.NET/C# instrumentation for search- based software testing

Amid Golmohammadi[1] · Man Zhang[1] · Andrea Arcuri[1,2]

Accepted: 6 July 2023 / Published online: 1 September 2023

Raport realizat de: Ene Cristian-Andrei (grupa 311)
Matei Serban-Petru (grupa 311)
Nicolae Mihnea-Vlad (grupa 311)
Nica Valentin-Teodor (grupa 312)

White-Box Software Testing

Prezentare generala

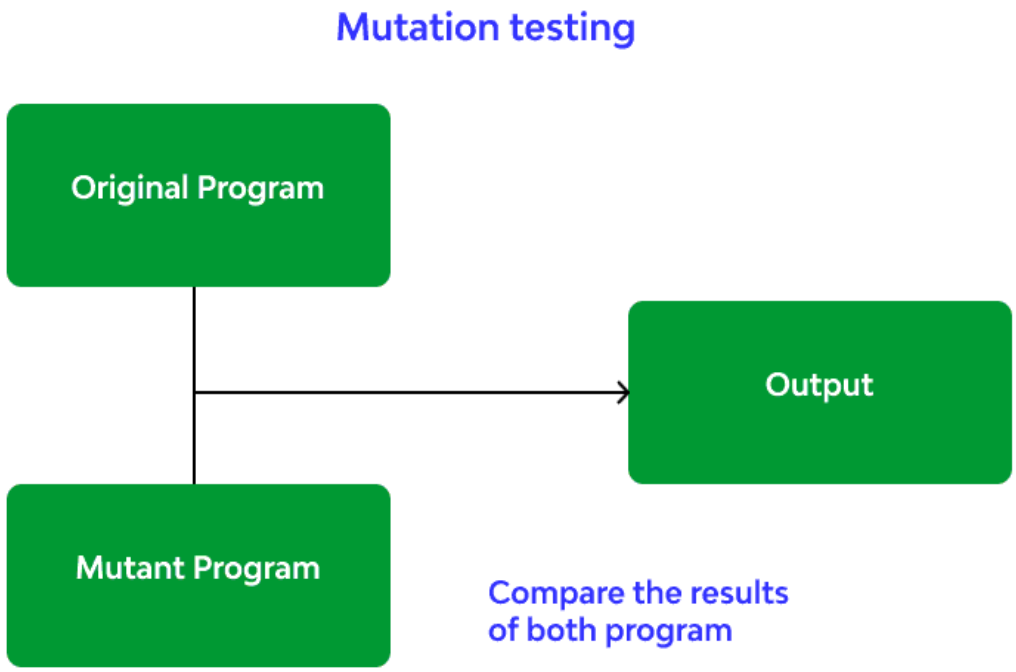
Algoritmul White Box este o metoda de testare a software-ului care se concentreaza pe examinarea si testarea interna a codului sursa. Acesta este numit si "testare structurala" sau "testare bazata pe cod", deoarece se bazeaza pe cunostinte detaliate ale codului pentru a proiecta si a executa testele. Iata cateva aspecte cheie ale algoritmului de testare White Box:

- Testarea bazata pe cunostinte interne: In contrast cu testarea Black Box, in care testerul nu cunoaste structura interna a codului, in testarea White Box, testerul are acces la codul sursa si intelege modul in care acesta este implementat.
- Structura de control: Algoritmul White Box se concentreaza pe testarea diferitelor cai de executie a programului. Acest lucru implica testarea conditiilor de decizie (if-else, switch), bucle (for, while) si alte structuri de control pentru a asigura ca toate caile posibile ale codului sunt acoperite.
- Criterii de acoperire a codului: In testarea White Box, sunt utilizate criterii specifice pentru a masura gradul de acoperire a codului sursa. Aceste criterii includ acoperirea instructiunilor (fiecare linie de cod este executata cel putin o data), acoperirea ramurilor (toate caile de decizie sunt testate) si acoperirea cailor (toate combinatiile posibile de cai sunt testate).
- Testarea unitatii: White Box este adesea utilizat in testarea unitatilor, unde fiecare componenta sau functie individuala este testata izolat, iar codul este verificat linie cu linie.
- Instrumente de testare automata: Exista o varietate de instrumente de testare automate disponibile pentru implementarea testarii White Box. Aceste instrumente pot ajuta la generarea automata a testelor si la evaluarea acoperirii codului.
- Avantaje: Testarea White Box ofera o acoperire detaliata a codului si poate detecta erori specifice de implementare sau probleme de performanta care pot fi greu de identificat prin alte metode. De asemenea, poate fi utilizata inca din stadiile incipiente ale dezvoltarii software-ului.

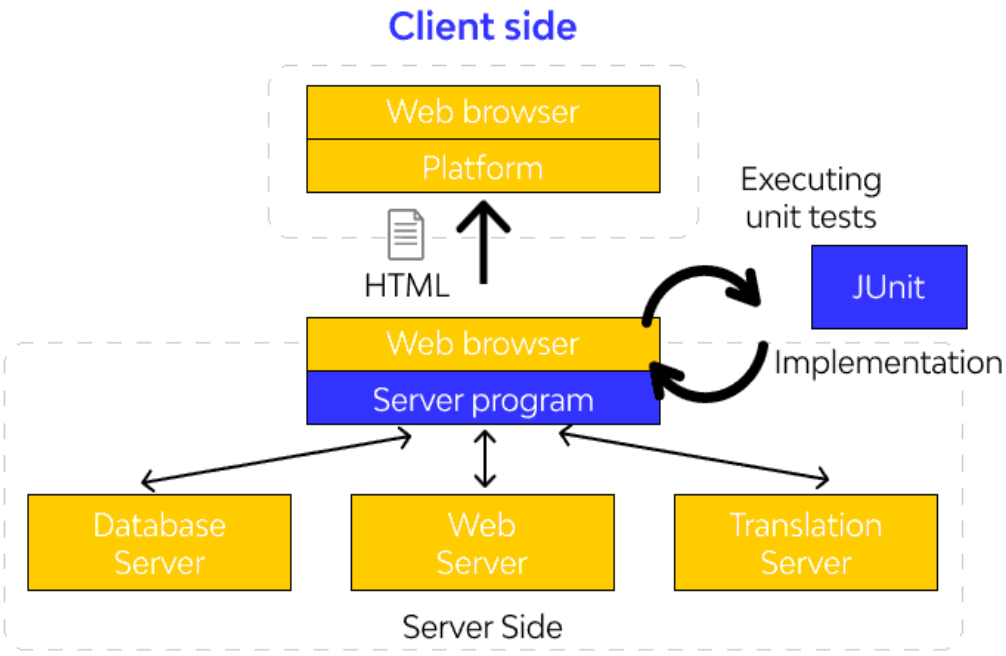
- Limitari: Testarea White Box necesita o cunoastere detaliata a codului sursa, ceea ce poate fi dificil de obtinut in unele cazuri. De asemenea, exista riscul de a omite anumite conditii sau cai de executie daca testele nu sunt proiectate corespunzator.

Tipuri de White Box Testing

Mutation Testing: Aceasta implica examinarea atenta a structurii interne a software-ului sau a codului pentru a identifica orice cai sau erori care ar putea determina sistemul sa se comporte in mod neasteptat. Este o modalitate eficienta de a aborda prabusiri si scurgeri neasteptate intr-un sistem.



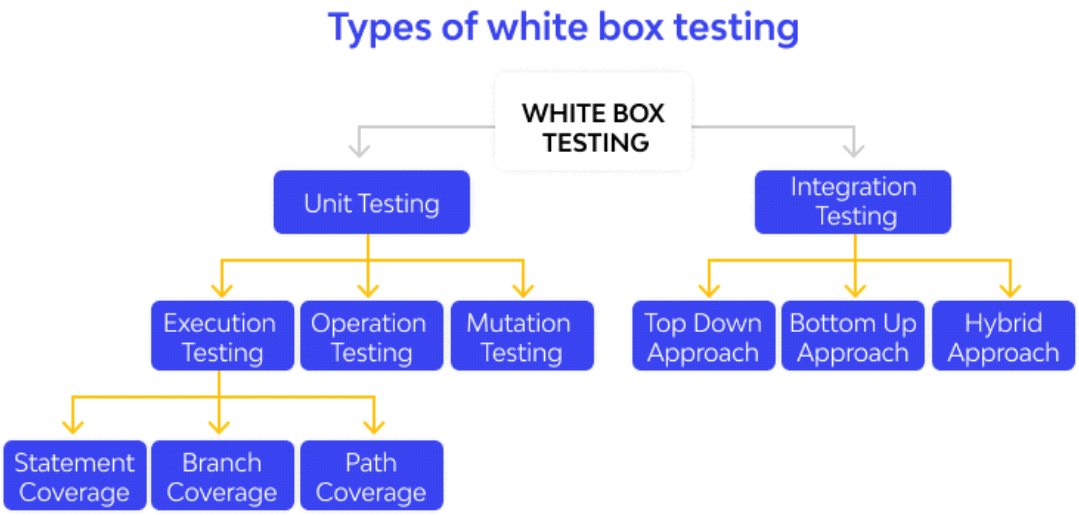
Unit Testing: In acest tip de testare, componentele aplicatiei sunt testate. Este conceput pentru a verifica daca fiecare unitate a structurii functioneaza conform asteptarilor. Acest tip de test este o idee excelenta pentru a detecta rapid erori in timpul dezvoltarii software-ului.



Integration Testing: Aceasta metoda este preocupata de combinarea unitatilor individuale sau a componentelor codului sursa si testarea lor ca grup. Scopul este de a expune orice erori in interactiunea diferitelor componente impreuna.

White box Penetration Testing: Aceasta metoda este folosita pentru a verifica daca erorile din afara sistemului pot afecta acesta. Este preocupata de integrarea software-ului cu factorii externi si de erorile din exterior care pot duce la caderea sistemului.

Static Code Analysis: Acest lucru se refera la analiza atenta a fiecarei linii de cod si verificarea erorilor. Erorile de baza din cod pot fi corectate, iar parti intregi pot fi inlocuite daca se constata ca nu sunt satisfacatoare.



Cele 3 tehnici principale de white box testing:

Statement Coverage

"Statement Coverage" este cel mai fundamental tip de examinare a includerii codului in testarea White Box a programelor. Acesta masoara numarul de instructiuni executate in codul sursa al unei aplicatii.

