, iecikošanās. Šo komponenti varētu pilnveidot tā, lai rīks konstatē ieciklošanos daudz ātrāk, neizpildot konkrēto zaru tūkstošiem reižu līdz sasniegts ierobežojuma nosacījums. Tas tiek aprakstīts sīkāk darba 3.2.2 punktā.

Pati sākotnējā komponente ir instrumentēšanas ietvars, kas veic testējamā koda instrumentāciju, veic datu un kontroles plūsmas uzraudzību. Kodā izveido atsauces, kas ļauj testēšanas dzinējam uzraudzīt izpildi. Atšķirībā no pārējām komponentēm, šī tiek darbināta tikai vienreiz, pašā testpiemēru ģenerēšanas sākumā. Arī šai komponentei ir iespējami

uzlabojumi, jo tika novēroti brīdinājumi lietotājam, kas ir saistīti ar dažādu matemātisko funkciju instrumentēšanu.

3.2.2 IntelliTest algoritma uzlabojumi

Rīku darbinot dažādām programmām tika novēroti ne tikai vispārēji tehniski ierobežojumi, bet arī algoritma nepilnības. Rīks *IntelliTest* izmanto dabīgo algoritmu – programmu izsaka kā koku, kur katram koka zaram konstruē ceļa iziešanas nosacījumus. Šos nosacījumus atrisina un tiek iegūts testpiemērs. Tāda veidā izveido testpiemēru kopu, kas realizēs visus realizējamos zarus [15].

Aprakstītais dabīgais algoritms ir par vāju, jo neatklāj ieciklošanos un nesasniedzamu kodu, ko pierāda iegūtie rezultāti iepriekšējo nodaļu punktos un zemāk arī šajā punktā. Šī rīka algoritms ir papildināts ar laika, izpildes reižu skaita nosacījumiem, kas ļauj tam neiecikloties un vienmēr beigt darbu. Šajā darba punktā jau apskatītās programmas tiek apvienotas, lai vēl uzskatāmāk parādītu šīs algoritma nepilnības. Tāpat arī apvienotajai programmai tiek veidots realizējamības koks, kas ir būtiska sastāvdaļa iespējamam algoritma uzlabojumam, koks atšķiras no dabīgā algoritma koka. Apvienotās programmas kods ir redzams attēlā 3.1. att.

```
public static int function1(int x)
        {
           Outer:   //---------------------------- (1)
            int value = x;
            switch (value)
            {
                case 1:
                    {
                        goto Outer;
                    }
                case 17:
                    {
                        return 17;
                    }
                case 3:
                    {
                        x = 17;
                        goto Outer;
                    }
                case 4:   //---------------------  (2)
                    int y;
                    if (x < 5)
                    {
                        y = x;
                        //pretrunīgs nosacījums
                        if (y >= 5) return 3; //--  (3)
                        return 4;
                    }
                    //šis nav sasniedzams
                    return 1;
                default:
                    return 0;
            }
        }
```

3.1. att. **Apvienotās programmas funkcijas kods**

Pretrunu un ieciklošanās programma ir realizēta vienā funkcijā. Programma saņem ieejas parametru *x*, kas ir *int* tipa. Funkcijas sākumā ir ievietota iezīme *Outer*, kas norāda vietu uz kuru atgriežas programmas izpilde, ja ir izpildīta attiecīga komanda – *goto Outer*. Programma ieciklojas, ja parametra *x* vērtība ir vienāda ar viens, izpildās pirmais *case* gadījums. Ir izveidots jauns zarojums, *case* gadījums – *case 4*, šis zarojums ietver pretrunīgus nosacījums. Šajā zarojumā nokļūst tikai tad, ja parametra *x* vērtība ir vienāda ar 4. Ir viegli redzēt, ka daži ceļi šī *case* zarojuma iekšienē nav izpildāmi. Programma nekad nesasniegs *return 3* komandu, jo tā ir aiz pretrunīgā nosacījuma. Nav iespējams sasniegt arī *return 1* komandu, jo programma jau iepriekš ir atgriezusi vērtību izpildot *if* nosacījumā ietverto kodu un beigusi darbu. Var apgalvot, ka aprakstītais *case* zarojums vienmēr atgriezīs vērtību "4". Arī šai programmai tika ģenerēti testpiemēri, skatīt 3.2. att.

IntelliTest Exploration Results - stopped

ProgramTest.function1(Int32 x) Run 2 Warnings

5 0 10/12 blocks, 0/0 asserts, 9 runs

	x	result	Summary / Exception	Error Message
1	0	0		
2	17	17		
3	4	4		
4	1		path bounds exceeded	
5	3	17		
6	2	0		

3.2. att. **IntelliTest darbināšanas rezultāts apvienotās programmas funkcijai**

Šie testpiemēri parāda rezultātus visiem realizējamiem programmas ceļiem, ir apskatīti visi *case* gadījumi. Kolonnā *result* ir attiecīgās atgrieztās vērtības – 0, 17 un 4 – dažādajiem *x* ieejas parametriem. Citas vērtības nav iespējamas. Arī šeit tiek ziņots par robežu pārsniegšanu, kas norāda uz ieciklošanos.

