KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Tomas Neverdauskas

Statinė CIL kodo analizė, remiantis simboliniu vykdymu

Magistro darbas

Darbo vadovas
Prof. E. Bareiša

Kaunas, 2010

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Tomas Neverdauskas

Statinė CIL kodo analizė, remiantis simboliniu vykdymu

Magistro darbas

Recenzentas
Prof. L. Nemuraitė
2010-05

Vadovas
Prof. E. Bareiša
2010-05

Atliko
IFM-4/2 gr. stud.
Tomas Neverdauskas
2010-05-31

Kaunas, 2010

Turinys

1. Įvadas 9
2. Simbolinio vykdymo taikymo modeliu paremtame testavime teorinis pagrindimas 10
    2.1. Programinės įrangos testavimo problematika magistro darbo kontekste 10
    2.2. Modeliu paremtas testavimas...11
    2.3. Orakulo problema 12
    2.4. „Baltos" ir „juodos" dėžės testavimo principai 12
        2.4.1. „Juodos dėžės" principas 13
        2.4.2. „Baltos dėžės" testavimas 14
    2.5. Vienetų testai 14
    2.6. CIL (Common Intermediate Language) 15
    2.7. Programos pavyzdžiai C# ir CIL kalba...16
    2.8. CIL instrukcijų tipai..17
    2.9. CIL instrukcijos 17
    2.10. JIT (Just-in-time) kompiliatorius...22
    2.11. AOT (Ahead-of-time) kompiliatorius...22
    2.12. Microsoft CCI 23
    2.13. SMT 23
    2.14. Simbolinis vykdymas..24
    2.15. SE algoritmo veikimas..24
3. Klaidų sekimo informacinė sistema „*CRUNCHBUG*" 27
    3.1. Tikslas...28
    3.2. Programų sistemos funkcijos...28
    3.3. Sistemos kontekstas 28
    3.4. Projekto įgyvendinimas 29
    3.5. Sistemos ribos...30
    3.6. Architektūriniai sprendimai 31
    3.7. Projektinio darbo rezultatai...32
4. Simbolinio vykdymo įrankių taikymo tyrimas .Net platformoje.......................................33
    4.1. Symex – simbolinio vykdymo variklis .Net platformai..33
    4.2. Symex architektūra 33
        4.2.1. Bendroji Symex architektūra ir veiklos kontekstas 33
        4.2.2. Klasių diagrama 35
    4.3. Programos apribojimai..35

4.4. Symex dabartinė būsena ir tolesni darbai ...36
4.5. Programos veikimo pavyzdžiai...36
4.6. Pex ...39
4.7. Atsitiktinio generavimo karkasas...40
5. Simbolinio vykdymo kodo analizės įrankių vienetų testams generuoti palyginimo eksperimentas...41
5.1. Eksperimento tikslai ...41
5.2. Eksperimentui naudota techninė ir programinė įranga ...41
5.3. Simboliniu vykdymu paremtas vienetų testų generavimo eksperimentas matematiniams metodams...41
5.4. Simboliniu vykdymu paremtas vienetų testų generavimo eksperimentas magistro projektui ...44
Išvados ...46
Literatūra...47
Terminų ir santrumpų žodynas ...50
Priedai ...51
1. Priedas: eksperimente naudota klasė matematiniams metodams tikrinti...51
2. Priedas: vienetų testų klasė sugeneruota Symex įrankiu ...52
3. Priedas: vienetų testų klasės sugeneruota Pex įrankiu...53
4. Priedas: Kodo eilučių skaičius ir ciklomatinis sudėtingumas „Crunchbug” projekte ..55
5. Priedas: Kodo eilučių padengimas vienetų testais „Crunchbug” projekte ...55
6. Priedas: publikacija iš IVUS 2010 konferencijos: „SYMEX - Symbolic Execution engine for .NET platform” ...56

Lentelių sąrašas

1 lentelė. CIL komandų sąrašas 17
2 lentelė. Eksperimente analizuojamos funkcijos...36
3 lentelė. Eksperimente tiriami klasių metodai..42
4 lentelė. Projektinės dalies kodo eilučių skaičius...55
5 lentelė. Projektinės dalies kodo padengimas vienetų testais..55

Paveikslėlių sąrašas

1 pav. Programinės įrangos testavimo problematika 10
2 pav. Modeliu paremto testavimo principas 11
3 pav. Principinė orakulo veikimo schema 12
4 pav. .Net platformos programų transliavimo infrastruktūra 15
5 pav. Vienos CIL pagrindu parašytos kalbos transformavimas į kitą 16
6 pav. Paruoštas vykdyti metodas 25
7 pav. Simbolinio vykdymo medis 26
8 pav. GNU projekto klaidų statistika 27
9 pav. Programos veiklos kontekstas 29
10 pav. Sistemos ribos pateiktos panaudos atveju diagrama 30
11 pav. Kuriamos sistemos architektūriniai sluoksniai 31
12 pav. „Crunchbug“ pagrindinis langas 32
13 pav. Symex bendroji architektūra ir veikimo kontekstas 34
14 pav. Symex klasių diagrama 35
15 pav. Metodo *GetSquareSurfaceArea* išeities tekstas 36
16 pav. Sugeneruotas *GetSquareSurfaceArea* bitinis kodas, kuris vykdomas simboliškai 37
17 pav. *GetSquareSurfaceArea* simboliniai vykdymo keliai, SMT lygčių pavidalu 37
18 pav. Sugeneruotas *GetSquareSurfaceArea* bitinis kodas, kuris vykdomas simboliškai 37
19 pav. Sugeneruotas *GetCubeSurfaceArea* bitinis kodas, kuris bus vykdomas simboliškai .. 38
20 pav. *GetSquareSurfaceArea* vykdymo keliai, suformuoti SMT lygčių pavidalu 39
21 pav. Symex analizuoja kodą (programos darbo langas) 39
22 pav. Atsitiktinio generavimo rezultatai 42
23 pav. SYMEX eksperimento rezultatai 43
24 pav. Pex rezultatai 43
25 pav. Apibendrinti įrankių palyginimo rezultatai 44
26 pav. Pex kodo padengimo projekte rezultatai 45

Static CIL code analysis using symbolic execution

Summary

Testing complex safety critical software always was difficult task. Development of automated techniques for error detection is even more difficult. Well known techniques for checking software are model checking static analysis and testing. Symbolic execution is a technique that is being used to improve security, to find bugs, and to help in debugging. A symbolic execution engine is basically an interpreter that figures out how to follow all paths in a program. It is a static code analysis technique.

This work presents symbolic execution background, current state, analysis the possibilities of implementation on the .Net framework and platform. The work describes the master project – bug tracking software “Crunchbug” and the tool – Symex (symbolic execution engine) for .Net platform. Symex is white box model based automatic unit test generator and it is evaluated against two other tools – Microsoft Pex and framework that generates unit test inputs random. Detailed experiments made to cover symbolic execution possibilities with proprietary benchmarks and real code from the master project.

Statinė CIL kodo analizė, remiantis simboliniu vykdymu

Santrauka

Programinės įrangos testavimas ir kokybės užtikrinimas yra svarbus programų sistemų inžinerijos kūrimo uždavinys, siekiant sukurti tinkamą naudojimui produktą. Yra daug skirtingų metodikų kuriamai programinei įrangai testuoti, tačiau vieningos sistemos, kuri būtų universali – nėra. Įvairūs tyrimai vykdomi programinės įrangos testavimo srityje duoda skirtingus rezultatus. Testavimo procesas taip pat svarbus ir praktikoje – be jo negali išsiversti nei vienas organizacija susijusi su programinės įrangos kūrimu ir plėtojimu.

Šis darbas remiasi modeliu paremto testavimo paradigma ir simboliniu vykdymo metodika. Darbe apžvelgiamos teorinės simbolinio vykdymo galimybės, jo pritaikymas .Net platformoje ir papildomos priemonės, kurios reikalingos įgyvendinti tokią sistemą. Taip pat trumpai pristatomas magistro projektinis darbas, aprašomi sukurti inžinerinio produkto svarbiausi aspektai.

Pagal teorinę medžiaga sukurtas simbolinio vykdymo variklis – Symex. Darbe nagrinėjamas praktinis tokio įrankio pritaikymas generuojant vienetų testus iš išeities kodo – eksperimentiškai tiriamos ir lyginamos simbolinio vykdymo ir atsitiktinių įėjimų vienetų testų kūrimo galimybės .Net platformoje.

1. Įvadas

Šis dokumentas – tai programų sistemų inžinerijos baigiamasis magistro darbas.

Pagrindinis darbo tikslas – išnagrinėti simbolinio vykdymo galimybes .Net platformoje – tiek teoriškai, tiek praktiškai, sukuriant reikalingą programinę įrangą ir eksperimentiškai patikrinant simbolinio vykdymo metodikos tinkamumą statiškai analizuojant empirinius kodo pavyzdžius ir magistro projektą, bei sukuriant vienetų testus duotam kodui padengti.

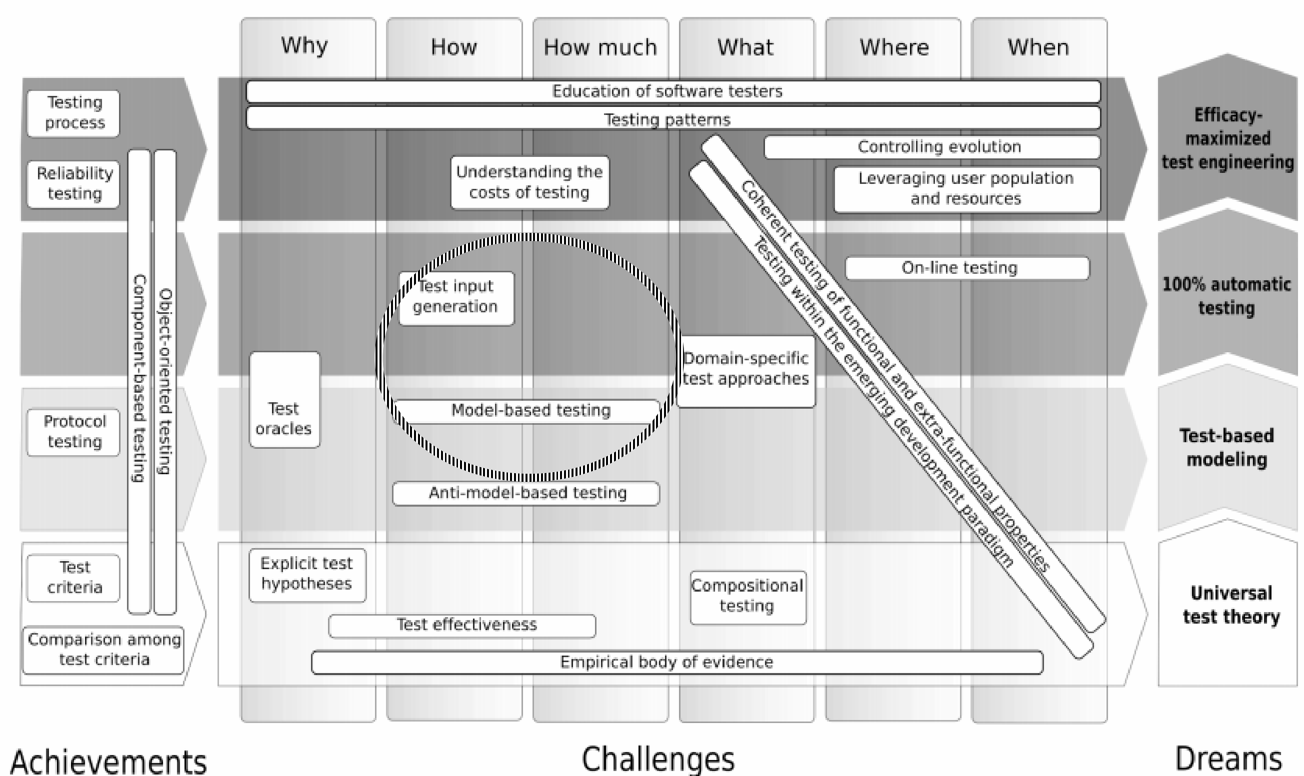
Magistro darbe suformuluotam tikslui pasiekti keliami tokie uždaviniai:

- Greitesnis ir patikimesnis automatinis vienetų testų kūrimas ir generavimas
- Vienetų testų kodo padengimas kuo artimesnis 100%
- Galimybė ištestuoti visus įmanomus programos vykdymo kelius
- Simbolinio vykdymo taikymas .Net platformoje
- Simbolinio vykdymo ir statinės kodo analizės įrankio kūrimas
- Eksperimentinė įrankių analizė ir išvadų apie simbolinį vykdymą pateikimas, šios metodikos vertinimas modeliu paremto testavimo kontekste

2. Simbolinio vykdymo taikymo modeliu paremtame testavime teorinis pagrindimas

Šioje dalyje teoriškai apžvelgiami .Net platformos kodo vykdymo būdai, pagrindiniai vienetų testavimo metodai ir sąvokos, nagrinėjamas simbolinio vykdymo veikimas ir prielaidos, reikalingos sukurti tokio tipo testavimo įrankius, teoriškai pagrindžiami tyrimo medžiagoje naudojami sprendimai ir aprašoma magistro darbo problematika.

2.1. Programinės įrangos testavimo problematika magistro darbo kontekste



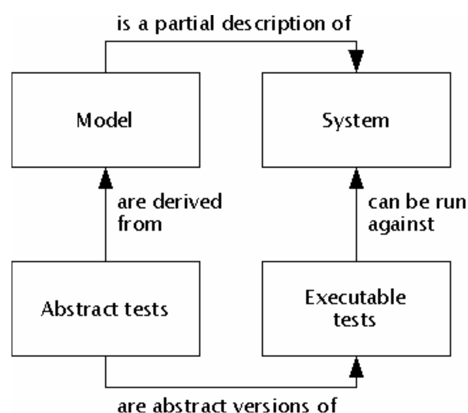
1 pav. Programinės įrangos testavimo problematika [2]

Bendrąją programinės įrangos testavimo problematika nusako pav. 1, kuriame vaizduojami testavimo būdai, priemonės, galimybės, tikslai ir „svajonės". Šio magistro darbo kontekstas yra testų įėjimų generavimas ir modeliu paremtas testavimas (taikant simbolinį vykdymą ir statinę kodo analizę), pav. 1, sritis pažymėta.

2.2. Modeliu paremtas testavimas

Šio tipo testavimo metodika (model – based testing) suformuota gan seniai – tai 1956 metais pasiūlė Moore [3]. Tačiau tik pastaraisiais metais šioje testavimo šakoje stebima vis daugiau mokslinio aktyvumo. Ši metodika nėra labai plačiai taikoma industrijoje, tačiau mokslininkai aktyviai stengiasi pašalinti kliūtis, kurios trukdo tai padaryti [2]. Pagrindinė modeliu paremto testavimo problema – kaip sukurti vieningą, tinkamą ir **efektyvų** aprašymą modeliui, pagal kurį galima būti testuoti ar kurti testus.

Dabar egzistuojantys sprendimai – pre / post sąlygos (pvz.: OCL kalba[4]), būsenos diagramos, scenarijais paremti būdai iškelia problemą, kaip juos tinkamai sujungti į bendrą visumą, nes pavieniai šių metodų taikymai neužtikrina tinkamo rezultato. Praktika rodo, kad programuotojai retai mėgsta ir naudoja papildomas priemones [5] (perteklines schemas ar įvairius programavimo kalbos papildymus, tokius kaip OCL). Tikrinimas pagal modelyje aprašytus (visus įmanomus) testavimo atvejus yra tinkama metodika, tačiau šio būdo didelis trūkumas – sudėtingas plečiamumas [6]. Statinės analizės metodika yra gerai plečiama ir lengvai prisitaiko, tačiau dažnai pasitaiko, kad statinės analizės metodai duoda klaidingus rezultatus [7]. Žmogaus testavimas, dažniausiai randa realias klaidas, bet tam tikros programos dalys visuomet lieka neištestuotos dėl įvairių žmoniškųjų faktorių [8].



2 pav. Modeliu paremto testavimo principas[9]

Nuo 7 – tojo dešimtmečio stengtasi išspręsti problemas, kurios kyla naudojant modeliu paremta testavimą, yra pasiekta gerų rezultatų, tačiau dauguma jų – moksliniame lygyje, tinkamų sprendimų realiam naudojimų vis dar nepakanka[2]. Fraser et al. [10] savo darbe aprašo testavimo priemonių, paremtų modeliu, problemas:

- Modelio sukūrimo laiko minimizavimas.
- Išimčių prioretizavimas
- Galimybės plėsti išimtinių atvejų aibę

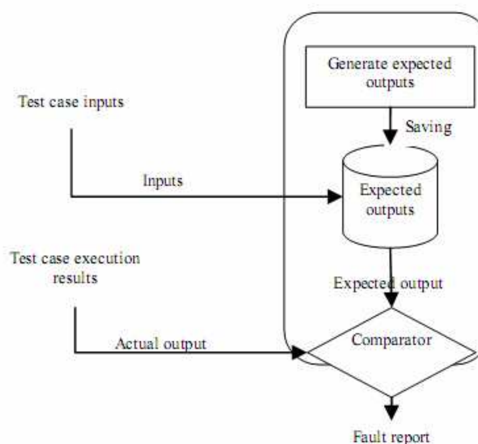
- Skirtingų modelių jungimas
- Alternatyvios išimtys
- Testavimo abstrahavimas

Simboliniu vykdymu paremtoje kodo analizėje **modelis yra pats kodas**, taigi labai didelis privalumas – nebereikia perteklinių modelių ir schemų, kurios nurodytų kaip turi elgtis sistema, tačiau atsiranda trūkumas – reikia žinoti ar kodas yra teisingai parašytas ir ar gautas testavimo rezultatas teisingas – iškyla orakulo problema. Verta pažymėti, kad šis tyrimas ir magistro tezės **nesprendžia ir nenagrinėja** orakulo problemos, o kodą laiko „absoliučia tiesa".

2.3. Orakulo problema

Orakulas - tai bet koks (žmogiškas ar mechaninis) agentas, kuris nusprendžia ar programa elgėsi teisingai ir ar sistema išlaikė testavimą ar ne. Tai rimta ir brangiai kainuojanti problema, kuri dažnai nuvertinama [11].

Orakulo pilnumą nusako aibė, kuri apibrėžia visus įmanomus testų atvejus duotus pagal sistemos specifikaciją. Magistro darbe orakulo problema neliečiama, todėl ši skyrius tik bendrais bruožais apibūdina testavimo orakulą.



3 pav. Principinė orakulo veikimo schema[9]

2.4. „Baltos" ir „juodos" dėžės testavimo principai

Tradiciškai testavimo metodikos skirstomos į dvi pagrindines šakas.

2.4.1. „Juodos dėžės" principas

Tai toks būdas, kai vidinis sistemos funkcionavimas neįtakoja testo sudarymo. Remiamasi tik funkcine specifikacija ir reikalavimais. Juodos dėžės metodas vertina testuojamą objektą kaip tokį, kurio veikimas nežinomas. Žinomi tik įėjimo poveikiai ir laukiamas rezultatas. Pagal gautus rezultatus, padarius norimus poveikius (ar įvedus duomenis, paspaudus mygtukus ir pan.), sprendžiama, ar programinės įrangos veikimas teisingas ir tinkamas. Šio tipo testai gali būti funkciniai (dažniausiai pasitaikantys) ir nefunkciniai. Testų kūrėjas parenka teisingus ir klaidingus įvesties duomenis bei aprašo laukiamą rezultatą prie kiekvieno iš šių įėjimo poveikių. Apie vidinį objekto veikimą žinių nėra. Šį metodą galima pritaikyti visiems testavimo lygiams: vienetų, integracijos, funkcinio ir tinkamumo. Kuo aukštesnis užbaigtumo lygis, tuo didesnė ir sudėtingesnė tampa užduotis, dėl to „juodos dėžės" metodas palengvina testavimą. Šis metodas gali aptikti dar nerealizuotas sistemos dalis, tačiau negali garantuoti, kad visi egzistuojantys keliai sistemoje bus patikrinti testo metu.

