Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Szimbolikus végrehajtás alapú tesztgenerálás támogatása és analízise

TDK-dolgozat

Készítette:
Honfi Dávid

Konzulensek:
Dr. Micskei Zoltán
Vörös András

2014.

Tartalomjegyzék

1. fejezet

Bevezető és motiváció

A napjainkban fejlesztett és használatos szoftverrendszerek alkalmazási köre egyre többször megköveteli, hogy a rendszer hibamentes működést biztosítson. A fejlesztő mérnökök számára ennek garantálása nem minden esetben megoldható, ám az erre való törekvés is magas ellenőrzési, tesztelési költségeket von maga után. Ökölszabályként elmondható, hogy a ma alkalmazott fejlesztési folyamatok során a *költségek és fejlesztési idő legalább a felét a tesztelés teszi ki* [23], emiatt tehát a tesztelés kiemelt szerephez jut a szoftverfejlesztésben.

Ez a kiemelt szerep, illetve a gazdasági indokok is motiválják a tesztelés módszereinek és eszközeinek fejlődését, amely napjainkban is tart. A *tesztelés automatizálása* az egyik olyan irány, amelyet nagyszámú kutatás övez, melyeknek egyik kiemelkedő eredménye a *szimbolikus végrehajtás alapú szoftverellenőrzés*. A technika egyik elterjedt használati esete az egységtesztelés hatékonyságának növelése *tesztbemenet-generálással*, mellyel a fejlesztés korai szakaszaiban, így kisebb költséggel fedhetők fel a szoftverek hibái, hiányosságai.

A szimbolikus végrehajtás során a vizsgált program bemeneteire szimbolikus változók kerülnek, amelyek segítségével kinyerhetők az egyes lefutási ágakat aktiváló konkrét bemeneti értékek. A technikát több eszköz is implementálja, ugyanakkor kisebb-nagyobb eltérések tapasztalhatók közöttük mind mószerben, mind hatékonyságban. Az ilyen eszközöknek, így magának a szimbolikus végrehajtásnak is az egyik legnagyobb kihívása azonban a *mérnöki alkalmazhatóság* megvalósítása.

A szimbolikus végrehajtás széleskörű, ipari használatban való elterjedését gátolja, hogy alapvetően egy összetett algoritmusról van szó, amelynek komplex szoftvereken történő használatához részletesen ismerni kell a technika belső működését (például a könnyen előforduló elakadások miatt), melyet saját, korábbi tapasztalataim is alátámasztanak. Ipari területen a mérnökök megismertetése új technikákkal ugyanakkor időt és pénzt is felemészt, ami valójában ellentétes a technika bevezetésének céljával. Ennek okán érdemes lehet olyan *új módszereket* kidolgozni, melyek segítik a szimbolikus végrehajtás ipari alkalmazását és *áthidalják a komplex, nagy méretű szoftverrendszerek vizsgálatakor felmerülő problémákat.*

A fellépő nehézségek enyhítésére egy megoldást adhat a *szimbolikus végrehajtás vizualizációja*, ami segít a tesztelést végző mérnök számára áttekinteni az egyes futtatásokat. A mérnök így könnyebben felderítheti a problémákat, és eredményesebben alkalmazhatja a szimbolikus végrehajtást, amivel visszznyerheti előnyét a technika bevezetése azáltal, hogy gyorsabban talál hibákat, vagy ugyanannyi idő alatt, de nagyobb számban. Az eddig említettek mellett saját tapasztalataim is motivációként járultak hozzá a TDK munkám elkészítéséhez

Munkám során kidolgoztam egy vizualizációs módszert, ami a szimbolikus végrehajtás gráf alapú reprezentációját valósítja meg. A vizualizációs technika lehetőséget ad a végrehajtások részletes metaadatainak közlésére is, mellyel továbbnövelhető az áttekinthetőség.

A működés demonstrálására elkészítettem egy eszközt is, mely megvalósítja a kidolgozott vizualizációs technikát.

Jelen dolgozatom felépítése a fent leírtakat követi, amely így a következő. A 2. fejezetben a későbbiekhez szükséges háttérismereteket foglalom össze, bemutatva általánosan a szoftvertesztelést, majd specifikusabban az egységtesztelést, a szimbolikus végrehajtást, mint kapcsolódó technikát, végül pedig egy ezt alkalmazó eszközt, a Microsoft Pex-et. A 3. fejezetben röviden áttekintem az automatikus szoftvertesztelés kihívásait, leírva saját tapasztalataimat és a kapcsolódó irodalmakat, melyek együttesen megadják kutatásom főbb kérdéseit. A 4. fejezet során a kidolgozott vizualizációs technikát mutatom be részletesen, amelyet követő 5. fejezetben pedig a technikát implementáló eszköz részleteit ismertetem, illetve példákon és méréseken keresztül véleményezem a vizualizációs módszer és az eszköz hatékonyságát. A 6. fejezetben egy alkalmazási lehetőséget mutatok be a vizualizációs technikára, mely egy továbbfejlesztési irányt vetít előre. Végül, a 7. fejezetben röviden összefoglalom és értékelem a munkámat, majd bemutatom a lehetséges továbbfejlesztési irányokat.

2. fejezet

Háttérismeretek

A napjainkban fejlesztett szoftverek mérete, és ezáltal komplexitása is növekvő tendenciát mutat. Az összetettségből közel nyilvánvalóan adódik a fejlesztés során vétett hibák számának növekedése is. Többek között ez indokolja, hogy a jelenleg is elterjedten alkalmazott fejlesztési életciklus modellek kiemelt hangsúlyt fektetnek a tesztelésre, ellenőrzésre.

A fejezet első felében bemutatom a szoftvertesztelés alapfogalmait, technikáit, majd részletesebben áttekintem az egységtesztelést és annak szerepét a szoftverek minőségében. A fejezet második részében ismertetem a szimbolikus végrehajtás algoritmusát, mint tesztbemenet-generálási módszert, illetve bemutatom a Microsoft Pex eszközt, amely az említett algoritmus egy variánsát implementálja.

2.1. Szoftvertesztelés

Az alfejezet betekintést ad a szoftvertesztelés alapvető definícióiba, fogalmaiba, illetve az alkalmazott technikáiba.

2.1.1. Áttekintés

Dolgozatomban szoftverhiba *(defect)* alatt a szabványos terminológiának [1] megfelelően olyan rendellenességet értek, amely által a szoftver nem képes megfelelni a követelményeinek és specifikációjának. A hibák hatásláncának [19] megfelelően az emberi vétségből eredő hibák *(error)* miatt létrejövő programhibák *(fault)* és az abból eredő hibajelenségek *(failure)* a szoftverhibák mindössze egy részhalmazát alkotják, hiszen a rendellenességek közé sorolható például a hiányos vagy akár a pontatlan működés is.

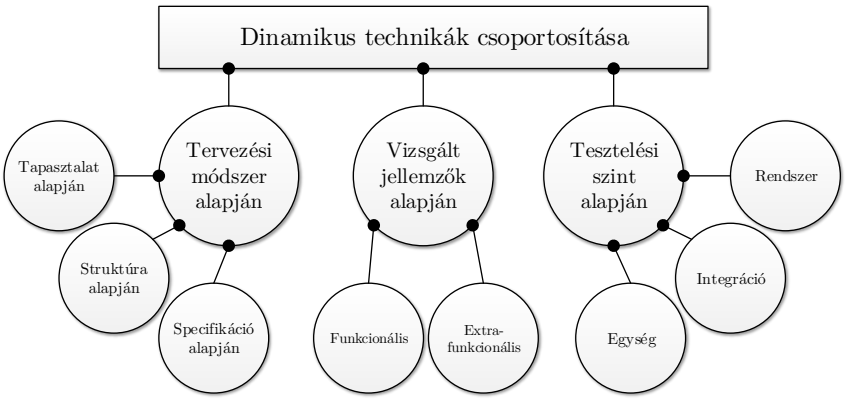
A szoftvertesztelés meglehetősen széles fogalomkörét nehéz tömören definiálni, habár több szabványosító törekvés is létezik. Az alábbiakban két különböző definíciót ismertetek a szoftvertesztelésre, melyek közül az egyik az IEEE *(Institute of Electrical and Electronics Engineers)*, míg a másik az ISTQB *(International Software Qualification Board)* nevéhez fűződik. Dolgozatomban az utóbbi szervezet definícióját használom fel.

- *IEEE:* A tesztelés egy olyan tevékenység, melyben a vizsgált rendszer, vagy komponens végrehajtódik, majd az ebből kapott eredmények bizonyos szempontok szerint kiértékelésre kerülnek. [17]
- *ISTQB:* Az összes szoftverfejlesztési életciklusban jelen lévő akár statikus, akár dinamikus folyamat, amely kapcsolódik a szoftvertermékek tervezéséhez, elkészítéséhez és kiértékeléséhez, így megállapíthatóvá téve, hogy a termék teljesíti-e követelményeit, megfelel-e céljának. A tesztelés felelős a szoftvertermék hibáinak megtalálásáért. [19]

Szoftverek tesztelése tehát a fejlesztés bármely fázisában végezhető, az egyes technikákat az áttekinthetőség érdekében érdemes azonban más szempontok szerint rendszerezve tárgyalni, megemlítve a folyamatban való elhelyezkedésüket.

2.1.2. Tesztelési metodikák rendszerezése

Az alábbiakban a teljesség igénye nélkül az ISTQB definíciójának megfelelő dinamikus technikákat mutatok be, amelyek fontosak a dolgozatom témájának megfelelő elhelyezéséhez. Ezen módszerek a kód végrehajtásán alapuló szoftverellenőrzést teszik lehetővé. A csoportosítási lehetőségek egy részét az alábbi, 2.1. ábra szemlélteti.



2.1. ábra. Dinamikus tesztelési módszerek csoportosításának lehetőségei.

Tervezési módszerek: A dinamikus teszttervezési módszerek három fő csoportját érdemes megkülönböztetni, melyek a következők.

- *Specifikáció alapján:* A szoftver specifikációja alapján történő tesztelés elősegíti annak ellenőrzését, hogy minden benne foglalt elvárás és kritérium teljesül-e. Az ide sorolható technikák másik elnevezése a *fekete-doboz (black-box) módszerek*, hiszen a tesztelés tervezése során a vizsgált rendszer struktúráját nem veszik figyelembe, fekete doboznak tekintik.
- *Struktúra alapján:* Az ide tartozó módszerek segítségével csupán a kód ismeretével lehetséges tesztek előállítása, így jól jellemezhetőek az előzővel ellentétben az *üveg-doboz (white-box) módszerek* elnevezéssel. Az ilyen technikák egyik fő motivációja, hogy a kód struktúrájának ne maradjon ellenőrizetlen része.
- *Tapasztalat alapján:* Kevésbé formális technikák gyűjtőcsoportja, amelyek hatékonysága nagymértékben függ a tesztelést végző személy saját tapasztalatától.

Vizsgált jellemzők: A lehetséges vizsgálati jellemzők két nagyobb csoportra bonthatók, amelyek jól el is különíthetők egymástól.

- *Funkcionális jellemzők:* Az ide sorolható tulajdonságok írják le az ellenőrzött szoftver viselkedését, működését. Általánosságban a vizsgált és az elvárt értékek is konkrétan kerülnek megadásra.
- *Extrafunkcionális jellemzők:* A szoftver működésének sajátosságait leíró tulajdonságok, mint például a rendszer skálázhatósága vagy a robusztusság. Több esetben itt azonban nem adható meg konkrét mérőszámmal elvárás vagy eredmény.

Tesztelési szintek: Az elterjedt szoftverfejlesztési életciklus modellek közel minden fázisában találhatóak teszteléshez kapcsolódó tevékenységek, így érdemes ilyen megközelítésben is felbontani és csoportosítani az egyes technikákat. A V-modell egy meglehetősen jó példának tekinthető, hiszen az implementációs fázis után minden lépésben a tervezési fázisban elkészült elemek verifikációja történik meg. Dolgozatom témájának szempontjából kiemelendők a részletes tervekkel szemben felállított és végrehajtott egységtesztek és az ezekhez tartozó támogató folyamatok.

2.2. Egységtesztelés

Az ISTQB szerint az egység jelen kontextusban az ellenőrzött alkalmazás legkisebb, még tesztelhető része. Az egységtesztelés ebből adódóan a szoftver forráskódjának ezen részegységeinek vizsgálatára irányuló tevékenység.

Fontos kiemelni, hogy az egységtesztelést sok esetben nehéz elkülöníteni az integrációs tesztektől. Habár az utóbbi az egyes komponensek, egységek közötti együttműködés vizsgálatért felelős, az egység tág határokkal, nagy granularitási szinttel való definiálása elmoshatja a két tesztelési szint közötti különbségeket. Emiatt elengedhetetlen, hogy az egységtesztelés során a vizsgálandó egységek megfelelő pontossággal legyenek definiálva és jól meghatározott határvonallal rendelkezzenek.