Iata ecuatia pentru calculul acestuia:

Statement coverage = (Number of statements executed/Total number of statements) * 100

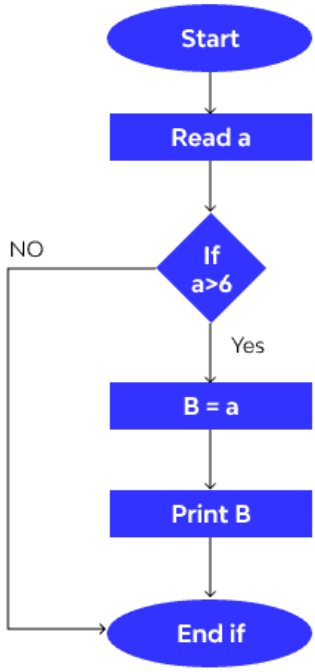
READ (a)

IF a>6 THEN

B = a
ENDIF

PRINT b

Here in this example Statement Coverage is 100%



Breach Coverage

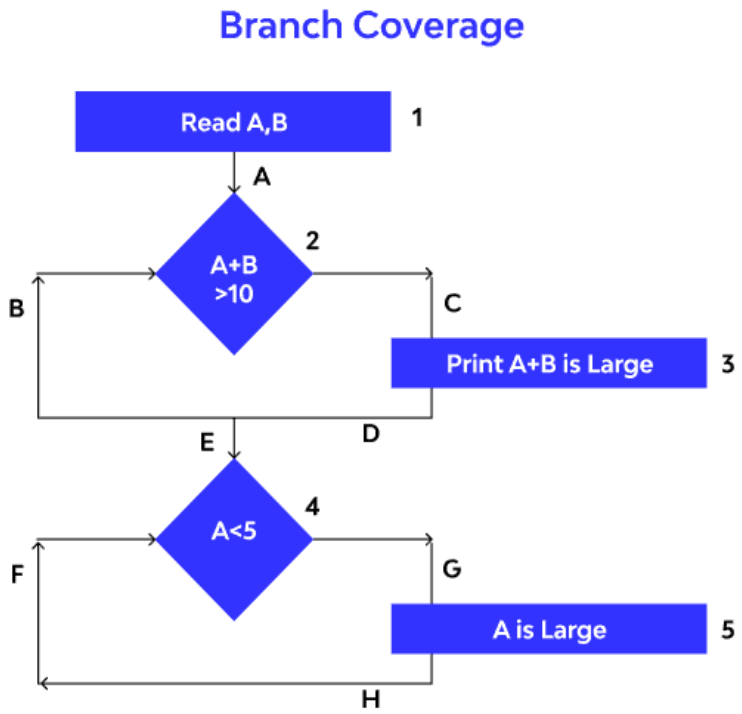
Acoperirea ramurii (Branch Coverage) este o metoda de testare a programelor de tip white box care masoara numarul de parti ale structurilor de control care au fost executate.

Poate verifica daca sunt evaluate instructiunile, declaratiile de caz si alte bucle restrictive prezente in codul sursa.

De exemplu, intr-o instructiune if, acoperirea ramurii poate determina daca ambele ramuri valide si false au fost executate.

Iata formula pentru calculul acesteia:

Branch coverage = (Number of branches executed/Total number of branches) * 100



Function Coverage

Acoperirea funcției (Function Coverage) evaluează numărul de funcții definite care au fost apelate. Un analizator de software poate, de asemenea, să ofere diferite parametri de date pentru a evalua dacă logica funcțiilor se comportă conform intenției.
Iată formula pentru calculul acestuia:

Function coverage = (Number of coverage executed/Total number of coverage) * 100

Abstract:

Articolul evidentiaza o lacuna in cercetarea utilizarii tehnicilor de testare a software-ului pentru aplicatiile C# si propune o abordare bazata pe cautare, impreuna cu un instrument open-source, pentru testarea in afara cutiei a acestor aplicatii. Metoda utilizeaza intrumente de byte-code.NET pentru a evalua acoperirea codului si defineste moduri de ghidare a procesului de cautare. Accentul este pus pe acoperirea ramurilor in codul sursa pentru a masura nivelul de apropiere.

1. Introducere

Desi C# este unul dintre cele mai utilizate limbaje de programare pentru dezvoltarea de aplicatii de afaceri de sine statatoare si web, inclusive aplicatii bazate pe cloud(conform The State of the Octoverse[1]), nu exista practici disponibile pentru generarea automata a testarii sistemului pentru aceste aplicatii. Metodele de testare bazate pe cautare(SBST-Search Based Software Testing) au avut success atat in industrie, cat si in cercetare(conform lui Harman et al., 2012[2]). m

Pana in present, nu exista instrument SBST pentru aplicatiile C#. Avand in vedere utilizarea frecventa a limbajului C# in industrie, acesta reprezinta o lacuna semnificativa in literature de cercetare.

EVOMASTER este un fuzzer open-source (instrument software gratuit si care are codul sursa deschis pentru a fi utilizat, modificat si distribuit de catre comunitate, cu scopul de a testa aplicatiile sau sistemele software in identificarea si exploatarea posibilelor vulnerabilitati sau erori in sofware) ce utilizeaza algoritmi evolutive pentru a efectua testarea automata atat black-box, cat si white-box a API-urilor REST si GraphQL. Cu toate acestea, este important de mentionat ca sunt acceptate doar API-urile bazate pe JVM si Node.JS pentru testarea in cutie.

Performanta EVOMASTER in modul white-box a fost evaluata in diverse studii empirice, comparandu-se cu alte tehnici. În acest studiu, se extinde EVOMASTER pentru a permite white-box fuzzing-ul API-urilor REST.NET/C#. Acest lucru implică utilizarea algoritmului de căutare și a funcției de fitness a EVOMASTER-ului, împreună cu heuristicile noastre SBST.NET/C#. Deoarece limbajul C# este orientat pe obiecte, acesta poate fi compilat în Common Intermediate Language (CIL).

În primul rând, se construiește o instrumentație a bytecode-ului.NET și se adaugă probe pentru a colecta acoperirea codului în timpul execuției. În plus, pentru heuristicile white-box SBST, se utilizează distanța între ramuri pentru a ghida căutarea într-un mod mai eficient, în special pentru tipurile de date numerice și de șiruri de caractere.

Pentru a evalua eficacitatea abordării noastre, am integrat instrumentele noastre de bytecode și heuristica bazată pe distanțe de ramură în EVOMASTER, denumită EVOMASTER.NET. Am utilizat EVOMASTER cu setările sale implicite, cum ar fi Resource-based Sampling și am activat configurările necesare pentru a permite testarea în C#. Am efectuat un experiment comparând EVOMASTER.NET cu o abordare de testare cu cutie gri pe trei API-uri REST.NET open-source. Două dintre acestea se concentrează pe probleme numerice (de exemplu, cs-rest-ncs) și probleme de șiruri (de exemplu, cs-rest-scs), în timp ce al treilea API gestionează meniul unui restaurant și interacționează cu o bază de date Postgres (menu-api). Rezultatele experimentului arată că abordarea noastră aduce o îmbunătățire semnificativă și clară în comparație cu abordarea de testare cu cutie gri pentru API-urile cs-rest-ncs și cs-rest-scs, iar pentru menu-api, obținem rezultate echivalente din punct de vedere statistic. Analiza acoperirii codului generat de teste arată că abordarea noastră poate rezolva majoritatea ramurilor numerice și de șir, evidențiată prin acoperirea ridicată a liniilor de cod (de exemplu, până la 98% pentru problemele numerice și până la 86% pentru problemele de șiruri).

Totuși, în cazul menu-api, care interacționează cu o bază de date, nu am observat o îmbunătățire semnificativă, sugerând că adaptarea tehnicilor de manipulare a bazelor de date SQL poate fi necesară pentru a obține rezultate mai bune .

EVOMASTER este un instrument de testare automată care a strâns peste 260 de stele pe GitHub și a fost descărcat de peste 1400 de ori, indicând utilizarea sa în practica industrială. Un exemplu notabil este integrarea cu succes a EVOMASTER în sistemele de integrare continuă ale companiei chineze Meituan, o mare platformă de comerț electronic. Această integrare vine în răspuns la o nevoie exprimată de practicieni din industrie pentru suport pentru C#/.NET în EVOMASTER. În ciuda numeroaselor instrumente prezentate în literatura de specialitate pentru fuzzing-ul API-urilor RESTful, majoritatea sunt cutii negre, ceea ce înseamnă că nu analizează codul sursă al aplicațiilor testate. Dezvoltarea unui fuzzer cutie albă pentru API-urile Web, în special pentru C#/.NET, a fost o provocare considerabilă, necesitând luni de muncă și peste 10.000 de linii de cod doar pentru instrumentare și driver. Acest lucru explică de ce EVOMASTER rămâne singurul fuzzer cu cutie albă pentru API-uri Web, chiar și după ce a fost open-source din 2016.

Principalele contribuții ale acestei lucrări includ:

a) Introducerea tehnicilor SBST cu cutie albă în literatura pentru aplicațiile .NET.

b) Dezvoltarea unei noi instrumentări de bytecode cu SBST heuristică pentru .NET, care poate fi integrată cu alte tehnici SBST existente.

c) Crearea unei soluții automate cu o implementare de instrumente open-source pentru activarea SBST a aplicațiilor .NET.

d) Realizarea unui studiu empiric în care am reușit să replicăm cu succes fuzing-ul API-urilor RESTful folosind tehnicile SBST.

2. Background

2.1. Programarea.NET

.NET[3] este o platformă de dezvoltare de la Microsoft, lansată în 2016 sub numele de .NET Core, care este open source și poate fi utilizată pentru construirea diverselor tipuri de aplicații. Această platformă suportă limbajele de programare C#, F# și Visual Basic, iar codul scris în aceste limbi este compilat în Common Intermediate Language (CIL), un set de instrucțiuni bytecode orientate spre obiecte și bazate pe stack. C# este cel mai popular limbaj de programare pentru dezvoltarea aplicațiilor .NET și, conform datelor din The State of the Octoverse[1], este printre cele mai utilizate limbaje de programare din lume.

```
        if(a > 0)
                Console.WriteLine("a is greater than 0");
```

Fragmentul de cod de mai sus conține o metodă care ia ca intrare a, care este de tip int. Apoi, imprimă un șir la consola dacă a este mai mare decât zero. Codul CIL pentru un astfel de program este următorul:

```
.method public hidebysig instance void
Foo(
int32 a
)       cil managed
{
        .maxstack 2
        .locals init (
        [0] bool V_0
        )

        IL_0000:nop

        IL_0001:  ldarg.1           //a
        IL_0002:  ldc.i4.0
        IL_0003:  cgt
        IL_0005:  stloc.0           // V_0

        IL_0006:  ldloc.0           // V_0
        IL_0007:  brfalse.s         IL_0014

        IL_0009:  ldstr             "a is greater than 0"
        IL_000e:  call              void [System.Console]System.Console::WriteLine(string)
        IL_0013:  nop
```

```
        IL_0014:  ret
} // end of method MyClass::Foo
```

În .NET, instrucțiunile de bytecode sunt similare (cu unele diferențe) cu cele ale mașinii virtuale Java (JVM), conform standardului ECMA-335[4]. În exemplul dat, instrucțiunile ldarg.1 împing valoarea de intrare pe stivă, în timp ce ldc.i4.0 împinge constanta 0. Instrucțiunea cgt compară aceste valori și împinge rezultatul (0 sau 1) pe stivă. Instrucțiunile stloc.0 și ldloc.0 sunt utilizate în scopuri de debugging și sunt eliminate în compilarea în modul "release". Instrucțiunea brfalse.s IL_0014 este un salt condiționat: dacă valoarea de pe stivă este 0, programul sare la eticheta IL_0014, altfel este afișat un mesaj. Instrucțiunea ldstr împinge un șir constant pe stivă, care este apoi afișat folosind WriteLine.