Juodos dėžės testavimo privalumai:

- Efektyvesnis už baltą dėžę didesnės apimties testavimo vienetams.
- Testuotojui nereikia žinoti programos veikimo specifikos ar specifinių programavimų kalbų sintaksės bei logikos.
- Testuotojas ir programuotojas vienas nuo kito yra nepriklausomi.
- Testai sudaromi iš vartotojo perspektyvos.
- Padeda išryškinti specifikacijų daugiareikšmiškumus ir nesuderinamumus.
- Testiniai rinkiniai gali būt pradėti kurti vos tik baigiamos rašyti specifikacijos.

Juodos dėžės testavimo trūkumai:

- Tik maža dalis galimų įvesties kombinacijų gali būti ištestuojama. Testas su visais/dauguma rinkinių užtruktų labai ilgai.
- Be aiškių ir nuoseklių specifikacijų sunku sudaryti tinkamus testinių duomenų rinkinius.
- Testiniai rinkiniai gali kartotis, jei programuotojas testuotojo neinformavo apie išbandytus variantus.
- Gali likti neištestuota dalis kelių.
- Sunku ar ne neįmanoma nukreipti testo į specifinius kodo segmentus, kurie gali būti itin sudėtingi (ir tuo pačiu gali slėpti daugiau klaidų)

- Daugiau testavimų tyrimų pritaikyta baltos dėžės modelių.

2.4.2. „Baltos dėžės” testavimas

Kitaip dar vadinamas: skaidri, stiklinė, permatoma dėžė arba struktūrinis testavimas - naudoja testų sudarymą atsižvelgiant į vidinę programos struktūrą. Visų įmanomų kelių išrinkimui sistemoje reikalingi programavimo įgūdžiai ir kalbos žinios. Testuotojas parenka įėjimo signalų rinkinius tokius, kad sistemos veikimas vyktų norimu algoritmo keliu. Kadangi šio tipo testai remiasi pagamintu produktu, jam keičiantis, greičiausiai teks keisti ir testus. Tai išaugina pakeitimų ir testavimo kainą kūrimo proceso metu, tačiau negarantuoja, kad pakeitimai neįneša naujų papildomų klaidų. Balta dėžė gali būti taikoma vienetams, integracijai ir sistemos testavimams. Dažniausiai taikoma vienetų testavimui, kur bandomi pasirinkti vykdymo keliai. Taip pat gali būti išbandoma sąveika tarp vienetų integracijoje, ar tarp posistemių, sistemos testo metu. Šiuo metodu gaunamas didelis ir/ar pilnas kodo padengimas testais, tačiau neaptinkamos nerealizuotos sistemos dalys aprašytos funkcinėse specifikacijose. Testai sudaromi taip, kad padengtų kodo eilutes, kelius, sąlygas, šakas. Simbolinio vykdymo metodika ištaiso kai kuriuos šio tipo testavimo trūkumus – pakeitus kodą, galima iš naujo automatiškai sugeneruoti testus.

2.5. Vienetų testai

Vienetų testavimas (unit testing) dažnai atliekamas testuojant iš abiejų „dėžių” perspektyvos. „Baltos dėžės” atveju testiniai atvejai gaunami iš programos struktūros. Testais bandoma patikrinami visus operatorius, visos sąlygų šakos, keliai. Vienetų testavimas naudojamas atskiriems programinės įrangos komponentams testuoti. Komponentai testuojami paduodant įėjimo duomenis, stebint išėjimus ir juos lyginant su laukiamais rezultatais. Vienetų testavimas įprastai yra vykdomas pačių programuotojų, o ne galutinių vartotojų.

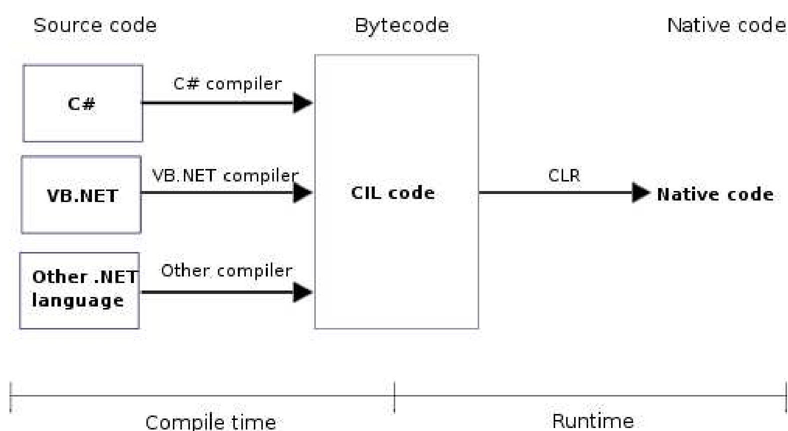
Programavimo kontekste vienetų testavimas yra procedūra naudojama parodyti, jog atskiri programinio kodo vienetai veikia teisingai. Vienetas - yra pati mažiausia testuojama taikomosios programos dalis. Procedūriniame programavime vienetu gali būti atskira paprogramė, funkcija, procedūra, metodas.

Svarbi vienetų testų metrika – *kodo padengimais %* (code coverage) vienetų testais. Ši metrika parodo, kiek kodo eilučių santykinai nuo visų kodo eilučių buvo įvykdyta, vykdant vienetų testavimą. Žinoma, verta pabrėžti, kad šios metrikos nereikia suabsoliutinti ir

100% kodo padengimas nereiškia, kad visas kodas yra ištestuotas ir teisingas, nes atsiranda papildomos problemos, pvz.: kodo mutacijos.

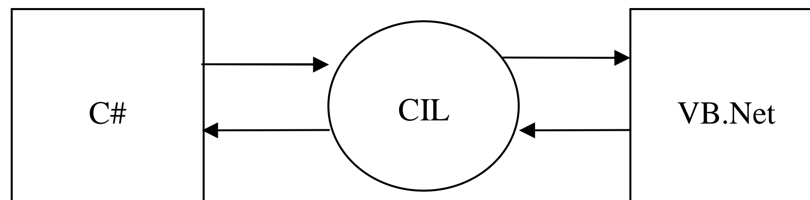
2.6. CIL (Common Intermediate Language)

CIL – tai tarpinė kodo kalba, kurią kodo interpretatorius verčia iš išeities kodo, o vėliau ją vykdo virtuali mašina arba kompiliatorius, versdamas į mašininį kodą. CIL ankščiau vadinta MSIL (Microsoft Intermediate Language), nes jos autorius yra korporacija Microsoft, tačiau visuotinai patvirtinus CLI kaip standartą, kurį naudoja .net ir Mono platformos, pavadinimas buvo pakeistas. Ši kalba paremta ECMA-335 arba ISO/IEC 23271:2006 standartu (dar kitaip vadinamu Common Language Infrastructure)[12].



4 pav. .Net platformos programų transliavimo infrastruktūra

CIL yra bendrinė, tarpinė, objektiškai orientuota kalba. Kalbos bendrumas leidžia sudaryti įvairias aukšto lygio kalbas, kurios interpretuojamos į vieną nuo platformos nepriklausančia kalbą. Labai tinkamas pavyzdys yra pati Microsoft .Net platforma. Joje galima programuoti C#, VisualBasic.Net, F# ir kitomis kalbomis, kurios visos bus konvertuotos į tokį pat CIL kodą. Be to, įmanomas ir atvirkštinis veiksmas, kurio metu CIL kodas konvertuojamas atgal į aukšto lygio kodą. Tai suteikia galimybę VB.Net kalba (ar bet kuria kita. standartą atitinkančia) parašyta programą paversti į C# (ar į bet kurią kitą, standartą atitinkančią) ir atvirkščiai. Kalbos tarpiškumas nusako, kad tai tėra tarpinis variantas tarp realiai procesoriaus vykdomo mašininio kodo ir žmogaus rašomo aukšto lygmens programinio kodo. Tai aiškiai vaizduojama pav. 1. CIL – objektiškai orientuota kalba, tai leidžia išsaugoti reikalingą papildomą (meta) informaciją – klasių, metodų, kintamųjų, struktūrų vardus, tipus, gražinimas reikšmes, etc.



5 pav. Vienos CIL pagrindu parašytos kalbos transformavimas į kitą

2.7. Programos pavyzdžiai C# ir CIL kalba

Programos pavyzdys, parašytas C# kalba:

```
public class Foo
{
        public static int Add(a, b)
        {
                a = a + b;
                return a;
        }
}
```

Ta pati programa CIL kalba:

```
.class public Foo
 {
     .metodą public static int32 Add(int32, int32) cil managed
     {
          .maxstack 2
          .locals init (
              [0] int32 num1,
              [1] int32 num2
          )

          ldloc.0
          ldloc.1
          add
          stloc.0     // a = a + b;
          ret         // return a;
     }
 }
```

Šiame pavyzdyje parodytas žmogui perskaitomas CIL kalbos pavidalas – taupant vietą ir sistemos resursus, kodo interpretatoriai paprastai verčia išeities tekstus į taip vadinamą CIL bitinį kodą (bytecode). Microsoft savo bitinį kodą saugo Common Language Runtime (CLR) formate, kuris vėliau perduodamas JIT kompiliatoriui ir vykdomas (ar kompiliuojamas į mašininį).

Programos iš CIL kalbos gali būti sukompiliuotos į mašininį kodą, skirta daugumai dabar egzistuojančių populiariausių mašininių platformų:

- x86

- x86-64
- MIPS
- SuperH
- PowerPC
- SPARC

2.8. CIL instrukcijų tipai

Šiuo metu naudojamos 236 CIL instrukcijos (opkodai), kurios skirstomos į šias grupes [13]:

- Reikšmių užkrovimo ir saugojimo (steke).
- Aritmetinės
- Tipų pervertinimo (float → int, etc.)
- Objektų kūrimas ir manipuliavimas
- Operandų steko manipuliavimas
- Šakojimosi (branch)
- Metodų kvietimas ir reikšmių grąžinimas
- Išimčių (exceptions) kvietimas
- Monitoriumi paremtas daugiagijiškumas (multithreading)

2.9. CIL instrukcijos

1 lentelė. CIL komandų sąrašas [13]

Opkodas	Instrukcija	Aprašymas
0x58	add	Sudedamos dvi reikšmės, grąžinamas rezultatas
0xD6	add.ovf	Sudėti natūralius skaičius su ženklu su patikrinimu
0xD7	add.ovf.un	Sudėti natūralius skaičius be ženklo su patikrinimu
0x5F	and	Bitinis ir
0xFE 0x00	arglist	Visų metodo argumentų sąrašas
0x3B	beq <int32 (target)>	Šakojimasis į tikslą, jeigu lygu.
0x2E	beq.s <int8 (target)>	Šakojimasis į tikslą, jeigu lygu, trumpoji forma.
0x3C	bge <int32 (target)>	Šakojimasis į tikslą, jeigu daugiau negu arba lygu.
0x2F	bge.s <int8 (target)>	Šakojimasis į tikslą, jeigu daugiau negu arba lygu, trumpoji forma.
0x41	bge.un <int32 (target)>	Šakojimasis į tikslą, jeigu daugiau negu arba lygu(be ženklo arba nesurikiuotas).
0x34	bge.un.s <int8 (target)>	Šakojimasis į tikslą, jeigu daugiau negu arba lygu(be ženklo arba nesurikiuotas), trumpoji forma
0x3D	bgt <int32 (target)>	Šakojimasis į tikslą, jeigu daugiau negu.
0x30	bgt.s <int8 (target)>	Šakojimasis į tikslą, jeigu daugiau negu, trumpoji forma.
0x42	bgt.un <int32 (target)>	Šakojimasis į tikslą, jeigu daugiau negu (be ženklo arba nesurikiuotas).
0x35	bgt.un.s <int8 (target)>	Šakojimasis į tikslą, jeigu daugiau negu (be ženklo arba

		nesurikiuotas), trumpoji forma.
0x3E	ble <int32 (target)>	Šakojimasis į tikslą , jeigu mažiau negu arba lygu.
0x31	ble.s <int8 (target)>	Šakojimasis į tikslą , jeigu mažiau negu arba lygu, trumpoji forma.
0x43	ble.un <int32 (target)>	Šakojimasis į tikslą , jeigu mažiau negu arba lygu(be ženklo arba nesurikiuotas).
0x36	ble.un.s <int8 (target)>	Šakojimasis į tikslą , jeigu mažiau negu arba lygu(be ženklo arba nesurikiuotas), trumpoji forma
0x3F	blt <int32 (target)>	Šakojimasis į tikslą , jeigu mažiau negu .
0x32	blt.s <int8 (target)>	Šakojimasis į tikslą , jeigu mažiau negu , trumpoji forma.
0x44	blt.un <int32 (target)>	Šakojimasis į tikslą , jeigu mažiau negu (be ženklo arba nesurikiuotas).
0x37	blt.un.s <int8 (target)>	Šakojimasis į tikslą , jeigu mažiau negu (be ženklo arba nesurikiuotas), trumpoji forma.
0x40	bne.un <int32 (target)>	Šakojimasis į tikslą, jeigu nelygu arba nesurikiuota.
0x33	bne.un.s <int8 (target)>	Šakojimasis į tikslą, jeigu nelygu arba nesurikiuota, trumpoji forma.
0x8C	box <typeTok>	Versti boxable reikšmę į boxed formą
0x38	br <int32 (target)>	Šakojimasis į tikslą.
0x2B	br.s <int8 (target)>	Šakojimasis į tikslą, trumpoji forma.
0x01	break	Informuoti, kad pasiektas breakpoint
0x39	brfalse <int32 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra nulis(false).
0x2C	brfalse.s <int8 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra nulis(false), trumpoji forma.
0x3A	brinst <int32 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra ne nulinis objektas(tas pats, kaip ir brtrue).
0x2D	brinst.s <int8 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra ne nulinis objektas, trumpoji forma (tas pats, kaip ir brtrue.s).
0x39	brnull <int32 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra null(tas pats, kaip ir brfalse).
0x2C	brnull <int8 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra null(tas pats, kaip ir brfalse.s), trumpoji forma.
0x3A	brtrue <int32 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra ne nulis (true).
0x2D	brtrue.s <int8 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra ne nulis (true), trumpoji forma.
0x39	brzero <int32 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra nulis(tas pats, kaip ir brfalse).
0x2C	brzero <int8 (target)>	Šakojimasis į tikslą, jeigu reikšmė yra nulis(tas pats, kaip ir brfalse.s), trumpoji forma.
0x28	call <method>	Kviesti metodą.
0x29	calli <callsitedescr>	Iškviesti metodą nurodyte steke su callsitedescr nurodytais argumentais
0x6F	callvirt <method>	Kviesti a metodą
0x74	castclass <class>	Versti į klasės tipą
0xFE 0x01	ceq	Įkelti 1 (tipas - int32) jei value1 equals value2, priešingu atveju Įkelti 0.
0xFE 0x02	cgt	Įkelti 1 (tipas - int32) jei value1 > value2, priešingu atveju Įkelti 0.
0xFE 0x03	cgt.un	Įkelti 1 (tipas - int32) jei value1 > value2, be ženklo arba nesurikiuotas, priešingu atveju Įkelti 0.
0xC3	ckfinite	Rodyti išimtį, jeigu skaičius nėra baigtinis.
0xFE 0x04	clt	Įkelti 1 (tipas - int32) jei value1 < value2, priešingu atveju Įkelti 0.
0xFE 0x05	clt.un	Įkelti 1 (tipas - int32) jeigu value1 < value2, be ženklo arba nesurikiuotas, priešingu atveju įkelti 0.
0xFE 0x16	constrained. <thisType> [prefix]	Kviesti virtualų metodą
0xD3	conv.i	Versti į native int, įkelti native int į steką.
0x67	conv.i1	Versti į int8, įkelti int32 į steką.
0x68	conv.i2	Versti į int16, įkelti int32 į steką.

0x69	conv.i4	Versti į int32, įkelti int32 į steką.
0x6A	conv.i8	Versti į int64, įkelti int64 į steką.
0xD4	conv.ovf.i	Versti į a native int (į steką kaip native int) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x8A	conv.ovf.i.un	Verst be ženklo to a native int (į steką kaip native int) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0xB3	conv.ovf.i1	Versti į int8 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x82	conv.ovf.i1.un	Verst be ženklo į int8 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0xB5	conv.ovf.i2	Versti į int16 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x83	conv.ovf.i2.un	Verst be ženklo į int16 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0xB7	conv.ovf.i4	Versti į int32 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x84	conv.ovf.i4.un	Verst be ženklo to int32 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0xB9	conv.ovf.i8	Versti į an int64 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x85	conv.ovf.i8.un	Verst be ženklo į int64 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0xD5	conv.ovf.u	Versti į be ženklo int (į steką kaip native int) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x8B	conv.ovf.u.un	Verst be ženklo be ženklo int (į steką kaip native int) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0xB4	conv.ovf.u1	Versti į be ženklo int8 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x86	conv.ovf.u1.un	Verst be ženklo į be ženklo int8 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0xB6	conv.ovf.u2	Versti į be ženklo int16 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x87	conv.ovf.u2.un	Verst be ženklo į be ženklo int16 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0xB8	conv.ovf.u4	Versti į be ženklo int32 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x88	conv.ovf.u4.un	Verst be ženklo į be ženklo int32 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0xBA	conv.ovf.u8	Versti į be ženklo int64 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x89	conv.ovf.u8.un	Versti be ženklo į be ženklo int64 (į steką kaip int32) ir rodyti išimtį, jeigu viršijama ribinė reikšmė.
0x76	conv.r.un	Versti be ženklo integer į floating-point, įkelti F į steką.
0x6B	conv.r4	Versti į float32, įkelti F į steką.
0x6C	conv.r8	Versti į float64, įkelti F į steką.
0xE0	conv.u	Versti į native be ženklo int, įkelti native int į steką.
0xD2	conv.u1	Versti į be ženklo int8, įkelti int32 į steką.
0xD1	conv.u2	Versti į be ženklo int16, įkelti int32 į steką.
0x6D	conv.u4	Versti į be ženklo int32, įkelti int32 į steką.
0x6E	conv.u8	Versti į be ženklo int64, įkelti int64 į steką.
0xFE 0x17	cpblk	Perkopijuoti iš vienos atminties vietos į kitą.
0x70	cpobj <typeTok>	Kopijuoti tipo reikšme iš šaltinio į tikslą
0x5B	div	Dalinti dvi reikšmes
0x5C	div.un	Dalinti dvi reikšmes, be ženklo
0x25	dup	Dubliuoti viršutinė steko reikšmę
0xDC	endfault	Išimties bloko pabaiga
0xFE 0X11	endfilter	Baigti išimties bloko filtrą
0xDC	endfinally	Finnaly bloko pabaiga
0x4C	idind.u8	Netiesiogiai užkrauti reikšme su tipu be ženklo int64 kaip int64 į steką (tas pats, kaip ir ldind.i8).