2.2.1. Izolált egység

Az egységet nehéz tehát megfelelően meghatározni, fontos azonban, hogy a hozzá tartozó külső függőségei ténylegesen leválasztva, izolálva legyenek. Ez lehetővé teszi, hogy a külső függőségekből esetlegesen eredő hibák ne fedjék el a komponens belsejében található rendellenességeket. A következő formalizált definíciók az izolált egység pontos leírását tűzik ki célul *objektumorientált* rendszerek esetén.

Definíció. *Az **egység** definiáláshoz szükséges segéddefiníciók a következők:*

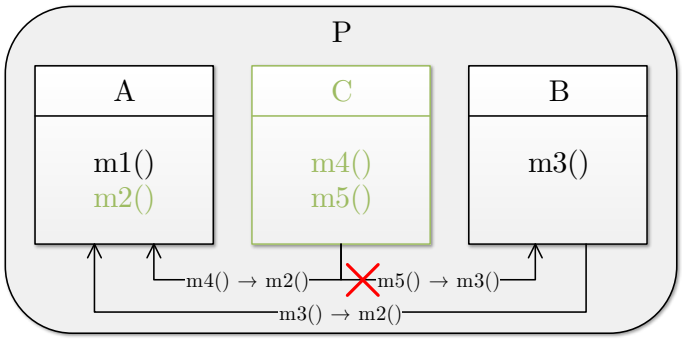
- *Legyen $C \in \mathcal{C}_\mathcal{P}$ egy osztály, $\mathcal{C}_\mathcal{P}$ pedig egy $\mathcal{P}$ program forráskódjában található **osztályok** halmaza.*
- *Legyen $M_C \in \mathcal{M}_C$ egy metódus, ahol $\mathcal{M}_C$ egy tetszőleges $C \in \mathcal{C}_\mathcal{P}$ osztály összes **metódusának** halmaza.*
- *Legyen $\mathcal{D}_{E,I} \subseteq \mathcal{D}_E$ az $E, I \in \mathcal{M} \cup \mathcal{C}_\mathcal{P}$ $(E \neq I)$ elemek (osztály vagy metódus) közötti **függések** halmaza úgy, hogy a függés iránya $E \to I$, azaz E függ I-től. Ilyen függés lehet például paraméterként való használat vagy akár metódushívás is.*

Definíció. *Legyen a **tesztelendő egység** egy olyan $\mathcal{U} \subset \mathcal{M} \cup \mathcal{C}_\mathcal{P}$ halmaz, melyre minden $U \in \mathcal{U}$ részegység esetén teljesül minden I-re, hogy minden $D \in \mathcal{D}_{U,I}$ függés izolálva van.*

1. példa. *A $\mathcal{P}$ példarendszer álljon az A, B, $C \in \mathcal{C}_\mathcal{P}$ osztályokból, melyek az alábbiak:*

- ***A:** $\mathcal{M}_A$:= {m1(),m2()}, $\mathcal{D}_A$:= {$\emptyset$}*
- ***B:** $\mathcal{M}_B$:= {m3()}, $\mathcal{D}_{B,A}$:= {m3() $\to$ m2()}*
- ***C:** $\mathcal{M}_C$:= {m4(),m5()}, $\mathcal{D}_{C,A}$:= {m4() $\to$ m2()}, $\mathcal{D}_{C,B}$:= {m5() $\to$ m3()}*

Válasszuk az ellenőrizni kívánt $\mathcal{U}$ egységnek a C osztályt és az A osztály $m2()$ metódusát, tehát U := { C, $m2()$ }. Ekkor tehát a leválasztandó külső függés mindössze a $m5() \to m3()$. Az összefüggéseket az alábbi, 2.2. ábra is szemlélteti, ahol zöld színnel jelöltem az ellenőrizendő egységhez tartozó elemeket, míg piros X-szel jelzem a leválasztandó függőséget.



2.2. ábra. A 1. példa felépítése és függőségei.

2.2.2. Az egységtesztelés jelentősége

Az egységteszelés jelentősége abban rejlik, hogy segítségével biztosítható a kód minőségének javulása, tartalmazott hibák számának csökkenése, így redukálva a szoftver éles használata során felbukkanó, robusztusságot veszélyeztető hibák (pl. nem kezelt kivételek, hibás lefutások) megjelenését.

Az alapvető megfontolás – mely szerint a lehető legkisebb tesztelhető egység legyen a vizsgálat határa – is biztosítja, hogy az adott egység minél tüzetesebben, részletesebben ellenőrzött legyen. Nagyobb granularitással hasonló részletességű vizsgálatok nem vagy nehezen biztosíthatóak. Az egységtesztelés használatának előnyei között tehát biztosan említethetőek a következők:

- *Felfedezett hibák:* Több hiba vagy hiányosság felfedése lehetséges a kisebb granularitásnak köszönhetően.
- *Részletesebb vizsgálat:* Az egyes külső, kiajánlott interfészeken, beavatkozási felületeken *(API)* nem feltétlenül érhető el minden funkció közvetlenül, az egységteszteléssel azonban azok átvizsgálása is elvégezhető.
- *Karbantarthatóság javítása:* Segíti a forráskód struktúrájának átlátható kialakítását azáltal, hogy az egységtesztelhetőséget szem előtt tartva már a fejlesztés fázisa során is pontosan definiált komponensek és függőségek szükségesek.
- *Költségcsökkentés:* A rendellenességek korai felfedése csökkenti kijavításának költségét, az egységteszteket pedig általában az implementációval párhuzamosan, vagy az után közvetlenül végzik, ami a teljes fejlesztési folyamatot tekintve korainak mondható.
- *Párhuzamosíthatóság:* Párhuzamosan is végezhető, így csökkentve az ellenőrzés összidejét, mely szintén a fejlesztési költségek csökkenéséhez vezethet.

2.3. Szimbolikus végrehajtás

A szimbolikus végrehajtás egy olyan *program analízis* technika, melynek lényege, hogy a vizsgált kód bemeneteire *szimbolikus* változók kerülnek a konkrét értékek helyett. Ekkor a kód végrehajtása valójában nem történik meg, hanem egy *interpreter* értelmezi és

feldolgozza az aktuális utasítást, kiértékeli a változásokat a szimbolikus változók szempontjából, majd lép tovább. Ez addig megy, amíg minden lehetséges forráskódútvonalat fel nem térképezett vagy egy előre definiált korlátot el nem ért.

Az interpreter végrehajtásának eredményeképp úgynevezett útvonalfeltételek (*Path Condition – PC*) állnak elő, melyek minden egyes elérhető végrehajtási útvonalakra feltételeket fogalmaznak meg a szimbolikus változók segítségével.

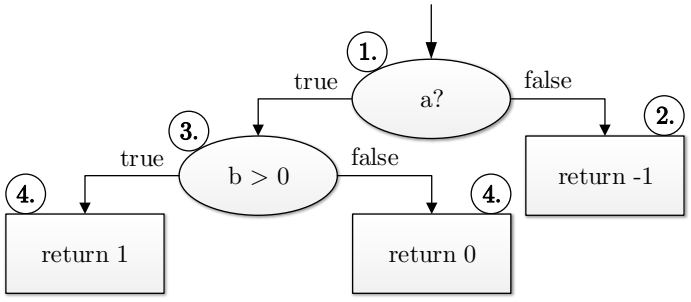
Tekintsük a következő, 2. példát és a hozzá tartozó 2.3. ábrát, amelyek segítenek áttekinteni a technika fő lépéseit.

2. példa. *Legyen egy Foo metódus egész visszatérési értékkel, illetve két bemeneti paraméterrel. A metódus a kapott paraméterek alapján dönt a visszatérésről:*

```
int Foo(bool a, int b) { return a ? (b > 0? 1 : 0) : -1; }
```

A fenti kódrészlet két, egymásba ágyazott elágazás egyszerűsített leírása. Az interpreter a feldolgozás során első lépésben az a *és* b *változók helyére behelyettesíti az* $\tilde{a}$, $\tilde{b}$ *szimbolikus változókat. Az interpreter elkezdi a program végrehajtását, majd ha egy elágazáshoz ér, akkor kettébontja a végrehajtást az adott feltételnek megfelelően. Jelen esetben ezek a lépések a következők.*

1. *Első elágazás felfedezése, ahol a feltétel az* a *változóra vonatkozik. Ekkor tehát a szimbolikus végrehajtás a feltételnek megfelelően két ágra válik, melynek feltételei* $\neg\tilde{a}$ *és* $\tilde{a}$.
2. *A tetszőlegesen választott végrehajtási ág legyen most a* $\neg\tilde{a}$. *Az interpreter itt megtalálja a* return *utasítást, amely a működés végét jelzi. Ekkor tehát az első megtalált végrehajtási útvonal feltétele* $\neg\tilde{a}$.
3. *Az interpreter kiválasztja a következő végrehajtási ágat, ami legyen most az egyetlen maradék:* $\tilde{a}$. *Az újonnan felfedezett elágazás feltételének megvizsgálását követően a szimbolikus változókra felírható egy újabb feltétel a két ágra:* $\tilde{b} \leq 0$ *és* $\tilde{b} > 0$.
4. *A következő lépésben tetszőlegesen kiválasztható a következő végrehajtandó ág, ám jelen esetben a rákövetkező lépésben mindkettő terminálódni fog, így már adottak a hozzájuk tartozó útvonalfeltételek:* $\tilde{a} \wedge \tilde{b} > 0$, *illetve* $\tilde{a} \wedge \tilde{b} \leq 0$



2.3. ábra. A szimbolikus végrehajtás egyszerű példájának vizualizált megjelenítése.

A szimbolikus végrehajtásból nyert útvonalfeltételek ugyan még nem használhatóak fel közvetlenül a tesztek előállításához, azonban kényszermegoldókat alkalmazva ez is elérhetővé válik tesztbemenetek előállításával. Általánosan, logikai formulák kielégíthetőségének problémáját a SAT *(Boolean SATisfiabaility Problem)* írja le, ahol azonban csak igaz-hamis

értékekkel történik a vizsgálat. Szimbolikus végrehajtás során nyert útvonalfeltételek esetén ellenben a SAT problémánál nehezebb feladatot kell megoldani, hiszen a kényszereket kielégítő modellek nem csak igaz vagy hamis értékeket vehetnek fel (ahogy jelen esetben a $\tilde{\mathsf{b}}$ szimbolikus változó sem), hanem gyakorlatilag bármilyen típust az adott változó típusának értékkészletéből.

Ennek leírását az úgynevezett *SMT (Satisfiability Modulo Theories)* problémák adják meg, amelyek logikai formulák felett értelmezett SAT döntési problémák háttérelméletekkel támogatva. Ilyen háttérelmélet lehet például az egészértékűségre történő megkötések halmaza, vagy akár egész adatstruktúrára vonatkozó kifejezések halmaza is. Ezek a háttérelméletek adják tehát a logikai formulák termjeire (logikai formulákban lévő változók, függvények) vonatkozó értékmegkötéseket.

A számítógépek viszonylagosan gyors, nagyléptékű fejlődésének és az SMT-megoldó algoritmusok hatékonyságának növekedése által a szimbolikus végrehajtás is egyre hatékonyabb, hiszen egy SMT megoldó kiválóan alkalmazható arra, hogy a végrehajtás során kinyert útvonalfeltételekhez konkrét értékbehelyettesítéseket rendeljen. Ez lehetővé teszi az adott útvonal konkrét lefutásához szükséges tesztbemenetek kinyerését.

Lényeges megemlíteni, hogy a szimbolikus végrehajtás segítségével alapvetően tesztbemeneteket lehetséges előállítani, így a bemenetekből esetlegesen előállított tesztesetekhez nem tartoznak elvárt kimenetek.

2.4. Microsoft Pex

Az előzőekben leírt fejlődés által tehát egyre nagyobb hírnevet nyernek a szimbolikus végrehajtás technikáját alkalmazó eszközök a tesztbemenet-generálás terén. Napjainkban is meglehetősen sok ígéretes eszköz (pl. [28, 14, 4, 26]) áll fejlesztés alatt, amelyek hosszútávú célja nem titkoltan az iparban való alkalmazhatóság fázisába történő eljutás. Ezen eszközök közé sorolható a Microsoft Research által fejlesztett Pex is.

A Pex egy automatikus üvegdoboz-egységteszt generáló eszköz .NET környezethez. A szoftver a szimbolikus végrehajtás egy továbbfejlesztett technikáját, a dinamikus szimbolikus végrehajtást (2.4.2) alkalmazza az útvonalfeltételek összegyűjtéséhez, a konkrét bemeneteket pedig parametrizált egységteszteken (2.4.1) keresztül adja át a vizsgált komponens, egység számára. Ekkor a paraméterek behelyettesítésével az eszköz beállításaitól függően teszteset fog keletkezni (a bemenetekből és a parametrizált egységteszt logikájából együttesen), amely eltárolható későbbi használatra. A Pex tehát mindig ezeken a paramétereken keresztül tudja csak vizsgálni a komponenst. Az útvonalfeltételekből konkrét bemeneteket az SMT problémákat is jól kezelő Z3 tételbizonyító [10] segítségével kapja meg.