Iegūtā testpiemēru kopa norāda uz to, ka algoritms darbojas korekti tikai ar realizējamām programmas daļām, bet nespēj atpazīt ieciklošanos un nesasniedzamus koda fragmentus. Paziņojumi norāda tikai uz cikla iespējamību. Nav nekādu brīdinājumu par nesasniedzamiem koda fragmentiem.

Lai to uzlabotu ir nepieciešams būvēt jau iepriekš minēto realizējamības koku, kas ir būtiska PTS algoritma sastāvdaļa. Ideja veidot šādu koku programmai ir aizgūta no *SMOTL* sistēmā realizētā PTS algoritma. Vispārīgs PTS algoritms ietver šādas darbības [15]:

1. “realizējamības koka konstruēšanu;
2. tādas ceļa kopas *K* izvēli, kas:
    - saskaņojas ar realizējamības koku;
    - satur visus dažādos realizējamības koka lokus;
    - beidzas ar apstāšanās komandu *Stop*.
3. katram ceļam no *K* sastāda ceļa iziešanas nosacījumus un atrisina tos.

Iegūtie atrisinājumi algoritma 3. punktā uzdod PTS, veido testpiemēru kopu.”

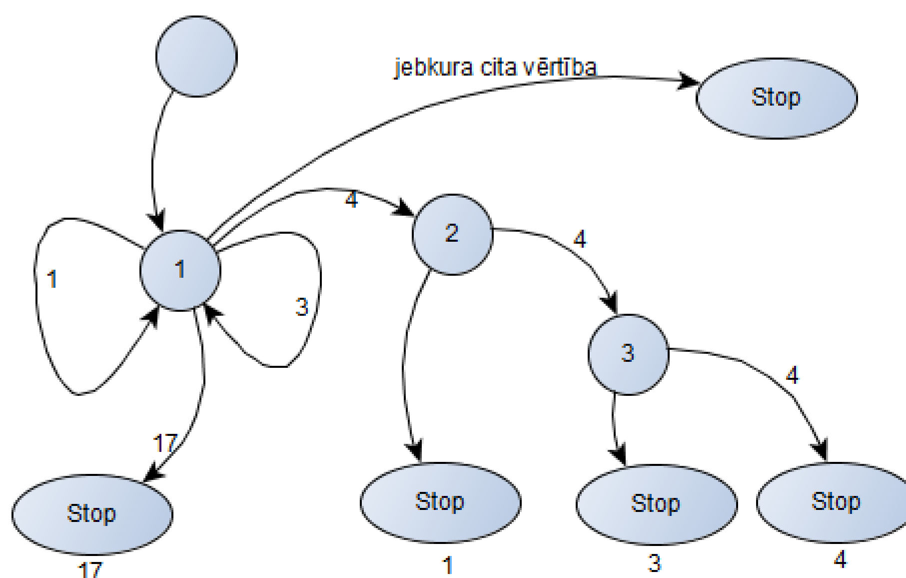
Realizējamības koka būvēšanas soļi *SMOTL* sistēmā ir sadalīti divās daļās – stratēģijā un analizatorā. Stratēģija ir aprakstīta mācību materiālā [16]: “Stratēģija veic vadības grafa virsotņu apstaigāšanu. Tiek izvēlēta virsotne, stāvoklis, turpinājums (lineārs gabals), kas tiek nodots analizatoram. Analizators pārbauda šī gabala realizējamību no dotā stāvokļa. Ja tas būs realizējams, tad tiks uzbūvēts stāvoklis pēc lineārā gabala iziešanas, kuru pierakstīs lineārā

gabala beigu virsotnei. Ja lineārais gabals izrādīsies nerealizējams, tad tiks izdots kļūdas paziņojums. Stratēģija analizē stāvokļu atkārtošanos."

Arī otrās daļas soļi, analizators, ir aprakstīts mācību materiālā [16]: "Analizators sastāda lineārā gabala iziešanas nosacījumus no padotā stāvokļa. Ja lineārais gabals ir realizējams, iziešanas nosacījumiem ir atrisinājums, tad tiek izveidots jauns stāvoklis. Ja atrisinājuma nav, tad lineārais gabals ir nerealizējams."

Pirms tiek būvēts realizējamības koks ir nepieciešams uzbūvēt vadības grafu. Vadības grafu veido virsotnes un loki, virsotnes ir komandas, bet loki – iespējamās vadības nodošanas [17]. Šajā kokā tiek iekļautas tikai būtiskās virsotnes, tās apzīmē kādas būtiskās vietas, komandas programmā. Materiālā [18] ir teikts: "Programma tiek sadalīta lineāros gabalos, no vienas būtiskas virsotnes līdz nākamai virsotnei, realizējamības koks tiek būvēts pa lineāriem gabaliem." Tātad, visa koka būve balstās uz minētajiem lineārajiem gabaliem.