0xFE 0x18	initblk	Visus atminties blokus nustatyti į duotą bito reikšmę
0xFE 0x15	initobj <typeTok>	Sukurti reikšmę duotu adresu
0x75	isinst <clkaips>	Tikrinti jei obj yra clkaips objektas, gražinti arba null arba patį objektą
0x27	jmp <method>	Išeiti iš metodo ir šokti į nurodytą metodą
0xFE 0x09	ldarg <uint16 (num)>	Užkrauti surikiuotą argumentą num į steką.
0x02	ldarg.0	Užkrauti argumentą 0 į steką.
0x03	ldarg.1	Užkrauti argumentą 1 į steką.
0x04	ldarg.2	Užkrauti argumentą 2 į steką.
0x05	ldarg.3	Užkrauti argumentą 3 į steką.
0x0E	ldarg.s <uin8 (num)>	Užkrauti surikiuotą argumentą num į steką, trumpoji forma.
0xFE 0x0A	ldarga <uint16 (argNum)>	Gauti argNum atminties adresą
0x0F	ldarga.s <uint8 (argNum)>	Gauti argNum atminties adresą, trumpoji forma.
0x20	ldc.i4 <int32 (num)>	Įkelti num tipas - int32 į steką kaip int32.
0x16	ldc.i4.0	Įkelti 0 į steką kaip int32.
0x17	ldc.i4.1	Įkelti 1 į steką kaip int32.
0x18	ldc.i4.2	Įkelti 2 į steką kaip int32.
0x19	ldc.i4.3	Įkelti 3 į steką kaip int32.
0x1A	ldc.i4.4	Įkelti 4 į steką kaip int32.
0x1B	ldc.i4.5	Įkelti 5 į steką kaip int32.
0x1C	ldc.i4.6	Įkelti 6 į steką kaip int32.
0x1D	ldc.i4.7	Įkelti 7 į steką kaip int32.
0x1E	ldc.i4.8	Įkelti 8 į steką kaip int32.
0x15	ldc.i4.m1	Įkelti -1 į steką kaip int32.
0x15	ldc.i4.M1	Įkelti -1 tipas - int32 į steką kaip int32 (tas pats, kaip ir ldc.i4.m1).
0x1F	ldc.i4.s <int8 (num)>	Įkelti num į steką kaip int32, trumpoji forma.
0x21	ldc.i8 <int64 (num)>	Įkelti num tipas - int64 į steką kaip int64.
0x22	ldc.r4 <float32 (num)>	Įkelti num tipas - float32 į steką kaip F.
0x23	ldc.r8 <float64 (num)>	Įkelti num tipas - float64 į steką kaip F.
0xA3	ldelem <typeTok>	Užkrauti elementą į steko viršų.
0x97	ldelem.i	Užkrauti elementą su tipu native int į steko viršų kaip native int.
0x90	ldelem.i1	Užkrauti elementą su tipu int8 į steko viršų kaip int32.
0x92	ldelem.i2	Užkrauti elementą su tipu int16 į steko viršų kaip int32.
0x94	ldelem.i4	Užkrauti elementą su tipu int32 į steko viršų kaip int32.
0x96	ldelem.i8	Užkrauti elementą su tipu int64 į steko viršų kaip int64.
0x98	ldelem.r4	Užkrauti elementą su tipu float32 į steko viršų kaip F
0x99	ldelem.r8	Užkrauti elementą su tipu float64 į steko viršų kaip F.
0x9A	ldelem.ref	Užkrauti elementą į steko viršų kaip objektą
0x91	ldelem.u1	Užkrauti elementą su tipu be ženklo int8 į steko viršų kaip int32.
0x93	ldelem.u2	Užkrauti elementą su tipu be ženklo int16 į steko viršų kaip int32.
0x95	ldelem.u4	Užkrauti elementą su tipu be ženklo int32 į steko viršų kaip int32.
0x96	ldelem.u8	Užkrauti elementą su tipu be ženklo int64 į steko viršų kaip an int64 (tas pats, kaip ir ldelem.i8).
0x8F	ldelema <clkaips>	Užkrauti elemento adresą į steko viršų.
0x7B	ldfld <field>	Įkelti lauko adresą objektui (arba reikšmės tipą) obj, į steką.
0x7C	ldflda <field>	Įkelti lauko adresą objektui obj į steką.
0xFE 0x06	ldftn <method>	Įkelti rodyklę į metodą nurodytą pagal method, į steką.
0x4D	ldind.i	Netiesiogiai užkrauti reikšme su tipu native int kaip native int į steką
0x46	ldind.i1	Netiesiogiai užkrauti reikšme su tipu int8 kaip int32 į steką.
0x48	ldind.i2	Netiesiogiai užkrauti reikšme su tipu int16 kaip int32 į steką.
0x4A	ldind.i4	Netiesiogiai užkrauti reikšme su tipu int32 kaip int32 į steką.
0x4C	ldind.i8	Netiesiogiai užkrauti reikšme su tipu int64 kaip int64 į steką.
0x4E	ldind.r4	Netiesiogiai užkrauti reikšme su tipu float32 kaip F į steką.

0x4F	ldind.r8	Netiesiogiai užkrauti reikšme su tipu float64 kaip F į steką.
0x50	ldind.ref	Netiesiogiai užkrauti reikšme su tipu object ref kaip O į steką.
0x47	ldind.u1	Netiesiogiai užkrauti reikšme su tipu be ženklo int8 kaip int32 į steką
0x49	ldind.u2	Netiesiogiai užkrauti reikšme su tipu be ženklo int16 kaip int32 į steką
0x4B	ldind.u4	Netiesiogiai užkrauti reikšme su tipu be ženklo int32 kaip int32 į steką
0x8E	ldlen	Įkelti masyvo ilgį į steką.
0xFE 0x0C	ldloc <uint16 (indx)>	Užkrauti lokalų kintamąjį su indeksu indx į steką.
0x06	ldloc.0	Užkrauti lokalų kintamąjį 0 į steką.
0x07	ldloc.1	Užkrauti lokalų kintamąjį 1 į steką.
0x08	ldloc.2	Užkrauti lokalų kintamąjį 2 į steką.
0x09	ldloc.3	Užkrauti lokalų kintamąjį 3 į steką.
0x11	ldloc.s <uint8 (indx)>	Užkrauti lokalų kintamąjį su indeksu indx į steką, trumpoji forma.
0xFE 0x0D	ldloca <uint16 (indx)>	Užkrauti adresą objekto su indeksu indx.
0x12	ldloca.s <uint8 (indx)>	Užkrauti adresą objekto su indeksu indx, trumpoji forma.
0x14	ldnull	Įkelti a null reference į steką.
0x71	ldobj <typeTok>	Kopijuoti reikšmę nurodyta adrese į steką
0x7E	ldsfld <field>	Įkelti lauko reikšmę į steką.
0x7F	ldsflda <field>	Įkelti statinio lauko reikšmę į steką
0x72	ldstr <string>	Įkelti eilutės objektą i eilutę
0xD0	ldtoken <token>	Versti metaduomenis į reikalingus vykdymo metu.
0xFE 0x07	ldvirtftn <method>	Įkelti virtualaus metodo adresą į steką.
0xDD	leave <int32 (target)>	Išeiti iš apsaugotos kodo zonos
0xDE	leave.s <int8 (target)>	Išeiti iš apsaugotos kodo zonos, trumpoji forma.
0xFE 0x0F	localloc	Išskirtį atmintį
0xC6	mkrefany <clkaips>	Įkelti a reference iš ptr tipo - clkaips į steką.
0x5A	mul	Dauginti reikšmės.
0xD8	mul.ovf.<type>	Dauginti sveikus skaičius
0xD9	mul.ovf.un	Dauginti be ženklo integer reikšmės.
0x65	neg	Neigiama reikšmė
0x8D	newarr <etype>	Sukurti masyvą su tipu - etype.
0x73	newobj <ctor>	Sukurti objektą ir kviesti konstruktorių ctor.
0xFE 0x19	no. typecheck, rangecheck, nullcheck [prefix]	Klaidos patikrinimo opkodų dalis, paprastai kviečiama kitų opkodų, nenaudojama
0x00	nop	Nedaryti nieko
0x66	not	Bitinis neigimas
0x60	or	Bitinis neigimas, sveikų skaičių
0x26	pop	Išmesti reikšmę iš steko.
0xFE 0x1E	readonly. [prefix]	Masyvas tik skaitymui
0xFE 0x1D	refanytype	Masyvas tik skaitymui su rodykle
0xC2	refanyval <type>	Įkelti adresą saugomą nurodytame šaltinyje
0x5D	rem	Atmintis dalinant vieną skaičių iš kito
0x5E	rem.un	Atmintis dalinant vieną skaičių iš kito be ženklo
0x2A	ret	Gražinamas metodas
0xFE 0x1A	rethrow	Pakartojama išimtis
0x62	shl	Perkelti sveiką skaičių į kairę (shift in zeros), grąžinamas sveikas skaičius.
0x63	shr	Perkelti sveiką skaičių į right (shift in sign), grąžinamas sveikas skaičius.
0x64	shr.un	Perkelti sveiką skaičių į right (shift in zero), grąžinamas sveikas skaičius.
0xFE 0x1C	sizeof <typeTok>	Įkelti dydį, baitais, be ženklo int32.
0xFE 0x0B	starg <uint16 (num)>	Išsaugoti reikšmę į argumentą numeruota num.
0x10	starg.s <uint8 (num)>	Išsaugoti reikšmę į argumentą numeruota num, trumpoji forma.
0xA4	stelem <typeTok>	Pakeisti masyvo elementą su indeksu su value į steką
0x9B	stelem.i	Pakeisti masyvo elementą su indeksu su i value į steką.

0x9C	stelem.i1	Pakeisti masyvo elementą su indeksu su int8 value į steką.
0x9D	stelem.i2	Pakeisti masyvo elementą su indeksu su int16 value į steką.
0x9E	stelem.i4	Pakeisti masyvo elementą su indeksu su int32 value į steką.
0x9F	stelem.i8	Pakeisti masyvo elementą su indeksu su int64 value į steką.
0xA0	stelem.r4	Pakeisti masyvo elementą su indeksu su float32 value į steką.
0xA1	stelem.r8	Pakeisti masyvo elementą su indeksu su float64 value į steką.
0xA2	stelem.ref	Pakeisti masyvo elementą su indeksu su ref value į steką.
0x7D	stfld <field>	Pakeisti lauko reikmę su objektu reikšme
0xDF	stind.i	Išsaugoti reikšmę su tipu native int atmintyje su adresu
0x52	stind.i1	Išsaugoti reikšmę su tipu int8 atmintyje su adresu
0x53	stind.i2	Išsaugoti reikšmę su tipu int16 atmintyje su adresu
0x54	stind.i4	Išsaugoti reikšmę su tipu int32 atmintyje su adresu
0x55	stind.i8	Išsaugoti reikšmę su tipu int64 atmintyje su adresu
0x56	stind.r4	Išsaugoti reikšmę su tipu float32 atmintyje su adresu
0x57	stind.r8	Išsaugoti reikšmę su tipu float64 atmintyje su adresu
0x51	stind.ref	Išsaugoti reikšmę su tipu object ref (type O) atmintyje su adresu
0xFE 0x0E	stloc <uint16 (indx)>	Išmesti reikšmę iš steko viršaus į lokalų kintamąjį indx.
0x0A	stloc.0	Išmesti reikšmę iš steko viršaus į lokalų kintamąjį 0.
0x0B	stloc.1	Išmesti reikšmę iš steko viršaus į lokalų kintamąjį 1.
0x0C	stloc.2	Išmesti reikšmę iš steko viršaus į lokalų kintamąjį 2.
0x0D	stloc.3	Išmesti reikšmę iš steko viršaus į lokalų kintamąjį 3.
0x13	stloc.s <uint8 (indx)>	Išmesti reikšmę iš steko viršaus į lokalų kintamąji indx, trumpoji forma.
0x81	stobj <typeTok>	Išsaugoti objekto typeTok adresą
0x80	stsfld <field>	Pakeisti lauko reikšmę duotąją lauke
0x59	sub	Atimti dvi reikšmes
0xDA	sub.ovf	Atimti dvi reikšmes (dviejų integer tipo skaičių)
0xDB	sub.ovf.un	Atimti dvi reikšmes (dviejų integer tipo skaičių, be ženklo)
0x45	switch <uint32, int32,int32 (t1..tN)>	Peršokti į vieną iš reikšmių.
0xFE 0x14	tail. [prefix]	Metodo pabaiga
0x7A	throw	Rodyti išimtį
0xFE 0x12	unaligned. (alignment) [prefix]	Rodyklės perkėlimas
0x79	unbox <valuetype>	Ištraukti reikšmę iš objekto
0xA5	unbox.any <typeTok>	Ištraukti reikšmę iš objekto
0xFE 0x13	volatile. [prefix]	Uždrausti operaciją
0x61	xor	Bitinis XOR

2.10. JIT (Just-in-time) kompiliatorius

JIT – tai metodas, naudojamas kompiliavime, kai programa nėra sukompiliuojama ir užkraunama visa į atmintį, bet kompiliuojamos tik tos programos dalys, kurios tuo metu reikalingos programai vykdyti.

2.11. AOT (Ahead-of-time) kompiliatorius

AOT – tai metodas, naudojamas kompiliavime, kai visa programa iškart verčiama į mašininį kodą. Jis naudingas tada, kai JIT kompiliavimas yra per lėtas ar per daug

sudėtingas. AOT kompiliatorius nuo įprasto skiriasi tuo, kad kompiliuoja iš bitinio kodo, o ne iš išeities tekstų.

2.12. Microsoft CCI

Microsoft Research Common Compiler Infrastructure [14] – tai bibliotekų rinkinys, skirtas palengvinti užduotis susijusias su programų analize ir kompiliatorių kūrimu. Jos leidžia efektyviai analizuoti ir keisti .Net kalbos CLR failus. Šias bibliotekas naudoja nemažai žymių .net skirtų kodo analizės įrankių, tokių kaip FxCop [15]. Bibliotekos labai palengvina statinę ir dinaminę programų analizę, leidžia naudotis ir operuoti bitininiame kode esančiais metaduomenimis. Šios bibliotekos gerokai palengvina kodo analizės, kompiliavimo ar kitokių, tiesiai su bitiniu kodu dirbančių įrankių kūrimą.

2.13. SMT

SMT – tai Satisfiability Modulo Theories [16]. Tai matematinės logikos šaka, kurios pagrindinis uždavinys yra spręsti teoremas (neigti jas arba įrodyti), kurios turi tenkinti tam tikras sąlygas ar apribojimus. Tai viena iš predikatų logikos dalių. Paprasčiau kalbant, turimas uždavinys: lygtis $x - y \leq c$ ir reikia rasti realias reikšmes kintamiesiems x, y ir konstantą c. Iovos universitete [17] buvo paskelbta iniciatyva sukurti testų rinkinį, kuris padėtų įvertinti SMT bibliotekų (programinių) kokybę, ilgainiui tai tapo standartu – buvo standartizuota matematinių išraiškų kalba SMT-LIB ir ja naudojantis galima programinėje įrangoje panaudoti bet kurią iš SMT-LIB iniciatyvos bibliotekų. Keletas skirtingų SMT bibliotekų, kurios buvo aktyvios darbo rašymo metu:

- Alt-Ergo [18]
- Barcelogic [19]
- Beavei [20]
- Boolector [21]
- MathSAT [22]
- OpenSMT [23]
- Yices [24]
- Z3 [25]

SMT naudojamas formalioje analizėje, optimizavimo ir verifikavimo uždaviniuose, gaminant programinę ir aparatūrinę įrangą. SMT kodo bibliotekoms padedant, galima realizuoti simbolinį vykdymą.

Tačiau dabartinės galimybės SMT bibliotekoms leidžia spręsti uždaviniuos su deterministinėmis sąlygomis, o tai yra nemažas trūkumas, jeigu norima pritaikyti realioms sistemos. **Dėl šio trūkumo SMT bibliotekos dar negali išspręsti lygčių su slankaus kablelio skaičiais [25]**. Tačiau darbo rašymo metu, buvo išleista nauja SMT-LIB [17] specifikacijos versija, kurioje aprašyti būdai, skaičiuoti realiųjų skaičių aibės priklausiančias reikšmes. Tai lems naujų įrankių atsiradimą ir ši problema tūrėtų būti išspręsta.

2.14. Simbolinis vykdymas

Simbolinio vykdymo (Symbolic execution) idėja nėra nauja, ją dar 1976 metais pasiūlė J.C King [1]. Autorius manė, kad ši technologija galėtų būti naudojama kartu su tuo metu progresyviais laikytais formalios analizės metodais.

Simbolis vykdymas – tai programos interpretavimo būdas, kai kodo interpretatorius operuoja ne su realiais duomenis, bet su simboliniais, t.y. programa vykdoma keičiant realias reikšmes į kintamuosius (simbolius) [1], tai yra statinės kodo analizės būdas. Idėja pristatyta dar 1976 metais, tačiau tik nuo maždaug 2006 [8] metų pradėta taikyti praktikoje. Tokio interpretatoriaus darbo rezultatas yra simbolinės sekos su visais įmanomais įvykdyti keliais iš analizuojamo kodo. Norint realizuoti SE, algoritmas turi turėti:

- Simbolines kiekvieno realaus kintamojo reikšmes.
- Simbolinio vykdymo medį (Symbolic execution tree, SET), kuriame saugomos visos įmanomos programos vykdymo šakos.
- Kelio būsena (Path condition, PC) – nurodo kokioje analizuojamos programos kodo vietoje vykdoma analizė.

Simbolinis vykdymas naudojamas daugiausiai programų testavimui, nes yra realu gauti visus įmanomus programos vykdymo kelius iš vykdymo medžio, analizuojant ją simboliškai.

2.15. SE algoritmo veikimas

Norėdami aiškiau suprasti metodologiją, įsigilinkime į paprastą pavyzdį, duotą pav. 4. Duota maža paprogramė, kurią reikia įvykdyti simboliškai.

```
int x, y;
[1] if (x > y)
[2] result = x - y;
[3] else
[4] result = y - x;
[5] assert (result > 0);
```

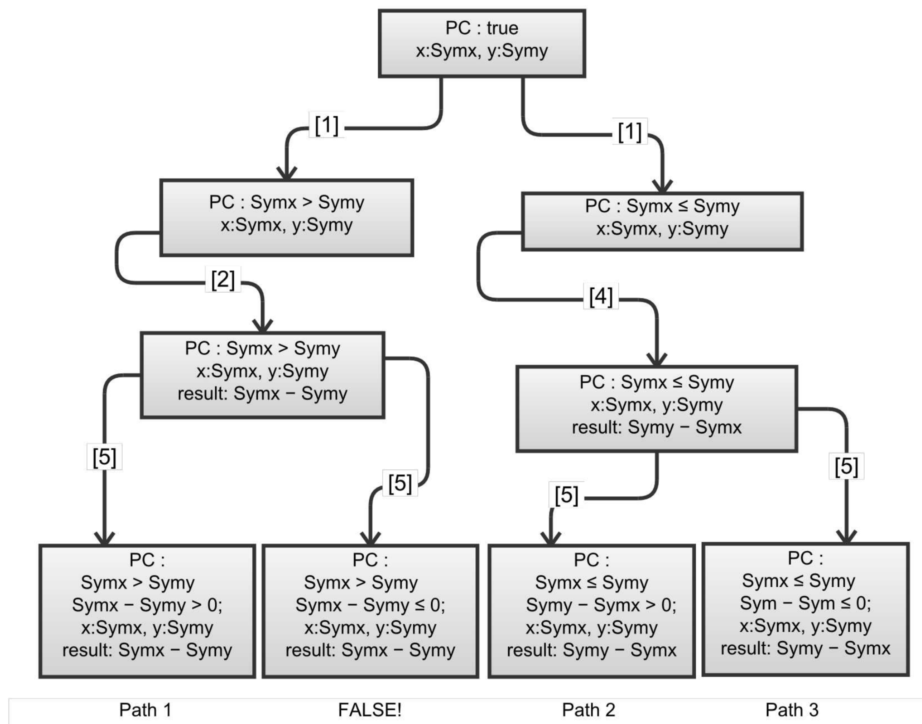
6 pav. Paruoštas vykdyti metodas

Pradedant simbolinį vykdymą PC visada turi reikšmę TRUE. Esantys realūs funkcijos įėjimo kintamieji x ir y keičiami simboliniais – šiuo atveju pavadintais SymX ir SymY. Nagrinėjant kodą sutinkama *if* (1 kodo eilutė) sakinys, algoritmas žino, kad programa skaidosi į dvi šakas, kuriama medžio struktūra su dvejomis šakomis. Dešinėje pusėje nuo 2 eilutės nagrinėjamas atvejis kai *symx* > *symy*. PC įsimena šią reikšmę. Programos kodas sekamas toliau ir 5 eilutėje sutinkamas matematinis veiksmas – atimtis. Į medžio struktūrą įrašomi reikalingi duomenys. Toliau analizuojama 5 kodo eilutė (kitos praleidžiamos, nes tai – *if* sakinio dalys, kurios šitame programos vykdymo kelyje neatitinka sąlygos ir yra nevykdomos), joje sutinkamas *assert*() funkcijos kvietimas. Ši funkcija grąžina duotos sąlygos teisingumą (true ar false) nusakančią reikšmę. Šiuo atveju tikrinama ar gautas atimties veiksmo rezultatas (*result* kintamasis) yra didesnis už nulį. *Assert*() funkcijoje vėl randamas skaidimosi veiksmas(palyginimas >), algoritmas skaido SET medžio šakas (nes vienu atveju nagrinėjama >, o kitu <= galimybė). Gaunamas pirmas programos vykdymo kelias (*path1*), programa pateks į šią būseną, kai iš *symx* atėmus *symy* bus gautas teigiamas rezultatas (> 0). Kitas kelias SET medyje neatsiranda (*FALSE!*), nes pagal analizuojama algoritmą, neįmanoma, kad programa patektų į tokią būseną (iš didesnio skaičiau atėmus mažesnį, visada bus gautas rezultatas didesnis už 0).