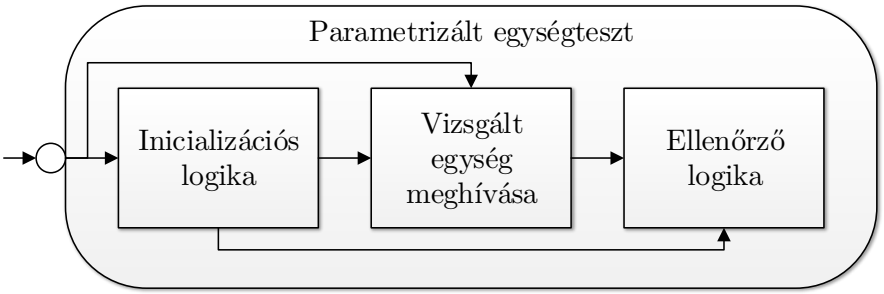
2.4.1. Parametrizált egységtesztek

A paraméteres egységtesztek *(Parameterized Unit Test – PUT)* adják tehát a Pex számára a beavatkozási felületet, így ezen keresztül kell vizsgálnia a komponens működését. A feladat formalizálva így a következőképp írható le [28].

Definíció. *Legyen adott egy szekvenciális P program S utasításhalmazzal. A* ***tesztelési probléma*** *szerint ki kell számítani az I bemenetek halmazát, hogy minden elérhető $s \in S$ utasításhoz létezzen $i \in I$ úgy, hogy $P(i)$ végrehajtja s-t.*

Konkrét megvalósítás tekintetében a paraméteres egységtesztek valójában metódusok adott paraméterlistával, így tekinthetjük úgy, mint egy *csomagoló interfész* az egység bemeneteihez. A tesztmetódusok valójában nem csupán az egység aktuálisan tesztelt

metódusainak hívásait tartalmazhatják, hanem ide kerülnek az inicializálási, izolációs és ellenőrzési logikák is. Az általános felépítést a 2.4. ábra szemlélteti.



2.4. ábra. A parametrizált egységtesztek általános felépítése.

A Pex által használt dinamikus szimbolikus végrehajtás egyik alap építőköve, hogy parametrizált egységteszteket használ az ellenőrizendő komponens vizsgálatánál. A paraméterlista jó megválasztása azonban a tesztelési problémából adódóan nem triviális feladat, hiszen eldönthetetlen, hogy pontosan mely I bemenethalmazra lesz szükséges a teljes S utasításhalmaz lefedéséhez. A lehetséges minimális paraméterlistára vonatkozó feltétel viszont megfogalmazható a következőképpen.

Definíció. *Legyen adott egy U tesztelés alatt álló egység, hozzá egy T_U parametrizált tesztmetódus, és a T_U-ban lévő metódushívások metódusaiból álló M_{T_U} halmaz. A T_U* ***minimális paraméterlistája*** *álljon azon p paraméterekből, amelyre igaz, hogy $\forall m \in M_{T_U}$ metódus P_m paraméterlistája esetén $p \in P_M$.*

Amennyiben ez a minimális paraméterlista nem biztosított, az magával vonhatja, hogy nem biztosítható az az I bemenethalmaz, amely segítségével minden elérhető utasítás lefedhető. Fontos azonban megjegyezni, hogy ez nem általánosítható, hiszen a programstruktúrából adódóan elképzelhető olyan eset, hogy az adott egység lefedése kívülről sohasem biztosítható (pl. nem teljes interfész), illetve előfordulhat az is, hogy az I bemenethalmaztól függetlenül biztosítható a teljes lefedés (pl. paraméterek továbbadása).

2.4.2. Dinamikus szimbolikus végrehajtás

A Pex által is alkalmazott dinamikus szimbolikus végrehajtás *(Dynamic Symbolic Execution – DSE)* egy olyan technika, amely a szimbolikus végrehajtást ötvözi a konkrét végrehajtással. A módszer a konkrét végrehajtást párhuzamosan végzi a szimbolikus végrehajtással, így elérhető, hogy a konkrét futásokból nyert információk segítsék és irányítsák a szimbolikus végrehajtást a lefedéshez szükséges irányokba.

A DSE első lépésben nagyon egyszerű, konkrét bemenetekkel (pl. egész számok esetén 0-val) indítja a végrehajtást, majd a szimbolikus kényszereket ennek a lefutásnak a mentén gyűjti össze. A következő lépésben egy keresési stratégia segítségével kiválaszt egy kibontandó elágazást: az ott megállapított elágazási kényszert negálja vagy kiegészíti továbbiakkal. Ezt az újonnan előálló kényszert lehet átadni a kényszermegoldónak, amely konkrét bemeneteket ad a következő konkrét végrehajtáshoz, amellyel ismét egy új lefutási út keletkezik. Az algoritmus innentől kezdve ciklikusan ismétli ezt addig, amíg van ilyen lehetséges, újonnan felfedhető útvonal (kielégíthető a megoldónak átadott kényszer), vagy idő- és/vagy memóriakorlátba ütközik. Az Algoritmus 1. a DSE alapvető vázát mutatja be [9].

Algoritmus 1 A dinamikus szimbolikus végrehajtás algoritmusa

```
1: J := ∅;                                  ▷ a már megvizsgált bemenetek halmaza
2: for each i ∉ J do      ▷ az i értékkészletére a bemeneti típusok adnak megkötéseket
3:     Végrehajt P(i);                  ▷ emellett párhuzamosan C útvonalfeltétel felvétele
4:     J := J ∪ C;      ▷ C, mint bemenethalmaz, ami az adott útvonalat eredményezte
5: end for
```

2.4.3. Belső működés analízise

A Pex belső működését immáron több évnyi használat és *reverse engineering* eszközök által sikerült megismernem. A tesztbemenet-generáló eszköz minden parametrizált teszthez egy-egy felfedezést (*exploration*) indít el. A felfedezésen belül minden lépésben a DSE-nél már leírt konkrét végrehajtásokat (itt *run*) hajtja végre. Egy felfedezés időtartama alatt a Pex egy úgynevezett végrehajtási fát tart fenn, ami az addig megtalált végrehajtási útvonalakból áll össze. A fa felépítése során a Pex – többek között – elágazásoknál és metódushívásoknál is létrehoz egy-egy csomópontot. Minden csomópont a belső reprezentációban címkézve van az odáig való eljutáshoz szükséges útvonalfeltétellel (itt *prefix*).

Az eszköz minden konkrét végrehajtás előtt a háttérben működő keresési stratégiája segítségével kiválasztja, hogy mely csomópont kerüljön kibontásra az adott iterációban. Ekkor az eszköz veszi a csomóponthoz tartozó prefixet, majd konjugálja az összes felfedett, belőle kimenő ág feltételének diszjunkciójának negáltjával (*flip* lépés). Az ebből előálló kényszert a Pex továbbítja a Z3 számára, majd ha kap konkrét értékeket, amelyek kielégítik a kényszert, akkor elindítja a végrehajtást a kapott értékekkel.

Fontos kiemelni, hogy az említett keresési stratégia nagymértékben befolyásolja az eszköz működésének hatékonyságát, hiszen nem megfelelő használata esetén könnyedén elakadhat a bejárás. Az algoritmus leállása ugyan véges számú végrehajtási útvonalú kódbázis esetén megoldható, viszont egyéb esetben szükség van korlátok definiálásra a működés terminálásának érdekében.

A kényszerek szimbolikus felállításához a Pex gyakorlatilag minden .NET típushoz biztosít absztrakt reprezentációt, illetve képes ezeket közvetlenül SMT problémaként megfogalmazni, amelyet a Z3 fel tud dolgozni.

2.4.4. Kiterjeszthetőségi vizsgálat

A Pex az alapértelmezett kiterjesztési felületet .NET attribútumokon keresztül biztosítja az eszközt körülölelő keretrendszer segítségével. Az ajánlott kiterjesztési mechanizmus három előre elkészített attribútumon keresztül történik. Emellet léteznek más interfészek is, azokhoz viszont nem található publikus dokumentáció, így használatuk nehézségekbe ütközik. Ez a három különböző attribútum a Pex futásának különböző időpillanataiban képes aktiválni a hozzájuk tartozó komponenseket, így különböző időben keletkező egyéb információkat tesz hozzáférhetővé a kiterjesztések számára. A három említett attribútumtípus a következő.

- *Execution:* A Pex teljes futási ideje alatt élő komponensek futtatását biztosítja, illetve a Pex indításakor és leállásakor lehetőség van üzleti logikát is végrehajtani.
- *Exploration:* A Pex egy-egy felfedezése során aktív komponensek futtatásának lehetőségét biztosítja. Ez az attribútum is lehetőséget biztosít a felfedezés előtt és után történő parancsok futtatására.

- *Run:* Az eszköz egy-egy lefutásakor aktiválódó komponensek megvalósítását biztosítja. Továbbá szintén lehetőséget ad lefutás előtt és lefutás után parancsok végrehajtására.

Ezen kívül a Pex saját keresési stratégia implementációjára is lehetőséget biztosít, amely segítségével az eszköz hatékonysága továbbnövelhető.

3. fejezet

Automatizált tesztelés kihívásai

A fejezet elején bemutatom az automatizált tesztelés során esetlegesen fellépő problémákat, kihívásokat a kapcsolódó kutatásokon és irodalmakon keresztül. A fejezet további részében korábbi munkáimból [8] összegyűjtött tapasztalatokat mutatok be. Elsőként egy akadémiai fejlesztésű Petri-háló modellező eszköz automatikus teszteléséből levont következtetéseket, majd pedig egy ipari fejlesztésű tartalomkezelő rendszerhez történő automatikus tesztbemenet-generálás kihívásait tekintem át.

3.1. Kapcsolódó kutatások és irodalom

Jelen dolgozatban és a munkám során is a tesztesetek generálására úgy tekintek, mint az automatizált tesztelés egyik technikájára. A tesztgenerálás, illetve a tesztbemenet-generálás témaköre meglehetősen aktív kutatási területnek számít jelenleg is, így az egyes tesztgenerálási módszerek összehasonlítására is léteznek eredmények (pl. [2]), melyek szerint a szimbolikus végrehajtás egyike a legígéretesebb technikáknak.

A szimbolikus végrehajtás technika több változatához készültek eszközimplementációk, melyek azonban meglehetősen eltérő képességekkel rendelkeznek. Több kutatás is ezen eszközök képességeinek összehasonlítását célozta meg [5, 27, 6], így megkönnyítve a választást közöttük. Xiao és társa a kutatásuk [27] során kitértek arra is, hogy a szimbolikus végrehajtás ipari, komplex szoftvereken való alkalmazása olyan többletfeladatokkal jár, mint például az egységek függőségeinek leválasztása vagy az eszközök bejárási stratégiájának megfelelő beállítása. Ezek a többletfeladatok megnövelik a tesztelés idejét, így annak költségét is, ami hátráltathatja az ipari alkalmazást.

A Microsoft Pex eszközét tekintve a kutatás szintén aktívnak tekinthető. Kiemelendő, hogy Tillmann és társai (az eszköz fejlesztői) a Pex ipari használati tapasztalatait, kutatási fázisból alkalmazási fázisba való migrációját is vizsgálták [29]. Tipikusan a *korai elfogadók*[1] problémáját említik az egyik legnagyobb kihívásnak, hiszen egy kutatási célból kifejlesztett módszer és eszköz mindennapi használatának mérnökökkel való elfogadtatása lassú és nehéz feladat. Megemlítik továbbá, hogy egy kísérleti eszköz sikeres esettanulmánya komplexebb, ipari méretekben sem feltétlenül jelenti azt, hogy az adott technológia készen áll a teljes körű, elterjedt használatra. Ez többek között annak köszönhető, hogy az ilyen és hasonló eszközök alkalmazása egy mérnök számára nem egyértelmű, nehézkes és betanulása többletfeladatot jelent. Így tehát a problémák enyhítésére mindenképpen érdemes közelebb hozni egymáshoz a két oldalt: a kutatásból megérett, kinőtt eszközöket a mérnöki alkalmazással.

[1] Az innováció elfogadás-elméletben azon csoport, akik az innovátorokhoz képest már széleskörűen alkalmazzák az adott újdonságot.

3.2. Petri-háló modellező

A modellező eszközhöz a tesztelés során a manuálisan előállított tesztesetek mellett a Pex több, az alább felsorolt használati szintjét is alkalmaztam tesztek további generálásához.

- *Automatikus felfedezés:* Az eszköz legegyszerűbb funkciója, amely minimális beavatkozási lehetőséget biztosít a tesztbemenet-generálás folyamatába, így kétségtelenül csak korlátozott esetekben használható.
- *Parametrizált tesztek:* A használati eset lehetőséget biztosít a paraméteres tesztmetódusok létrehozására és törzsük kitöltésére, így a belőlük konkrét bemenetek segítségével származtatott tesztesetek már elmenthetővé válnak későbbi használatra. Továbbá itt nyílik lehetőség az egység izolációjának megvalósítására, illetve azon metódusok létrehozására, amelyeket a Pex igénybe vesz egy komplex objektum példányosításához, amikor paraméterben szeretné átadni *(Factory minta)*.
- *Kibővített parametrizált tesztek:* Ez a használati eset biztosítja, hogy a Pex segítségével funkcionális jellemzők is vizsgálhatóak legyenek. Ehhez a Pex speciális utasításait kell alkalmazni, amelyek irányítani fogják a felfedezést. A speciális utasítások specifikáció alapján történő megadásával funkcionális tulajdonságokat ellenőrző tesztesetek hozhatóak létre.