2.2. Algoritmul MIO

În contextul testării sistemului cu white-box, gestionarea eficientă a zecilor sau chiar a sute de mii de ținte de testare poate fi dificilă. Pentru a gestiona această cantitate mare de ținte, MIO utilizează managementul dinamic al populațiilor, unde fiecare țintă de testare are asociată o populație cu o dimensiune maximă n. Aceste populații sunt gestionate dinamic în timpul căutării. Inspirat de (1+1) Evolutionary Algorithm(este o variantă simplificată a algoritmilor genetici, utilizată în special pentru optimizarea problemelor de optimizare combinatorială. În acest algoritm, este gestionată o singură soluție candidată (denumită părinte) și se încearcă să se găsească o soluție mai bună (denumită copil) prin aplicarea unei operații de mutație asupra părintelui), MIO este alcătuit din doi operatori principali: eșantionare și mutator. Acești operatori contribuie la optimizarea procesului de căutare și la îmbunătățirea eficienței testării sistemului cu white-box.

```
Algorithm 1: Pseudo-code of the MIO Algorithm
        Input :Stopping condition C, Fitness function δ, Population size n,
                Probability for random sampling Pr , Start of focused search F
        Output:Archive of optimised individuals A

        T ← SetOfEmptyPopulations()
 A ← {}
 while ¬C do
  if Pr > rand() then
    p ← RandomIndividual()
  else
    p ← SampleIndividual(T)
    p ← Mutate(p)
  end
  foreach element k ∈ ReachedTargets(p) do
    if NewTarget(k) then
      T ← T ∪ Tk
    end
    T□ ← Tk  ∪ {p}
    if IsTargetCovered(k) then
        UpdateArchive(A,p)
        T ← T \ Tk
```

```
    else if | T□ |>n then
      RemoveWors(T□, δ)
    end
  end
  UpdateParameters(F,Pr,n)
end
```

În algoritmul prezentat (Algoritmul 1), căutarea începe cu un set gol de populații T și o arhivă goală A care stochează soluții fezabile pentru ținte. În fiecare iterație, fie se eșantionează un test, fie se mută un test preluat din T, acțiunile fiind controlate de o probabilitate Pr. Aceste acțiuni pot genera un nou test p, care este apoi executat. Cu toate acestea, în timpul execuției, obiectivele de testare nu sunt întotdeauna atinse, acoperite sau satisfăcute. Un exemplu ar putea fi o țintă de ramură pentru instrucțiunea if(x == 42), unde se evaluează dacă valoarea lui x este egală cu 42.

```
public int foo(x){
  if(x <= 0) return -1;
  if(x == 42) return 1;{
    return 0;
  }
}
```

În testarea sistemului cu white-box, există unele ținte de testare care pot fi imposibile, iar utilizatorii sunt interesați în special de acoperirea acestor ținte, mai degrabă decât de apropierea lor heuristicoasă. Pentru a gestiona eficient aceste ținte, MIO utilizează managementul dinamic al populațiilor, unde fiecare țintă are asociată o populație, actualizată în timpul căutării. În acest proces, se iau în considerare mai mulți factori:

a) Dacă o țintă este atinsă pentru prima dată, se creează o nouă populație care conține soluția și este adăugată la setul de populații.

b) Dacă ținta este acoperită, soluția este adăugată în arhiva A, iar populația corespunzătoare este eliminată.

c) În caz contrar, soluția este adăugată la populația corespunzătoare; dacă aceasta depășește dimensiunea maximă, se elimină cea mai slabă soluție.

MIO integrează, de asemenea, eșantionarea orientată către feedback pentru a se concentra pe țintele cu îmbunătățiri recente și pentru a face schimb între explorare și exploatare în peisajul de căutare, gestionând dinamic parametrii cum ar fi probabilitatea de mutație și dimensiunea populației. De exemplu, în etapele inițiale ale căutării, se pune accent pe explorare, iar apoi se trece treptat la exploatare pentru a se concentra pe acoperirea obiectivelor atinse.

2.3. Distanta dintre ramuri

Pentru a atinge o acoperire extinsă a codului în testarea sistemului, este esențial să se folosească heuristici pentru a ghida căutarea și a genera intrări care să rezolve constrângerile din sistemul testat (SUT), cum ar fi predictele complexe din declarațiile if. Una dintre cele mai comune heuristici utilizate în literatură este distanța de ramură, inițial concepută pentru a gestiona constrângerile numerice și ulterior extinsă pentru a acoperi și constrângerile de șir. De exemplu, pentru o afirmație simplă ca if(x==42), distanța de ramură calculează diferența absolută dintre valoarea lui x și 42. Astfel, căutarea este ghidată să modifice x într-un mod care minimizează această distanță.

În acest articol, o provocare majoră abordată este aplicarea acestor funcții de distanță de ramură la bytecode-ul .NET CIL (Common Intermediate Language). Prin utilizarea acestor distanțe de ramură adaptate la specificul .NET, se îmbunătățește capacitatea de ghidare a căutării pentru a obține o acoperire mai eficientă a codului.

3. Related Work

EvoSuite este o unealta SBST care produce cazuri de testare unitara cu afirmatii pentru clasele Java. EvoSuite face aceasta prin implementarea unei tehnici hybride care produce si optimizeaza toate seriile de teste pentru a indeplini obiectivele de acoperare. EvoSuite este probabil cea mai cunoscuta unealta SBST si face instrumentatie bytecode pentru JVM, sustinand ramurile calculelor la distanta. Pex foloseste executia dinamica simbolica pentru a genera mici serii de testare unitara pentru programe dezvoltate cu .NET. Pex reuseste sa faca aceasta prin determinarea datelor de intrare ale testului pentru testarile unitare parametrizate de un program de sistemizare si analiza. Pex invata comportamentul programului si genereaza noi date de intrare ale testului cu comportament variat al programului cu ajutorul unui constrangator.

Randoop este o unealta pentru Java si .NET care creeaza testari unitare cu ajutorul unei tehnici de testare aleatorie directionata prin feedback. Scopul acestuia este de a evita producerea ilegala si redundanta a datelor de intrare prin utilizarea feedback-ului de executie de la executarea datelor de intrare ale testarii in timp ce acestea sunt create. Randoop construieste secvente de metode una cate una, alegand metoda “call at random” (apelarea aleatorie) si selectand argumente din secventele construite anterior, din care denota faptul ca Randoop actioneaza ca un chid in crearea noilor date de intrare.

Cu privire la fuzzing RESTful APIs, majoritatea uneltelor au fost prezentate in literatura, pe langa EvoMaster, cum ar fi Restler, RestTestGen, Restest, RestCT, bBOXRT si Schemathesis, care sunt toate black-box testing. Din ceea ce se stie, nu exista nicio tehnica SBST in literatura pentru testarea white-box a programelor .NET. Procedura din acest articol le permite practicantilor sa utilizeze testarea white-box pentru aplicatiile .NET, ceea ce asigura rezultate mai bune decat testarea black-box cand sursa cod este accesibila.

4. Instrumentare .NET

4.1 Instrumentare Bytecode

Instrumentatia implementata pentru programele .NET este efectuata cu ajutorul librariei “Mono.Cecil”[5], care face posibila analiza si modificarea codului CIL. Instrumentatia trebuie sa fie integrata cu o tehnica SBST pentru a genera teste. Pentru efectuarea experimentelor din acest articol au fost folosite abilitatile uneltei EvoMaster, care acesta genereaza cazuri de testare la nivel de sistem pentru RESTful APIs. Exista doua componente principale in EvoMaster: proces de baza si proces de driver. Driver-ul asigura functionalitatile ca RESTful APIs, cu care nucleul comunica prin HTTP. Instrumentarea e implementata ca aplicatie a consolei de baza .NET Core. Principala metoda a acestei aplicatii preia path-ul tintei SUT ca parametru de intrare, efectueaza instrumentatia pe el si intr-un final salveaza fisierul instrumentat in locatia specificata. Pentru a utiliza EvoMaster pentru un SUT, s-a implementat un driver scris in C# care implementeaza aceleasi puncte finale ca driver-ul original JVM. Astfel, EvoMaster este capabil sa genereze cazuri de test care sunt secvente de apeluri HTTP catre puncte finale diferite ale SUT bazate pe xUnit.

4.2 Code Coverage

Un program .NET este format dintr-un numar de asamblari, fiecare continand cateva clase. Fiecare clasa contine metode, lucrate una cate una. Fiecare afirmatie din metoda va deveni o tinta de testare, in care se insereaza probe inainte si dupa ele pentru a urmari de fiecare data faptul cand sunt acoperite in timpul cautarii. Scopul uneltei EvoMaster este de a genera cazuri de teste la nivel de sistem care realizeaza cel mai mare numar de tinte acoperite.

Pentru fiecare instructiune se ia in considerare coordonarea de inceput a metodei in sursa cod ca indicator a unei noi afirmatii. Acea informatie este obtinuta de la un obiect de tip SequencePoint care este atribuita instructiunii. Aceasta instructiune nu este doar un indicator al unei noi afirmatii, dar poate insemna si sfarsitul unei alte afirmatii, exceptand cazul in care afirmatia este prima din metoda. Ca rezultat, se insereaza alta proba CompletedStatement pentru a semnala sfarsitul afirmatiei anterioare.

```
public class MyClass {
  public void MyMethod() {
    int x = 123;
  }
}
```

Acest fragment de cod defineste o clasa numita MyClass. In interiorul acestei clase exista o metoda numita MyMethod, care este de tip void, ceea ce inseamna ca nu returneaza niciun rezultat. In corpul metodei MyMethod, este declarata o variabila locala de tip int numita x, iar aceasta este initializata cu valoarea 123. Cu alte cuvinte, atunci cand metoda MyMethod este apelata, variabila x este creata si initializata cu valoarea 123. Este important de retinut ca variabila x este accesibila doar in interiorul metodei MyMethod si nu poate fi utilizata in afara acesteia.

```
public class MyClass {
  public void MyMethod() {
    Probes.MarkStatementForCompletion("MyClass", "MyMethod", 2, 32);
    Probes.EnteringStatement("MyClass", "MyMethod", 3, 13);
     int num = 123;
     Probes.CompletedStatement("MyClass", "MyMethod", 3, 13);
     Probes.MarkStatementForCompletion("MyClass", "MyMethod", 4, 9);
  }
}
```

Fragmentul de cod oferă o definiție simplă a unei clase numită MyClass. În interiorul acestei clase, există o metodă numită MyMethod. În corpul acestei metode sunt inserate apeluri către funcții sau metode ale unei clase numită Probes. Aceste apeluri par să fie parte dintr-un instrument de analiză sau de debugare care marchează și urmărește execuția codului.

Probes.MarkStatementForCompletion("MyClass", "MyMethod", 2, 32); - Marchează o instrucțiune pentru a fi marcată ca finalizată în metoda MyMethod din MyClass. Aceasta este o funcție care marchează începutul unei instrucțiuni.

Probes.EnteringStatement("MyClass", "MyMethod", 3, 13); - Marchează începutul unei instrucțiuni în metoda MyMethod din MyClass.

int num = 123; - Aici este declarată o variabilă locală numită num și este inițializată cu valoarea 123.

Probes.CompletedStatement("MyClass", "MyMethod", 3, 13); - Marchează finalizarea unei instrucțiuni în metoda MyMethod din MyClass.