Analogiškai vykdoma ir dešinė medžio pusė. Baigus simbolinį vykdymą gauti trys programos vykdymo keliai:

- *symx* – *symy* > 0
- *symy* – *symx* < 0
- *symy* – *symx* >= 0

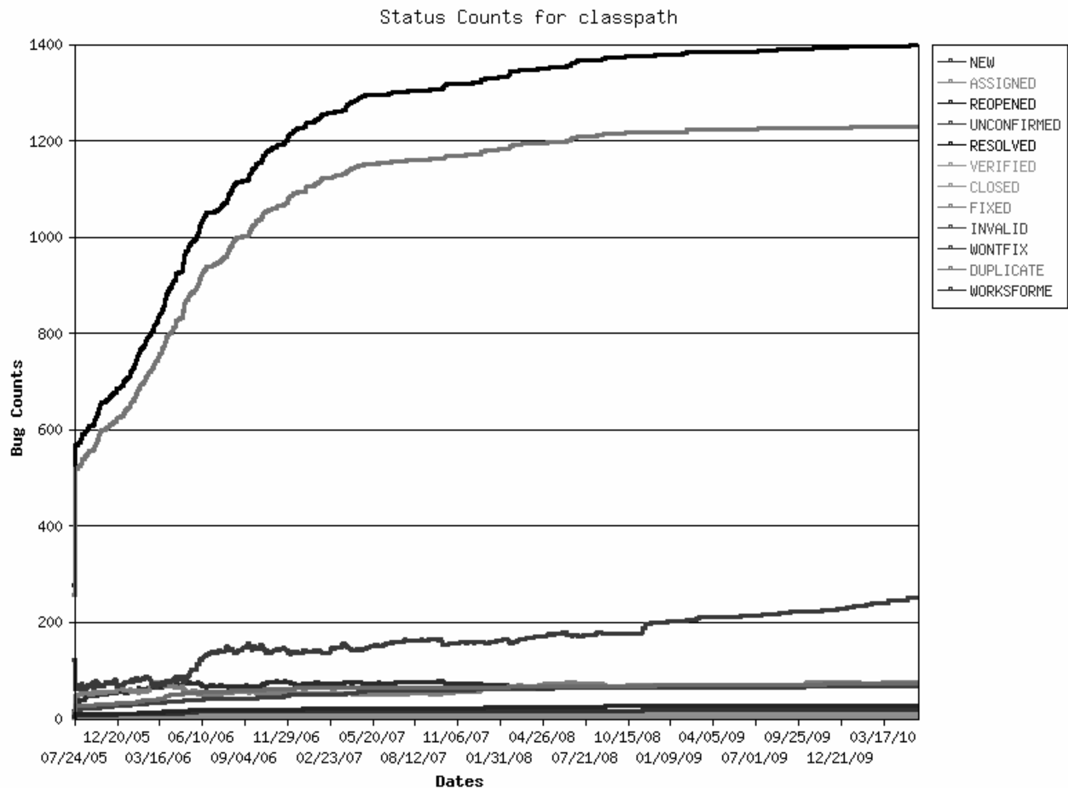
Gautas rezultatas 100% apibrėžia galimą programos vykdymo aibę, šiuo atveju gauname 3 skirtingus programos vykdymo kelius.



7 pav. Simbolinio vykdymo medis

3. Klaidų sekimo informacinė sistema „*CRUNCHBUG*”

Programinės įrangos klaidos – tai žmonių, daug rečiau kompiliatorių, sukurtos programinės įrangos kodo netikslumai, dėl kurių teisingas programos darbas sutrinka. Deja, žmogiškoji prigimtis lemia tokių klaidų atsiradimą, todėl tai verčia ieškoti priemonių, kaip mažinti klaidų skaičių, jų taisymo spartą ir tuo pačiu didinti programuotojų darbo efektyvumą ir įmonės pelną.



8 pav. GNU projekto klaidų statistika [26]

Šioje diagramoje (pav. 8) pateikta informaciją apie GNU projekte praneštas klaidas ir jų taisymus. Kaip matyti, netgi žymūs projektai neišvengia didelio skaičiaus klaidų, todėl būtinas tam tikra sistema joms sekti, apie jas informuoti, keisti informaciją apie jų būklę ir t.t.

Pasiekti absoliučiai teisingą programinės įrangos veikimą nelabai įmanoma, tam egzistuoja daugybė priežasčių. Viena pagrindinių – programuotojas neturi absoliučiai visų reikalingų žinių toje srityje [27].

Taigi, programinė įranga (informacinė sistema), skirta klaidų administravimui ir registravimui yra aktuali ir yra panaudojama kiekvienam didesniam programinės įrangos projektui.

3.1. Tikslas

Projekto tikslas yra sukurti klaidų registracijos ir sekimo informacinę sistemą, kuri leistų efektyviai ir patogiai organizuoti klaidų taisymo procesą, turėtų galimybę patogiai vartotojams pranešti apie klaidas, integruotųsi su kitomis programomis, kurių darbą reikia stebėti.

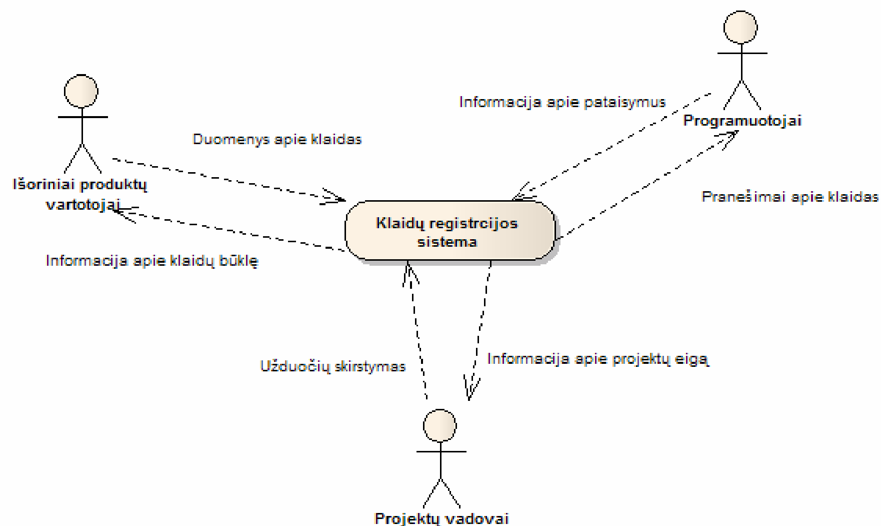
3.2. Programų sistemos funkcijos

Kuriamos programinės tikslai yra užtikrinti tinkamą kuriamų programų klaidų taisymo mechanizmo įgyvendinimą, sąveikų tarp skirtingų profesijų įmonės darbuotojų ir įmonės klientų įgyvendinimą, taisant pastebėtas klaidas, klaidų stebėjimas ir analizė. Pagrindinės sistemos funkcijos yra:

- Klaidų administravimas
- Sistemos administravimas
- Projektų, pagal kuriuos skirstomos klaidos administravimas
- Detali klaidų paieška.
- Patogus ir greitas būdas pranešti apie klaidas.
- Keičiama ir pritaikoma darbo metodika (custom workflow).
- Darbo jėgos paskirstymas ir laiko apskaita.
- Programinės įrangos gyvavimo ciklo palaikymas.
- Atskiras įrankis, skirtas surinkti klaidų pranešimams iš sistemos (Reporter)
- Galimybė integruotis su kitomis sistemomis
- Ekrano filmavimo galimybė ir video medžiagos prisegimas prie klaidos pranešimo

3.3. Sistemos kontekstas

Sistema veikia klientas – serveris principu. Yra viena centrinis serveris, į kurį galima kreiptis ir per naršyklę ir per taikomąją programą. Sistema veikia aplinkoje, kurioje dirba programuotojai, projektų vadovai ir programinės įrangos naudotojai.



9 pav. Programos veiklos kontekstas

3.4. Projekto įgyvendinimas

Bandant įgyvendinti klaidų sekimo sistemos tenka susidurti su keletu projektavimo ir įgyvendinimo problemų – tinkamu platformos ir architektūros pasirinkimu, kuris leistų efektyviai tobulinti ir plėtoti sistemą, sistemos tipu – internetinė ar paprasta.

Technologiniam įgyvendinimui pasirinkta Microsoft .Net platforma. Tai įrankių rinkinys, suteikiantis programuotojams galimybę dirbti savo darbą produktyviau ir efektyviau.

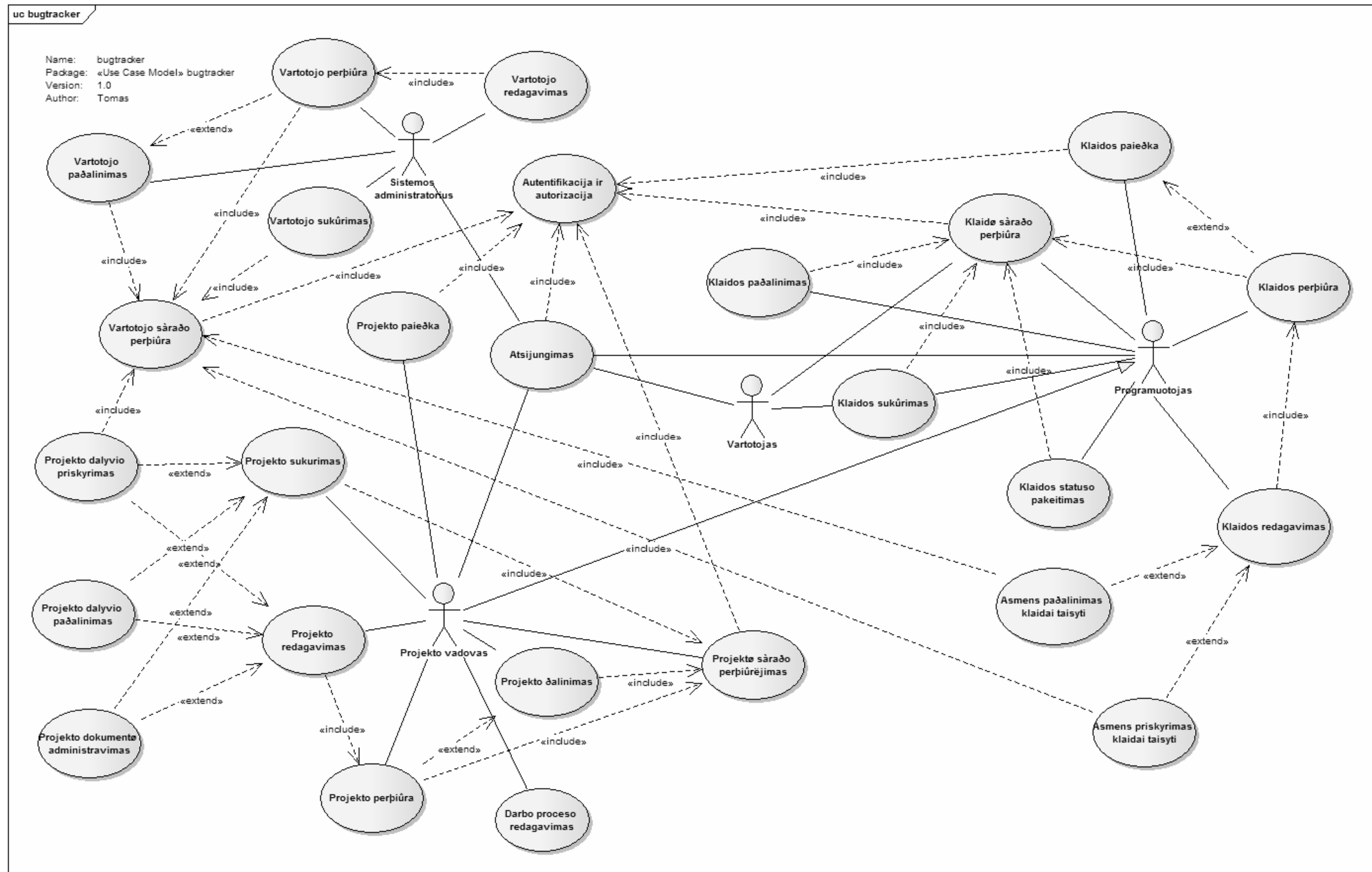
Pagrindiniai .Net platformos privalumai – integracija su Windows platforma, tačiau egzistuoja sprendimai, leidžiantis programoms veikti ir kitose platformose, užtikrinamas saugumas dėl atminties valdymo, kodo matomumo, duomenų perdavimo. Ši technologija dėl savo daugelio lygių architektūros leidžia efektyviai kurti tiek paprastas, tiek žiniatinklio programas su minimaliu kodo keitimu.

Pagrindiniai privalumai .net platformos, kuriuos pateikia Microsoft [28]:

- Saugi, daugiakalbė programavimo platforma
- Solidi, į modelį orientuota kūrimo paradigma
- Puikus naujų internetinių technologijų palaikymas
- Saugūs, lankstūs WEB servisai.
- Galima naudoti įvairiose platformose.

Naudojantis „Entity Framework ORM", kuris atsirado .NET 3.5 SP1 versijoje, galima išspręsti problemas dėl DBVS pasirinkimo, bus galima panaudoti dauguma SQL veikiančių DBVS. Produktas taps nesusiejamas prie konkrečios duomenų bazių programinės įrangos.

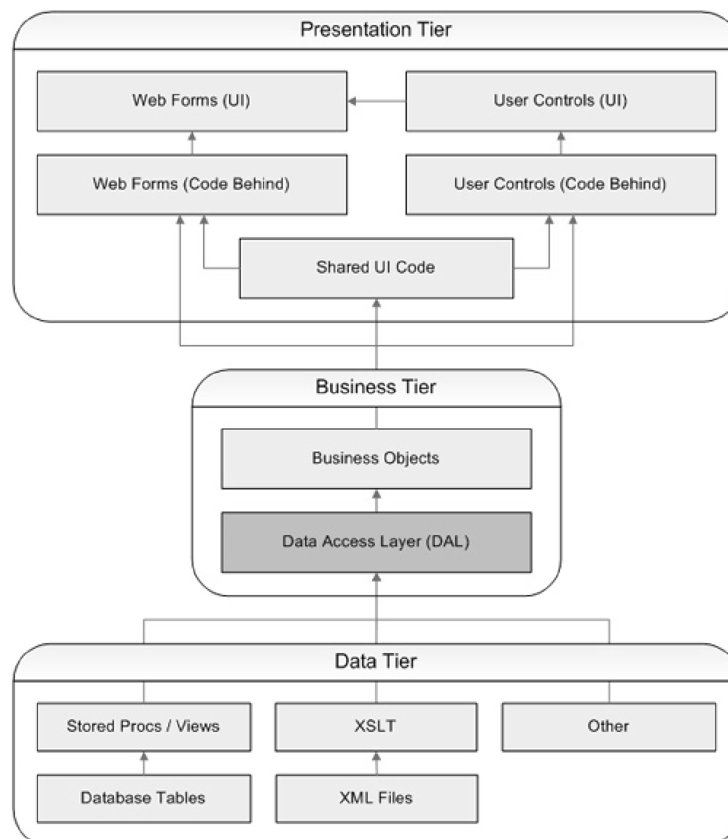
3.5. Sistemos ribos



10 pav. Sistemos ribos pateiktos panaudos atveju diagrama

3.6. Architektūriniai sprendimai

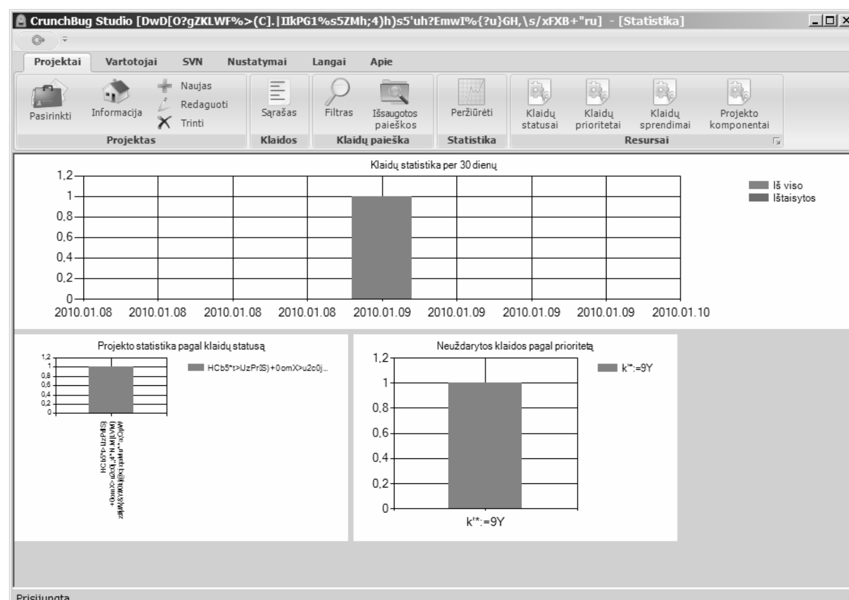
- Programinę įranga realizuoti pasirinkta naudojant 3-lygių architektūrą.
- Tokios architektūros pasirinkimas leis kuriamai programai būti greitai veikiančiai, lengvai plečiamai, lanksčiai.
- Realizacija atlikta .Net 3.5 aplinkoje C# kalba, Postgresql DBVS.
- Architektūra leidžia naudoti skirtingus DBVS, nekeičiant arba minimaliai keičiant programos kodą.
- Realizuojant architektūrą, turi būti atsižvelgta į kūrimo charakteristikas, apibrėžtas reikalavimų specifikacijoje.
- GUI paketuose daug metodų generuojama IDE pagalba, vizualiai modeliuojant formas, jų klasių diagramoje nevaizduojama, dėl didelės pasikeitimo galimybės.



11 pav. Kuriamos sistemos architektūriniai sluoksniai

3.7. Projektinio darbo rezultatai

Projekte sėkmingai pavyko įgyvendinti klaidų informacinė sistemą. Sukurta sistema naudojama ir yra atvirojo kodo, todėl ją gali toliau tobulinti kito programuotojai. Tarp įdomiausių sistemos savybių galima paminėti galimybę filmuoti ekraną ir gautus rezultatus pridėti prie siunčiamos klaidos pranešimo, tuo būdu sprendžiama labai opi problema – vartotojų nenoras ar kompetencijos neturėjimas tinkamai užpildyti klaidos pranešimus. Sukurta PĮ susideda iš dviejų dalių – apžvalginio/administravimo įrankio, kuriame peržiūrima informacija apie klaidas (Bugtracker), sekama informaciją apie atnaujinimus, reikalingos klaidos priskiriamos programuotojams ir klientinės dalies (Reporter), kuri veikdama foniniu rėžimu registruoja gedimus, ji taip pat gali būti integruojama su kita programine įranga iš kurios gauna klaidų pranešimus ir persiunčia juos į „Crunchbug". Sistemą galima rasti adresu: http://topazas.byethost11.com/crunchbug



12 pav. „Crunchbug" pagrindinis langas

4. Simbolinio vykdymo įrankių taikymo tyrimas .Net platformoje

Tyrime nagrinėjamos galimybės realioms sistemoms .Net platformoje vykdyti kodą simboliškai. Tam tikslui buvo sukurtas įrankis SYMEX. Tačiau jo galimybių dar nepakanka išnagrinėti (sistema kuriama neilgai) tyrimo objektą dėl pastarojo sudėtingumo – klaidų sekimo informacinę sistemą „CRUNCHBUG", kuri buvo kuriama projekto darbo metu, todėl pasitelkiamas dar vienas simbolinio vykdymo ir analizės įrankis – Pex.

Simbolinis vykdymo programų galutinis rezultatas – vienetų testai, kurie generuojami automatiškai iš kodo.