A Pex segítségével történő tesztbemenet-generálás során a parametrizált teszteket alkalmaztam, hiszen fontos szempont volt, hogy későbbiekben is felhasználó tesztesetek álljanak elő. A szoftverhez 371 teszteset született, amelyek körülbelül 99%-os kódblokk[2] lefedettséget biztosítottak a kódban, ezek közül pontosan 100 idézett elő kivételt. A szoftver fejlesztőinek visszajelzései alapján ezeknek körülbelül fele volt fals hibajelzés, a másik fele viszont potenciális hibának/rendellenességnek volt tekinthető. A felfedett hibák okai között legnagyobb részt a defenzív programozás hiánya, azaz a rosszindulatú bemenetekre való felkészületlenség említhető. Emellett sikerült felfedni több inkonzisztens változtatást is a kódban, amelyek dinamikusan elérhetetlen kódrészleteket eredményeztek.

A tíz manuálisan megalkotott, funkcionális tulajdonságokat vizsgáló teszteset közül mindössze egyet nem tudott a Pex generált teszttel lefedni. Ez szintén jó iránymutatás volt az eszköz kiváló alkalmazhatóságra hasonlóan komplex rendszerekben (kb. 3000 sor tényleges kód).

Fontos kiemelni azonban, hogy a tesztbemenet-generálás első néhány fő lépése meglehetősen lassú volt, köszönhetően a szoftver komponensei között lévő komplex függőségeknek (amelyeket le kellett választani az egységekről) és a Pex felől való pontos, egzakt hibajelzések hiányának.

3.3. Tartalomkezelő rendszer

Egy ipari partner által biztosított tartalomkezelő rendszer szerveroldali komponense volt a másik olyan esettanulmány, amelynek ellenőrzése során meglehetősen sok értékes tapasztalatot sikerült gyűjteni a Pex segítségével történő tesztelés terén. A szoftver komplexitása egy nagyságrenddel nagyobb (kb. 10000 sor tényleges kód) volt, mint a modellező eszköz esetén.

Külső szoftver lévén a kódbázis teljesen ismeretlen volt számomra, így az első lépések a Pex használatában jóval több előkészületet igényeltek, mint amire számítani lehetett az előző esettanulmányból levont tapasztalatok alapján. Fontos megjegyezni, hogy a tartalomkezelő rendszer fejlesztése a tesztelés elkezdésekor már befejeződött, ami nyilvánvalóan

[2]Olyan kódrészlet, melynek pontosan egy be- és kimenete van.

növeli a hibakeresés nehézségét a nagyobb komplexitás miatt. A szoftver ellenőrzését két fő fázisra bontottam a vizsgált egységek mérete alapján.

1. *Méretkorlát nélküli vizsgálat:* Az egységtesztelés íratlan szabályait áthágva a Pex számára nem definiáltam korlátot, azaz egyáltalán nem határoltam be a tesztelt egység környezetét.

2. *Kis méretű egységekkel történő vizsgálat:* Az egységek méretét néhány osztályra vagy metódusra korlátozva egyszerűsödik a tesztelési probléma (ld. 2.4.1), így megkönnyíthető a nagyméretű, komplex szoftverekben történő tesztelés.

Az *1. fázis* vizsgálatának célja volt, hogy felfedjem a Pex korlátait, azaz hogy megállapítsam milyen kódméret és komplexitás mellett nem képes már megfelelő hatékonysággal vizsgálni a rendszert. A Pex futtatás során azonban meglehetősen kisszámú teszteset született egy-egy metódusból indítva a vizsgálatot, ráadásul nagyon alacsony lefedettséggel. Nagyléptékű hátráltató tényező volt, hogy a Pex futásai a kódbázis mérete miatt gyakorlatilag átláthatatlanok voltak, hiszen akár több tíz metódushíváson is keresztülmentek, amire terminálódott egy-egy. Mindemellett, a terminálódások és elakadások, illetve a tesztesetek hiányának okait nem lehetett közvetlenül kideríteni, ami továbblassította az egyébként is nehézkes tesztbemenet-generálást.

A *2. fázis* során már viszonylagosan átlátható működést kaptam, azonban a tesztesetek hiányát így sem volt triviális megállapítani. A fázisban történő vizsgálat során 10 metódust vizsgáltam meg, amelyekhez átlagosan körülbelül 4 teszteset született. A felderített hibák közül kiemelhető volt egy olyan funkcionális rendellenesség, amely során egy azonosító nem adódik át az adatelérési és az üzleti logikai réteg között. Ez a hiba több helyen kivételt eredményezett a futtatások során. Emellett a Pex segítségével sikerült felfedni több paraméterkezelési hiányosságot, illetve egy potenciális biztonsági hibát is a rendszerben.

A 2. fázis során – hasonlóan a Petri-háló modellező eszközhöz – az egységek komplex külső függőségei miatt az első tesztesetek előállítása sok előkészületet igényelt, valamint ezek felderítéséhez speciális eszközökre volt szükség. Ez a probléma mindkét vizsgált szoftver esetében fennállt, és habár a forráskódok viszonylagosan ismeretlen mivoltában fakadt, ezt a Pex ipari alkalmazása esetén is érdemes feltételezni.

3.4. Tapasztalatok összegzése és elemzése

Összefoglalva a tapasztalatokat kijelenthető, hogy a Pex komplex és nagyméretű szoftverek esetén is rendelkezik potenciállal. Ki kell emelni azonban, hogy az említett hátráltató tényezők által csak meglehetősen sok munka befektetésével, illetve korábbi tapasztalatok segítségével lehet jelen állapotában az eszközt ilyen környezetekben hatékonyan alkalmazni tesztbemenet-generálásra. Konklúzióként az eredmények és a két esettanulmány alapján is megállapítható, hogy a Pex használatát ipari méretű szoftverkörnyezetekben úgy kell támogatni, hogy a tesztelést végző mérnök minél kevesebb időbefektetéssel tudja hatékonyan alkalmazni a tesztesetek automatikus származtatására a generált bemenetekből.

3.5. Kutatási kérdések és célok

A kapcsolódó kutatások és a korábbi saját eredményeim is egybehangzóan azt mutatják, hogy a szimbolikus végrehajtás és kapcsolódó technológiái, eszközei napjainkban érik el azon küszöbszintet, amely az ipari alkalmazhatóság minimuma.

A technika és az eszközök szempontjából a legfőbb feladat tehát most a könnyű alkalmazhatóság és a hatékonyság közötti egyensúly megtalálása, amely többek között a

mérnöki alkalmazási szempontok figyelembevételével történhet meg. Ez a már fentebb említett korai elfogadók csoportjának alkalmazásba történő bevonásának egyik elengedhetetlen része.

Az eddigiek alapján tehát a dolgozatom következő fejezeteiben az alábbi fő kérdés és a mellette felmerülő alkérdések megválaszolásához vezető utat fogom bemutatni.

Hogyan támogatható a szimbolikus végrehajtás alapú tesztbemenet-generálás mérnöki alkalmazása?

1. A lefutások milyen jellegű vizualizációja segíti a megértést?
2. Milyen jellegű információk nyerhetők ki a szimbolikus végrehajtás során?
3. Milyen információkat érdemes a lefutásokhoz megjeleníteni?
4. Hogyan hidalhatók át a nem megfelelően izolált komponensekből eredő tesztbemenet-generálási hiányosságok?

A kutatási kérdések megválaszolására a következő Pex kiegészítéseket tartom alkalmasnak, így dolgozatomban is ezek elméleti hátterének majd implementációjának bemutatása a célom.

- Az eszköz egy felfedezés során történt összes lefutásának megjelenítése egyben az átláthatóság növelése érdekében.
- Pontosabb hibajelzési értesítések a tesztelést végző mérnök számára, amellyel könnyebben megállapítható az elakadás oka.
- A kódrészletekhez útvonalfeltételek rendelése, amely segítségével például lehetőség nyílik a Z3 által nem megoldott kényszerek megállapítására.
- Automatikus izoláció, amely segítségével mérnöki beavatkozás nélkül absztrahálhatók az egységből kivezető hívások, ezzel is többletidőt és átláthatóságot biztosítva.

4. fejezet

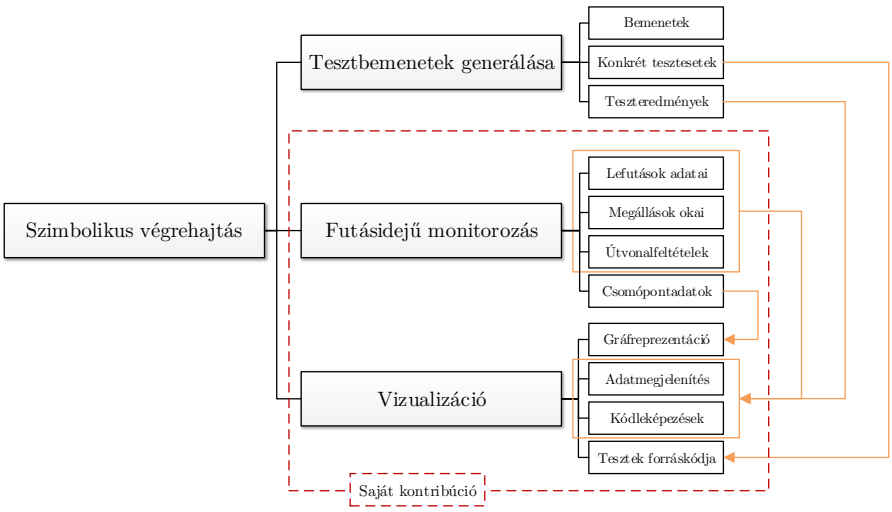
Szimbolikus végrehajtás vizualizációja

A fejezet során bemutatom azt a módszert, amely megteremtheti a kapcsolatot a szimbolikus végrehajtás mérnöki alkalmazása és annak elméleti háttere között az által, hogy segíti és támogatja a technika használatát a tesztelés folyamatában.

4.1. Javasolt módszer

A munkámban egy olyan vizualizációs technikát dolgoztam ki, amely több metainformációval is kiegészítve, megoldásként szolgálhat a szimbolikus végrehajtás mérnöki alkalmazhatósága körül felmerült problémák egy részére. A módszer emellett kiindulási alapot jelenthet további kiegészítések számára, mint például az automatikus egységizoláció.

A vizualizáció megfelelő eljárás lehet arra, hogy a teszteket generáló mérnök a szimbolikus végrehajtás hátterét kevésbé ismerve is képes legyen az eljárást nagy hatékonysággal használni a tesztesetek előállításához. Ehhez azonban a lefutások információi mellett olyan metainformációk is szükségesek, amelyek a működés átláthatóságát növelik. A 4.1. ábra bemutatja a módszer alapvető ötleteit és azok kapcsolatait. Az ábrán szaggatott keretezéssel jelöltem a saját munkámat.



4.1. ábra. Az általam javasolt módszer alapvető fogalmai.

A módszer lényege tehát, hogy a szimbolikus végrehajtás során történő futásidejű monitorozás segítségével vizuálisan is megtekinthetők legyenek az egyes lefutások, illetve a hozzájuk tartozó adatok. Ennek segítségével az adott lefutáshalmazról és a generált tesztekről egy általános kép kapható, amellyel felhasználóként a tesztelő mérnök azonosíthatja a hiányosságokat, vizsgálatból kimaradt részeket, így könnyedén megválaszthatja a további teendőket, lépéseket.

A további alfejezetekben részletesen is bemutatom a javasolt módszer hátterét a reprezentáció felépítésétől kezdve a megjelenítésig.

4.2. A reprezentáció felépítése

A szimbolikus végrehajtás vizualizációjához a futás során össze kell gyűjteni a megjelenítéshez szükséges adatokat egy később felhasználható formában. Jelen alfejezet során ennek részleteit tárgyalom.

4.2.1. Csomópontok és élek

A szimbolikus végrehajtás bemutatása (2.3) során is már lefutási ágakról és elágazásokról tettem említést. A két fogalomból természetszerűleg adódik, hogy a lefutások vizuális megjelenítéséhez a legmegfelelőbb forma egy fa struktúrájú gráf. Munkám során így én is ezt a reprezentációt választottam, hiszen a mérnöki szem számára is megszokott, jól áttekinthető megjelenítést biztosít. Ugyanakkor a technika működésének leírásából nem feltétlenül egyértelmű, hogy mit reprezentál egy-egy csomópont vagy egy-egy él két csomópont között. Ennek feloldásához a következő definíció ad segítséget.