Izveidotajā programmas vadības grafā ir attēlotas būtiskās virsotnes, kas apzīmē komandas, vietas programmā, šīs vietas ir atzīmētas arī apskatāmajā programmā, skatīt 3.1. att. koda komentārus. Pirmā no šīm vietām ir iezīme *Outer*, kas apzīmē ciklu, atgriešanās vietu programmā. Nākamā vieta – *case 4* gadījums, kas ietver sevī *if* zarojumu. Vēl cita vieta – šī zarojuma iekšiene, kas satur vēl vienu *if* zarojumu, kas ir pretrunīgs. Par būtisku vietu varētu uzskatīt katru *case* zarojumu, taču tas netiek darīts, lai vienkāršotu grafu. Tas arī nav nepieciešams, jo ir iespējams ietvert visus *case* zarojumus grafā arī tad, ja neveido katram no tiem atsevišķu virsotni. Programmas vadības grafs ir attēlots 3.3. att.



3.3. att. **Vadības grafs apvienotās programmas funkcijai**

Izveidotais grafs sastāv no lineāriem gabaliem. Grafs sastāv no dažādu veidu virsotnēm, lokiem ar vērtībām. Būtiskās virsotnes ir numurētas, virsotne bez numura var tikt uzskatīta par ieejas pozīciju. *Stop* virsotnes apzīmē programmas darbības beigas. Šajā gadījumā *Stop* virsotnes apzīmē *return* komandas programmas kodā. Grafā ir redzams, ka 1. virsotne ir saistīta ar visām pārējām virsotnēm. Pirmā virsotne ar attiecīgo loku, kura vērtībā vienāda ar viens, apzīmē ieciklošanos, tas ir *case 1* gadījums. Arī ar loku, kura vērtība vienāda ar trīs, ir redzama atgriešanās uz to pašu virsotni. Taču ir zināms, ka *case 17* gadījumā programma beigs darbu, attiecīgi pāriet uz virsotni ar nosaukumu *Stop*. Šādā veidā ar vairākiem lokiem un vienu virsotni ir aprakstīti vairāki *case* gadījumi, bet izmantota tikai pati būtiskākā komanda, vieta programmā. Tiek izcelts *case 4* gadījums, atsevišķa virsotne ar nosaukumu "2", jo tas ietver vairākas būtiskas komandas, zarošanos. Virsotne "3" apzīmē pretrunīgo *if* zarojumu, kas ir pirmā *if* nosacījuma iekšienē. No pirmās virsotnes ar jebkuru citu saņemto vērtību programma beidz darbu, tiek aprakstīts *default* gadījums.

Vadības grafā netiek analizēta ceļu realizējamība, tāpēc tiek paradīti visi ceļi, arī neiespējamie. Grafā pie *Stop* virsotnēm ir uzrādītas vērtības, kas apzīmē iespējamo atgriežamo vērtību, ja vadība nonāk šajā virsotnē, tas ļauj vieglāk orientēties grafā.

Izmantojot vadības grafu tiek būvēts realizējamības koks. Ar realizējamības koka palīdzību ir iespējams:

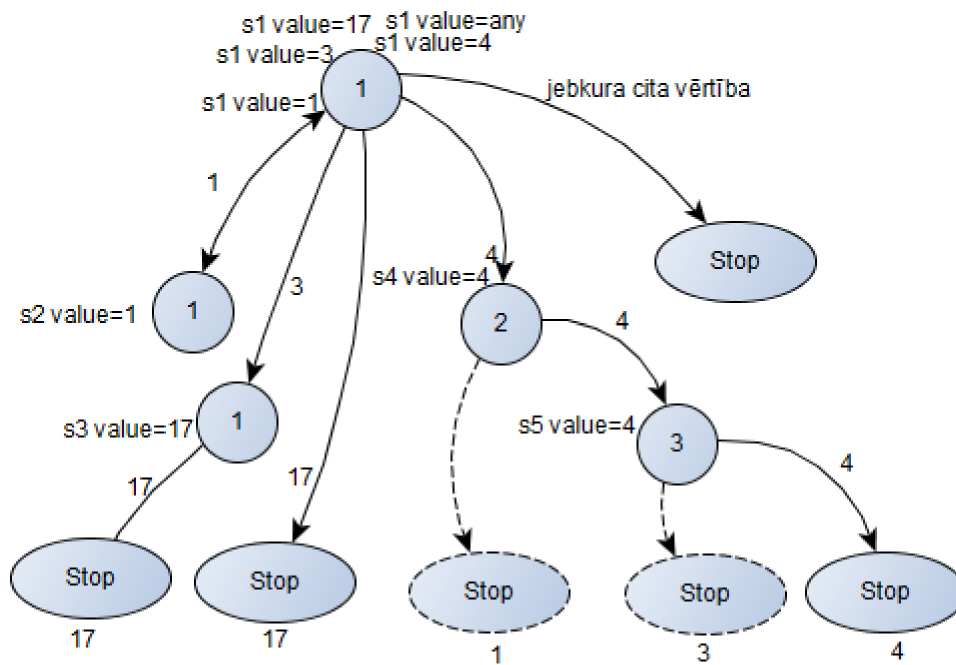
1. ģenerēt pārklājumu;
2. atrast ieciklošanos;
3. atrast nesasniedzamus koda fragmentus;
4. optimizēt testu skaitu, ieejas datus.