4.1. Symex – simbolinio vykdymo variklis .Net platformai

„Symex (Symbolic Execution) engine" – tai programa, sukurta programų sistemos inžinerijos katedros studentų Tomo Neverdauskas ir Justino Prelgausko. Jos uždavinys – generuoti iš kodo vienetų testus, kurie turėtų 100% kodo padengimą, naudojant simbolį vykdymą. Sukurti šį įrankį paskatino kitų įrankių .Net buvimo stoka, nes dabar vienintelis egzistuojantis sprendimas – Pex, tačiau ši programinė įranga yra mokama ir uždaro kodo, tai neleidžia jos tinkamai nagrinėti ir plėsti moksliniais bei praktiniais tikslais. SYMEX – tai:

- „Baltos dėžės" principo analizės įrankis.
- Kodo interpretatorius, simboliškai vykdantis .NET platformos CIL kodą.
- SMT formulių generatorius.
- Vienetų testų generatorius.

Ši programinė įranga plėtojama, norint ištirti simbolinio vykdymo galimybes .Net platformoje, ji atviro kodo ir darbo rašymo metu aktyviai kuriama toliau.

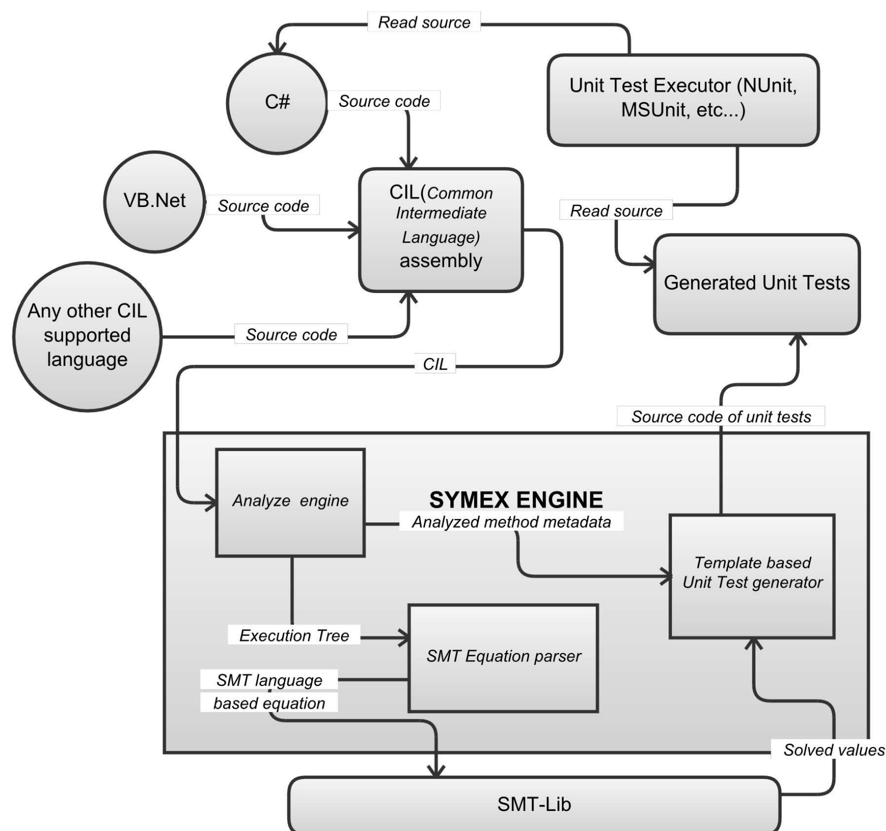
4.2. Symex architektūra

4.2.1. Bendroji Symex architektūra ir veiklos kontekstas

Symex įėjimo duomenys, tai .Net CLR tipo, kompiliuoti į bitinį kodą DLL ir EXE failai. Toks pasirinkimas leidžia išplėsti analizės galimybes, net nėra apsiribojama viena kalba – programos išeities kodas gali būti pateiktas bet kuria iš CLI standartą palaikančių kalbų. Symex dekompiliuoja dll failus padedant Microsoft CCI bibliotekų rinkiniui.

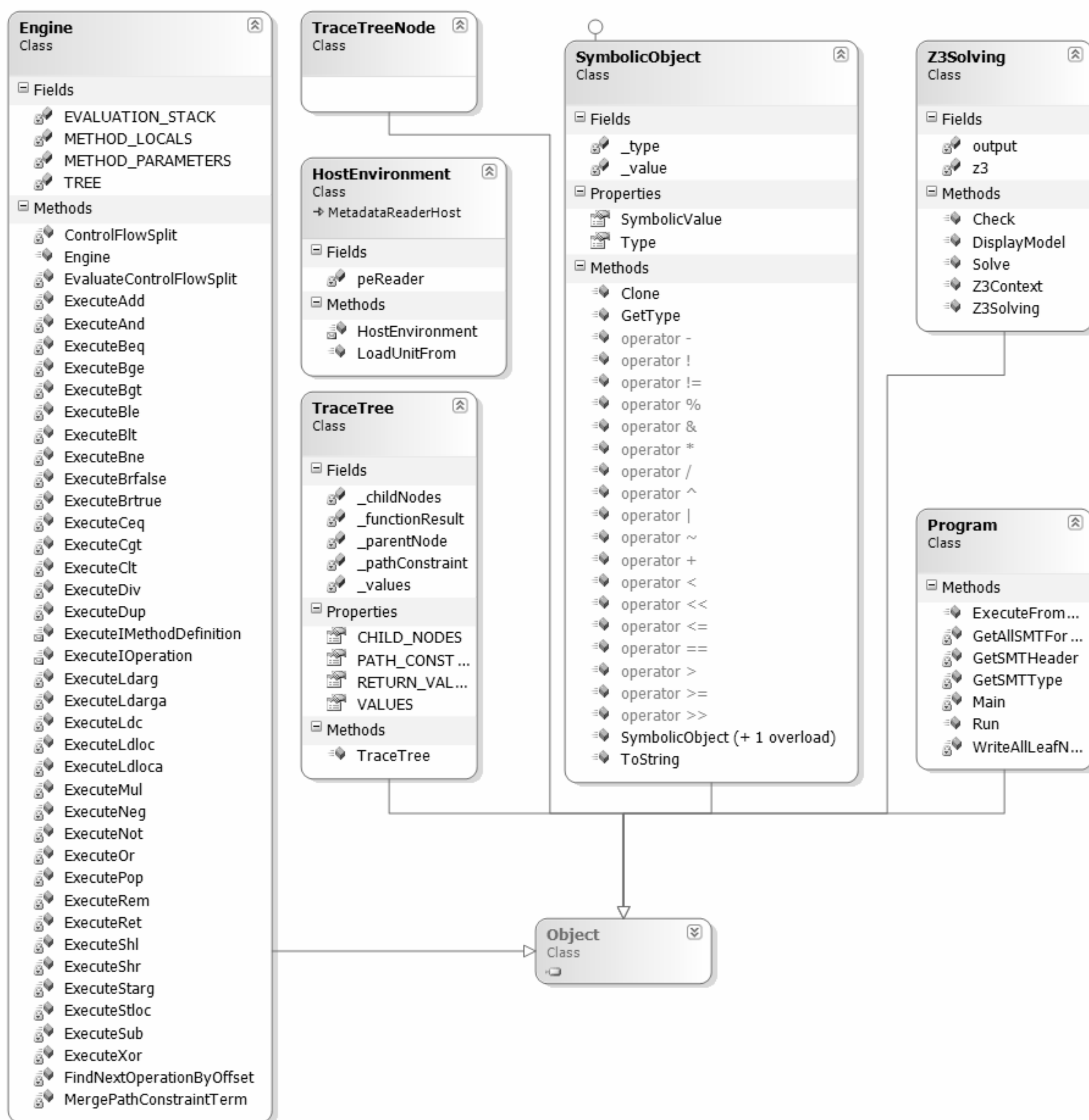
Nuskaityti metodai iš duoto šaltinio yra vykdomi simboliškai. Gautas simbolinio vykdymo rezultatas perduodamas į SMT lygčių formavimo modulį, kur standartine SMT-Lib kalba sudaromos loginės formulės. Jos perduodamos į išorinė SMT biblioteką. Kadangi SMT-Lib standartizuota, galima naudoti bet kurią iš dabar kuriamų SMT bibliotekų. Symex naudoja Z3 biblioteką dėl savo greičio ir patogios integravimo sąsajos su .Net platforma. Gauti iš SMT rezultatai, kartu su reikalingais meta duomenimis (klasių, metodų vardai) perduodami į vienetų testų generatorių, kuris sugeneruoja vienetų testų aibes, padengiančias analizuojamą kodą 100%. Tai programos darbo rezultatas. Vienetų testų programa savo ruoštu gautus vienetų testus ir testuojamą kodą įvykdo ir pateikia gautus rezultatus.

Programos architektūra kurta tai, kad kuo daugiau vos dalių būtų universalios – galimybės generuoti skirtingo formato vienetų testus, naudoti skirtingas SMT bibliotekas.



13 pav. Symex bendroji architektūra ir veikimo kontekstas

4.2.2. Klasių diagrama



14 pav. Symex klasių diagrama

4.3. Programos apribojimai

Dėl SMT teorijos apribojimų Symex negali dirbti su slankaus skaičiaus skaičiais ir operacijomis. Taip pat nėra aišku, kaip elgtis su nedeterministinėmis funkcijomis, tokiomis kaip laikas, pseudo-atsitiktiniai skaičiai ir pan.

Šiuo metu Symex palaiko:

- Bitines operacijas
- Aritmetines operacijas

- Loginės operacijos
- Sąlygos sakiniai
- Ciklai

4.4. Symex dabartinė būsena ir tolesni darbai

Darbo rašymo metu, Symex aktyviai kuriamas ir tobulinimas, tolesniuose darbuose numatytas eilučių (string) tipo, išorinių funkcijų kvietimo, daugiagijiškumo palaikymas. Taip pat realioms sistemos reikia pritaikyti būdą, kaip skaičiuoti slankaus kablelio skaičius, nes darbo rašymo metu SMT bibliotekose šios galimybės dar nėra realizuotos.

4.5. Programos veikimo pavyzdžiai

Pateikiama viena klasė C# kalba ir trys jos funkcijos, kurios bus vykdomos simboliškai.

2 lentelė. Eksperimente analizuojamos funkcijos

Pavadinimas	**Aprašymas**
GetSquareSurfaceArea	Skaičiuoti kvadrato plotą, jeigu a ir b teigiami.
GetCubeSurfaceArea	Skaičiuoti kubo paviršiaus plotą, jeigu a, b ir c teigiami.

```
        public int GetSquareSurfaceArea(int a, int b)
        {
            if (a > 0)
            {
                if (b > 0)
                {
                    return a * b;
                }
            }

            return 0;
        }
```

15 pav. Metodo *GetSquareSurfaceArea* išeities tekstas

```
Method 'GetSquareSurfaceArea':
-----------
OPCODE=Nop;              OFFSET=0;
OPCODE=Ldarg_1;          VALUE=a;         OFFSET=1;
OPCODE=Ldc_I4_0;         OFFSET=2;
OPCODE=Cgt;              OFFSET=3;
OPCODE=Ldc_I4_0;         OFFSET=5;
OPCODE=Ceq;              OFFSET=6;
OPCODE=Stloc_1;          VALUE=local_1;   OFFSET=8;
OPCODE=Ldloc_1;          VALUE=local_1;   OFFSET=9;
OPCODE=Brtrue_S;         VALUE=32;        OFFSET=10;
```

```
OPCODE=Nop;             OFFSET=12;
OPCODE=Ldarg_2;         VALUE=b;        OFFSET=13;
OPCODE=Ldc_I4_0;        OFFSET=14;
OPCODE=Cgt;             OFFSET=15;
OPCODE=Ldc_I4_0;        OFFSET=17;
OPCODE=Ceq;             OFFSET=18;
OPCODE=Stloc_1;         VALUE=local_1;  OFFSET=20;
OPCODE=Ldloc_1;         VALUE=local_1;  OFFSET=21;
OPCODE=Brtrue_S;        VALUE=31;       OFFSET=22;
OPCODE=Nop;             OFFSET=24;
OPCODE=Ldarg_1;         VALUE=a;        OFFSET=25;
OPCODE=Ldarg_2;         VALUE=b;        OFFSET=26;
OPCODE=Mul;             OFFSET=27;
OPCODE=Stloc_0;         VALUE=local_0;  OFFSET=28;
OPCODE=Br_S;            VALUE=36;       OFFSET=29;
OPCODE=Nop;             OFFSET=31;
OPCODE=Ldc_I4_0;        OFFSET=32;
OPCODE=Stloc_0;         VALUE=local_0;  OFFSET=33;
OPCODE=Br_S;            VALUE=36;       OFFSET=34;
OPCODE=Ldloc_0;         VALUE=local_0;  OFFSET=36;
OPCODE=Ret;             OFFSET=37;
-----------
```

16 pav. Sugeneruotas *GetSquareSurfaceArea* bitinis kodas, kuris bus vykdomas simboliškai

```
formula (= (> a 0) false)
Custom model display:
(define a int) |-> 0
num consts: 1
formula (not (= (> a 0) false))
formula (= (> b 0) false)
Custom model display:
(define a int) |-> 1
(define b int) |-> 0
num consts: 2
formula (not (= (> a 0) false))
formula (not (= (> b 0) false))
Custom model display:
(define a int) |-> 1
(define b int) |-> 1
num consts: 2
```

17 pav. *GetSquareSurfaceArea* simboliniai vykdymo keliai, suformuoti SMT lygčių pavidalu

```
        public int GetCubeSurfaceArea(int a, int b, int c)
        {
            if (a > 0)
            {
                if (b > 0)
                {
                    if (c > 0)
                    {
                        return (a * b) * 4 + (a * c) * 2;
                    }
                }
            }
            return 0;
        }
```

18 pav. Sugeneruotas *GetSquareSurfaceArea* bitinis kodas, kuris bus vykdomas simboliškai

```
Method 'GetCubeSurfaceArea':
```

```
-----------
OPCODE=Nop;             OFFSET=0;
OPCODE=Ldarg_1;         VALUE=a;         OFFSET=1;
OPCODE=Ldc_I4_0;        OFFSET=2;
OPCODE=Cgt;             OFFSET=3;
OPCODE=Ldc_I4_0;        OFFSET=5;
OPCODE=Ceq;             OFFSET=6;
OPCODE=Stloc_1;         VALUE=local_1;   OFFSET=8;
OPCODE=Ldloc_1;         VALUE=local_1;   OFFSET=9;
OPCODE=Brtrue_S;        VALUE=53;        OFFSET=10;
OPCODE=Nop;             OFFSET=12;
OPCODE=Ldarg_2;         VALUE=b;         OFFSET=13;
OPCODE=Ldc_I4_0;        OFFSET=14;
OPCODE=Cgt;             OFFSET=15;
OPCODE=Ldc_I4_0;        OFFSET=17;
OPCODE=Ceq;             OFFSET=18;
OPCODE=Stloc_1;         VALUE=local_1;   OFFSET=20;
OPCODE=Ldloc_1;         VALUE=local_1;   OFFSET=21;
OPCODE=Brtrue_S;        VALUE=52;        OFFSET=22;
OPCODE=Nop;             OFFSET=24;
OPCODE=Ldarg_3;         VALUE=c;         OFFSET=25;
OPCODE=Ldc_I4_0;        OFFSET=26;
OPCODE=Cgt;             OFFSET=27;
OPCODE=Ldc_I4_0;        OFFSET=29;
OPCODE=Ceq;             OFFSET=30;
OPCODE=Stloc_1;         VALUE=local_1;   OFFSET=32;
OPCODE=Ldloc_1;         VALUE=local_1;   OFFSET=33;
OPCODE=Brtrue_S;        VALUE=51;        OFFSET=34;
OPCODE=Nop;             OFFSET=36;
OPCODE=Ldarg_1;         VALUE=a;         OFFSET=37;
OPCODE=Ldarg_2;         VALUE=b;         OFFSET=38;
OPCODE=Mul;             OFFSET=39;
OPCODE=Ldc_I4_4;        OFFSET=40;
OPCODE=Mul;             OFFSET=41;
OPCODE=Ldarg_1;         VALUE=a;         OFFSET=42;
OPCODE=Ldarg_3;         VALUE=c;         OFFSET=43;
OPCODE=Mul;             OFFSET=44;
OPCODE=Ldc_I4_2;        OFFSET=45;
OPCODE=Mul;             OFFSET=46;
OPCODE=Add;             OFFSET=47;
OPCODE=Stloc_0;         VALUE=local_0;   OFFSET=48;
OPCODE=Br_S;            VALUE=57;        OFFSET=49;
OPCODE=Nop;             OFFSET=51;
OPCODE=Nop;             OFFSET=52;
OPCODE=Ldc_I4_0;        OFFSET=53;
OPCODE=Stloc_0;         VALUE=local_0;   OFFSET=54;
OPCODE=Br_S;            VALUE=57;        OFFSET=55;
OPCODE=Ldloc_0;         VALUE=local_0;   OFFSET=57;
OPCODE=Ret;             OFFSET=58;
-----------
```

19 pav. Sugeneruotas *GetCubeSurfaceArea* bitinis kodas, kuris bus vykdomas simboliškai

```
formula (= (> a 0) false)
Custom model display:
(define a int) |-> 0
num consts: 1
formula (not (= (> a 0) false))
formula (= (> b 0) false)
Custom model display:
(define a int) |-> 1
(define b int) |-> 0
num consts: 2
```

```
formula (not (= (> a 0) false))
formula (not (= (> b 0) false))
formula (= (> c 0) false)
Custom model display:
(define a int) |-> 1
(define b int) |-> 1
(define c int) |-> 0
num consts: 3
formula (not (= (> a 0) false))
formula (not (= (> b 0) false))
formula (not (= (> c 0) false))
Custom model display:
(define a int) |-> 1
(define b int) |-> 1
(define c int) |-> 1
```

20 pav. *GetSquareSurfaceArea* vykdymo keliai, suformuoti SMT lygčių pavidalu

```
file:///C:/Documents and Settings/Tomas/Desktop/x242.soften.ktu.lt/phd_dev/trunk...
Methods in types from 'FunctionsUnderTest':
Method 'GetSquareSurfaceArea':
-----------
OPCODE=Nop;             OFFSET=0;
OPCODE=Ldarg_1;         VALUE=a;        OFFSET=1;
OPCODE=Ldc_I4_0;                OFFSET=2;
OPCODE=Cgt;             OFFSET=3;
OPCODE=Ldc_I4_0;                OFFSET=5;
OPCODE=Ceq;             OFFSET=6;
OPCODE=Stloc_1;         VALUE=local_1;  OFFSET=8;
OPCODE=Ldloc_1;         VALUE=local_1;  OFFSET=9;
OPCODE=Brtrue_S;                VALUE=32;       OFFSET=10;
OPCODE=Nop;             OFFSET=12;
OPCODE=Ldarg_2;         VALUE=b;        OFFSET=13;
OPCODE=Ldc_I4_0;                OFFSET=14;
OPCODE=Cgt;             OFFSET=15;
OPCODE=Ldc_I4_0;                OFFSET=17;
OPCODE=Ceq;             OFFSET=18;
OPCODE=Stloc_1;         VALUE=local_1;  OFFSET=20;
OPCODE=Ldloc_1;         VALUE=local_1;  OFFSET=21;
OPCODE=Brtrue_S;                VALUE=31;       OFFSET=22;
OPCODE=Nop;             OFFSET=24;
OPCODE=Ldarg_1;         VALUE=a;        OFFSET=25;
OPCODE=Ldarg_2;         VALUE=b;        OFFSET=26;
OPCODE=Mul;             OFFSET=27;
OPCODE=Stloc_0;         VALUE=local_0;  OFFSET=28;
OPCODE=Br_S;            VALUE=36;       OFFSET=29;
OPCODE=Nop;             OFFSET=31;
OPCODE=Ldc_I4_0;                OFFSET=32;
OPCODE=Stloc_0;         VALUE=local_0;  OFFSET=33;
OPCODE=Br_S;            VALUE=36;       OFFSET=34;
OPCODE=Ldloc_0;         VALUE=local_0;  OFFSET=36;
OPCODE=Ret;             OFFSET=37;
-----------
formula (= (> a 0) false)
Custom model display:
(define a int) |-> 0
num consts: 1
formula (not (= (> a 0) false))
formula (= (> b 0) false)
Custom model display:
(define a int) |-> 1
```

21 pav. Symex analizuoja kodą (programos darbo langas)

4.6. Pex

Pex, kaip ir Symex – baltos dėžės principu veikiantis simbolinio vykdymo metodu paremtas vienetų testų generatorius, skirtas generuoti parametrizuotus vienetų testus, kurie padengtų kodą kuo didesniu procentu [29], tačiau šis įrankis plėtojamas Microsoft korporacijos ir yra mokamas, bei uždaro kodo, todėl sunku spręsti apie naudojamus algoritmus, vidinę programos struktūrą.