Definíció. *Legyen a* ***szimbolikus végrehajtási gráf*** *egy olyan $G = (V, E)$ irányított élű fa*[1]*, amely tetszőleges $\mathcal{P}$ program $\mathcal{U}$ egységének szimbolikus végrehajtás alapú vizsgálata során történt lefutásait reprezentálja.*

- *A gráf* ***csomópontjainak*** *V halmaza a forráskódot tekintve a következő elemeket tartalmazhatja.*
  - Elágazás: *Bármely elágazástípus legyen csomópont, pl. egy egyszerű* `if-else`*, vagy akár a* `for` *ciklus iterációinak elején történő vizsgálat is.*
  - Blokk kezdete: *Minden programszervezési blokk eleje legyen megfeleltethető egy csomópontnak, hogy könnyen visszakövethető legyen egy-egy lefutás.*
  - Blokk vége: *Hasonlóan megfontolások miatt a blokkok végei is legyenek csomópontoknak megfeleltethetők.*
- *A G gráf bármely két $v_1, v_2 \in V$ csomópontja között akkor vezet* ***irányított él****, ha a szimbolikus végrehajtás során egy lefutásban egymás után következtek. Ennek segítségével érhető el, hogy a lefutásokat egy gráfon megjelenítve kirajzolódjon egy – a bejárás jellegének megfelelő – fa.*

A fenti, csomópontokra vonatkozó felsorolás nem köti meg, hogy minden egyes kódban található adott típusú elemhez legyen csomópont, azaz elképzelhető, hogy egy metódushívás jelzése nem kerül bele külön a fába, hanem csak a hívott metódus törzsének első csomópontja (pl. egy elágazás).

A gráf csomópontjainak **összegyűjtése és tárolása** a szimbolikus végrehajtás során az alábbiak szerint történik.

[1] Olyan gráf, melynek bármely két csúcsát pontosan egy út köti össze, azaz körmentes, összefüggő.

1. Új csomópont felfedezése
2. Egyéni azonosító, mélység hozzárendelése
3. Útvonalfeltétel megállapítása
4. Forráskód-hozzárendelés (ha lehetséges)
5. Eltárolás metaadatokkal azonosító szerint

Fontos, hogy minden egyes csomópont a tulajdonságai alapján **legyen megkülönböztethető** a vizualizációban, így jól elkülönülő jelöléseket kell alkalmazni a különböző tulajdonságok leírására (pl. történt-e kényszermegoldó felé hívás). Ezt részletesebben a 4.2.2. szakaszban is tárgyalom.

A csomópontok adatain kívül a megfelelő vizualizációhoz szükség van az egyes lefutások részletesebb adataira is. Az összegyűjtendő három ilyen adat az alábbi, amelyek segítségül lehetnek az átláthatóság növelésében.

Definíció. *A szimbolikus végrehajtás minden egyes lefutásához legyen rendelve tehát a következő címkehalmaz.*

- *Lefutás sorszáma*
- *Lefutáshoz tartozó csomópontok azonosítói*
- *Utolsó csomópontnál a megállás indoka: pl. hibamentes, hibás, elakadás időtúllépéssel*

4.2.2. Csomópontok külső megjelenése

Az egyes csomópontok felcímkézése metainformációkkal tehát elengedhetetlen az egyszerű használathoz. A következő két szakaszban (4.2.3 és 4.2.4) két ilyen tulajdonságot részletesebben is tárgyalok, de előtte a következő definíciók segítségével pontos leírást adok a tulajdonságok vizualizációban való megjelenítésével kapcsolatban.

Tervezői döntés alapján a végrehajtási gráfok a következő tulajdonságokkal rendelkeznek a megfelelő áttekinthetőség érdekében.

- Legyen a vizualizációban egy csomópont alakja attól függő, hogy **történt-e kényszermegoldó felé hívás**. Ha igen, akkor ellipszis, egyébként pedig téglalap alakú.
- Ha az adott csomóponthoz **létezik forráskód-hozzárendelés**, akkor dupla, egyébként pedig egyszeres körvonallal rendelkezzen.
- A csomópontok színe alapvetően legyen fehér.
- Legyen a csomópont narancssárga színű, ha lefutás végén található, de az adott lefutásból **nincs konkrét teszteset** származtatva.
- Egy adott csomópont legyen zöld színű, ha valamely lefutás végén található és tartozik hozzá egy olyan **konkrét teszteset is, amely hibamentesen** lefut.
- Valamely lefutás végén található csomópont színe legyen piros, ha a lefutáshoz tartozó **konkrét teszteset hibát fedezett fel**.

Az egyéb, csomópontokhoz tartozó tulajdonságok (pl. metódus, útvonalfeltételek, lefutás befejezésének indoka) a csomópontból könnyedén elérve legyenek megtekinthetőek.

4.2.3. Forráskód-hozzárendelés

A mérnöki alkalmazás szempontjából kiemelkedően fontos, hogy a szimbolikus végrehajtást használó tesztelő számára nyilvánvalóvá váljon egyes végrehajtási gráf csomópontok

és a forráskód részleteinek összetartozása. A **kétirányú leképezés** pontosságát az alábbi két szinten lehet meghatározni.

- *Kódsor-szintű:* A gráf csomópontjaihoz kódsorokat rendelünk, amelynek legfőbb előnye, hogy teljes pontossággal megállapíthatók a bejárás helyszínei. A megoldás hátránya azonban, hogy köztes nyelveket[2] használó programnyelvek esetén nem vagy nehezen kivitelezhető, mivel ilyenkor legtöbb esetben a szimbolikus végrehajtás a köztes nyelven történik.
- *Kódblokk-szintű:* A végrehajtási gráf csúcsaihoz mindössze blokk pontossággal rendeljük az elemeket. A megoldás előnye, hogy a megvalósítása egyszerű (elég, ha a blokk elejére mutat minden benne foglalt csomópont). Hátránya, hogy nem adható részletes leképezés, ami csökkentheti a bejárás áttekinthetőségét a vizualizációban.

Általánosságban elmondható, hogy a forráskód-hozzárendelést a legpontosabban érdemes megvalósítani, hiszen a kódsor-szintű hozzárendelés növeli az áttekinthetőséget. Ugyanakkor kiemelendő, hogy a minden csomóponthoz a hozzá tartozó *névtér, osztálynév és metódusnév* álljon rendelkezésre, azaz minden csomópont legyen címkézve a hozzá tartozó metódusnak az FQN-jével (*Fully Qualified Name*).

4.2.4. Útvonalfeltételek

Definíció. *Legyen egy $G = (V, E)$ szimbolikus végrehajtás gráf. Ekkor egy $v \in V$ csomóponthoz tartozó útvonalfeltétel, az oda történő eljutáshoz szükséges logikai feltételek konjunkciója. A konjunkció szétbontható különálló logikai formulákra, amelyeknek halmazát jelölje PC_v egy adott $v \in V$ csomópont esetében.*

A 2. példát tekintve, a 4. lépésben a két levélcsomóponthoz való eljutás feltétele az $\tilde{a} \wedge \tilde{b} > 0$ és a $\tilde{a} \wedge \tilde{b} \leq 0$.

Az útvonalfeltételek a végrehajtási gráf minden egyes csomópontjához legyenek hozzárendelve, hogy könnyen látható legyen az oda való eljutás feltétele. A feltételek elsőrendű logikai formulákként fogalmazhatóak meg, így tehát a programnyelvekből általános megszokott jelölésekkel érdemes helyettesíteni az egyes operátorokat, ezáltal kódközelibbé téve a feltételeket.

Kiemelendő, hogy komplex, mély bejárások esetén előfordulhat, hogy az útvonalfeltétel nem áttekinthető az emberi szem számára. A munkám során azonban a vizualizáció egyik fő célja, hogy segítse a mérnöki munkát, így az útvonalfeltételek leképzését a csomópontokra kétféleképpen is megoldottam.

Definíció. *Legyen adott egy $G = (V, E)$ szimbolikus végrehajtási gráf. Az útvonalfeltételek a $v \in V$ csomópontokra a következőféleképpen lehetnek leképezve.*

- *Teljes leképezés: A végrehajtási gráf minden egyes csomópontjához a hozzá tartozó teljes útvonalfeltételt rendeljük, tehát minden $v \in V$ csomóponthoz a teljes PC_v halmaz tartozzon.*
- *Inkrementális leképezés: Minden $v_1, v_2 \in V$ csomóponthoz, ahol v_2 szülője[3] v_1, v_2-höz csak a $PC_{v_2} \setminus PC_{v_1}$ útvonalfeltétel-halmazt rendeljük. Azaz egyes csomópontokhoz mindig csak a szülő csomópontjához képest hozzáadott új logikai feltételeket rendeljük, amivel a lefutások az irányított élek mentén könnyedén visszakövethetőek.*

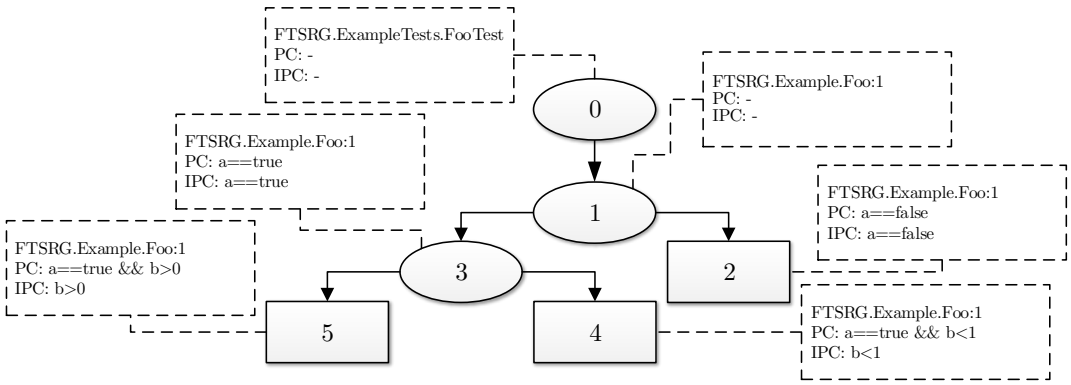
[2] Például Java esetén ilyen a Java bájtkód, vagy .NET esetén az Intermediate Language.

[3] Az adott csomópontba vezető irányított éllel rendelkező csomópont, amely a fában így eggyel magasabb szinten áll.

A megfogalmazott kényszereknek mindig jól leképezhetőnek kell lennie a programban található változókra, azaz a program változóinak nevének meg kell jelennie a logikai változók neveiben is.

3. példa. *Tekintsük újra a 2. példában szereplő kódrészletet. Ekkor az eddigiek alapján a példához tartozó szimbolikus végrehajtási fának a 4.2. ábrán láthatóhoz hasonló módon kell kirajzolódnia.*

Látható, hogy az útvonalfeltételek inkrementálisan és teljesen is leképeződnek a csomópontokra. Az egyes csomópontok egyéni azonosítója a bejárási sorszámai. Az ábrán a PC jelöli a teljes útvonalfeltételt, míg az IPC az inkrementálisat. Kiemelendő, hogy az ábrán megjelennek a metódusnevek és a sorszámok is. A gyökércsomópont a parametrizált tesztmetódus hívását reprezentálja a `Foo` *metódus felé, a többi csomópont pedig a metódus első sorához tartozik, hiszen a példa szereplője mindössze egy sorból áll.*



4.2. ábra. A 2. példa szimbolikus végrehajtási fája az útvonalfeltételekkel kiegészítve.

4.2.5. Egységizoláció megjelölése

A 2.2.1-ben már említést tettem az egység pontos definiálásáról és 2.2.2-ben az egységtesztelés kiemelt szerepéről. A szimbolikus végrehajtás során is alkalmazandó izolációval elérhető a tesztelés komplexitás növekedésének megakadályozása, ám az egység függőségeinek leválasztása meglehetősen összetett feladat, ahogyan a 3. fejezetben is kifejtettem.

A tesztelő mérnök számára ez az izolációs feladat azonban könnyebbé tehető szintén vizualizációs technika felhasználásával. Az előzőekben definiált szimbolikus végrehajtási gráf kiegészíthető azon jelölésekkel, amelyek megmutatják a tesztelendő egységből való kilépési pontokat.

Definíció. *Legyen adott egy tetszőleges $\mathcal{P}$ programhoz tartozó $\mathcal{U}$ tesztelt egység és ezen egység szimbolikus végrehajtásából kapott $G = (V, E)$ végrehajtási gráf. Ekkor bármely két olyan $v_1, v_2 \in V$ csomópontot véve, hogy*

- *v_1 és v_2 között vezet $v_1 \to v_2$ irányított él,*
- *v_1 forráskód-leképezéséhez tartozó metódus eleme $\mathcal{U}$-nak és*
- *v_2 forráskód-leképezéséhez tartozó metódus* nem *eleme $\mathcal{U}$-nak,*

akkor a $v_1 \to v_2$ irányított él legyen jól elkülöníthető és azonosítható a többi éltől, így biztosítva az egységből történő kilépés jelzését.