Probes.MarkStatementForCompletion("MyClass", "MyMethod", 4, 9); - Marchează o altă instrucțiune pentru a fi marcată ca finalizată în metoda MyMethod din MyClass.

Acest cod este o simplă demonstrație a cum se utilizează anumite funcționalități ale clasei Probes în interiorul unei metode.

Probele, EnteringStatement, care se insereaza inainte, la inceputul unei afirmatii si CompletedStatement, sunt inserate inainte si dupa atribuire variabila. Ele sunt pur si simplu invocatii ale metodelor statice care sunt inauntru aplicatiei consola dezvoltata pentru a ajuta procesul instrumentelor. Instructiunile nop and ret care sunt pentru a deschide si a inchide curly braces sunt folositoare pentru ca acestea afla daca o metoda este atinsa, chiar daca are lipsa sau daca executia unei metode este completa.

Probele (de exemplu MarkStatementForCompletion), care marcheaza completarea curly braces, sunt inserate la liniile 3 si 7 in codul de sus. De fiecare data cand e executat EnteringStatement, tintele pentru clasa si linie sunt marcate ca reusite setand valoarea lor la 1. Pentru afirmatie, e setat la 0.5 pentru ca afirmatiile pot face exceptii si nu se stie daca nicio exceptie nu este luata in considerare pana ce afirmatia nu este executata in intregime. Daca CompletedStatement este atinsa, atunci valoarea euristica pentru afirmatii este setata la 1.

Inserarea EnteringStatement and CompletedStatement nu este mereu directa in exemplul de sus. Cand vine vorba de instructiuni care schimba cursul controlului, programul se poate corupe sau logica s-ar putea sa se schimbe daca nu este manevrata cu grija. Atunci cand se plaseaza marcajul EnteringStatement inainte de o instructiune, exista posibilitatea ca acea instructiune sa fie destinatia unui salt in alta parte a codului, ceea ce poate face ca EnteringStatement sa nu fie atinsa. Pentru a rezolva aceasta problema, se efectueaza o verificare pentru a determina daca instructiunea curenta este destinatia unui salt. In caz afirmativ, se modifica tinta saltului astfel incat sa fie prima instructiune a lui EnteringStatement.

In ceea ce priveste CompletedStatement, principala problema este evitarea plasarii sale dupa instructiunile care efectueaza salturi sau iesiri in mod conditionat. Daca acest lucru se intampla, CompletedStatement trebuie plasata inainte de acele instructiuni pentru a le marca ca finalizate. Aceasta nu ar trebui sa fie o problema, deoarece nu exista alte instructiuni intre care ar putea sa apara o exceptie sau sa schimbe fluxul de control.

O alta problema intampinata in timpul instrumentarii este ca exista instructiuni care au si o forma scurta si una lunga. Cand se modifica codul CIL, acest lucru poate duce la schimbarea numarului de argumente, variabile locale sau octeti ai metodei, ceea ce poate provoca un overflow si afecta codul CIL. Pentru a preveni acest lucru, fiecare instructiune este convertita in forma sa lunga prin utilizarea functiei SimplifyMacros din Mono.Cecil. Dupa finalizarea instrumentarii, se apeleaza o alta metoda numita OptimizeMacros, care le converteste inapoi la forma lor scurta, daca este posibil.

Table 1 Branch instructions that our instrumentation deals with for distance calculation

Name	Branch instructions	Description
One-arg jump	brtrue, brtrue.s brfalse, brfalse.s	Jump instruction with one argument
Two-arg compare	ceq, clt, clt.un cgt, cgt.un	Compare instruction with two arguments
Two-arg jump	bgt, bgt.s, bgt.un, bgt.un.s bge, bge.s, bge.un, bge.un.s ble, ble.s, ble.un, ble.un.s blt, blt.s, blt.un, blt.un.s beq, beq.s, bne.un, bne.un.s	Jump instruction with two arguments

4.3 Branch distance

4.3.1 Numeric

Este greu sa acoperi un numar acceptabil de tinte fara a lua in considerare instructiunile branch. Predicate complexe, cum ar fi conditiile din if statements, pot afecta fluxul de control al sistemului de testat. Precum in exemplul din sectiunea 2.1, if statement-ul ar trebui compilat in cgt, iar brfalse.s trebuie sa conduca fluxul de control spre un rezultat(0 sau 1) al predicatului a>0 al if-ului. Exista diferite tipuri de instructiuni care se pot identifica ca fiind instructiuni branch. In functie de valorile scoase din stiva de evaluare, tipurile numerice pot produce una sau doua instructiuni. Instructiunile branch se pot imparti in trei grupuri: one-arg jump, two-arg compare si two-arg jump.

Pentru a calcula distanta branchului, trebuie folosite alte probe in cod. Aceste probe au nevoie ca valorile sa fie transmise prin branch pentru a calcula cat de departe sunt de a indeplini conditia. Totusi, din cauza ca valorile sunt in varful stivei de evaluare, instructiunile branch le scot pe acestea din stiva, asa ca aceste trebuie sa fie duplicate. Pentru one-arg jumps, duplicatele sunt directe. In codul prezentat mai jos, brfalse.s iese din stiva de evaluare si face un salt pana la instructiunea de la IL_001C daca valoarea extrasa este false(adica 0). Exemple de salturi si ale versiunii instrumentate ar fi acestea:

```
IL_0014: ldloc.1    //push local variable at index 1 onto the stack
IL_0015: brfalse.s  IL_001c
```

```
IL_0008a: ldloc.1    //push local variable at index 1 onto the stack
IL_0008b: dup
IL_0008c: ldstr     "brfalse" //opCode
IL_0091:  ldstr     "TriangleClassificationImpl"  // className
IL_0096:  ldc.i4.6   //lineNo
IL_0097:  ldc.i4.4   //branchCounter
IL_0098:  call       void [EvoMaster.Instrumention]
```

```
   EvoMaster.Instrumention.Probes::
   ComputeDistanceForOneArgJumps(int32, string, string, int32, int32)
IL_0009d:  brfalse.s    IL_0101
```

Proba pentru a calcula distanta branchului este inserata fix inaintea instructiunii de branch. In afara de valorile de care avem nevoie pentru a marca tinta branchului, opCode, className, lineNo si branchCounter, mai este nevoie si de valoarea pe care brfalse.s o scoate, de asemenea. Aceasta se face prin adaugarea instructiunii dup care duplica valoarea inserata de cea anterioara in stiva de evaluare.

Calcularea distantei branchului pentru instructiuni two-arg compare si two-arg jump prezinta dificultati. Una dintre acestea ar fi faptul ca aceste tipuri de instructiuni au nevoie de doua argumente ca input, insa nu este posibil sa duplici primele doua valori din varful stivei ca la one-arg jump. Ca urmare, folosim metoda bytecode. De fiecare data cand o instructiune de tip two-arg compare si two-arg jump este atinsa, o inlocuim cu o metodacare are aceeasi utilitate din punct de vedere semantic cu instructiunea originala pe langa calcularea distantei branchului.

Table 2 Mapping two-arg jump instructions to two-arg compare and one-arg jump instructions

Original instruction	Converted instructions	Original instruction	Converted instructions
bgt	cgt + brtrue	ble	cgt + brfalse
beq	ceq + brtrue	blt	clt + brtrue
bge	clt + brfalse	bne	ceq + brfalse

```
IL_0020: ldarg.2  //Load the argument at index 2 onto the evaluation stack
IL_0021: ldarg.3  //Load the argument at index 3 onto the evaluation stack
IL_0022: ceq
IL_0024: br.s     IL_0027
```

De exemplu, instructiunea ceq scoate doua valoridin argumentele metodei, le compara si introduce 1 daca sunt egale, iar 0 daca nu sunt. Codul instrumentat pentru codul de mai sus este acesta:

```
IL_012a: ldarg.2            //Load the argument at index 2 onto the evaluation stack
IL_012b: ldarg.3            //Load the argument at index 3 onto the evaluation stack
IL_012c: ldstr              "ceq"    //pushes the opcode string
IL_0131: ldstr              "TriangleClassificationImpl"
IL_0136: ldc.i4.s           10       //push lineNo
IL_0138: ldc.i4.1           // push branchCounter
IL_0139: call               int32[EvoMaster.Instrumentation]
   EvoMaster.Instrumentation.Probes::
   CompareAndComputeDistance(int32, int32, string, string, int32, int32)
IL_013e: br.s   IL_0141
```

Instructiunile ceq sunt inlocuite de o metoda numita CompareAndComputeDistance. Mai intai, metoda calculeaza distanta prin pasarea primelor doua valori numerice introduse la IL_012a si IL_012b. Apoi, determina valoarea care trebuie introdusa in stiva bazata pe string-ul opcode introdus la IL_012c.

Pentru instructiunile two-arg jump, metoda de inlocuire este diferita. Instructiunile de acest tip, mai intai, compara valorile din cele doua inputuri si, apoi, sare catre alt punct bazat pe rezultatul comparatiei care se afla in stiva de evaluare. Aceasta ne permite sa le inlocuim pe fiecare dintre ele cu o instructiune two-arg compare uramata de un two-arg jump. Tabelul 2 contine informatii despre cum mapam aceste instructiuni.

A doua problema ar fi legata de detectarea tipului de date, ceea ce este foarte important pentru a chema proba corecta. O solutie ar fi sa folosesti o metoda care utilizeaza valori de tipul 'object' ca input, ceea ce este imposibil pentru ca acestea trebuie mai intai sa fie convertite in object. Drept urmare, o solutia mai buna ar fi creearea mai multor metode prin polimorfism, fiecare primind alt tip de date in input si sa fie detectate tipurile de date scoase de ultimele doua instructiuni de tip two-arg jump pentru a insera proba corecta.

Tipul detectat depinde de ultimele doua instructiuni. Instructiunile branch pot aparea sub forma mai multor tipuri de instructiuni care introduc valori in stiva de evaluare. Avand in vedere ca ambele valori sunt mereu de acelasi tip, detectarea tipului oricareia dintre ele este suficient. Pentru FieldDefinition, VariableDefinition, si MethodReference, tipul poate fi deduse din casting pentru operandul instructiunii si returnarea tipului acestea. Daca instructiunea se incarca din argumentul metodei, tot ce avem de facut este sa identificam indexul si sa gasim elementul care are acelasi index in parametrii metodei care pot fi dedusi folosind Mono.Cecil din metadata metodei curente si sa returneze tipul sau de date. O alta posibilitate ar fi ca instructiunea anterioara sa incarce o variabila locala. Aceste instructiuni incarca variabila locala la un index specific in stiva de evaluare. Orice variabila ldloc apare dupa un stloc care stocheaza o valoare la indexul respetiv din lista variabilelor locale. Totusi, aceste doua instructiuni nu functioneaza mereu impreuna. Pentru a rezolva aceasta problema, trebuie sa stocam toate numele variabilelor locale si tipul acestora intr-un dictionar detectand instructiunile stloc. De fiecare data cand se intalneste o instructiune de acest tip, tot ce trebuie facut este sa se ia tipul variabilei locale curente referindu-ne la dictionar. De asemenea, exista si un grup de instructiuni al caror tip de date se regaseste in titlul acestora. De exemplu, Ldc_I4,Ldc_I8,Ldc_R4 si Ldc_R8 introduc o valoare de tip int, long, double sau float in stiva. Astfel, putem detecta tipul de date doar prin parsarea OpCode-ului lor.