Pex taip pat išplėtė tradicines simbolinio vykdymo galimybes, pridėdamas naują metodą – dinaminį simbolinį vykdymą. Šiuo atveju programa analizuojama ne tik simboliškai, bet ir paraleliai vykdoma ir renkami papildomi duomenys, kurie naudojami parametrizuotų vienetų testų sukūrimui. Tokiu būdu Pex išplečia savo galimybių aibę, nes

sudaromos prielaidos analizuoti kodą, kuris nėra tiesiogiai pasiekiamas. Galima tikėtis, kad Pex daugeliu atveju duos geresnį rezultatą negu Symex, nes pastarasis nenaudoja dinaminio simbolinio vykdymo. Tačiau Pex taip turi trūkumų, vis dar būdingų šio tipo programoms:

- Nevykdomas neinstrumentuotas kodas. Tai toks kodas, kuris reikalauja tiesioginio atminties valdymo, .Net platformoje dar vadinamas kaip „nesaugus", nes pastaroji platforma vykdoma virtualioje mašinoje ir atminties valdymu rūpinasi pastaroji.
- Gijų (threads) palaikymas – Pex moka analizuoti tik tokias programas, kuriose naudojami linijiniai procesai, o ne paraleliniai.
- Apribota C# kalba, Pex dėl dinaminio simbolinio vykdymo dabar supranta tik programas parašytas C# kalba.
- Dėl SMT apribojimų nėra palaikomi slankaus kablelio skaičiai.

4.7. Atsitiktinio generavimo karkasas

Eksperimentinėje dalyje numatyta mėginti simbolinį vykdymą lyginti su atsitiktinai generuojamais įėjimais vienetų testams. Todėl reikalingas įrankis, gebantis atsitiktinai generuoti įvairias duomenų struktūras, Šiam tikslui panaudotas – „Random generation framework" [30]. Šis įrankis geba atsitiktinai generuoti šias duomenų struktūras:

- Logines reikšmes
- Sveikų skaičių reikšmes
- Eilutės (String) tipo reikšmes
- Slankaus kablelio reikšmes
- Datas

5. Simbolinio vykdymo kodo analizės įrankių vienetų testams generuoti palyginimo eksperimentas

5.1. *Eksperimento tikslai*

Eksperimentinė dalis susideda iš dviejų dalių:

- Pirmoje dalyje testuojamos matematinės funkcijos
- Antroje dalyje projektinis darbas analizuojamos Pex įrankiu.

Pagrindiniai eksperimento tikslai yra nustatyti, kuris vienetų testų generavimo būdas yra greitesnis ir efektyvesnis, kodo padengimo ir vienetų testų palyginimų skaičiaus prasme, išanalizuoti ir pamatuoti sukurtą magistro projektą įrankiu Pex ir nustatyti, kiek kodo galima padengti vienetų testais automatiškai (naudojantis pačiu kaip modeliu). Taip pat buvo palyginti atsitiktinių vienetų testų generavimo ir simbolinio vykdymo testų generavimui būdų efektyvumas eksperimentiškai.

5.2. *Eksperimentui naudota techninė ir programinė įranga*

Kompiuterio parametrai naudoti eksperimentams atlikti – programinė įranga:

- Visual Studio 2010 RC1 v. 10.0.3
- .Net Framework v. 3.5 SP1.
- Visual Studio Unit Tests (integruota į VS 2010)
- Random Generation Framework [30]
- SYMEX
- Microsoft Pex 2010 x86 (0.19.41110.1)
- Windows XP SP3 OS

Aparatūrinė įranga:

- Pentium M 1.7 GHz
- 1.5 GB Ram

5.3. *Simboliniu vykdymu paremtas vienetų testų generavimo eksperimentas matematiniams metodams*

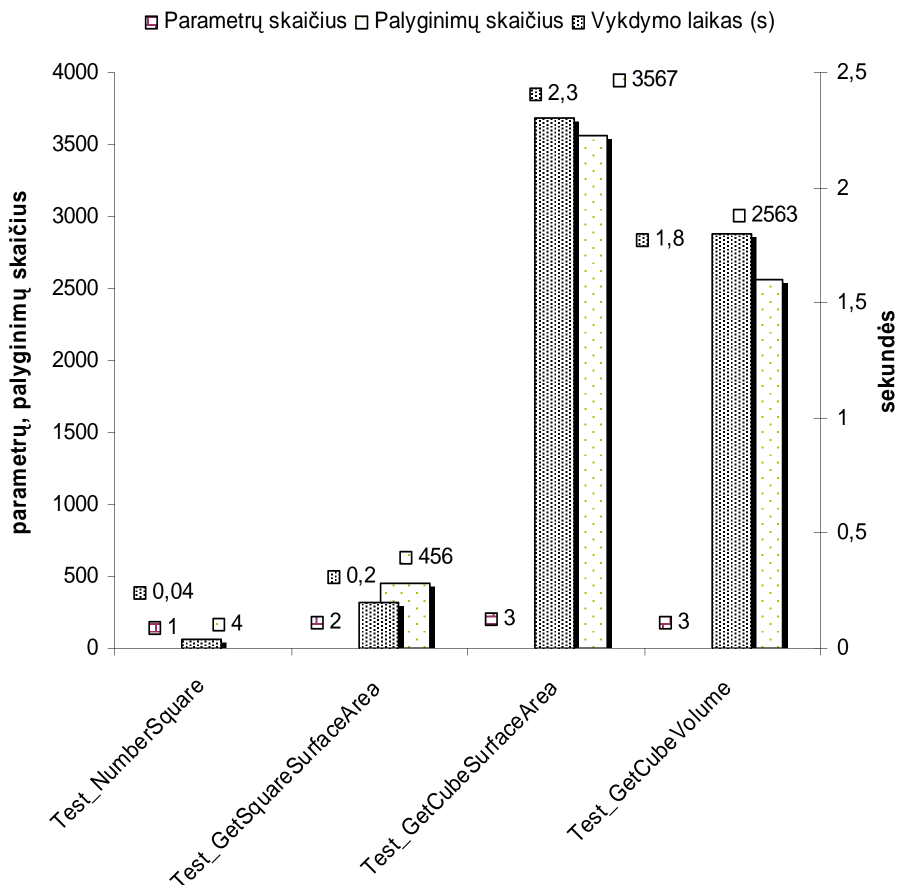
Dėl dalinių apribojimų sukurtam įrankiui Symex buvo parašyti matematiniai metodai priede nr. 1 pateiktos realizacijos), pagal kuriuos buvo generuojami vienetų testai. Eksperimente naudota programinė įranga: Microsoft PEX, Symex ir „Random Generation Framework". Dėl atsitiktinio generavimo specifikos šiai daliai eksperimentas kartotas 20 kartų ir išvesti gautų rezultatų vidurkiai, norint gauti optimalesnį ir tikslesnį rezultatą. Kitos sąlygos visiems testams buvo vienodos.

Eksperimente buvo matuojamas vienetų testų vykdymo laikas ir tikrinimų skaičius kiekviename vienetų teste (asserts). Būtina sąlyga – visi testai analizuojamą kodą turi padengti 100%.

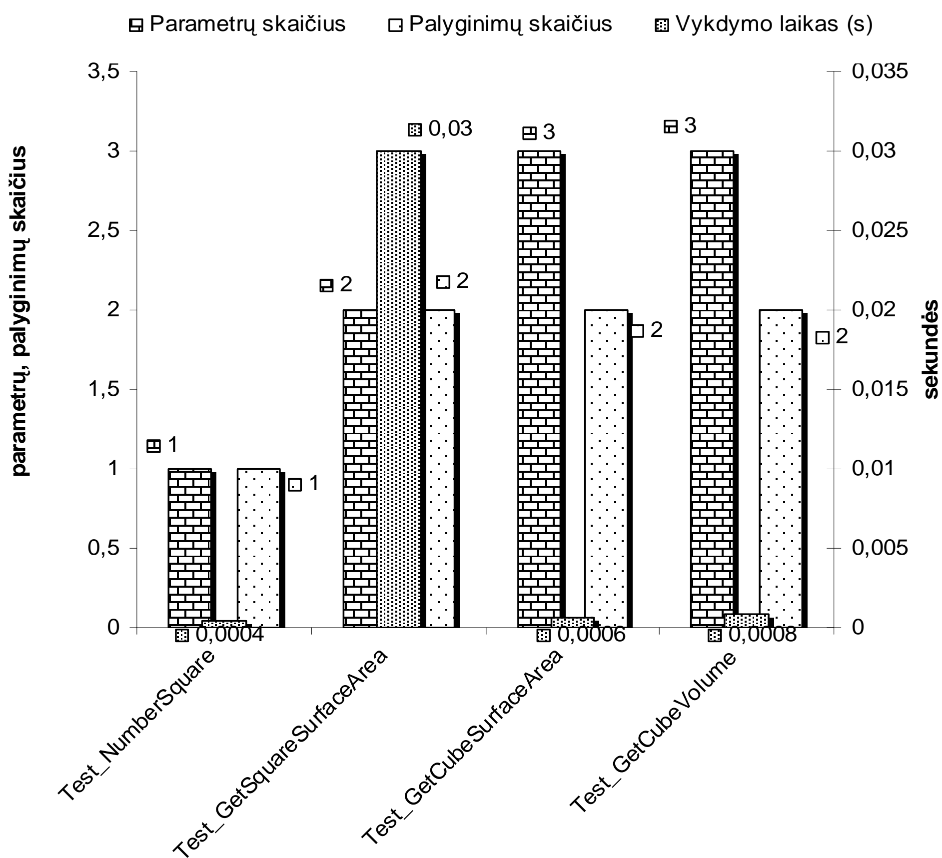
3 lentelė. Eksperimente tiriami klasių metodai

Pavadinimas	Aprašymas	Įeinančių parametrų skaičius
NumberSquare	Suskaičiuoti skaičiaus kvadratą	1
GetSquareSurfaceArea	Skaičiuoti kvadrato plotą, jeigu a ir b teigiami.	2
GetCubeSurfaceArea	Skaičiuoti kubo paviršiaus plotą, jeigu a, b ir c teigiami.	3
GetCubeVolume	Skaičiuoti kubo (gretasienio) tūrį, jeigu a, b ir c teigiami.	3

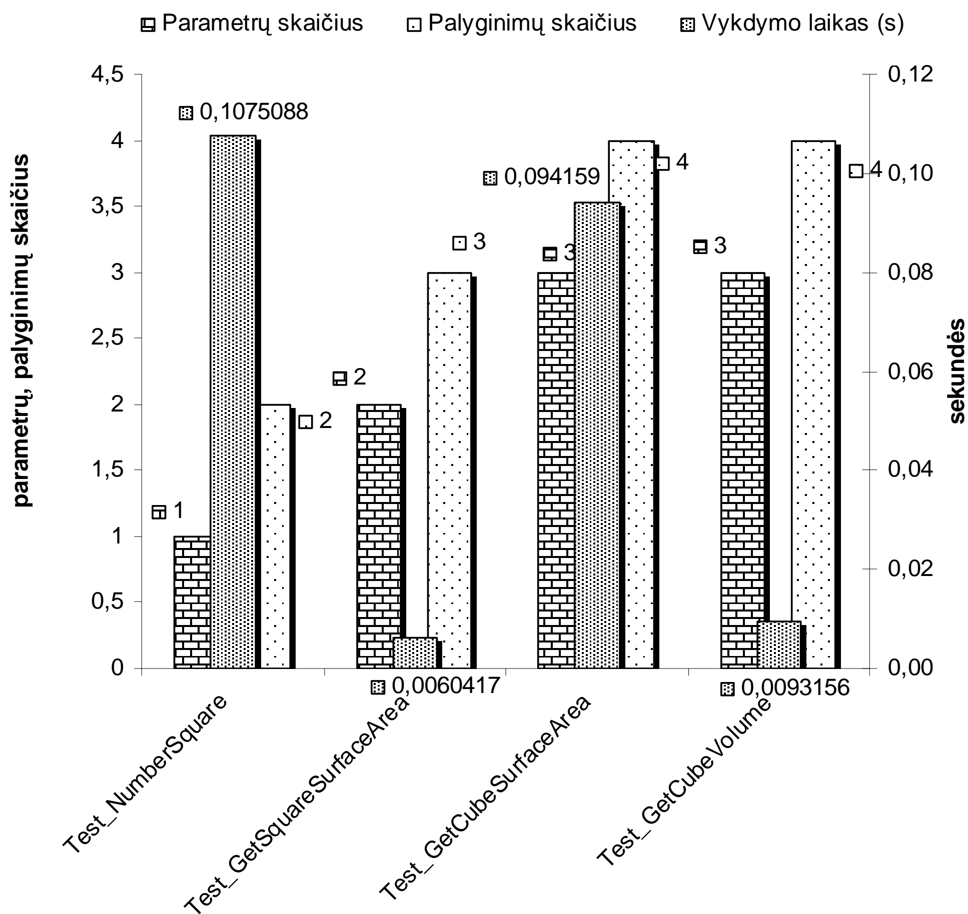
Atlikus eksperimentą, gauti tokie rezultatai:



22 pav. Atsitiktinio generavimo rezultatai

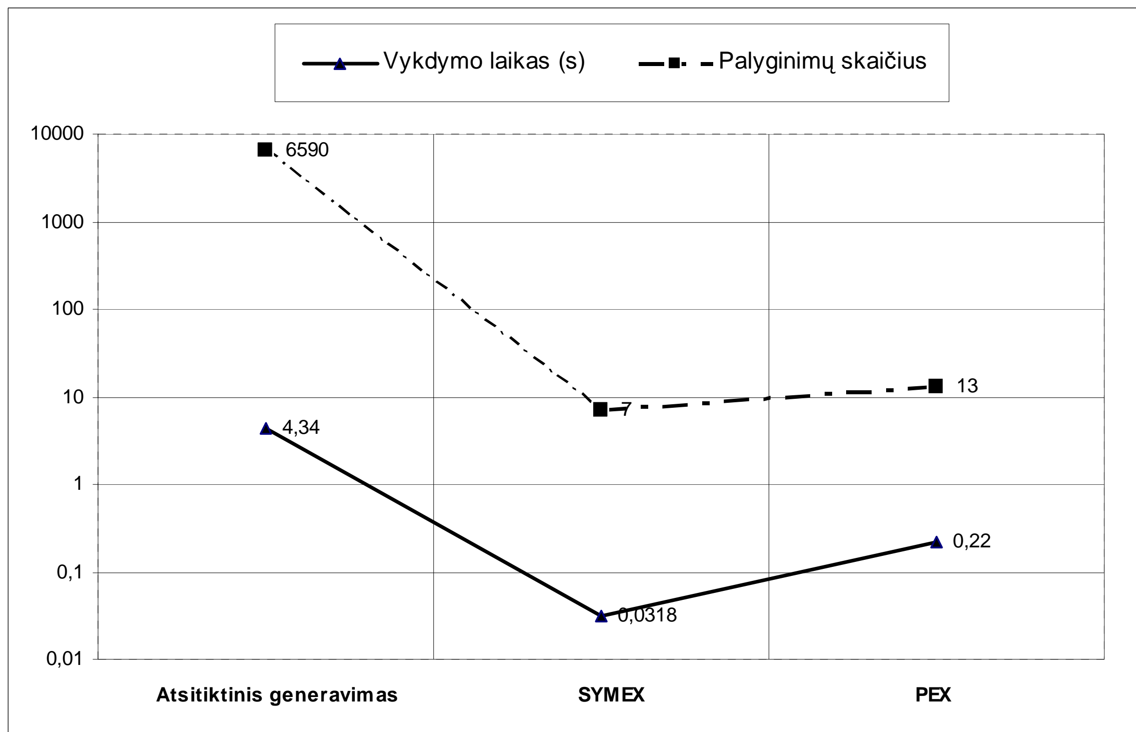


23 pav. SYMEX eksperimento rezultatai



24 pav. Pex rezultatai

Apibendrintuose rezultatuose (pav. 25) rodomi susumuoti ankstesnių bandymų rezultatai.



25 pav. Apibendrinti įrankių palyginimo rezultatai

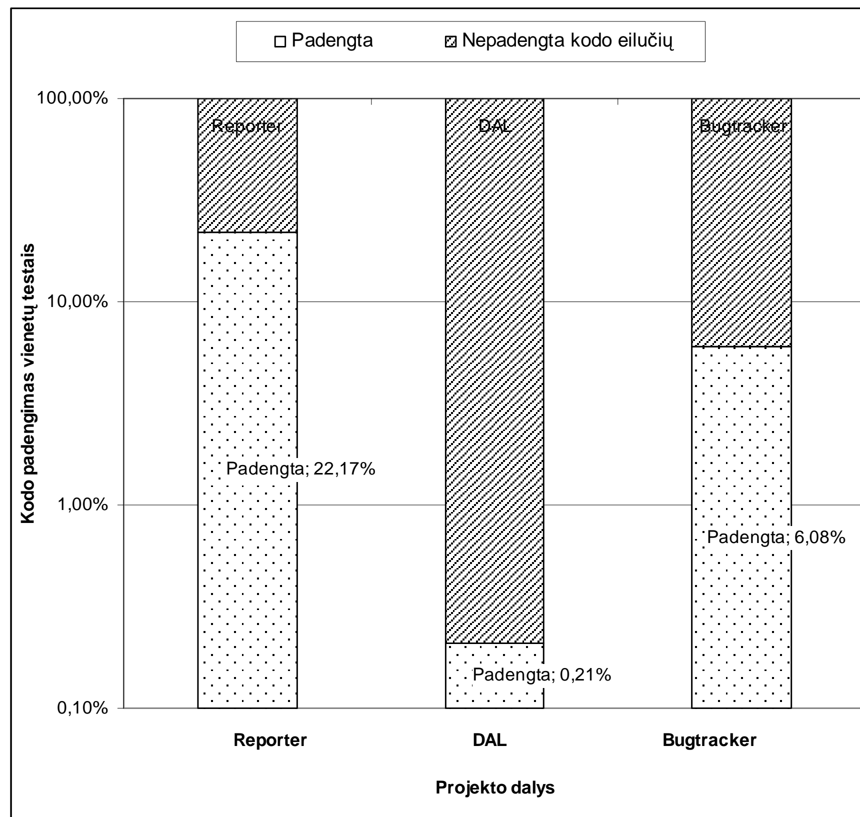
Iš eksperimento rezultatų akivaizdu, kad atsitiktinio generavimo priemonė atlikdama tą patį darbą, smarkiai nusileidžia laiko ir palyginimų skaičiumi simbolinės analizės įrankiams. Galima teigi, vienetų testų generavimo rezultatai tarp Symex ir Pex yra labai panašūs ir duoda kokybiškai labai artimą rezultatą bet Pex naudoja dinaminę simbolinė analizę, todėl jo sukurtų testų aibė yra šiek tiek didesnė ir vykdymo laikas truputį ilgesnis.

5.4. Simboliniu vykdymu paremtas vienetų testų generavimo eksperimentas magistro projektui

Šiame bandyme testo objektas yra magistro projekte sukurta programinė įranga – klaidų administravimo informacinė sistema „Crunchbug". Tai realiai veikianti sistema, turinti per 8 tūkst. eilučių kodo (detaliau nurodyta priede nr. 4). Kadangi Symex įrankis dar nepalaiko reikalingų duomenų struktūrų, analizė buvo atlikta įrankiu Pex. Buvo generuojami vienetų testai, po to vykdomas vienetų testavimas ir pateikti gauti rezultatai. Projektą sudaro trys dalys:

- DAL – projekto dalis, atsakinga už sąveiką su duomenų baze (data access layer)
- Reporter – įrankis klientui pranešti apie klaidas.

• Bugtracker – klaidų valdymo informacinė sistema.



26 pav. Pex kodo padengimo projekte rezultatai

Gauti rezultatai parodo, kad didžiausia automatinį kodo padengimą pavyko pasiekti paprasčiausiam projekto daliai (Reporter), o DAL (atsakingą už sąveika su DB) automatiškai padengti pavyko vos 0,21%. Tai nėra netikėtas rezultatas, nes veiksmai su DB gerokai pralenkia dabartines simbolinio vykdymo galimybes. Vis dėlto galima daryti išvadą, kad automatinio kodo padengimo rezultatai rodo nemažą simbolinio vykdymo potencialą, o ypač geri rezultatai gali būti pasiekti, taikant šią techniką tinkamose srityse.