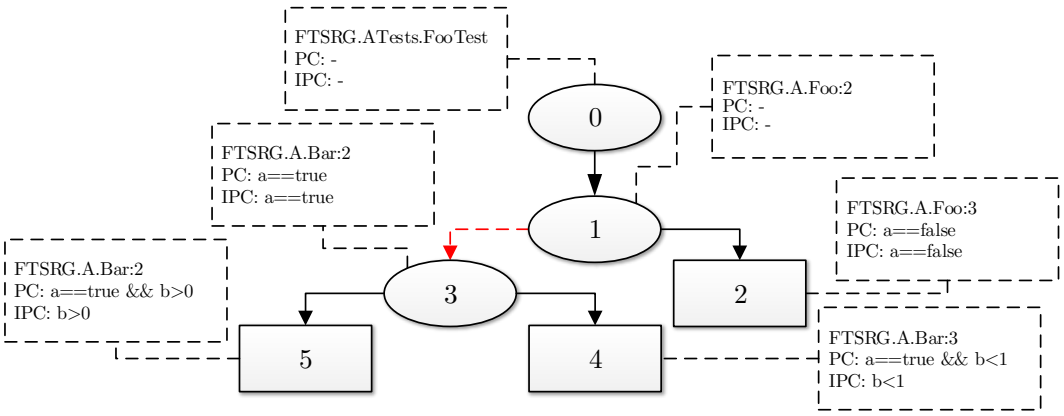
Az izolációt tehát a definiált egységből kivezető élek megjelölésével lehetséges segíteni a mérnök szempontjából. Így lehetővé válik számára azon metódushívások beazonosítása, amelyeket le kell választania az egységről, hiszen a forráskód-hozzárendelés megmutatja a hívási pontokat is. Természetesen ehhez a technikához elengedhetetlen, hogy a szimbolikus végrehajtás első néhány futtatása a *forráskód struktúrájának feltérképezését* célozza meg és ne a tesztbemenet-generálást. A technikát a következő, 4. példán mutatom be a gyakorlatban.

4. példa. *Tekintsük a következő objektumorientált pszeudokód-részletet.*

```
class A
{
    // Foo metódus
    int Foo(bool a, int b)
    {
        if(a) return Bar(b);
        else return -1;
    }

    // Bar metódus
    int Bar(bool b)
    {
        if(b) return 1;
        else return 0;
    }
}
```

Legyen a tesztelendő $\mathcal{U}$ *egység itt egyedül a* `Foo` *metódus, tehát formalizálva* $\mathcal{U} :=$ {`Foo(int,int)`} *és* $\mathcal{D}_\mathcal{U} :=$ {`Boo(bool)`}*, azaz egyedül a* `Boo` *metódus tekinthető külső függésnek.*

Ekkor a `Foo` *metódus egy megfelelő paraméterezésű tesztmetódusából kiinduló szimbolikus végrehajtás a következő, 4.3. ábrán látható végrehajtási fát adhatja. Az ábrán jól láthatóan vizuálisan pirossal kiemelt a külső hívást reprezentáló él.*



4.3. ábra. Az egységizolációja jelölésének egy példája a szimbolikus végrehajtási fán.

A technika mérnöki alkalmazásából adódik egy továbbfejlesztési lehetőség, ami a függőségek automatikus leválasztását hivatott megoldani. Erről a 6. fejezetben még részletesen említést teszek.

4.3. Megjelenés és megjelenítés

Az alfejezet során az eddig bemutatott vizualizációs technika egy konkrét megvalósítását fogom bemutatni. Az eddigiekben adott leírás tekinthető egyfajta metamodellnek, míg a következőkben ennek egy példánymodelljéről lesz szó, amely már eszköz-specifikus részleteket tartalmaz.

A konkrét eszköz jelen esetben a Microsoft Pex, így tehát az eddigi leírásokat és definíciókat a Microsoft Pex segítségével történő megvalósítás tekintetében fogom tárgyalni. Az alfejezet során részletesen kitérek a kihívásokra és azon részletekre, amelyekben a Pex által biztosított szolgáltatások miatt esetlegesen az eddigi leírástól bizonyos mértékben eltérő megvalósítást kellett alkalmazni.

Fontos kiemelni továbbá, hogy a Pex a dinamikus szimbolikus végrehajtást alkalmazza, míg az eddigi leírások az egyszerű szimbolikus végrehajtásra vonatkoztak. Ebből adódóan is megemlíthetők különbségek a vizualizáció adatgyűjtésének szempontjából.

4.3.1. Gráfreprezentáció

A Pex kiegészítési lehetőségeiről a 2.4.4. szakaszban már említést tettem, amelyből jól látható, hogy zárt forráskódú szoftver lévén csak korlátozott mértékű beavatkozást enged a belső működésébe, amelyből kiolvashatók az aktuális végrehajtás információi.

A szimbolikus végrehajtás gráf csomópontjainak reprezentálásához így a Pex által nyújtott információkat lehet felhasználni. A végrehajtás során két kiegészítési interfész ad lehetőséget a feldolgozás kiolvasására: saját keresési stratégia vagy a futások előtt és után lefutó saját komponens.

A Pex a saját keresési stratégia megvalósításához felfedi működésének azon részletét, melyben azokat a csomópontokat kezeli, amiket már említettem 2.4.3.-ban[4]. Ennek használatával azonban a későbbiek szempontjából fontos, csomópontokra vonatkozó metainformációk nem nyerhetők ki az eszköz belső működése miatt, így nem alkalmazható a vizualizáció megvalósítására. Fontos kiemelni továbbá, hogy a Pex leírása nem tartalmaz arra vonatkozólag információt, hogy *pontosan* mely kódelemekből lesznek végrehajtási csomópontok[5], így a fenti leírásnak való megfelelés sérülhet bizonyos esetekben.

Saját keresési stratégia helyett egy saját komponens alkalmazása kézenfekvőbb, amelyet a Pex minden egyes lefutás előtt és után is meghív. Az eszköz ekkor minden lefutás után hozzáférést biztosít a futás során végigjárt csomópontok listájához, amely felhasználható a vizualizációhoz. Ekkor a Pex már kiegészíti a fontos metainformációkkal a csomópontokat. A felfedezés végéig a csomópontokat egyedileg tárolni kell, amelyet a Pex-hez egy új, általam felvett belső komponens biztosít szolgáltatásként. Ez lehetővé teszi, hogy a felfedezés végén kiolvashatók a csomópontok, amelyekből a szimbolikus végrehajtási gráf felépülhet.

A végrehajtási gráf megjelenítéséhez azonban irányított élek is szükségesek. A Pex biztosítja minden egyes végrehajtási csomópont esetén az utódok listájához való hozzáférést. Ekkor elég a csomópontokon iterálva, az egymás után következő csomópontokat összekötni, azonban a különböző futásokból származó duplikált éleket ki kell szűrni, így a már ismert éleket el kell tárolni és csak az újakat kirajzolni. A szimbolikus végrehajtási gráf kirajzolásának folyamatát a következő algoritmus (2.) írja le. Az algoritmus az egyes lefutások után végrehajtandó feladatokat mutatja be, így ennek implementációja található a lefutások végén lefutó komponensben.

A vizuális reprezentációt tekintve egy csomópontot alapesetben egy téglalap jelöl, egy irányított élet pedig egy egyszerű nyíl a két adott téglalap között. Fontos a vizualizáció

[4]Ez a Pex esetében az *ExecutionNode*, azaz a végrehajtási csomópont nevet kapta.

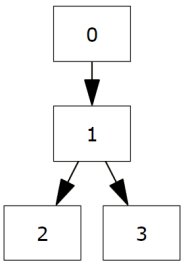
[5]Tapasztalat alapján elágazás, visszatérés, metódushívás biztosan eredményezhet csomópontot.

Algoritmus 2 A csomópontok és éleik kirajzolásának algoritmusa

```
1: for each node in nodesInPath.ExcludeFirst() do      ▷ az első csúcs nélkül iterálunk
2:     prevNode := nodesInPath[node.Index -1];                  ▷ elkérjük az előzőt
3:     if !prevNode.KnownSuccessors.Contains(node) then
4:         PrintEdge();                                          ▷ kiírjuk az élet
5:         prevNode.KnownSuccessors.Add(node);          ▷ most már tudunk róla
6:     end if
7:     nodeStorage.Add(node);                          ▷ eltároljuk a csomópontot
8: end for
```

szempontjából, hogy a gráf megfelelő elrendezést kapjon. Ezt a Pex által automatikusan, minden csomóponthoz biztosított mélység tulajdonsággal értem el, amely megadja az adott csomópont mélységét az adott lefutásban. Az alábbi, nagyon egyszerű kódrészlethez a 4.4. ábrán látható fa tartozik, amely már a saját implementáció segítségével készült.

```
public bool Branching(bool condition)
{
    if (condition)
    {
        return true;
    } else {
        return false;
    }
}
```



4.4. ábra. A Branching metódushoz tartozó Pex végrehajtási fa tulajdonságok reprezentációja nélkül.

4.3.2. Metainformációk reprezentációja

A végrehajtási gráfhoz tartozó metainformációk fontos szerepet töltenek be az áttekinthetőség növelésében, így az egyes tulajdonságoknak jól megkülönböztethetőnek kell lenniük, ahogy a 4.2.2. szakaszban bemutattam.

A **kényszermegoldó felé történő hívásokat** a Pex eseményeivel oldottam meg, amelyek biztosítanak olyan eseménykezelő regisztrálására lehetőséget, ami egy logikai probléma megoldása során hívódik meg. Ennek segítségével pont a Z3 felé történő esetleges hívások jelölhetőek meg a csomópontoknál.

A **forráskód-hozzárendelést** több lépésben kellett megoldani. Elsőként ellenőrizni kell, hogy az adott csomópontnak egyáltalán van-e információja a saját helyéről, mivel a Pex például rendszerszintű könyvtáraknál ezt nem biztosítja. Második lépésben, amennyiben rendelkezésre áll helyinformáció, akkor azt lekérve ellenőrizni kell, hogy az adott dokumentum (fájl), amelyben az adott sor/blokk található, rendelkezésre áll-e. Csak akkor végezhető el a hozzárendelés a vizualizációban, ha ez utóbbi kérdésre a válasz igenlő. A leképzés működését a következő (3.) algoritmus mutatja be. A `MapToFile` és `MapToLine`

általam készített metódusok közül az előbbi a fájlba történő leképzésért felelős, míg a másik a .NET IL kódot fordítja forráskódsorra úgynevezett szekvenciapontok segítségével. A forráskódhoz tartozóan érdemes említést tenni az adott csomópont metódusának nevéről, amelyről a Pex mindig biztosít információt, így természetesen ezt is megjelenítem a csomópont részletes adatai között.

Algoritmus 3 A forráskód-hozzárendelés algoritmusa

```
1:  function MapToSourceCode(ExecutionNode node)
2:      if node.Location then                          ▷ ha tartozik hozzá információ
3:          file := MapToFile(node.Location);                    ▷ leképezés fájlba
4:          if Exists(file) then                                 ▷ ha van ilyen fájl
5:              line := MapToLine(node.Location.Offset);         ▷ leképezés sorba
6:              return file + line;                  ▷ visszatérés fájllal és sorral
7:          end if
8:      end if
9:      return null;                                 ▷ egyébként semmivel térünk vissza
10: end function
```

A **lefutások végén lévő csomópontokhoz tartozó információkat** a Pex alapvetően több forrásból biztosítja. Adott lefutáshoz elkérhető a csomópontok listája, a tesztesetek generálódására pedig eseményen keresztül biztosít feliratkozást az eszközt, ahol hozzáférhető, hogy melyik futáshoz tartozik. Így már csak a három információt (csomópont, lefutás, teszteset) össze kell rendelni az azonosítóin keresztül. Ezzel a technikával megjeleníthetőek a csomópontoknál említett színezésükre való megkötések (piros, zöld, narancssárga).

Az **egységizoláció megjelölése** a metódusnév alapján történik. A tesztelő mérnök számára lehetőséget adok a parametrizált egységteszt attribútumokkal való megjelölésére, melyben megadhatja, hogy mely metódusok/osztályok/névterek tartoznak a tesztelni kívánt egységbe. A Pex felfedezés során a metódusnevek FQN-jét minden egyes csomópontnál ellenőrzöm, és amennyiben egységből való kilépést tapasztalok, akkor piros színnel megjelölöm az adott élet.

Az egyes **csomópontok részletes adatai** között a következőket jelenítem meg, melyek a csomópontból könnyedén elérhetőek a tesztelő mérnök számára.