4.3.2 String

Pentru a determina distanta de branch pentru date de tip string avem nevoie sa identificam toti operatorii si metodele din System.String care returneaza un boolean, cum ar fi operatorul “==”, Equals, Contains, StartsWith sau EndsWith in timpul instrumentarii . Apoi, vom inlocui metoda cu o proba care sa calculeze distanta si sa faca operatia intentionata. Consideram codul a.Equals(b, StringComparison.OrdinalIgnoreCase). Codul sau CIL echivalent este urmatorul:

```
IL_0001: ldarg.1    // a
IL_0002: ldarg.2    // b
IL_0003: ldc.i4.5
IL_0004: callvirt   instance bool [System.Runtime]System.
   String::Equals(string, valuetype, [System.Runtime] System.StringComparison)
```

Codul de mai sus ia a si b din parametrii metodei, ii compara si introduce rezultatul in stiva de evaluare. In timpul instrumentatiei, odata ce System.String::Equals este identificat , il inlocuim cu o metoda care sa permita euristica white-box SBST. Versiuniea instrumentata este:

```
IL_003a: ldarg.1            // a
IL_003b: ldarg.2            // b
IL_003c: ldstr              "System.Boolean System.String::Equals(System.String)"
IL_0041: ldstr              "MyClass"
IL_0046: ldc.i4.s           14  //line number
IL_0048: ldc.i4.0           //branchcounter
IL_0049: call       bool [EvoMaster.Instrumentation]
   EvoMaster.Instrumentation.Probes::StringCompare(string, string, string, string, int32, int32)
```

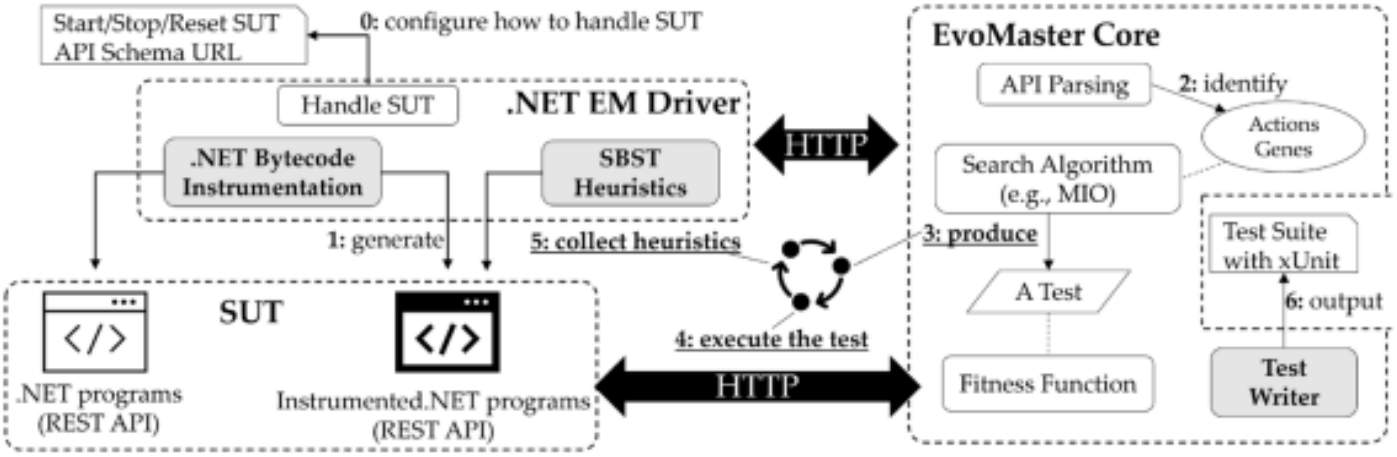
```
public static bool Equals(string caller, string  anotherString,
StringComparison comparisonType, string  idTemplate) {
   ObjectExtensions.RequireNonNull<string>(caller);
   ExecutionTracer.HandleTaintForStringEquals(caller, anotherString, comparisonType);
   bool result = caller.Equals(anotherString, comparisonType);
   if (idTemplate == null) return result;
   if (anotherString == null) {
      ExecutionTracer.ExecutedReplacedMethod(idTemplate,
      ReplacementType.BOOLEAN, new Truthness(DistanceHelper.H_REACHED_BUT_NULL,
      1));
      return false;
   }
   Truthness t;
   if (result) { t = new Truthness(id, DistanceHelper.H_NOT_NULL);}
      else {
      double baseS = DistanceHelper.H_NOT_NULL;
      double distance = DistanceHelper.GetLeftAlignmentDistance(caller, anotherString,
   comparisonType);
      double h = DistanceHelper.HeuristicFromScaledDistanceWithBase(baseS, distance);
      t = new Truthness(h, id);
   }
   ExecutionTracer.ExecutedReplacedMethod(idTemplate, ReplacementType.BOOLEAN, t);
   return result;
}
```

Precum versiunea instrumentata, metoda Equals este inlocuita de apelul unei metode(precum StringCompare) care calculeaza distanta branch, face comparatia si introduce rezultatul in stiva de evaluare. Mai sunt alte patru instructiuni CIL intre apelul metodei si instructiunile folosite pentru operatia de comparare. Folosindu-ne de cele patru argumente, definim un unic target ID ca idTemplate pentru comparatia aceasta. Avand in vedere operandul de comparatie pe stringuri, acum avem “==”, Equals, Contains, StartsWith si EndsWith. Implementarea lui Equals este iin exemplul de mai sus. Metoda are caller, anotherString si comparisonType ca input. Operatorul de comparatie ID aduce doua

noi tinte de testare, precum true_branch si flase_branch pentru apelul metodei. Trebuie evidentiat ca pe langa true_branch, consideram si false_branch ca tinta de testare pentru a oferi indrumare mai buna cautarii. Mai intai, metoda de inlocuire se asugura ca caller nu este null. Altfel, arunca NullReferenceException.

Apoi, la linia 4, prezinta aceeasi semantica precum metoda originala Equals. Privind distanta, folosim Truthness care este o clasa de utilitate ce stocheaza doua valori euristice dintre 0 si 1, reprezentand rezultatele posibile. Unu reprezinta ca rezultatul este acoperit. Dupa cum se observa la linia 6, anotherString este null, asa ca setam ofTrue ca fiind o valoare constanta reprezentand cazul Null si setam ofFalse cu 1. Observam ca in loc sa atribuim valoarea 0 lui ofTrue, folosim valori pozitive mici pentru a indica cautarii ca branchul este “reached but far from being covered”.

Liniile 11-17 sunt pentru a acoperi sitiarii cand result este true sau false. Cu privire la true, distanta este data ca fiind 1 pentru ofTrue si o valoare constanta reprezentand cazul NotNull pentru ofFalse. Daca rezultate este false, liniile 13-15 calculeaza distanta h pentru a reprezenta cat de aproape este de adevar. Pentru compararea stringurilor, folosim aceeasi distanta ca EvoSuite si apoi, scalam valoarea cu H_NOT_NULL. Acum, putem calcula valoarea euristica a true_branch su false_branch, apoi le actualizam la linia 18. Aceste euristici vor fi utilizate mai departe de catre cautarea pentru a evalua un test.



4.4 Integrarea cu EvoMaster for testarea .NET

EvoMaster este o unealta open-source destinata generarii automate a testelor de sistem cu SBST (Search-Based Software Testing), ce sustine atat testarea white-box, cat si cea black-box. Foloseste algoritmul evolutiv Many Independent Objective (MIO) pentru a rezolva problema testarii sistemelor white-box si pentru generarea automata a suitelor de testare.

In contextul testarii programelor .NET, aceasta tehnica este integrata in EvoMaster printr-un driver .NET EM ce contine Instrumentatia Bytecode .NET si Euristicile SBST, iar nucleul este extins cu un Test Writer ce genereaza suite de teste scrise in C# (folosind xUnit). Acest Test Writer poate fi utilizat si de EvoMaster in aplicarea testarii black-box.

EvoMaster este compus din doua parti principale: driver si nucleu. La nivelul driver-ului, utilizatorii trebuie sa specifice manual modul de gestionare a SUT (System Under Test) si sa furnizeze un URL pentru accesarea schemei API-ului. Apoi, cu Instrumentatia Bytecode .NET, se genereaza automat un SUT instrumentat, unde este injectat un set de sonde. Din partea nucleului, Parsarea API-ului analizeaza schema API-ului pentru a identifica punctele finale disponibile si datele ce pot fi manipulate. Apoi, cautarea produce un test cu un operator de cautare aplicabil, iar Functia de Fitness executa testul pe SUT si colecteaza informatii despre obiectivele atinse de aceasta executie.

Informatiile obtinute despre acoperirea in timpul executiei permit Functiei de Fitness sa evalueze un test; de exemplu, cu euristicile white-box, se poate determina ca x=50 este mai bun decat x=100 pentru acoperirea ramurii if(x==42).

5. Studiu emaapiric

5.1. Intrebarile cercetarii:

Pentru a evalua tehnica noastră, am efectuat un studiu empiric pentru a răspunde la următoarele întrebări de cercetare:

RQ1: Aceasta abordare permite euristici SBST white-box eficiente pentru a ghida căutarea pentru fuzzing în API-urile RESTful .NET/C#?

RQ2: Ce tipuri de constrângeri pot fi rezolvate? Și care dintre ele nu pot?

RQ3: Care este impactul aplicării timpului ca criteriu de oprire asupra performanței abordării?

5.2. Setup-ul experimentului:

Noua noastră tehnică permite utilizarea euristicilor SBST cutie albă pentru testarea programelor .NET. Pentru a evalua această abordare, am integrat-o în EVOMASTER.NET (cunoscută și sub numele de EVOMASTER.NET discutată în secțiunea 4.4) și am efectuat experimente pe trei API-uri .NET REST, respectiv C# REST Numeric Studiu de caz (cs-rest-ncs), C # REST String Case Study (cs-rest-scs) și Menu API (menu-api). Primele două studii de caz au fost inițial concepute pentru a investiga abordările de testare a unităților în rezolvarea problemelor numerice și de șiruri de caractere. Acestea două au fost re-implementate ca API-uri RESTful în diferite limbaje de programare (cum ar fi Java și JavaScript) pentru a evalua problemele de generare a testelor cutiei albe.

Aici, le-am re-implementat cu C# și le-am făcut accesibile prin intermediul unei API REST. Ultimul studiu de caz, adică menu-api, este unul dintre serviciile de backend ale aplicației populare Restaurant-App, care este un proiect open-source existent pe GitHub[6]. În contrast cu celelalte două studii de caz, menu-api gestionează o bază de date PostgreSQL. Tabelul 3 prezintă statisticile pentru aceste API-uri. Pentru a ușura replicarea acestui studiu, toate aceste API-uri sunt incluse în depozitul EMB (EvoMaster Benchmark (EMB), 2022[7]).