Šajā gadījumā mērķis ir 2. un 3. punktā minētās iespējas.

PTS algoritma izstrādātāji noteica un pierādīja vairākas lemmas, būtiskākās no tām konkrētajam gadījumam [15] [17]:

1. Ceļš programmā ir realizējams tad un tikai tad, ja tā iziešanas nosacījumiem ir atrisinājums. Ceļa iziešanas nosacījumu atrisinājums uzdod testu, kurš realizē atbilstošo ceļu.
2. Ja divi ceļi *C1* un *C2* beidzas ar vienu un to pašu komandu programmā un stāvokļi pēc šo ceļu izpildes *S(C1)* un *S(C2)* ir vienādi *(S(C1) = S(C2))*, tad katrs *C1* realizējams turpinājums ir realizējams turpinājums ceļam *C2* un otrādi.

Apskatītās funkcijas realizējamības koks ar vērtībām pie stāvokļiem, balstoties uz vadības grafu, ir attēlots 3.4. att.



3.4. att. **Realizējamības koks apvienotās programmas funkcijai**

Realizējamības kokā ir tās pašas virsotnes, kas ir vadības grafā, tomēr tās var arī atkārtoties, šeit tiek sauktas par stāvokļiem, kuriem piemīt vērtība. Tā stāvokļos ir iekļauta informācija nevis par visiem mainīgajiem, bet tikai par būtiskajiem. Tās ir vērtības, kuras ir tieši saistītas ar konkrēto ceļu iziešanu, tiek iekļautas ceļu iziešanas nosacījumos [18]. Katrs loks pie būtiskajām virsotnēm no vadības grafa tiek analizēts un tikai tad iekļauts realizējamības kokā. Loki vienmēr pāriet jaunā stāvoklī vai arī pāriet uz *Stop* stāvokli, kokā nav stāvokļu ar loku pašiem uz sevi. Virsotnes un loki no grafa tiek secīgi iekļauti realizējamības kokā, sākot ar kreiso pusi. Vadības grafā attēlotā pirmā virsotne ar loku, kura vērtība “1” šeit tiek apzīmēta ar diviem stāvokļiem, kurus savieno konkrētais loks. Virsotņu, kuras savieno konkrētais loks, nosaukumi ir vienādi, taču tie tiek uzskatīti par dažādiem stāvokļiem, attiecīgi tiem ir apzīmējumi *s1 value=1* un *s2 value=1*. Šādā veidā pie stāvokļiem tiek pievienota būtiskā vērtība, stāvokļa apzīmējums. Nākošais loks ar vērtību “3” arī pāriet jaunā stāvoklī *s3*. Attiecīgajā vietā programmā uzstāda jaunu būtiskā mainīgā *x* vērtību, tāpēc būtiskā mainīgā vērtība šajā stāvoklī – v*alue=17*. No šī stāvokļa pāriet *Stop* stāvoklī, loks ar vērtību “17” apzīmē *case 17* gadījumu. Ar šiem diviem lokiem, kas savieno stāvokļus *s1*, *s3* un attiecīgo *Stop* stāvokli ir aprakstīti *case 3* un *case 17* gadījumi. Gadījums *case 17* tiek vēlreiz attēlots ar atsevišķu loku, tā vērtība “17”, tas apraksta atsevišķu gadījumu – ja jau sākotnēji būtiskā mainīgā vērtība ir vienāda ar 17.

Līdzīgi kā vadības grafā, arī programmas *case 4* gadījums tiek attēlots realizējamības kokā, taču ir būtiskas izmaiņas, kas tiks pamatotas zemāk. Vadība nonāk *Stop* stāvoklī, kas atgriež vērtību "4", ja visos stāvokļos *s3*, *s2*, *s1* tika saņemta šāda vērtība. Visbeidzot, realizējamības kokā tiek attēlots *default* gadījums, pārejā no pirmā stāvokļa ar jebkuru citu vērtību nonāk stāvoklī *Stop*, kas atgriež vērtību nulle.

Realizējamības kokā ir attēloti tikai iespējamie programmas stāvokļi un ceļi, tāpēc minētais *case 4* gadījums realizējamības kokā tiek attēlots citādāk. Daži ceļi ir izcelti, ar raustītu līniju ir attēloti nesasniedzamie, neiespējamie ceļi. Šādi ceļi realizējamības kokā neparādās, taču ir izcelti konkrētajā kokā uzskatāmības dēļ. Vadības grafu pārveidojot par realizējamības koku analizators nosaka šo ceļu realizējamību, kas tika aprakstīts pie realizējamības koka būvēšanas soļiem.