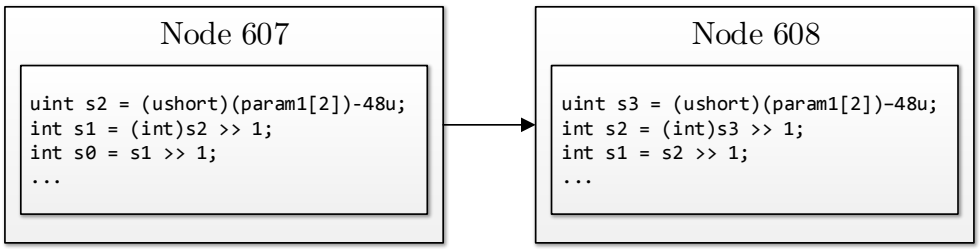
- Útvonalfeltételek
- Teszteset kódja, ha generálódott az adott lefutás végén
- Lefutás befejeződésének indoka

A csomópontok részletes adatai között az **útvonalfeltételek** az egyik legjelentősebbek, így a mérnökök számára is érthető formában kell megjeleníti őket. A Pex a tárolásukat egy saját reprezentációs formában oldja meg, így át kell azokat alakítani könnyen olvasható, érthető formába (*pretty printing*). Ezt a Pex belső szolgáltatásainak felhasználásával lehet megvalósítani, azonban ezek megismeréséhez meglehetősen mély *reverse engineering* kutatásra volt szükség.

A tesztelő mérnök számára fontos lehet, hogy az útvonalfeltételeket könnyedén átlássa, ez viszont több tíz mélységű fa esetében már meglehetősen nehézkes lehet, hiszen a gyakorlatban a mélységgel arányos számú konjunkcióból áll a logikai feltétel. Ennek megoldásaképp bevezettem az úgynevezett **inkrementális útvonal-leképezés** fogalmát már korábban (4.2.4.). A megoldás implementációja a Pex esetében ugyanakkor jelentős nehézségekbe ütközik az alábbiak okán.

A Pex a bonyolult útvonalfeltételek esetén már változókat vezet be, amelyek átláthatóbbá teszik a nagyszámú tagból álló logikai feltételeket. Ekkor az útvonalfeltétel egy

adott csomópontnál két részre bomlik: 1) inicializáció, 2) konjunkció. Előbbiben deklarálódnak és inicializálódnak a logikai változók, majd az utóbbi részben pedig konjunkcióba szervezi őket az eszköz a változó nélküli feltételekkel együtt. Az inkrementális megjelenítés nehézségét az adja, hogy ezen deklarált logikai változókat a Pex csomópontonként nemdeterminisztikus módon nevezi el[6], így az egymást követő feltételek valójában megfeleltethetőek egymásnak logikai értékeik szerint, azonban a szöveges reprezentációjukat tekintve nem. Az alábbi ábra (4.5) egy példát mutat be a problémára, ahol két egymás utáni csomópontban megváltoznak az ugyanazon változóra vonatkozó változónevek.



4.5. ábra. Egy valós példából kivett útvonalfeltételben látható változónév probléma.

Megoldásként a felmerült problémára az alábbi két lehetőség adódhat.

1. Kényszermegoldóval összevetve a két logikai feltétel erősségét és venni a kettő közötti különbséget, illetve a hozzá tartozó kényszert.
2. Szöveg alapú mintaillesztéssel megállapítani a két feltétel közötti különbséget az eltérő változónevek feloldásával.

Az 1. lehetőség a legkézenfekvőbb módszernek tűnik, azonban ki kell emelni, hogy a Pex-ből kinyert útvonalfeltételeket vissza kellene alakítani az eszköz reprezentációs formájába, vagy SMT-LIB[7] formátumba a Z3 számára való átadáshoz. Ez további összetett problémákat vetne fel, ráadásul a Pex futásába újabb, többlet számítást hozna be.

Így a 2. lehetőséget választottam az inkrementális útvonalfeltételek megvalósításához. A feltételek szétbontása azonban nem a Pex futási idejében történik, így a probléma összetettebb, hiszen első lépésben meg kell találni az adott csomópont ősét az inkrementum megállapításához. Az inkrementum megállapítását a következő (4.) algoritmus mutatja be. Az algoritmusban található feltételezés szerint két csomópont között az egyező változók neveiben lévő szám legfeljebb hárommal nőhet az előzőhöz képest. Ezt a feltételezést meglehetősen nagyszámú mérés és ellenőrzés alapján tettem meg.

A megvalósítás bemutatásához tekintsük a következő példakódot és a hozzá tartozó 4.6. ábrát, amelyen már az elkészült eszköz egy részlete látható (ezt részletesebben a következő fejezetben mutatom be). A példakód egy metódust tartalmaz, amely megfordítja a paraméterben átadott `string` típusú adatot, majd visszaadja azt.

[6] A változók s{i} alakú névvel rendelkeznek, ahol i egy nemdeterminisztikusan választott szám.
[7] SMT problémák leírására szolgáló szabványosított formátum. [3]

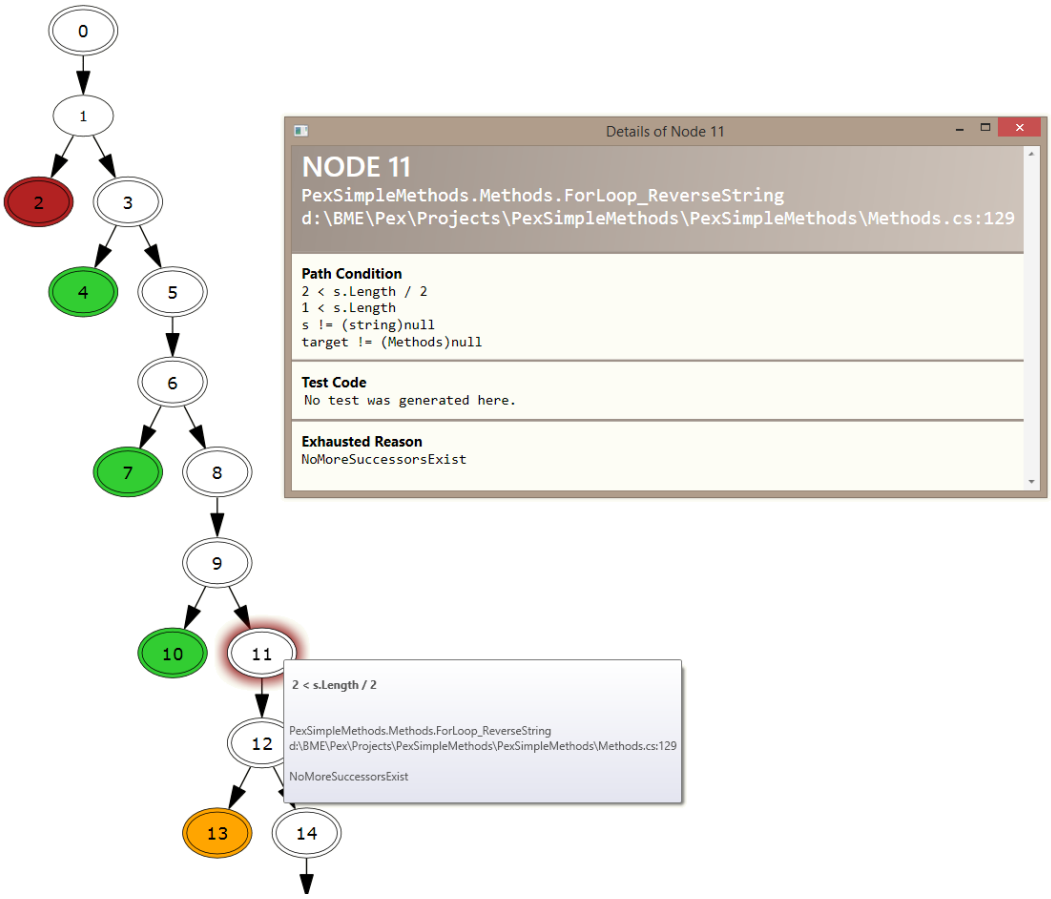
Algoritmus 4 Az inkrementális útvonalfeltételeket előállító algoritmus

```
1: function GetIncrementalPathCondition(ExecutionNode node)
2:     remainedPC[ ];
3:     nodePC[ ] := SplitByLine(node.PathCondition);                ▷ sorokra bontás
4:     OrderByLetter(nodePC);                                       ▷ rendezés ábécé szerint
5:     prevNode := FindParent(node);                                ▷ szülő megkeresése
6:     prevNodePC[ ] := SplitByLine(prevNode.PathCondition);        ▷ sorokra bontás
7:     OrderByLetter(prevNodePC);                                   ▷ rendezés ábécé szerint
8:     if !HasInitializations(prevNodePC) then                      ▷ ha nincs inicializáló rész
9:         for each s in NodePC do
10:            if !prevNodePC.Contains(s) then
11:                remainedPC.Add(s);          ▷ hozzáadjuk, ha előzőben nem volt benne
12:            end if
13:        end for
14:    else
15:        remainedPC := nodePC;                                    ▷ egyébként feldolgozzuk az egészet
16:    end if
17:    for i = 1 to 3 do          ▷ feltételezés: a változónévben a szám max. 3-mal nőhet
18:        for each line in prevNodePC do
19:            incrementedLine := IncrementVariableName(line,i);
20:            remainedPC.Remove(incrementedLine);                  ▷ kivesszük, ha benne van
21:        end for
22:    end for
23: end function
```

```
public string ForLoop_ReverseString(string s)
{
    char[] chars = s.ToCharArray();
    int j = s.Length - 1;
    for (int i = 0; i < s.Length/2; i++)
    {
        char c = chars[i];
        chars[i] = chars[j];
        chars[j] = c;
        j--;
    }
    return new string(chars);
}
```

A 4.6. ábrán a megnyitott ablakban a 11. csomóponthoz tartozó részletes adatok láthatóak, míg a szürke üzenetdobozban szintén a 11-es csomópont inkrementális útvonalfeltétele is megtalálható. Mindemellett az ábrán találhatóak különböző színű csomópontok is, amelyek közül például a piros jelöli azon esetet, amikor a paraméterben kapott `s` változó `null` értéket vesz fel, így hiba keletkezik.

Elmondható összességében, hogy minden olyan metainformációt megjelenítettem vizualizációban, amelyeket szükségesként definiáltam a fejezet korábbi szakaszaiban (4.2.2.). Ezzel tehát megválaszoltam a kutatási kérdések közül az 1-3. sorszámúakat. A következő fejezetben részletesebben is bemutatom a 4.6. ábrán látható, a kidolgozott módszerem alapján készült eszközt.



4.6. ábra. A ForLoop_ReverseString metódushoz tartozó végrehajtási fa egy részlete.

5. fejezet

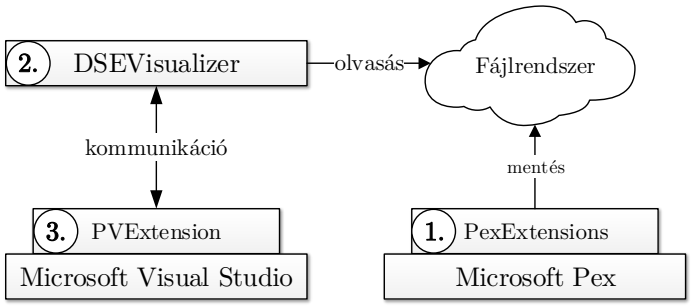
Kiértékelés

A fejezet során először részletesen bemutatom az elkészült vizualizációs eszköz funkcióit és azt, hogy azok hogyan segítik a mérnököket a szimbolikus végrehajtás alkalmazásában. Ezt követően bemutatom az eszköz és a kidolgozott módszer működését példákon keresztül, majd mérési eredmények segítségével alátámasztom a hatékonyságot. Ez után megemlítem a módszer és az eszköz korlátait, majd a fejezet végén bemutatok egy használati módszert, végül pedig kitérek a kapcsolódó kutatásokra, amelyek a szimbolikus végrehajtás vizualizációját tűzték ki célul.

5.1. Az elkészült eszköz

Az előző fejezetben tárgyalt általánosabb, majd specifikusabb leírása a szimbolikus végrehajtás vizualizációjának megfelelő alapot adott arra, hogy elkészítsem a leírásoknak megfelelő eszközt, ami képes minden ott definiált funkció és döntés megvalósítására. Az eszköz segítségével átültettem a gyakorlatba is a vizualizáció kínálta lehetőségeket.

Az elkészült alkalmazás a *DSEVisualizer* és két, hozzá tartozó komponens. A követelmények alapján a három komponenst szorosan együttműködőként állítottam össze (5.1. ábra).



5.1. ábra. A DSEVisualizer architektúrája.

1. `PexExtensions`: Ez felel a Pex által nyújtott információk összegyűjtéséért és fájlokba mentéséért úgy, hogy megvalósít egy futások előtt és után végrehajtódó, illetve egy felfedezés előtt és után végrehajtódó komponenst együttesen.
2. `DSEVisualizer`: A kinyert adatok megjelenítéséért felelős komponens, amelyet az előző komponensből kapott információk alapján végez.