Tabelul 3: Statistici privind API-urile utilizate,
inclusiv numărul de linii de cod (#LOCs)
și numarul de puncte terminale
REST (#Endpoints)

În acest studiu, API-urile cs-rest-ncs și cs-rest-scs au fost alese pentru a ne asigura că heuristica noastră SBST pentru .NET funcționează corect pentru constrângerile numerice și de șiruri de caractere. Deoarece rezultatele anterioare pentru JVM arată rezultate bune pentru tehnicile SBST pe aceste API-uri ar trebui să ne așteptăm la aceleași rezultate pozitive pentru .NET dacă tehniciile noastre funcționează conform intenției. Pentru a arăta aplicarea tehnicilor noastre la API-urile reale RESTful, am căutat pe GitHub pentru API-uri .NET RESTful, acordând prioritate în funcție de popularitate (reprezentată prin numărul de stele). Din păcate, deși C#/.NET este foarte popular în industrie, din motive istorice (de exemplu, datorită instrumentelor aproape sursă și legăturilor cu platforma Windows), este mai puțin popular printre proiectele open-source. Găsirea API-urilor potrivite printre proiectele open-source s-a dovedit a fi destul de provocatoare. API-ul menu-api a fost primul pe care l-am găsit care îndeplinește criteriile noastre.

După cum am discutat în secțiunea 3, în cea mai mare parte, în cadrul cunoștințelor noastre, nu există nicio tehnică SBST existentă pentru testarea cutiei albe a programelor .NET. Astfel, pentru a evalua eficiența abordării noastre, am efectuat o comparație între doi algoritmi dezvoltați în EVOMASTER, adică MIO și random, în ceea ce privește performanțele obținute prin testele generate. MIO este algoritmul implicit folosit cu heuristica SBST în EVOMASTER pentru generația de teste, în timp ce random este doar o căutare aleatorie naivă, utilizată ca bază. Rețineți că algoritmul aleatoriu ar putea fi considerat ca un test de cutie gri, deoarece încă urmărește testele pe baza a ceea ce obiective sunt acoperite (de exemplu, linia de acoperire) și produce cele mai bune dintre ele la sfârșit. Nu comparăm cu alte fuzzer-uri cutie neagră din acest articol, deoarece EVOMASTER oferă deja cele mai bune rezultate în comparațiile de instrumente existente.

În contextul testării software-ului, am utilizat patru criterii pentru a evalua performanța tehnicilor, și anume, acoperirea țintelor , acoperirea liniilor (%Liniile), acoperirea ramurilor (%Franșe) și defectele detectate (# Defecte). #Obiective este criteriul agregat care ia în considerare toate metricile de acoperire pentru care EVOMASTER optimizează . Cu instrumentele noastre, permitem acoperirea claselor, a liniilor, a declarațiilor și a ramurilor ca părți ale #Obiectivelor pentru a fi optimizate. În ceea ce privește %Liniile și %Franșe, acestea sunt măsurători aplicate pe scară largă pentru a evalua abordările de testare în practică. Defectele sunt detectate pe baza codului de stare HTTP returnat 500 și pe neconcordanțe în răspunsurile bazate pe schemele API.

SUT	Maaetrics	MIO	Random	$\hat{A}_{12}$	p-value	Relative
cs-rest-ncs	#aTargets	**992.1**	656.4	**1.00**	**≤0.001**	**+51.14%**
	%Lines	**85.5%**	55.4%	**1.00**	**≤0.001**	**+54.36%**
	%Branches	**76.4%**	51.6%	**1.00**	**≤0.001**	**+47.93%**
	#Faults	**6.0**	5.0	**1.00**	**≤0.001**	**+20.00%**
cs-rest-scs	#Targets	**967.8**	617.6	**1.00**	**≤0.001**	**+56.71%**
	%Lines	**73.6%**	57.4%	**1.00**	**≤0.001**	**+28.03%**
	%Branches	**32.5%**	25.9%	**1.00**	**≤0.001**	**+25.33%**
	#Faults	**1.0**	**1.0**	0.50	NaN	+0.00%
menu-api	#Targets	333.5	**333.7**	0.43	0.588	–0.06%
	%Lines	**29.1%**	**29.1%**	0.50	NaN	+0.00%
	%Branches	**1.8%**	1.7%	0.55	0.651	+2.70%
	#Faults	22.7	**23.0**	0.35	0.077	–1.30%

Tabelul 4: Rezultatele medii și comparațiile pairwise pentru MIO și random cu patru metrici, adică #Obiective, %Liniile, %Ramuri și #Defecte.

În ceea ce privește setările de parametri, în primul set de experimente, bugetul de căutare pentru cei doi algoritmi a fost alocat ca 100.000 de apeluri HTTP, o setare obișnuită în cercetările existente utilizând EVOMASTER. În al doilea set de experimente, am folosit 1 oră ca criteriu de oprire. Pentru ceilalți parametri (de exemplu, F și Pr), am aplicat valorile implicite ca în EVOMASTER. Având în vedere randomizarea moștenită de la algoritmii de căutare, am repetat experimentele noastre de 10 ori pentru MIO și algoritmul aleatoriu pe cele trei studii de caz, urmând liniile directoare comune din literatură. Experimentul a fost efectuat pe un laptop DELL cu următoarele specificații: Procesor Gen 11 Intel(R) Core(TM) i9-11950 H @2.60GHz 2.61 GHz; RAM 32 GB; Sistem de operare Windows 10 pe 64 de biți.

5.3. Rezultatele experimentului

5.3.1. Rezultate pentru RQ1:

Pentru a răspunde la RQ1, Tabelul 4 prezintă rezultatele medii de acoperire obținute de către MIO și random pentru #Obiective, %Liniile, %Ramuri și #Defecte. Bazându-ne pe rezultatele medii, MIO obține în mod constant cele mai bune rezultate atât pentru problemele numerice, cât și pentru cele de șiruri de caractere, pe toate cele patru metrici. În tabel, am efectuat și o analiză statistică pentru a compara MIO și random cu cele patru metrici folosind teste U Mann–Whitney-Wilcoxon (valoarea p) și dimensiuni ale efectului Vargha-Delaney (Â 12). În tabel, valoarea în bold indică faptul că MIO este semnificativ mai bun, adică p-value < 0,05 (nivel de semnificație) și Â 12 > 0,5. Luând în considerare rezultatele prezentate în Tabelul 4 pentru cs-rest-ncs și cs-rest-scs, MIO depășește semnificativ random

pentru toate metricile, cu efecte mari și valori mici ale p-ului pentru studiile de caz. Pentru menu-api, nu există o diferență semnificativă între cele două algoritme.

Figura 3 prezintă linii de grafic care arată schimbările în numărul de obiective acoperite obținute de MIO și random pe parcursul căutării, colectate la diferite intervale de timp. Pentru cs-rest-ncs și cs-rest-scs, MIO prezintă un avantaj clar față de random pe întreaga căutare. Acest lucru demonstrează în continuare eficacitatea euristicii SBST cutie albă pentru a rezolva problemele numerice și de șiruri de caractere.

În ceea ce privește numărul de erori găsite (adică, #Faults), MIO a reușit să găsească încă unul în comparație cu aleatorii pentru cs-rest-ncs. Cu toate acestea, nu se obține nicio îmbunătățire pentru cs-rest-scs și menu-api. Acest lucru este de înțeles pentru menu-api, deoarece MIO nu a avut nici o vârstă de acoperire mai mare, ceea ce reduce probabilitatea de a detecta mai multe defecte.

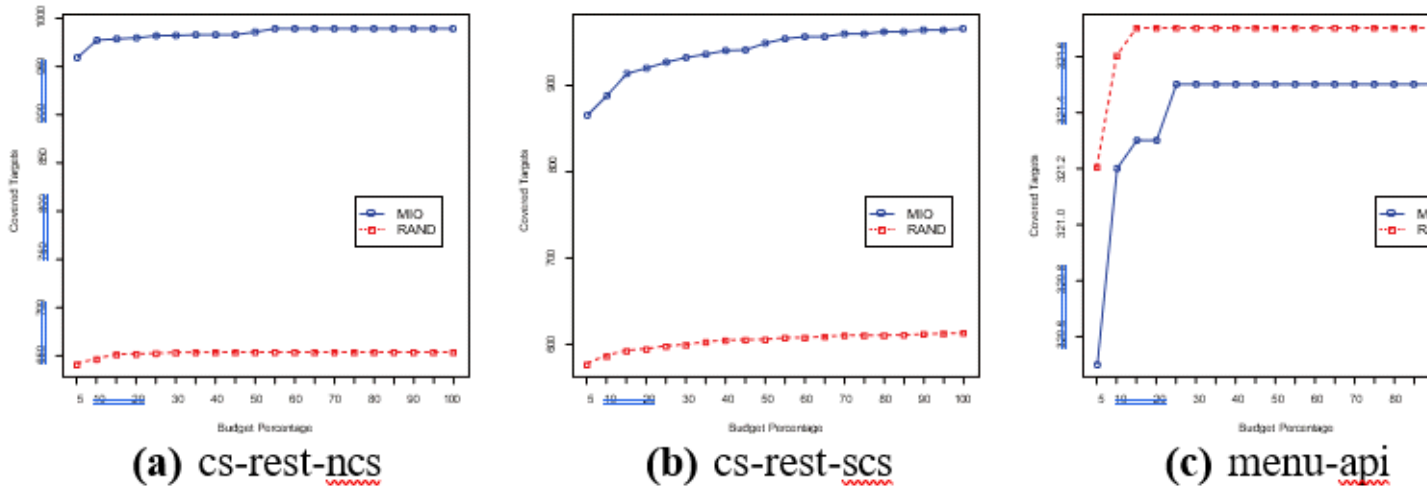


Fig. 3 Obiectivele medii acoperite (axa y) atinse de MIO (albastru) și aleator (roșu) pe tot parcursul căutării (RQ1) la fiecare 5% dintre intervalele bugetului cheltuit de căutare (x-axis)

Cu toate acestea, nu observăm nicio îmbunătățire pentru cs-rest-scs și menu-api. Aceasta este înțeleasă în cazul menu-api, deoarece MIO nu a reușit să atingă o acoperire mai mare, ceea ce reduce probabilitatea de a detecta mai multe defecte. O acoperire mai largă ar putea conduce la detectarea unui număr mai mare de erori.

În ceea ce privește cs-rest-scs, deoarece este un sistem artificial de testare cu mai puțină flexibilitate în comparație cu menu-api, singurul defect detectat ar putea fi singurul defect potențial existent în cod. Figura 4 prezintă un caz de testare generat pentru menu-api care detectează o eroare internă a serverului. Această eroare apare atunci când clientul încearcă să acceseze un anumit element alimentar, dar codul de stare HTTP rezultat este 500, indicând o eroare pe partea serverului.

Concluzie pentru RQ1: Cu rezultatele obținute în două dintre studiile de caz, tehnicianul nostru de cutie albă a obținut o acoperire semnificativ mai mare a codului în comparație cu testarea aleatorie cu cutie gri în două din cele trei studii de caz. Acest lucru demonstrează eficacitatea SBST în ghidarea testării cutii albe a programelor numerice și string în.NET.