Saskaņā ar pirmo lemmu konkrētie ceļi nav realizējami, jo to iziešanas nosacījumiem nav atrisinājumu. Saskaņā ar otro lemmu ir noteikta ieciklošanās starp stāvokļiem *s1* un *s2*, jo stāvokļos vērtība nemainās, tā ir vienāda ar 1. Tāda veidā, ar šī koka un augstāk aprakstītā algoritma palīdzību ir pierādīta ieciklošanās ar ieejas vērtību "1" un nesasniedzamība koda fragmentiem *return 1* un *return 3*.

Lai iegūtu PTS, saskaņā ar darba sākumā doto definīciju, programmas testpiemēriem jāizpilda visus vadības grafa lokus un vienmēr jābeidz darbu. Šādus testpiemērus ir iespējams izveidot izmantojot izveidoto realizējamības koku.

3.3 Iespējamie uzlabojumi EvoSuite

Šajā apakšnodaļā tiek aprakstīti iespējamie uzlabojumi, kas varētu padarīt *EvoSuite* ģenerēto testpiemēru kopu precīzāku un kvalitatīvāku.

Viens no būtiskiem uzlabojumiem, kuru šī rīka izstrādātāji ir ieviesuši un aprakstījuši [19] [20] – meklēšanas algoritma papildināšana ar dinamisko simbolisko izpildi, kas ir arī *IntelliTest* darbības pamatā. Rīka izstrādātāji apgalvo, ka tas ļauj palielināt testpiemēru kopas koda pārklājumu, jo uzlabojums apkopo abu pieeju priekšrocības. Darba autoram neizdevās iedarbināt rīku ar šo papildus iestatījumu un par to pārliecināties.

Autors uzskata, ka ir nepieciešama sīkākas konfigurācijas iespējas *Eclipse* spraudnim. Ir nepieciešama precīzāka dokumentācija, lai atvieglotu darbu ar šo rīku, rīks nav lietotājam draudzīgs. Ir nepieciešams izveidot pilnvērtīgu spraudni *Eclipse* videi, izveidot arī pilnvērtīgu spraudni citām, populārām izstrādes vidēm.

SECINĀJUMI

Veikto pētījumu rezultātā ir iegūti sekojoši secinājumi:

- rīki *IntelliTest* un *EvoSuite* ir piemēroti testpiemēru ģenerēšanai, taču veikt analīzi un ģenerēt testpiemērus var tikai programmām *C#* un *Java* valodās; Abus rīkus var izmantot, lai testētu reālas programmas, taču tiem ir dažādi ierobežojumi;
- *IntelliTest* veido testpiemērus, lai sasniegtu iespējami lielāku koda pārklājumu;
- sistēma *SMOTL* un *IntelliTest* izmanto līdzīgas tehnikas, lai ģenerētu testpiemērus testējamai programmai – simbolisko programmas izpildi, nosacījumu sistēmu risināšanu;
- *SMOTL* un *IntelliTest* spēj noteikt neapstrādātus izņēmumgadījumus, potenciāli bīstamas vietas programmās, ko programmētājs var nepamanīt;
- *IntelliTest* rīkam ir definētas darbības robežas ar noklusētajām vērtībām, lai tas vienmēr beigtu darbu, tās var mainīt, arī *SMOTL* rīkam ir laika darbības robeža;
- rīks *IntelliTest* nespēj atpazīt ieciklošanos un nesasniedzamus koda fragmentus, *SMOTL* to spēj;
- ar noklusētajām izpētes robežām nevar vienmēr uzģenerēt pietiekošu testpiemēru kopu, tas ir raksturīgs rīkiem *IntelliTest* un *EvoSuite*, ko pierāda apskatītā programma ar vienkāršu *for* ciklu;
- ir iespējams iegūt dažādas uzģenerēto testpiemēru kopas, ja programmētājs veic izmaiņas parametrizētā vienībtesta kodā, pievienojot pieņēmumus un novērtējumus;
- izmantojot *SMOTL* sistēmas algoritma idejas *IntelliTest* algoritmu var papildināt un uzlabot, lai tas atpazītu gan ieciklošanos, gan nesasniedzamus koda fragmentus;
- darbā izmantotās funkcionāli vienkāršās programmas ir labs līdzeklis, lai varētu atklāt testpiemēru ģenerēšanas rīku darbības īpašības, noteikt ierobežojumus.

Darba ievadā izvirzītais mērķis ir sasniegts, uzdevumi izpildīti.