Išvados

1. Teorinėje dalyje buvo išnagrinėtas *simbolinis vykdymas* (symbolic execution), jo panaudojimas programinės įrangos testavime, aptarti būdai, kaip kodo vykdymas vyksta .Net platformoje ir kaip *simbolinį vykdymą* pritaikyti šiai platformai. Taip pat nagrinėti papildomi reikalavimai, kurie reikalingi paruošti tinkamą simbolinio vykdymo programinę įrangą. *Simbolinis vykdymas* teoriškai sukuria pagrįstas prielaidas išplėsti modeliu paremtą automatinį testavimą.
2. Magistro projekte sukurta atviro kodo klaidų sekimo ir registracijos informacinė sistema .Net platformai „Crunchbug”, turinti išskirtinių savybių rinkos egzistuojančių sprendimų atžvilgiu. Sistema sėkmingai naudojama praktikoje.
3. Remiantis teorine dalimi buvo sukurtas įrankis Symex, skirtas vienetų testų generavimui iš kodo. Jis buvo panaudotas eksperimentinėje dalyje ir parodė testuotose sistemose gerus rezultatus.
4. Eksperimentinėje dalyje buvo palyginti trys įrankiai: atsitiktinio vienetų testų įėjimų generavimo, Symex ir Pex. Eksperimento rezultatai aiškiai parodo simbolinio vykdymo pranašumą prieš atsitiktinį generavimą laiko ir testų kiekio atžvilgiu, nors sisteminės aplinkos ir sąlygos testams buvo vienodos. Tačiau ištyrus magistro projektą paaiškėjo, kad tokio tipo įrankiai turi būti tobulinami, jeigu norima juos naudoti realiuose sistemose arba jų taikymas neapsiribotų tam tikrų (pvz. matematinių ar loginių skaičiavimų) sistemų testavimu.

Literatūra

[1] King, J.C.: „Symbolic execution and program testing”, Commun. ACM, 1976, 19, (7), pp. 385-394

[2] Bertolino, A.: „Software Testing Research: Achievements, Challenges, Dreams”, in Editor (Ed.)^(Eds.): „Book Software Testing Research: Achievements, Challenges, Dreams” (2007, edn.), pp. 85-103

[3] Moore, E.: „Gedanken Experiments on Sequential Machines”: „Automata Studies” (Princeton U., 1956), pp. 129-153

[4] Bouabana-Tebibel, T.: „Formal validation with OCL”, in Editor (Ed.)^(Eds.): „Book Formal validation with OCL” (2006, edn.), pp. 2736-2741

[5] Utting, M., and Legeard, B.: „Practical Model-Based Testing: A Tools Approach” (Morgan Kaufmann, 2007, 1 edn. 2007)

[6] Hashimoto, Y., and Nakajima, S.: „Modular Checking of C Programs Using SAT-Based Bounded Model Checker”, in Editor (Ed.)^(Eds.): „Book Modular Checking of C Programs Using SAT-Based Bounded Model Checker” (2009, edn.), pp. 515-522

[7] Evans, D., and Larochelle, D.: „Improving security using extensible lightweight static analysis”, Software, IEEE, 2002, 19, (1), pp. 42-51

[8] Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., and Pape, M.: „Combining unit-level symbolic execution and system-level concrete execution for testing nasa software”. Proc. Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, WA, USA2008

[9] Shahamiri, S.R., Kadir, W.M.N.W., and Mohd-Hashim, S.Z.: „A Comparative Study on Automated Software Test Oracle Methods”, in Editor (Ed.)^(Eds.): „Book A Comparative Study on Automated Software Test Oracle Methods” (2009, edn.), pp. 140-145

[10] Fraser, G., Wotawa, F., and Ammann, P.: „Issues in using model checkers for test case generation”, J. Syst. Softw., 2009, 82, (9), pp. 1403-1418

[11] Roper, M.: „Artifical Immune Systems, Danger Theory, and the Oracle Problem”, in Editor (Ed.)^(Eds.): „Book Artifical Immune Systems, Danger Theory, and the Oracle Problem” (2009, edn.), pp. 125-126

[12] http://www.ecma-international.org/publications/standards/Ecma-335.htm, žiūrėta 2010-05

[13] ECMA: „Standard ECMA-335”, in Editor (Ed.)^(Eds.): „Book Standard ECMA-335” (2010, edn.), pp.

[14] CCI, M.: „http://ccimetadata.codeplex.com/”, žiūrėta 2010-05

[15] FxCop: „http://msdn.microsoft.com/en-us/library/bb429476%28VS.80%29.aspx", , žiūrėta 2010-05, FxCop

[16] Cimatti, A.: „Beyond Boolean SAT: Satisfiability modulo theories", in Editor (Ed.)^(Eds.): „Book Beyond Boolean SAT: Satisfiability modulo theories" (2008, edn.), pp. 68-73

[17] http://goedel.cs.uiowa.edu/smtlib/: „The Satisfiability Modulo Theories Library", žiūrėta 2010-05

[18] Fran, Bobot, o., Conchon, S., Contejean, E., St, , and Lescuyer, p.: „Implementing polymorphism in SMT solvers". Proc. Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, Princeton, New Jersey2008

[19] Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodriguez-Carbonell, E., and Rubio, A.: „The Barcelogic SMT Solver". Proc. Proceedings of the 20th international conference on Computer Aided Verification, Princeton, NJ, USA2008

[20] Jha, S., Limaye, R., and Seshia, S.: „Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic": „Computer Aided Verification" (2009), pp. 668-674

[21] Brummayer, R., and Biere, A.: „Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays" (2009), pp. 174-177

[22] Bruttomesso, R., Cimatti, A., Fran\, A., Griggio, A., and Sebastiani, R.: „The MathSAT 4 SMT Solver". Proc. Proceedings of the 20th international conference on Computer Aided Verification, Princeton, NJ, USA2008 pp. Pages

[23] Bruttomesso, R., Pek, E., and Sharygina, N.: „OpenSMT 0.2 System Description", 2009

[24] Dutertre, B., and de Moura, L.: „The Yices SMT solver", in Editor (Ed.)^(Eds.): „Book The Yices SMT solver" (2006, edn.), pp.

[25] L. De Moura, and Bjorner, N.: „Z3: An efficient SMT solver", Lecture Notes in Computer Science, 2008, vol. 4963, pp. 337

[26] http://www.gnu.org/software/classpath/bugs.html: „Status Counts", žiūrėta 2009-11-10

[27] Wilfredo, T.: „Software Fault Tolerance: A Tutorial", in Editor (Ed.)^(Eds.): „Book Software Fault Tolerance: A Tutorial" (NASA Langley Technical Report Server, 2000, edn.), pp.

[28] http://www.microsoft.com/net/Overview.aspx: „.NET Framework Overview", žiūrėta 2008-11-13

[29] Tillmann, N., and de Halleux, J.: „Pex–White Box Test Generation for .NET”, in Beckert, B., and Hähnle, R. (Eds.): „Tests and Proofs” (Springer Berlin Heidelberg, 2008), pp. 134-153

[30] Kheyrollahi, A.: „Random Generation Framework - A .NET generics framework for generating random values", http://www.codeproject.com/KB/cs/RandomGenerationFramework.aspx, 2009

Terminų ir santrumpų žodynas

SMT – Satisfability modulo theories. Pirmos eilės matematinės logikos šaka, predikatų logikos dalis.

CIL – Common immiadiate language. Bendrinė tarpinė kalba .Net platformoje.

CLI – Common language infrastructure. Standartas apibrėžiantis CIL.

.Net – Microsoft programų kūrimo platforma, kurios programavimų kalbos pagrindas yra CIL.

Bytecode – CIL kodo saugojimo būdas, bitinis kodas.

Priedai

1. Priedas: eksperimente naudota klasė matematiniams metodams tikrinti

```
public class Benchmarks
{

    public int GetSquareSurfaceArea(int a, int b)
    {
        if (a > 0)
        {
            if (b > 0)
            {
                return a * b;
            }
        }
        return 0;
    }
    public int GetCubeSurfaceArea(int a, int b, int c)
    {
        if (a > 0)
        {
            if (b > 0)
            {
                if (c > 0)
                {
                    return (a * b) * 4 + (a * c) * 2;
                }
            }
        }
        return 0;
    }

    public int GetCubeVolume(int a, int b, int c)
    {
        if (a > 0)
        {
            if (b > 0)
            {
                if (c > 0)
                {
                    return a * b * c;
                }
            }
        }
        return 0;
    }
    public int NumberSquare(int a)
    {
        if (a > 0)
        {
            return a * a;
        }
        return 0;
    }
}
```

2. Priedas: vienetų testų klasė sugeneruota Symex įrankiu

```
public class BenchmarksTest
{
      [TestMethod]
        public void Test_GetSquareSurfaceArea()
        {
            Benchmarks b = new Benchmarks();
            int result;
            result = b.GetSquareSurfaceArea(1, 0);
            Assert.AreEqual(result, 0);

            result = b.GetSquareSurfaceArea(1, 1);
            Assert.AreEqual(result, 1);
        }

        [TestMethod]
        public void Test_NumberSquare()
        {
            Benchmarks b = new Benchmarks();
            int result;
            result = b.NumberSquare(1);
            Assert.AreEqual(result, 1);

            result = b.GetSquareSurfaceArea(0);
            Assert.AreEqual(result, 0);
        }


        [TestMethod]
        public void Test_GetCubeSurfaceArea()
        {
            Benchmarks b = new Benchmarks();
            int result;

            result = b.GetCubeSurfaceArea(1, 1, 0);
            Assert.AreEqual(result, 0);

            result = b.GetCubeSurfaceArea(1, 1, 1);
            Assert.AreEqual(result, 6);
        }

        [TestMethod]
        public void Test_GetCubeVolume()
        {
            Benchmarks b = new Benchmarks();
            int result;

            result = b.GetCubeVolume(1, 1, 0);
            Assert.AreEqual(result, 0);

            result = b.GetCubeVolume(1, 1, 1);
            Assert.AreEqual(result, 1);
        }
}
```

3. Priedas: vienetų testų klasės sugeneruota Pex įrankiu

```
public partial class BenchmarksTest
    {
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetCubeSurfaceArea01()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetCubeSurfaceArea(s0, 0, 0, 0);
            Assert.AreEqual<int>(0, i);
            Assert.IsNotNull((object)s0);
        }
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetCubeSurfaceArea02()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetCubeSurfaceArea(s0, 1, 0, 0);
            Assert.AreEqual<int>(0, i);
            Assert.IsNotNull((object)s0);
        }
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetCubeSurfaceArea03()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetCubeSurfaceArea(s0, 1, 1, 0);
            Assert.AreEqual<int>(0, i);
            Assert.IsNotNull((object)s0);
        }
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetCubeSurfaceArea04()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetCubeSurfaceArea(s0, 1, 1, 1);
            Assert.AreEqual<int>(6, i);
            Assert.IsNotNull((object)s0);
        }
    }
```

```
public partial class BenchmarksTest
    {
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetCubeVolume01()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetCubeVolume(s0, 0, 0, 0);
            Assert.AreEqual<int>(0, i);
            Assert.IsNotNull((object)s0);
        }
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetCubeVolume02()
        {
```

```
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetCubeVolume(s0, 1, 0, 0);
            Assert.AreEqual<int>(0, i);
            Assert.IsNotNull((object)s0);
        }
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetCubeVolume03()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetCubeVolume(s0, 1, 1, 0);
            Assert.AreEqual<int>(0, i);
            Assert.IsNotNull((object)s0);
        }
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetCubeVolume04()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetCubeVolume(s0, 1, 1, 1);
            Assert.AreEqual<int>(1, i);
            Assert.IsNotNull((object)s0);
        }
    }
```

```
public partial class BenchmarksTest
    {
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetSquareSurfaceArea01()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetSquareSurfaceArea(s0, 0, 0);
            Assert.AreEqual<int>(0, i);
            Assert.IsNotNull((object)s0);
        }
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetSquareSurfaceArea02()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetSquareSurfaceArea(s0, 1, 0);
            Assert.AreEqual<int>(0, i);
            Assert.IsNotNull((object)s0);
        }
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void GetSquareSurfaceArea03()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.GetSquareSurfaceArea(s0, 1, 1);
            Assert.AreEqual<int>(1, i);
            Assert.IsNotNull((object)s0);
        }
    }
```

```
public partial class BenchmarksTest
    {
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void NumberSquare01()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.NumberSquare(s0, 0);
            Assert.AreEqual<int>(0, i);
            Assert.IsNotNull((object)s0);
        }
        [TestMethod]
        [PexGeneratedBy(typeof(BenchmarksTest))]
        public void NumberSquare02()
        {
            int i;
            Benchmarks s0 = new Benchmarks();
            i = this.NumberSquare(s0, 1);
            Assert.AreEqual<int>(1, i);
            Assert.IsNotNull((object)s0);
        }
    }
```

4. Priedas: Kodo eilučių skaičius ir ciklomatinis sudėtingumas „Crunchbug" projekte

4 lentelė. Projektinės dalies kodo eilučių skaičius

Programos dalis	Kodo eilučių skaičius	Ciklomatinis indeksas
Bugtracker.DAL	3129	413
Bugtracker.GUI	3.957	559
Bugtracker.Reporter	879	323

5. Priedas: Kodo eilučių padengimas vienetų testais „Crunchbug" projekte

5 lentelė. Projektinės dalies kodo padengimas vienetų testais

Projekto dalis	Kodo padengimas %	Kodo padengimas blokais	Nepadengtas kodas	Nepadengti kodo blokai
Reporter	**22,17%**	**329**	**77,83%**	**1155**
Ftp	76,92%	30	23,08%	9
FtpClient_	0,00%	0	100,00%	604
FtpClient_.FtpException	0,00%	0	100,00%	4
Hook	41,67%	15	58,33%	21
Program	0,00%	0	100,00%	20
Recorder	0,00%	0	100,00%	169
ReporterForm	47,89%	284	52,11%	309
ReporterForm.<>c__DisplayClass5	0,00%	0	100,00%	14
Bugtracker.Reporter.Properties	0,00%	0	100,00%	5
Settings	0,00%	0	100,00%	5
DAL	**0,21%**	**2**	**99,79%**	**961**
BugtrackerDb	3,45%	1	96,55%	28

MyExceptionDB	0,00%	0	100,00%	15
attachment	0,00%	0	100,00%	42
bug	0,00%	0	100,00%	159
bugtrackerEntities	0,90%	1	99,10%	110
comment	0,00%	0	100,00%	59
component	0,00%	0	100,00%	41
log	0,00%	0	100,00%	22
priority	0,00%	0	100,00%	32
product	0,00%	0	100,00%	59
project	0,00%	0	100,00%	97
resolution	0,00%	0	100,00%	35
status	0,00%	0	100,00%	44
tag	0,00%	0	100,00%	35
users	0,00%	0	100,00%	98
users_has_bug	0,00%	0	100,00%	24
usersgroup	0,00%	0	100,00%	26
version	0,00%	0	100,00%	35
Bugtracker	**8,34%**	**403**	**91,66%**	**4429**
LoginForm	0,00%	0	100,00%	160
MyException	70,00%	14	30,00%	6
Program	0,00%	0	100,00%	14
ProjectForm	0,00%	0	100,00%	466
QueryBug	68,69%	147	31,31%	67
RegistryAccess	100,00%	35	0,00%	0
SplashScreen	88,32%	174	11,68%	23
SvnForm	0,00%	0	100,00%	140
UserForm	0,00%	0	100,00%	533
WorkFlow	93,33%	14	6,67%	1
bugFindForm	86,36%	19	13,64%	3
bugNewForm	0,00%	0	100,00%	158
mainForm	0,00%	0	100,00%	874
osForm	0,00%	0	100,00%	382
priorityForm	0,00%	0	100,00%	382
productsForm	0,00%	0	100,00%	399
tagsForm	0,00%	0	100,00%	382
versionsForm	0,00%	0	100,00%	439
Visas projektas:	***8,09%***	***734***	***91,91%***	***8343***

6. Priedas: publikacija iš IVUS 2010 konferencijos: „SYMEX - Symbolic Execution engine for .NET platform”

Abstract — **Symbolic execution is a state of the art technique that is being used in very interesting projects to improve security, to find bugs, and to help in debugging. A symbolic execution engine is basically an interpreter that figures out how to follow all paths in a program. In this work we present our symbolic execution engine and unit test generator for .NET platform "SYMEX". We evaluate it against other tool that generates test inputs randomly.**

Keywords - symbolic execution; white-box testing; automated unit data generation; test coverage; SMT solvers;

Introduction

Testing complex safety critical software always was a difficult task. Development of automated techniques for error detection is even more difficult. Well known techniques for checking software are model checking [1, 2], static analysis [3], and testing.

Model checking analyzes all program executions in a systematic way, but its weak side is scalability[4]. Static analysis is scalable and exhaustive, but it may give many warnings that are not real[5]. Testing, on the other hand, reports errors that are real, but it may miss errors since it can only analyze some of the program executions and in many times is performed by humans[6].

Symbolic Execution (SE) [7] is not a new approach to checking software. Today a few developed tools exist in industry. For Java is PathFinder [6], for .Net platform - Pex [8], for C exist CUTE[9].

Background

This tool is useful and work only then, when tested source code methods have inputs. This program does not solve oracle problem[10], that occurs in field of software testing. SYMEX is a white-box testing tool that is why source code of tested software is required. Our software is based on couple technologies (description follows):

- Symbolic execution
- Constraint solving

Symbolic execution

Symbolic execution [7] (also known as symbolic evaluation) refers to the analysis of programs by tracking symbolic rather than actual values (a case of abstract interpretation). If program is symbolically executed, a special symbolic variable (as distinct from the program's variables) is associated with the values in the analyzed function.

```
int x, y;
[1] if (x > y)
[2] result = x - y;
[3] else
[4] result = y - x;
[5] assert (result > 0);
```

Figure 1. Example for symbolic execution

The state of a symbolically executed program includes the Symbolic values of program variables, a path condition (PC) and a program counter, representing next statement to be executed. A symbolic execution tree (SET) characterizes the execution paths followed during the symbolic execution of a program.

For illustration of differences between concrete and symbolic execution, consider the simple example in Fig. 1. Assume that the values of the input parameters are x=2 and y=1. Concrete execution will follow only one program path, corresponding to the true branch of the if statement at line 1.

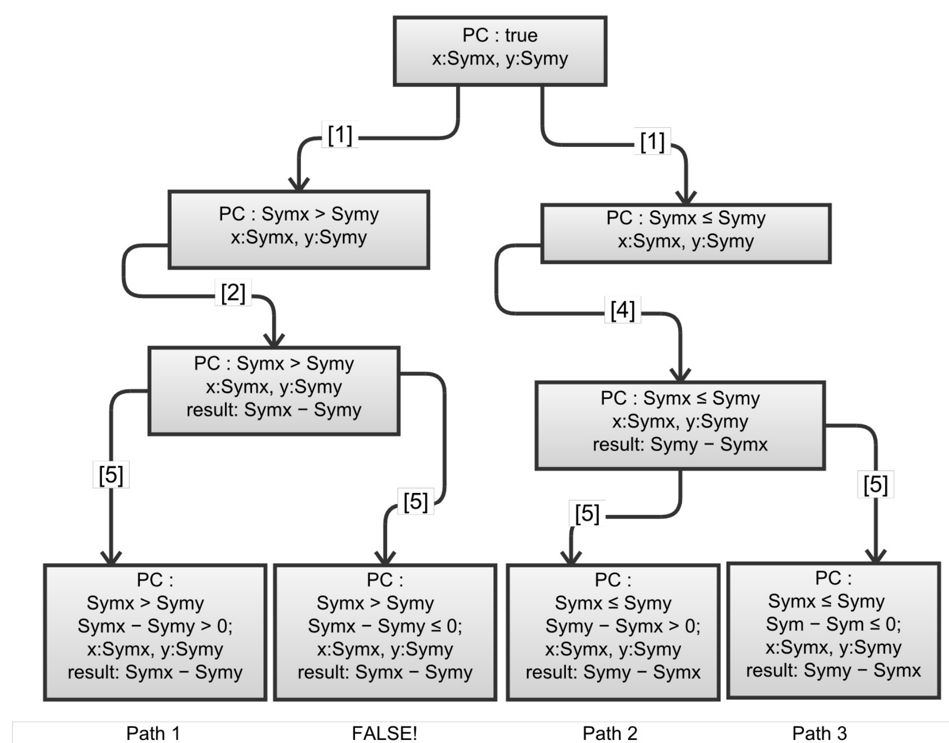


Figure 2. Symbolic execution tree

In opposite, symbolic execution starts with symbolic, values, let's say, x = Symx, y = Symy, and the initial value of path condition (PC) is true. The symbolic execution tree is illustrated in Fig. 2. At each tree leaf, PC is updated with constraints on the inputs in order to choose between alternative paths. After executing first line in the code, both alternatives of the if statement are possible, and PC is updated accordingly. If the path condition becomes "false", it means that the corresponding path is infeasible and symbolic execution does not continue for that path.