5.3.2 Rezultate pentru RQ2:

Pe baza rezultatelor din Tabelul 4, MIO a obținut o acoperire a liniilor de 85,5% pe CS-rest-ncs, 73,6% pe CS-rest-scs și 29,1% pe meniu-api, în medie, cu 10 repetiții. Pentru a analiza performanța în detaliu, s-a investigat în continuare acoperirea codului,executând cele mai bune și cele mai rele teste pe SUT-uri folosind JetBrains Rider[8]. Rețineți că acoperirea liniilor raportată de instrumentele EVOMASTER exclude acoperirea obținută la pornire. Prin urmare, acoperirea raportată de Rider ar fi mai mare decât cea din Tabelul 4.

```
[Fact]
public async Task text_16_with500(){
 Client.DefaultRequestHeaders.Clear();
 Client.DefaultRequestHeaders.Add("Accept", "*/*");
 var res_0 = await Client.GetAsync(_fixture.baseUrlOfSut + "/api/v1/Foods/Ohc");
 Assert.Equals(500, (int) res_0.StatusCode); // Statement_FoodRepository_00031_4
 Assert.True(string.IsNullOrEmpty(await res0.Content.ReadAsStringAsync()));
}
```

Fig. 4 Un exemplu de caz de testare generat care detectează o eroare potențială bazată pe codul de stare HTTP 500

```
[Fact]
public async Task text_17() {
 Client.DefaultRequestHeaders.Clear();
 Client.DefaultRequestHeaders.Add("Accept", "*/*");
 var res_0 = await Client.GetAsync(_fixture.baseUrlOfSut + "/api/triangle/653/653/653");
 Assert.Equals(200, (int) res_0.StatusCode);
 Assert.Contains("application/json", res0.Content.Headers.GetValues("Content-Type").First());
 dynamic body_1 = JsonConvert.DeserializeObject(await res0.Content.ReadAsStringAsync())
 Assert.True(body_1.result == "3");
}
```

Fig. 5 Exemplu de test generat pentru cs-rest-ncs

Figura 5 prezintă codul unui test generat pentru cs-rest-ncs. În acest test, se efectuează un apel asincron HTTP către punctul final /api/triangle/653/653/653. Apoi, testul verifică că codul de stare al solicitării HTTP este 200, corpul răspunsului este de tip JSON, și în cele din urmă, răspunsul JSON conține un câmp numit "rezultat" cu o valoare egală cu 3. Este important să remarcați că, atunci când este generată o suită de teste care conține mai multe teste, este generat și codul de rulare, similar cu cel prezentat în Figura 6. Aici, API-ul este pornit, resetat la fiecare execuție a testului, și închis odată ce toate testele sunt finalizate (folosind clasa de driver direct, cum ar fi NcsDriver.EmbeddedEvoMasterController() în acest caz). Această funcționalitate este esențială pentru a putea utiliza aceste tipuri de teste pentru teste de regresie.

```
public class ControllerFixture : IDisposable {
```

```
 public ISutHandler controller {get; private set;}
 public string baseUrlOfSut {get; private set;}
 public ControllerFixture(){
    controller = new NcsDriver.EmbeddedEvoMasterController();
    controller.SetupForGeneratedTest();
    baseUrlOfSut = controller.StartSut();
    Assert.NotNull(baseUrlOfSut);
 }
 public void Dispose(){
  controller.StopSut();
 }
}

public class EM_MI0_1_Test : IClassFixture <ControllerFixture>{
   private ControllerFixture _fixture;
   private static readonly HttpClient Client = new
     HttpClient();
   public ControllerFixture _fixture;
   private static readonly HttpClient CLient = new HttpClient();
   public EM_MI0_1_Test (ControllerFixture fixture){
     _fixture = fixture;
     _fixture.controller.ResetStateOfSut();
   }
 }
```

Fig. 6 Exemplu de scafandru de testare generat

În ceea ce privește cs-rest-ncs, s-a observat că majoritatea ramurilor numerice au fost acoperite, cu acoperiri între 93% și 98% pentru spațiul de nume IMP pe Fisher, Remainder, Triangle, Bessj, Expint și Gammq. Cu toate acestea, s-a constatat că unele ramuri nu au fost acoperite din cauza codului mort, cum ar fi verificările preliminare care împiedică accesul la anumite metode. În consecință, cs-rest-ncs poate fi considerată o problemă rezolvată.

În ceea ce privește cs-rest-scs, acoperirea codului pe spațiul de nume SCS.Imp variază între 72% și 86%. În general, MIO obține peste 73% acoperire pentru mai multe module precum Costfuns, DateParse, NotyPevear, Ordered4, Title, Text2Txt, Calc, Cookie, Regex și Pat. Cu toate acestea, există diferențe semnificative între cel mai rău și cel mai bun rezultat, de exemplu, modulul Ordered4 are o acoperire de doar 33% în cel mai rău caz, comparativ cu 100% în cel mai bun caz. Aceste diferențe sunt legate de metodele care întâmpină dificultăți în a fi acoperite din cauza lipsei de heuristici adecvate sau a necesității unui buget de căutare mai mare. Pentru a aborda aceste probleme, pot fi necesare mai multe investigații și ajustări ale strategiei de testare.

În plus, am observat că atât în cel mai rău, cât și în cel mai bun caz, acoperirea liniilor este de 73% pentru modulul Regex și de doar 16% pentru FileSuffix. Pentru a acoperi anumite ramuri, cum ar fi cele legate de Regex, poate fi necesară implementarea unor transformări de testare specifice. Pentru FileSuffix, care nu a prezentat o acoperire semnificativă, trebuie să explorăm strategii suplimentare pentru a îmbunătăți testarea.

```
var fileParts = file.Split(".");
var lastPart = fileParts.Length - 1;
if (lastPart <= 0) return "" + result;
```

Ținta ramurii ar putea fi rezolvată prin verificarea prezenței cel puțin a unui punct în șirul fișierului. Cu toate acestea, în prezent, activăm doar înlocuirea metodelor String care sunt legate de predicate boolean, conform discuției din Secțiunea 4.3.2. Pentru a aborda eficient astfel de separări de șiruri, sunt necesare metode de analiză și înlocuire care să sprijine metodele, cum ar fi Split, care încă nu au fost tratate. Astfel, fără o manipulare suplimentară a acestor metode cu ajutorul unei euristici de cutie albă, ramurile conexe nu pot fi ușor rezolvate în cadrul bugetului dat de 100k apeluri HTTP. Cu toate acestea, aceste metode ar putea fi susținute în continuare, de exemplu, prin includerea separatorului de șiruri Split ca parte a analizei noastre de contaminare.

În ceea ce privește menu-api, nu există diferențe între cel mai rău și cel mai bun caz. Ambele au raportat o acoperire a liniilor de 67% în spațiul de nume Menu.API. În plus, rezultatele nu au arătat îmbunătățiri semnificative în niciuna dintre metricele furnizate de MIO. Acest lucru se datorează în principal lipsei de suport pentru baze de date în instrumentele noastre. Pentru testarea cutiei albe în JVM, EVOMASTER suportă prelucrarea SQL, care poate calcula heuristici pentru interogările SQL și poate insera date direct în baza de date. Cu toate acestea, pentru acest studiu de replicare, încă nu am implementat o astfel de tehnică în instrumentația noastră de bytecode, deoarece necesită un efort complex de inginerie. Acest aspect este unul care trebuie îmbunătățit în viitor. Pe baza acestei analize, putem concluziona următoarele:

SUT	Metrics	MIO	Random	$\hat{A}_{12}$	p-value	Relative
cs-rest-ncs	#Targets	**991.8**	656.6	**1.00**	**≤0.001**	**+51.05%**
	%Lines	**85.8%**	55.5%	**1.00**	**≤0.001**	**+54.63%**
	%Branches	**76.1%**	51.5%	**1.00**	**≤0.001**	**+47.69%**
	#Faults	**6.0**	5.0	**1.00**	**≤0.001**	**+20.00%**
cs-rest-scs	#Targets	**916.2**	661.0	**1.00**	**≤0.001**	**+38.61%**
	%Lines	**70.8%**	60.7%	**1.00**	**≤0.001**	**+16.72%**
	%Branches	**32.0%**	27.7%	**0.95**	**≤0.001**	**+15.81%**
	#Faults	**1.0**	**1.0**	0.50	NaN	+0.00%
menu-api	#Targets	**333.7**	333.6	0.55	0.681	+0.03%
	%Lines	**29.1%**	**29.1%**	0.50	NaN	+0.00%
	%Branches	**1.7%**	1.7%	0.55	0.681	+2.78%
	#Faults	**23.0**	**23.0**	0.50	NaN	+0.00%

Tabelul 5: Rezultate de comparaţie medie şi în perechi a cursului cu bugetul de timp de 1 oră pentru MIO şi aleatoare cu patru metrici, adica #Targets, %Lines, %Branches and #Faults.

Concluzie pentru RQ2: Cu o analiză suplimentară a acoperirii codului în detaliu, am constatat că heuristicile noastre SBST cu cutie albă sunt capabile să fuzz.NET/C# API REST, adică acoperirea liniilor obținute prin testele generate este între 67 și 98%. Cu toate acestea, identificăm, de asemenea, unele limitări din cauza lipsei de manipulare a bazei de date, care pot fi abordate în lucrările viitoare.

5.3.3. Rezultate pentru RQ3

Pentru a răspunde la primele două întrebări de cercetare, evaluarea a folosit criteriul de oprire de 100.000 de acțiuni evaluate (apeluri HTTP), deoarece fiecare caz de testare putea avea un număr diferit de acțiuni, iar numărul de evaluări de adecvare nu ar fi fost o metrică echitabilă pentru comparații. Cu un criteriu de oprire de maxim 100.000 de apeluri HTTP, timpul mediu de execuție pe test a fost de 23 de minute. Totuși, timpul petrecut pentru diferite studii de caz sau configurații ar putea varia.

Pentru a investiga impactul diferitelor criterii de oprire, cum ar fi numărul maxim de apeluri HTTP versus bugetul de timp, s-a efectuat un alt experiment cu un buget de timp de 1 oră ca criteriu de oprire. Tabelul 5 conține rezultatele acestor noi experimente.

În general, rezultatele nu au arătat nicio diferență semnificativă față de cele din tabelul 4. Aceasta sugerează că utilizarea timpului ca criteriu de oprire nu a avut un impact semnificativ asupra rezultatelor.