IZMANTOTĀ LITERATŪRA

1. **J. Bičevskis u.c**, “SMOTL – A System to Construct Samples for Data Processing Program Debugging,” 1979 [tiešsaiste]. – [atsauce 16. janvāris, 2016.]. Pieejams: https://www.researchgate.net/publication/3189191_SMOTL-A_System_to_Construct_Samples_for_Data_Processing_Program_Debugging
2. **U. Straujums**, “ONTO6 metodoloģija informatizācijas konceptualizācijai,” 2011 [tiešsaiste]. – [atsauce 16. janvāris, 2016.]. Pieejams: https://dspace.lu.lv/dspace/handle/7/4758
3. **J. Bičevskis**, “Автоматическое построение полных систем примеров,” 1977 [tiešsaiste]. – [atsauce 12. janvāris, 2016.]. Pieejams: https://dspace.lu.lv/dspace/handle/7/636
4. **Z. Micskei**, “Code-based test generation” [tiešsaiste]. – [atsauce 19. janvāris, 2016.]. Pieejams: http://mit.bme.hu/~micskeiz/pages/code_based_test_generation.html
5. Code Contracts [tiešsaiste]. – [atsauce 5. marts, 2016.]. Pieejams: https://msdn.microsoft.com/en-us/library/dd264808%28v=vs.110%29.aspx
6. Generate unit tests for your code with IntelliTest [tiešsaiste]. – [atsauce 8. janvāris, 2016.]. Pieejams: https://msdn.microsoft.com/en-gb/library/dn823749.aspx
7. IntelliTest (Smart Unit Tests) [tiešsaiste]. – [atsauce 13. janvāris, 2016.]. Pieejams: https://csharp.today/intellitest-smart-unit-tests-parameterized-unit-tests-will-be-supported-in-visual-studio-2015/
8. Build Better Software with Smart Unit Tests [tiešsaiste]. – [atsauce 18. janvāris, 2016.]. Pieejams: https://msdn.microsoft.com/en-us/magazine/dn904672.aspx
9. **N. Tillmann**, **J. Halleux**, “Parameterized Unit Testing with Microsoft Pex,” 2010 [tiešsaiste]. – [atsauce 18. janvāris, 2016.]. Pieejams: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.4711&rep=rep1&type=pdf
10. **Lingming Zhang u.c**, “Test Generation via Dynamic Symbolic Execution for Mutation Testing” [tiešsaiste]. – [atsauce 18. janvāris, 2016.]. Pieejams: http://www.utdallas.edu/~lxz144130/publications/icsm2010.pdf
11. EvoSuite - Automatic Test Suite Generation for Java [tiešsaiste]. – [atsauce 1. marts, 2016.]. Pieejams: www.evosuite.org

12. **G. Fraser**, **A. Acuri**, “Evolutionary Generation of Whole Test Suites” [tiešsaiste]. – [atsauce 22. marts, 2016.]. Pieejams: http://www.evosuite.org/wp-content/papercite-data/pdf/qsic11.pdf
13. AgitarOne: Putting Java to the Test [tiešsaiste]. – [atsauce 3. marts, 2016.]. Pieejams: http://www.agitar.com/solutions/products/agitarone.html
14. Z3 Theorem Prover [tiešsaiste]. – [atsauce 20. marts, 2016.]. Pieejams: https://github.com/Z3Prover/z3
15. **J. Bičevskis**, Mācību materiāli, kurss DatZ5013: Programmatūras testēšana, Pilnas testu sistemas (algoritms) [tiešsaiste]. – [atsauce 21. aprīlis, 2016.]. Pieejams: http://estudijas.lu.lv/mod/resource/view.php?id=7013
16. **J. Bičevskis**, Mācību materiāli, kurss DatZ5013: Programmatūras testēšana, Pilnas testu sistemas (realizācija) [tiešsaiste]. – [atsauce 21. aprīlis, 2016.]. Pieejams: http://estudijas.lu.lv/mod/resource/view.php?id=7015
17. **J. Bičevskis**, Mācību materiāli, kurss DatZ5013: Programmatūras testēšana, Pilnas testu sistemas (jēdzieni) [tiešsaiste]. – [atsauce 21. aprīlis, 2016.]. Pieejams: http://estudijas.lu.lv/mod/resource/view.php?id=7011
18. **J. Bičevskis**, Mācību materiāli, kurss DatZ5013: Programmatūras testēšana, Pilnas testu sistemas (vispārinājumi) [tiešsaiste]. – [atsauce 21. aprīlis, 2016.].Pieejams: http://estudijas.lu.lv/mod/resource/view.php?id=7014
19. **J.P. Galeotti u.c**, “Extending a Search-Based Test Generator with Adaptive Dynamic Symbolic Execution” [tiešsaiste]. – [atsauce 20. marts, 2016.]. Pieejams: http://www.evosuite.org/wp-content/papercite-data/pdf/issta14_dse.pdf
20. **J.P. Galeotti u.c**, “Improving Search-based Test Suite Generation with Dynamic Symbolic Execution” [tiešsaiste]. – [atsauce 20. marts, 2016.]. Pieejams: http://www.evosuite.org/wp-content/papercite-data/pdf/issre13_dse.pdf

PIELIKUMI

1. pielikums. Testpiemēru ģenerēšanas rīku saraksts

Zemāk uzskaitītie rīki izmanto programmas kodu, lai ģenerētu testpiemērus. Sarakstā ir gan komerciālie, gan nekomerciālie rīki. Katrs ir paredzēts kādai noteiktai programmēšanas valodai. Jāatzīmē, ka saraksts nav pilnīgs, sarakstā var būt iekļauti arī tādi rīki, kuri netiek regulāri atjaunoti un uzturēti. Šie rīki ir [4]:

- AgitarOne;
- AutoTest;
- CATG;
- CAUT;
- CREST;
- EvoSuite (apskatīts darbā);
- GRT;
- GUITAR;
- Jalangi;
- JSeft;
- Jtest;
- JTExpert;
- KLEE;
- MergePoint/Mayhem;
- PathCrawler;
- Pex/IntelliTest (apskatīts darbā);
- QuickCheck;
- Randoop;
- SAGE;
- Symbolic PathFinder;
- T3.