Symbolic execution explores all three different paths and discovers that the fourth path is infeasible. For test case generation, the obtained path conditions are solved (using Constraint solving) and the solutions are used as test inputs that are guaranteed to exercise all the paths through this code and 100% coverage of source code by unit tests.

White and black-box testing

The black-box approach is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure[11]. It is also termed data-driven, input/output driven[12]. Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing – a testing method emphasized on executing the functions and examination of their input and output data.

Contrary to black-box testing, software is viewed as a white-box, or glass-box in white-box testing, as the structure and flow of the software under test are visible to the tester.[12] Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure.

Constraint solving

Constraint solving is mathematical problem defined as a set of objects whose state must satisfy a number of constraints or limitations. In this case we use Satisfiability Modulo Theories (SMT)[13] solvers witch generalize Satisfiability Techniques (SAT)[14] by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers and other useful first-order theories.

SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is certifiable. Systems for SMT have applications in formal verification, compiler optimization, and scheduling, among others.

A remarkable initiative is STM-LIB[15], whose goal is to establish a standard language and a library of benchmarks for satisfiability with respect to background theories. There were a number of tools under active development in 2010: Alt-Ergo[16], Barcelogic[17], Beaver[18], Boolector[19], MathSAT[20], OpenSMT[21], Yices[22], Z3[23] and others.

SYMEX Tool

SYMEX is automated white-box[11] unit test generation tool and *does not tries to solve or solves oracle problem*, that occurs in field of software testing [10]. Our approach to the problem says that the checked source code by itself is correct. To prove that the model and source code is correct (validation and verification) other software solutions is needed. The data flow in the usage context of the tool shown in the Figure 3. SYMEX reads CIL instructions from compiled DLL files (this gives the ability to be independent from concrete programming language in .Net platform), searches for inputs in class methods, executes the code symbolically and forms symbolic execution tree with all possible execution paths. The SMT equation parser parses data into standard SMT language and calls SMT solver to calculate concrete values. SYMEX architecture allows using any of SMT solvers from SMT-Lib initiative (in this study we use Microsoft Z3 SMT solver). In final step solved values and method metadata are passed to unit test file generator and the generator outputs generated unit test file (-s). Our tool is useful in practice if we have test oracle and tested methods at least have inputs (parameters).

Example

We use a simple C# code example to illustrate how SYMEX generates unit tests from tested methods.

Method *GetCubeVolume* in class *Benchmark* under test:

```
public int GetCubeVolume(int a, int b, int c)
   if (a > 0  && b > 0 && c > 0){
         return a * b * c;
  } else {
         return 0;
        }
```

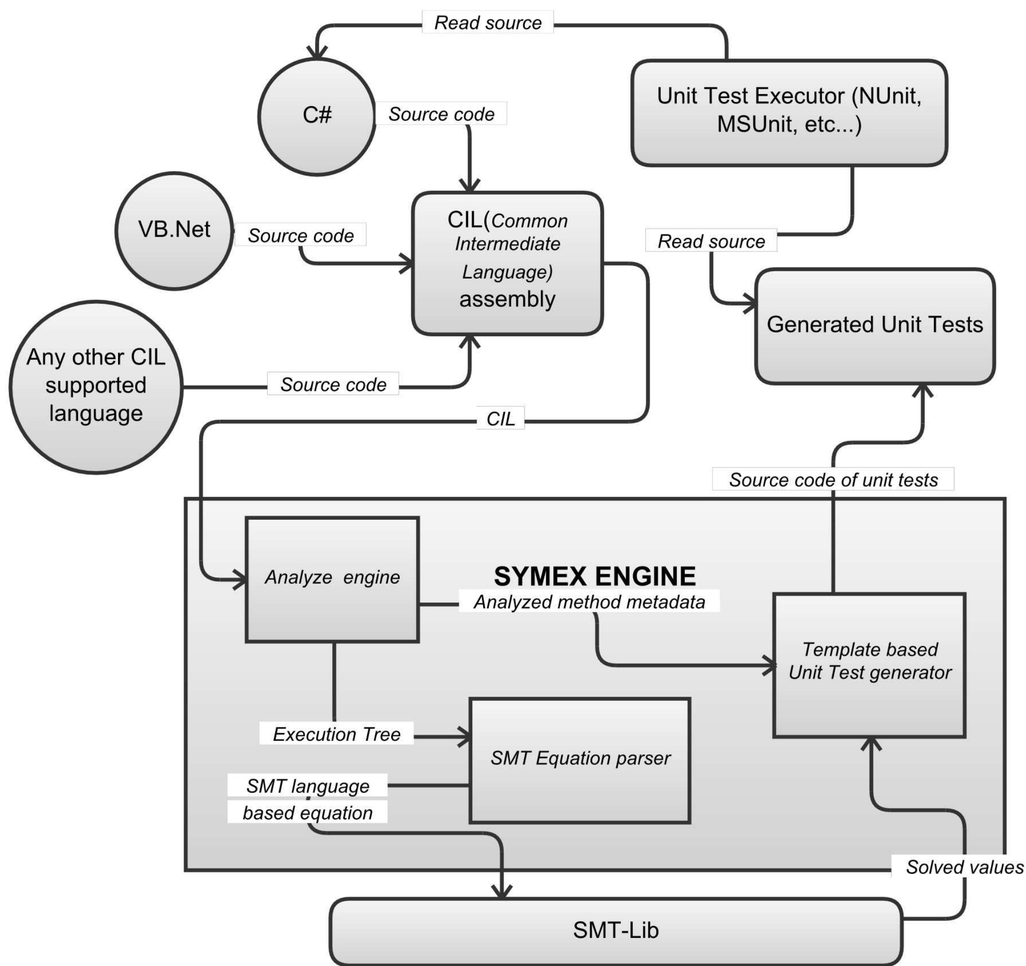


Figure 3. SYMEX architecture and working context

Generated Unit test with 100% code coverage for that method:

```
public void Test_GetCubeVolume(){
    Benchmarks b = new Benchmarks();
    int result;
    result = b.GetCubeVolume(1, 1, 0);
Assert.AreEqual(result, 0);
    result = b.GetCubeVolume(1, 1, 1);
    Assert.AreEqual(result, 1);
}
```

Experiment

The purpose of this experiment is to prove that SE methods for function analyses and generated test suites are more efficient than random generated test suites. The tool used for input parameters random generation in unit test is Random Generation Framework [24] that basically generates random data structures (integers, floats, string, arrays, etc.). Because of current SYMEX implementation limits described in Part V benchmark methods have been chosen proprietary. Benchmark method list and description is shown in Table 1.

TABLE 1 - BENCHMARK METHOD DESCRIPTIONS

No.	Benchmark method	Description	Lines of Code
1.	NumberSquare	Calculates the square number of integer	3
2.	GetSquareSurface Area	Calculates the area size of the square shape if all given edge values are positive	7
3.	GetCubeSurfaceArea	Calculates the area size of the cube shape if all given edge values are positive	10
4.	GetCubeVolume	Calculates the volume size of the cube if all given edge values are positive	10

List of software used in this experiment:

- Visual Studio 2010 RC1 v. 10.0.3
- .Net Framework 3.5v.
- Visual Studio Unit Tests (Integrated version)
- Random Generation Framework [24]
- SYMEX
- Windows XP SP3 OS

Hardware used for experiment:

- Pentium M 1.7 GHz
- 1.5 GB Ram

Results

SYMEX results were better. An important condition was that both software tools should generate test suites that achieve

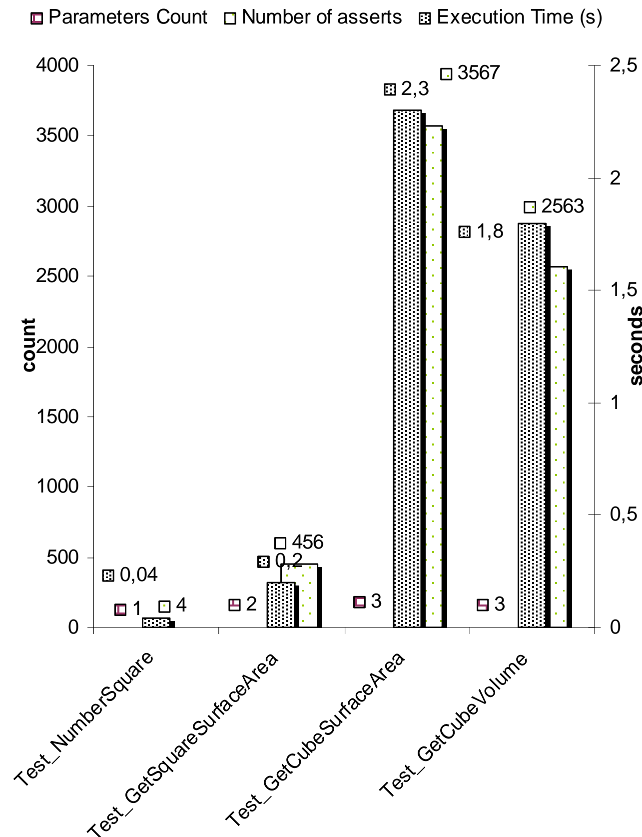


Figure 4. Random generation tool results

100% coverage of tested source code. Because one tool uses random generation algorithms, test was repeated **20** times and average results presented in Figure 4. In random generation execution time and assertation count grows exponentially when tested methods input parameters increases linear.

As a result of Symbolic Execution, test suite that was generated by SYMEX was executed within far better timeframe (average 230 times faster than random) (Fig. 6). Random generation tool is black-box tool that does not know anything about testing method logic[12]. It tries to cover all method source branches with random generated inputs. That requires a lot of input data to achieve 100% coverage (Fig. 4) even for very simple class methods.

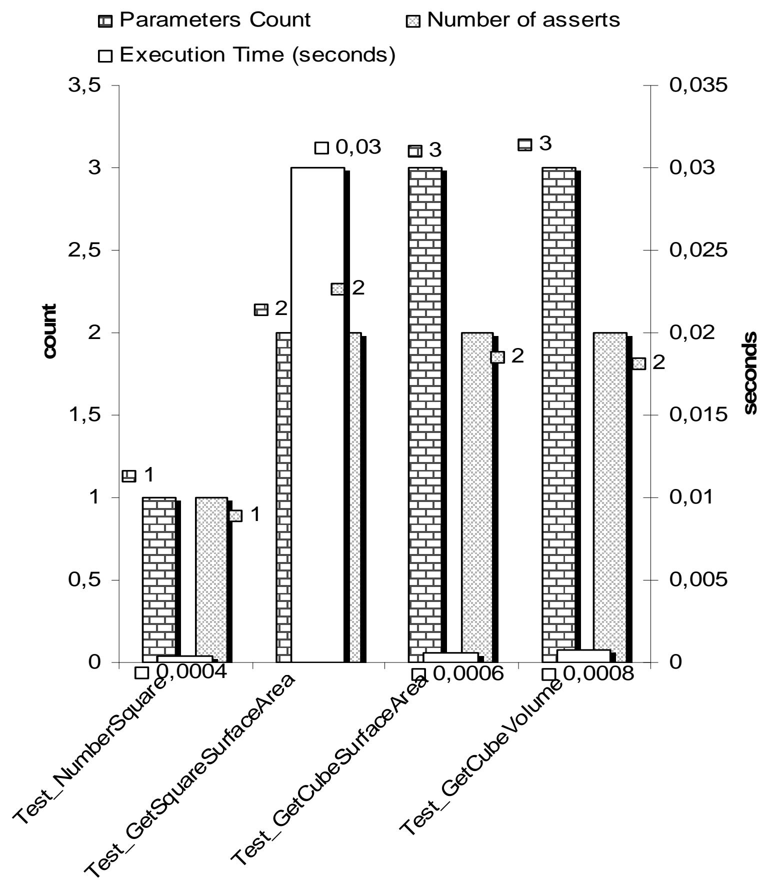


Figure 5. SYMEX results

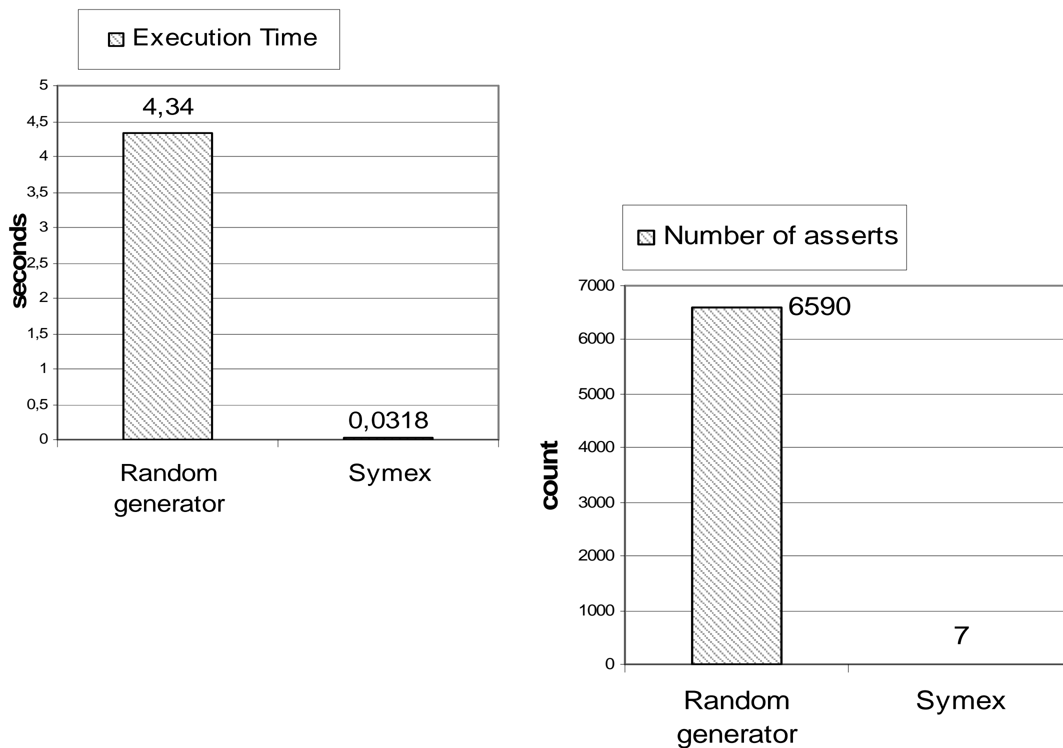


Figure 6. Comparison between two tools

Current state and future work

At the time of writing article, SYMEX tool was actively developed. Now it supports (SE engine part) all mathematical, logical and byte-shift expressions, if-then and cyclic (for, while, do…while) statements. However, for real world applications it is not enough. SYMEX currently supports only integer data types. Taken experiment is proof-of-concept solution, relevant to further research and development. The full potential of this tool will be tested in the future with more complex and real world software, as the development of SYMEX continues.

In the future we plan to implement calls for outside functions and methods, more types of data structures and types (floats, strings, array, list, etc.) and support multithreading application testing.

Experiment does not show mutation testing. In the future work such test muss made, because of importance in code quality of this measurement type.

References

[1] Godefroid, P.: ‘Model checking for programming languages using VeriSoft’. Proc. Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Paris, France1997 pp. Pages
[2] Henzinger, T., Jhala, R., Majumdar, R., and Sutre, G.: ‘Software Verification with BLAST’ (2003), pp. 624-624
[3] Louridas, P.: ‘Static code analysis’, Software, IEEE, 2006, 23, (4), pp. 58-61
[4] Hashimoto, Y., and Nakajima, S.: ‘Modular Checking of C Programs Using SAT-Based Bounded Model Checker’, in Editor (Ed.)^(Eds.): ‘Book Modular Checking of C Programs Using SAT-Based Bounded Model Checker’ (2009, edn.), pp. 515-522
[5] Evans, D., and Larochelle, D.: ‘Improving security using extensible lightweight static analysis’, Software, IEEE, 2002, 19, (1), pp. 42-51

[6] Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., and Pape, M.: ‘Combining unit-level symbolic execution and system-level concrete execution for testing nasa software’. Proc. Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, WA, USA2008 pp. Pages
[7] King, J.C.: ‘Symbolic execution and program testing’, Commun. ACM, 1976, 19, (7), pp. 385-394
[8] Tillmann, N., and de Halleux, J.: ‘Pex–White Box Test Generation for .NET’, in Beckert, B., and Hähnle, R. (Eds.): ‘Tests and Proofs’ (Springer Berlin Heidelberg, 2008), pp. 134-153
[9] Sen, K., Marinov, D., and Agha, G.: ‘CUTE: a concolic unit testing engine for C’. Proc. Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, Lisbon, Portugal2005 pp. Pages
[10] Brown, D.B., Roggio, R.F., Cross, J.H., II, and McCreary, C.L.: ‘An automated oracle for software testing’, Reliability, IEEE Transactions on, 1992, 41, (2), pp. 272-280
[11] Perry, W.E.: ‘A standard for testing application software’ (Auerbach Publications, 1986. 1986)
[12] Myers, G.J., and Sandler, C.: ‘The Art of Software Testing’ (John Wiley \& Sons, 2004. 2004)
[13] Cimatti, A.: ‘Beyond Boolean SAT: Satisfiability modulo theories’, in Editor (Ed.)^(Eds.): ‘Book Beyond Boolean SAT: Satisfiability modulo theories’ (2008, edn.), pp. 68-73
[14] Zhang, L., and Malik, S.: ‘The Quest for Efficient Boolean Satisfiability Solvers’. Proc. Proceedings of the 14th International Conference on Computer Aided Verification2002 pp. Pages
[15] Ranise, S., and Tinelli, C.: ‘The SMT-LIB Standard: Version 1.2’, Department of Computer Science, The University of Iowa, 2006
[16] Fran\, Bobot, o., Conchon, S., Contejean, E., St., and Lescuyer, p.: ‘Implementing polymorphism in SMT solvers’. Proc. Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning, Princeton, New Jersey2008 pp. Pages
[17] Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodriguez-Carbonell, E., and Rubio, A.: ‘The Barcelogic SMT Solver’. Proc. Proceedings of the 20th international conference on Computer Aided Verification, Princeton, NJ, USA2008 pp. Pages
[18] Jha, S., Limaye, R., and Seshia, S.: ‘Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic’: ‘Computer Aided Verification’ (2009), pp. 668-674
[19] Brummayer, R., and Biere, A.: ‘Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays’ (2009), pp. 174-177
[20] Bruttomesso, R., Cimatti, A., Franz\, A., Griggio, A., and Sebastiani, R.: ‘The MathSAT 4 SMT Solver’. Proc. Proceedings of the 20th international conference on Computer Aided Verification, Princeton, NJ, USA2008 pp. Pages
[21] Bruttomesso, R., Pek, E., and Sharygina, N.: ‘OpenSMT 0.2 System Description’, 2009
[22] Dutertre, B., and de Moura, L.: ‘The Yices SMT solver’, in Editor (Ed.)^(Eds.): ‘Book The Yices SMT solver’ (2006, edn.), pp.
[23] L. De Moura, and Bjorner, N.: ‘Z3: An efficient SMT solver’, Lecture Notes in Computer Science, 2008, vol. 4963, pp. 337
[24] Kheyrollahi, A.: ‘Random Generation Framework - A .NET generics framework for generating random values ’, http://www.codeproject.com/KB/cs/RandomGenerationFramework.aspx, 2009