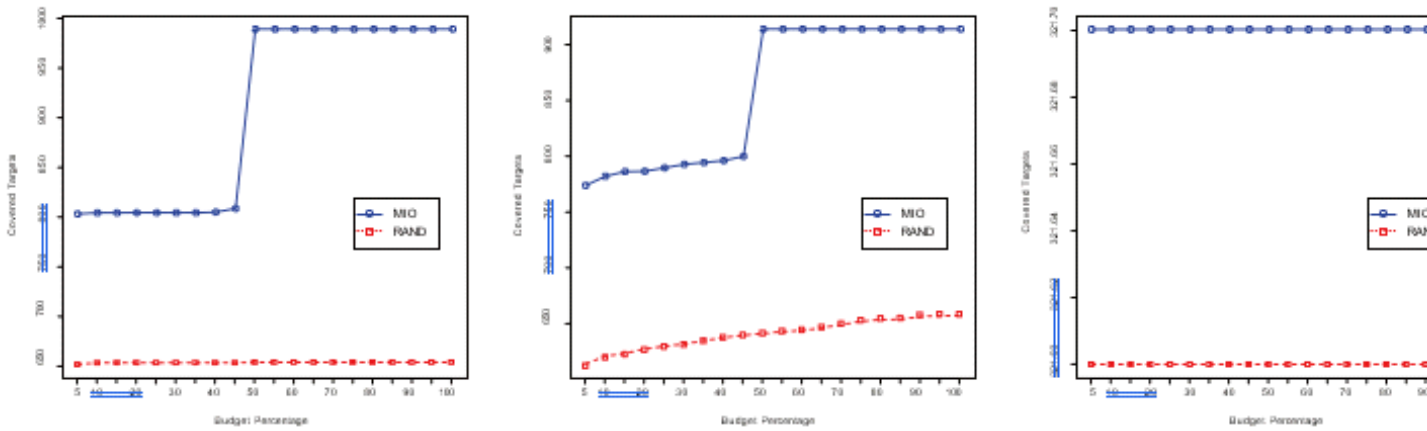


Fig. 7 Obiectivele medii acoperite (axa y) atinse de MIO (albastru) și aleator (roșu) pe parcursul căutării (RQ3) la fiecare 5% intervale din bugetul de timp al 1-lea cheltuit de căutare (x-axis)

În plus, Figura 7 ilustrează modificările în numărul de obiective acoperite atinse de MIO și algoritmul aleator pe durata experimentului cu bugetul de timp de 1 oră. Similar cu Figura 3, MIO a obținut rezultate superioare în comparație cu algoritmii aleatori atunci când a fost aplicat pe cs-rest-ncs și cs-rest-scs. Cu toate acestea, nu s-au observat îmbunătățiri semnificative prin aplicarea MIO pe menu-api, deoarece implică manipularea bazelor de date, care în prezent nu sunt susținute în heuristica noastră de instrumentare.

Concluzie pentru RQ3: Am aplicat bugetul de timp de 1 oră ca criteriu de oprire pentru algoritmii de căutare, însă rezultatele nu au indicat nicio diferență semnificativă între MIO și algoritmii aleatori.

6. Amenințări la adresa validității

Valabiliatatea concluziei:

1) Am repetat experimentul de 10 ori pentru a evita rezultatele obținute la întâmplare, luând în considerare randomizarea algoritmului de căutare.

2) Am folosit metode de analiză statistică, cum ar fi Mann-Whitney-Wilcoxon U-teste și dimensiunile efectului Vargha-Delaney, pentru a trage concluzii.

Validitate internă:

1) Implementarea și studiile de caz sunt disponibile online, permitând oricui să revizuiască și să reproducă studiul.

2) Cu toate acestea, este dificil să garantăm absența bug-urilor în implementare.

Validitate externă:

1) Acest studiu a fost realizat cu două API-uri.NET REST artificiale și unul open-source.

2) Pentru generalizarea rezultatelor, ar fi necesar să se includă mai multe studii de caz, dar este dificil să găsim proiecte open-source relevante în contextul testării API-urilor REST.

3) Rezultatele evidențiază necesitatea gestionării bazelor de date SQL înainte de a extinde acest tip de studiu de caz, dar acest lucru implică un efort major de inginerie și cercetare.

4) Tehnicile noastre ar putea fi aplicate și în alte contexte, cum ar fi generarea unităților de testare, dar fără validare empirică, eficacitatea lor în aceste contexte nu este garantată.

7. Exemple - Exemple de algoritmi White Box creati de noi

Exemplul 1

Am creat un exemplu de testare de tipul White Box pentru o clasa care gestioneaza operatii matematice intr-un proiect .NET. Am folosit framework-ul de testare NUnit pentru a crea si executa testele.

Presupunem ca avem o clasa numita Calculator care contine mai multe metode matematice, cum ar fi adunare, scadere, inmultire si impartire. Vom crea teste pentru fiecare dintre aceste metode.

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
    public int Subtract(int a, int b)
    {
        return a - b;
    }
    public int Multiply(int a, int b)
    {
        return a * b;
    }
    public int Divide(int a, int b)
    {
        if (b == 0)
        throw new ArgumentException("Nu se poate divide cu 0");
        return a / b;
    }
}
```

```
using NUnit.Framework;
using System;
```

```
[TestFixture]
public class CalculatorTests
{
    [Test]
    public void TestAdd()
    {
        Calculator calculator = new Calculator();
        int result = calculator.Add(5, 3);
        Assert.AreEqual(8, result);
    }
    [Test]
    public void TestSubtract()
    {
        Calculator calculator =  new Calculator();
        int result = calculator.Subtract(10,4);
        Assert.AreEqual(6, result);
    }
    [Test]
    public void TestMultiply()
    {
        Calculator calculator = new Calculator();
        int result = calculator.Multiply(7,3);
        Assert.AreEqual(21, result);
    }
    [Test]
    public void TestDivide()
    {
        Calculator calculator = new Calculator();
        int result = calculator.Divide(15, 3);
        Asser.AreEqual(5, result);
    }
    [Test]
    public void TestDivide_ByZero()
    {
        Calculator calculator = new Calculator();
        Assert.Throws<ArgumentException>(() => calculator.Divide(10, 0));
    }
}
```

```
> dotnet test --collect:"XPlat Code Coverage" -- DataCollectionRunSettings.DataCollectors.DataCollector.Configuration.F
```

Pentru a evalua acoperirea codului, am folosit Coverlet pentru a genera un raport. Coverlet este integrat in proiectul nostru folosind CLI (ne puteam folosi si de pachetele NuGet). Coverlet genereaza un raport de acoperire a codului. Acest raport va indica ce parti ale codului au fost testate si cat de bine.

Exemplul 2

Cerinte:

- Utilizatorii pot introduce o adresa de email si o parola pentru a se autentifica.
- Parola trebuie sa aiba cel putin 8 caractere si sa contina cel putin o litera majuscula, o litera minuscula si un numar.
- Adresa de email trebuie sa fie valida conform formatului specific unei adrese de email.
- Utilizatorii pot avea diferite roluri (de exemplu, "Utilizator" sau "Administrator").

```
public class AuthenticationService
{

   public bool Authenticate(string email, string password)
   {
      if (!IsValidEmail(email))
         return false;
      if (!IsValidPassword(password))
         return false;
        //Verifica autentificarea in baza de date
      //returneaza true daca autentificarea este reusita, altfel false
      return true;
   }
   private bool IsValidEmail(string email)
   {
      return email.Containe("@") && email.Contains(".");
   }
   private bool IsValidPassword(string password)
   {
      // Verifica daca parola respecta cerintele de complexitate
      return password.Length >= 8 &&
            password.Any(char.IsUpper) &&
            password.Any(char.IsLower) &&
            password.Any(char.IsDigit);
   }
}
```

```
using NUnit.Framework
[TestFixture]
public class AuthenticationServiceTests
{
   [Test]
   public void TestAuthenticate_ValidCredentials_ReturnsTrue()
   {
      AuthenticationService authService = new AuthenticationService();
```

```
    bool result = authService.Authenticate("test@example.com","Password123");
    Assert.IsTrue(result);
  }
  [Test]
  public void TestAuthenticate_InvalidEmail_ReturnsFalse()
  {
    AuthenticationService authService = new AuthenticationService();
    bool result = authService.Authenticate("invalid email", "Password123")
    Asser.IsFalse(result);
  }
  [Test]
  public void TestAuthenticate_InvalidPassword_ReturnsFalse()
  {
    AuthenticationService authService = new AuthenticationService();
    bool result = authService.Authenticate("test@example.com","password")
    Assert.IsFalse(result);
  }
  [Test]
  public void TestAuthenticate_ValidCredentials_AdminRole_ReturnsTrue()
  {
    AuthenticationService authService = new AuthenticationService();
    bool result = authService.Authenticate("admin@example.com", "Admin123");
    Assert.IsTrue(result);
  }
}
```

8. Concluzie

.NET/C# este una dintre cele mai populare limbaje de programare utilizare in special in industrii pentru construirea aplicatiilor cloud-based si internet-connected. Cu toate acestea, nu exista o tehnica SBST in literatura pentru testarea automata white-box a programelor .NET/C#.
In acest articol, s-a creat o instrumentatie .NET bytecode pentru a aplica existenta white-box SBST euristici bazate pe ramura distantei. Cu asemenea tehnici, se poate activa colectia runtime coverage ceea ce ofera indicatii eficiente in cautarea testarii aplicatiilor C#, reproducand SBST success stories existente pentru limbajele de programare Java si JavaScript. Doua din trei experimente .NET RESTful APIs arata ca prin integrarea novel techniques ca extensie a uneltei open-source EvoMaster, au adus o performanta mai buna decat tehnica de testare gray-box random. Insa, avand in vedere rezultatele ale uneia dintre cazurile studiate care se ocupa cu baza de date, metoda de integrare nu aduce performante mai bune decat testarea random. Pe langa acestea, in urma unei analizari aprofundate pe acoperirea codului realizat de testele generate, s-a observat ca metoda folosita este destul de eficienta in rezolvarea ramurilor numerice si de tip string. Aceasta acopera intre 67% (minim) si 98% (maxim), printre cele 10 petitii pe doua dintre cazurile studiate.

In teorie, orice aplicatie care este convertita in cod CIL poate folosi componenta instrumentatiei. Fiind bazat pe EvoMaster, care depinde de OpenAPI schema, s-a propus o abordare empiric testata pentru REST APIs. Totusi, este dificila determinarea performantei pe alte tipuri de aplicatii fara

informatie empirica adecvata. Solutia pentru aceasta nu avanseaza in mod direct testarea black-box; in schimb, se concentreaza pe testarea white-box.

Aceasta este prima lucrare din literatura privind aplicarea SBST de tip white-box pentru aplicatiile .NET/C#. Pentru a scala aceste tehnici la sisteme industriale mari, este necesar sa se faca mai mult (de exemplu, suport eficient pentru baze de date). Un alt posibil viitor demers poate fi imbunatatirea eficacitatii generarii intrarilor de test prin utilizarea tehnicilor de invatare automata pentru deducerea relatiilor potentiale intre actiunile si parametrii REST APIs. Cu toate acestea, aceasta lucrare ofera primii pasi stiintifici initiali importanti in aceasta directie.

Referinte

[1] The State of the Octoverse. https://octoverse.github.com/. Online, Accessed 3 August 2023.
[2] Harman, M., Mansouri, S. A., & Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR), 45(1), 11.
[3] .NET Platform. https://github.com/dotnet. Online, Accessed 3 August 2023.
[4] ECMA-335. Common Language Infrastructure (CLI). https://www.ecma-international.org/publicationsand-standards/standards/ecma- HYPERLINK "https://ecma-international.org/publications-and-standards/standards/ecma-335/"335 HYPERLINK "https://ecma-international.org/publications-and-standards/standards/ecma-335/"/. Online, Accessed 3 August 2023.
[5] https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/
[6] https://github.com/chayxana/Restaurant-App
[7] EvoMaster Benchmark (EMB). https://github.com/EMResearch/EMB. Online, Accessed 20 May 2022.
[8] JetBrains Rider. https://www.jetbrains.com/rider. Online, Accessed 3 August 2023.