2. pielikums. EvoSuite ģenerētie testpiemēri

Šajā pielikumā ir katras programmas uzģenerēto testpiemēru kopa ar datiem.

Zarošanās programmas ar trijstūriem testpiemēri

```
/*
 * This file was automatically generated by EvoSuite
 */


import org.junit.Test;
import static org.junit.Assert.*;
import org.EvoSuite.runtime.EvoRunner;
import org.EvoSuite.runtime.EvoRunnerParameters;
import org.junit.runner.RunWith;

@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism =
true, useVFS = true, useVNET = true, resetStaticState = true,
separateClassLoader = true)
public class Triangle_ESTest extends Triangle_ESTest_scaffolding {

  @Test
  public void test00()  throws Throwable  {
      Triangle triangle0 = new Triangle(251, 3744, 3537);
      String string0 = triangle0.triangleType();
      assertEquals("dazadmalu", string0);
  }

  @Test
  public void test01()  throws Throwable  {
      Triangle triangle0 = new Triangle(3689, 3689, 1);
      String string0 = triangle0.triangleType();
      assertEquals("vienadsanu", string0);
  }

  @Test
  public void test02()  throws Throwable  {
      Triangle triangle0 = new Triangle(2375, 33, (-1654));
      String string0 = triangle0.triangleType();
      assertEquals("nav iespejams", string0);
  }

  @Test
  public void test03()  throws Throwable  {
      Triangle triangle0 = new Triangle(775, 0, 2679);
      String string0 = triangle0.triangleType();
      assertEquals("nav iespejams", string0);
  }

  @Test
  public void test04()  throws Throwable  {
      Triangle triangle0 = new Triangle((-247), (-247), (-247));
      String string0 = triangle0.triangleType();
      assertEquals("nav iespejams", string0);
  }

  @Test
  public void test05()  throws Throwable  {
      Triangle triangle0 = new Triangle(1641, 31, 1641);
```

```
    String string0 = triangle0.triangleType();
    assertEquals("vienadsanu", string0);
}

@Test
public void test06()  throws Throwable  {
    Triangle triangle0 = new Triangle(604, 775, 775);
    String string0 = triangle0.triangleType();
    assertEquals("vienadsanu", string0);
}

@Test
public void test07()  throws Throwable  {
    Triangle triangle0 = new Triangle(1, 1, 1);
    String string0 = triangle0.triangleType();
    assertEquals("vienadmalu", string0);
}

@Test
public void test08()  throws Throwable  {
    Triangle triangle0 = new Triangle(3744, 1434, 2317);
    String string0 = triangle0.triangleType();
    assertEquals("dazadmalu", string0);
}

@Test
public void test09()  throws Throwable  {
    Triangle triangle0 = new Triangle(775, 3689, 1);
    String string0 = triangle0.triangleType();
    assertEquals("nav iespejams", string0);
}

@Test
public void test10()  throws Throwable  {
    Triangle triangle0 = new Triangle(1883, 1, 948);
    String string0 = triangle0.triangleType();
    assertEquals("nav iespejams", string0);
}

@Test
public void test11()  throws Throwable  {
    Triangle triangle0 = new Triangle(1, 1616, 2093);
    String string0 = triangle0.triangleType();
    assertEquals("nav iespejams", string0);
}

@Test
public void test12()  throws Throwable  {
    Triangle triangle0 = new Triangle(689, 689, 0);
    String string0 = triangle0.triangleType();
    assertEquals("nav iespejams", string0);
}

@Test
public void test13()  throws Throwable  {
    Triangle triangle0 = new Triangle(1587, (-628), 1587);
    String string0 = triangle0.triangleType();
    assertEquals("nav iespejams", string0);
}

@Test
public void test14()  throws Throwable  {
    Triangle triangle0 = new Triangle(0, 2955, 2955);
```

```
        String string0 = triangle0.triangleType();
        assertEquals("nav iespejams", string0);
    }

    @Test
    public void test15()  throws Throwable  {
        Triangle triangle0 = new Triangle(1702, 1702, 2955);
        String string0 = triangle0.triangleType();
        assertEquals("vienadsanu", string0);
    }
}
```

Vienkāršas *for* cikla programmas testpiemēri

```
/*
 * This file was automatically generated by EvoSuite
 */


import static org.EvoSuite.runtime.EvoAssertions.assertThrownBy;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

import org.EvoSuite.runtime.EvoRunner;
import org.EvoSuite.runtime.EvoRunnerParameters;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism =
true, useVFS = true, useVNET = true, resetStaticState = true,
separateClassLoader = true)
public class cycle1_ESTest extends cycle1_ESTest_scaffolding {

    @Test
    public void test0()  throws Throwable  {
        // Undeclared exception!
        try {
          cycle1.vaiSatur('}', (String) null);
          fail("Expecting exception: NullPointerException");

        } catch(NullPointerException e) {
           //
           // no message in exception (getMessage() returned null)
           //
           assertThrownBy("cycle1", e);
        }
    }

    @Test
    public void test1()  throws Throwable  {
        int int0 = cycle1.vaiSatur('d', "@#+@IE:E':Td");
        assertEquals(1, int0);
    }

    @Test
    public void test2()  throws Throwable  {
        int int0 = cycle1.vaiSatur('M', ",(p3 YiARa_N:1n+s");
        assertEquals(0, int0);
    }
```

```
  @Test
  public void test3()  throws Throwable  {
      String[] stringArray0 = new String[3];
      cycle1.main(stringArray0);
  }

  @Test
  public void test4()  throws Throwable  {
      cycle1 cycle1_0 = new cycle1();
  }
}
```

Objektu programmas ar bankas kontiem testpiemēri

```
/*
 * This file was automatically generated by EvoSuite
 */


import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNull;

import org.EvoSuite.runtime.EvoRunner;
import org.EvoSuite.runtime.EvoRunnerParameters;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism =
true, useVFS = true, useVNET = true, resetStaticState = true,
separateClassLoader = true)
public class BankAccount_ESTest extends BankAccount_ESTest_scaffolding {

  @Test
  public void test00()  throws Throwable  {
      BankAccount bankAccount0 = new BankAccount(0.0F, "");
      String string0 = bankAccount0.getNosauk();
      assertEquals("", string0);
  }

  @Test
  public void test01()  throws Throwable  {
      BankAccount bankAccount0 = new BankAccount((-1514.305F), "]/");
      int int0 = bankAccount0.withdraw((-1514.305F));
      assertEquals(0.0F, bankAccount0.getBalance(), 0.01F);
      assertEquals(0, int0);
  }

  @Test
  public void test02()  throws Throwable  {
      BankAccount bankAccount0 = new BankAccount(21.8268F, "E-,");
      int int0 = bankAccount0.deposit(0.0F);
      assertEquals(0, int0);
      assertEquals(21.8268F, bankAccount0.getBalance(), 0.01F);

  }

  @Test
  public void test03()  throws Throwable  {
      BankAccount bankAccount0 = new BankAccount(0.0F, "");
      bankAccount0.setNosauk("");
      assertEquals(0.0F, bankAccount0.getBalance(), 0.01F);
  }
```

```java
    @Test
    public void test04()  throws Throwable  {
        BankAccount bankAccount0 = new BankAccount((-497.98523F), "{");
        assertEquals(0.0F, bankAccount0.getBalance(), 0.01F);

        bankAccount0.setBalance((-497.98523F));
        float float0 = bankAccount0.getBalance();
        assertEquals((-497.98523F), float0, 0.01F);
    }

    @Test
    public void test05()  throws Throwable  {
        BankAccount bankAccount0 = new BankAccount((-943.9F), "");
        int int0 = bankAccount0.withdraw(1);
        assertEquals((-1.0F), bankAccount0.getBalance(), 0.01F);
        assertEquals(1, int0);
    }

    @Test
    public void test06()  throws Throwable  {
        BankAccount bankAccount0 = new BankAccount((-943.9F), "");
        bankAccount0.deposit(1.0F);
        float float0 = bankAccount0.getBalance();
        assertEquals(1.0F, float0, 0.01F);
    }

    @Test
    public void test07()  throws Throwable  {
        BankAccount bankAccount0 = new BankAccount((-497.98523F), "{");
        int int0 = bankAccount0.deposit((-1.0F));
        assertEquals(0.0F, bankAccount0.getBalance(), 0.01F);
        assertEquals(0, int0);
    }

    @Test
    public void test08()  throws Throwable  {
        BankAccount bankAccount0 = new BankAccount(0.0F, "xx<(j");
        int int0 = bankAccount0.withdraw(0.0F);
        assertEquals(0, int0);
    }

    @Test
    public void test09()  throws Throwable  {
        BankAccount bankAccount0 = new BankAccount((-1514.305F), "]/");
        String string0 = bankAccount0.getNosauk();
        assertNull(string0);
        assertEquals(0.0F, bankAccount0.getBalance(), 0.01F);
    }

    @Test
    public void test10()  throws Throwable  {
        BankAccount bankAccount0 = new BankAccount((-497.98523F), "{");
        float float0 = bankAccount0.getBalance();
        assertEquals(0.0F, float0, 0.01F);
    }
}
```

Maģistra darbs: **Testu automātiska ģenerēšana**

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: ______________________________

(Autora paraksts)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: ______________________________

(Vadītāja paraksts)

Darbs iesniegts **maģistrantūras sekretariātā** ________________________.

(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Studiju metodiķe: ______________________________.

(Metodiķes paraksts)

Recenzents: ___________________________________

(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_________________________ prot. Nr. _________

(Darba aizstāvēšanas datums)

Komisijas sekretārs: ________________________________

(Sekretāra paraksts)