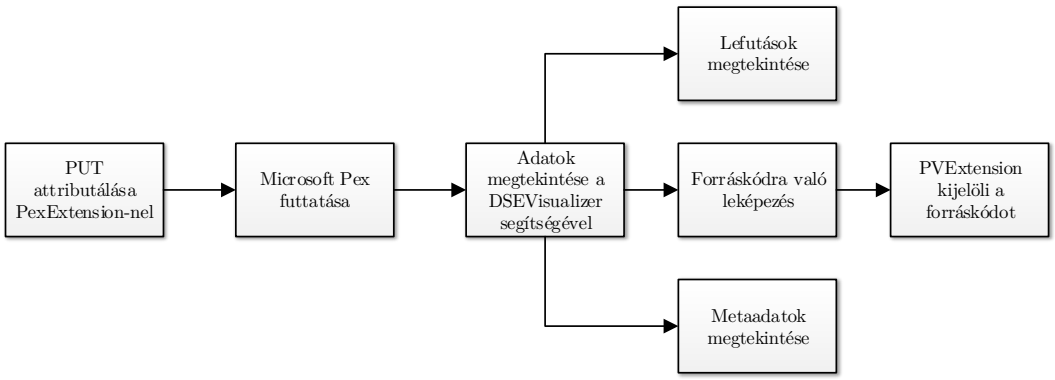
3. `PVExtension`: A komponens biztosítja, hogy a csomópontok forráskód-leképezése a gyakorlatban is kétirányú legyen, így a `DSEVisualizer`-rel szoros kapcsolatban áll.

A `PexExtensions`(1) komponens valósítja meg azokat a Pex kiegészítő modulokat, amelyeket említettem az előző fejezetben, így tehát a futás során eltárolja a bejárt csomópontokat, lefutásokat, majd egy-egy felfedezés végén ezeket a fájlrendszerbe menti. A használata meglehetősen egyszerűen a parametrizált teszt felattributálásával történik: használni kell a `PexGraphBuilderPackage` és `PexGraphBuilderPackageHelper` attribútumokat, melyek számára paraméterként a mentendő fájl elérési útját, illetve a tesztelendő egységhez tartozó osztályok vagy metódusok neveit kell megadni. A kimentett fájl egy *.pviz* kiterjesztésű csomagolt állomány, amelyben *GraphViz*[1] kompatibilis gráfleírások is találhatóak.

A megjelenítő `DSEVisualizer`(2) egy WPF[2] technológián alapuló alkalmazás, amely *kirajzolja és böngészhetővé teszi* a kimentett állomány megnyitásával elért információkból a szimbolikus végrehajtási gráfot, illetve megjeleníti a hozzá tartozó információkat egy mérnökök számára is könnyen átlátható, gyorsan megtanulható kezelőfelületen. A komponens ezenkívül képes kommunikálni a `PVExtension` modullal, amelynek átadja az aktuálisan kijelölt csomópont információit, így lehetővé téve a csomópontról forráskódra történő leképezést.

A harmadik komponens, a `PVExtension`(3) egy Visual Studio beépülő modul, amely folyamatosan figyeli a `DSEVisualizer` által küldött kéréseket és kijelöli azon sort a forráskódban, amelyre a kérés vonatkozott. Emellett biztosítja a leképezés kétirányúságát, azaz képes egy adott sorhoz csomópontot rendelni úgy, hogy visszaküldi a kódinformációt a megjelenítő komponens számára, majd az megkeresi a megfelelő csomópontot a végrehajtási gráfban és kijelöli azt.

Az eszköz működése és használata az architektúra alapján már egyértelműen adódik. Az 1-es komponens kinyeri a szimbolikus végrehajtás információt a Pex-ből, majd átadja a 2-es számára, ami vizualizálja azokat. A 3-as komponens pedig biztosítja a forráskód és a végrehajtási gráf közötti kétirányú leképezést, ha lehetséges. A 5.2. ábra illusztrálja a működés egyes lépéseit.



5.2. ábra. A DSEVisualizer használatának lehetséges lépései.

Összefoglalva tehát az eszköz képes biztosítani olyan funkciókat, amely a Pex, mint szimbolikus végrehajtást alkalmazó eszköz működését átláthatóvá teszi az alkalmazó mérnök számára a vizualizáció segítségével.

[1] Egyike a legismertebb gráfvizualizációs alkalmazásoknak. [20]

[2] Windows Presentation Foundation – a .NET felhasználói felületekért jelenleg felelős technológiája.

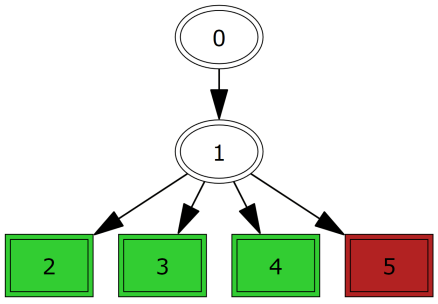
5.2. Demonstrációs példák

A *DSEVisualizer* és a hozzá tartozó két komponens működésének bemutatására elkészítettem egy általános programozási eseteket összegyűjtő .NET osztálykönyvtárat C# nyelven, mely harminc metódust tartalmaz. A könyvtárba vegyesen állítottam össze olyan belső logikákat, melyek a sűrűn használt utasítások, blokkok, illetve a BCL struktúrák részeit lehetőség szerint lefedjék. Az alábbiakban három metódust választottam közülük, hogy illusztrálva bemutassam az elkészített vizualizációs eszköz működését.

5. példa. *Legyen a példához tartozó vizsgálandó metódus az alábbi* `SwitchFourBranches`*. A metódus törzsében mindössze egy darab, négy ágú elágazás található, amelyek közül az egyik kivételt dob. A metódus teszteléséhez a vizualizációhoz megfelelően felattributált parametrizált egységteszt-metódust használtam, amelynek törzsében mindössze meghívódik maga a metódus.*

```
public int SwitchFourBranches(int a)
{
    switch (a)
    {
        case 0:
            return 0;
        case 1:
            return 1;
        case 2:
            return 2;
        default:
            throw new Exception("'a' must be between 0 and 2.");
    }
}
```

A metódushoz pontosan négy darab bemenet generálódott, amelyek összesen lefedik a `switch` *szerkezet minden ágát. A kimentett információk alapján a DSEVisualizer a következő gráfot (5.3) jelenítette meg, ami megfelel az elvárásoknak a négy ággal. Az ábrán jól látható a vörös színű csomópont ami a* `default` *ágon el nem kapott kivételt jelenti jelen esetben, továbbá szembetűnő, hogy minden egyes csomóponthoz tartozik forráskód-információ (dupla körvonalazás).*

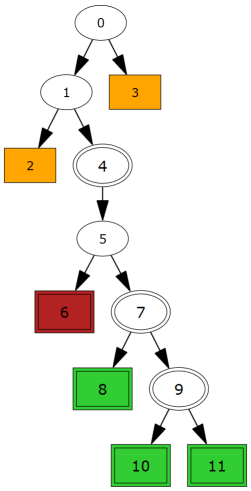


5.3. ábra. A *SwitchFourBranches* metódus szimbolikus végrehajtási gráfja.

6. példa. *Tekintsük a következő példát, ahol a Pex feladata azon tömb átadása paraméterként, amelynek második eleme 99. A parametrizált metódus ebben az esetben is megfelelően paraméterezett, illetve feltételezem, hogy nem jön a tömb helyén* `null` *paraméter és, hogy a kapott index nem negatív. Az előfeltételeket a Pex beépített interfészein keresztül adtam meg.*

```
public bool ArraySample_02(int[] array, int index)
{
    var element = array[index];
    if (index == 1)
    {
        if (element == 99)
        {
            return true;
        }
    }
    return false;
}
```

A metódushoz itt is pontosan négy darab bemenetpár generálódott, amelyből egy kivételt is eredményezett (tömb túlindexelése). A vizualizált szimbolikus végrehajtási gráf a következő ábrán látható (5.4). A narancssárga csomópontok jelölik az előfeltételt megsértő lefutásokat, hiszen azokhoz természetesen nem fog tartozni teszteset sem. A gráfon található továbbá két olyan csomópont is, melyhez nem tartozik leképezés a forráskódra. Ezt a tömb számára, a .NET háttérében történő memóriafoglalás okozza.

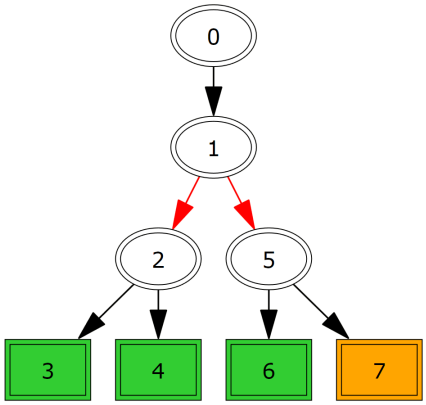


5.4. ábra. Az *ArraySample_02* metódus szimbolikus végrehajtási gráfja.

7. példa. *A harmadik példa két metódusból áll, melyek közül egyik a tesztelni kívánt egység része (*`Foo`*), míg a másik (*`External` *osztály* `Bar` *metódusa) külső függőségnek tekinthető. A* `Foo` *teszteléséhez tehát ismételten létrehoztam a megfelelő parametrizált egységteszt-metódust, amelyben jeleztem a vizualizációs komponens számára, hogy az egységhez csak a* `Foo` *metódus tartozik.*

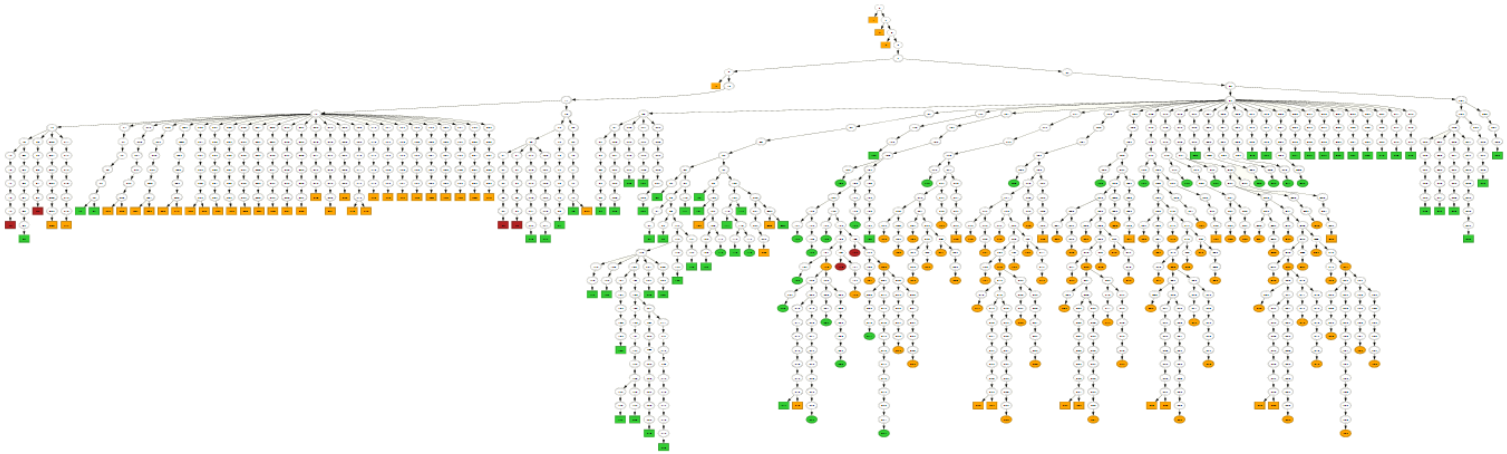
```
public class Current
{
    public int Foo(int a, bool b)
    {
        double paramValue = 0;
        if (a % 2 == 0) paramValue += a / 2; else paramValue += a;
        var external = new External();
        var bar = external.Bar(paramValue);
        if (bar) return 10; else return 0;
    }
}

public class External
{
    public bool Bar(double c)
    { if (c < 0) return true; else return false; }
}
```

A Pex összesen három tesztbemenethalmazt generált, amelyek teljesen lefedték a vizsgált metódust. A 5.5. ábrán látható, hogy a narancssárga csomóponttal végződött futás nem hozott új tesztesetet. Ennek oka, hogy a Pex alapbeállítása esetén akkor generálódik tesztset egy lefutásból, ha az eddigiekhez képest új blokkot fed le, ami jelen esetben nem teljesült. Az ábrán szintén szembetűnő a két piros él, amely jelzi a tesztelendő egységből való kilépést a mérnök számára.



5.5. ábra. A *Foo* és *Bar* metódus szimbolikus végrehajtási gráfja.

A következő alfejezetben nagyobb komplexitású példákat ismertetek, azonban a nagy méretük miatt csak mérési eredményeiket tárgyalom részletesen. A nagy komplexitást jól reprezentálja az 5.6. ábrán látható szimbolikus végrehajtási gráf, melyet egy, a mérések során felhasznált metódus futtatása során nyertem ki.



5.6. ábra. Egy komplex szimbolikus végrehajtásból összeállított végrehajtási fa.

5.3. Mérési eredmények

A kidolgozott vizualizációs módszer és az elkészült eszköz *hatékonyságának* alapfeltételei, hogy 1.) a vizualizáció ne befolyásolja jelentős mértékben a szimbolikus végrehajtás futási idejét, 2.) a megoldás skálázható legyen nagyobb méretű forráskódokra is. A következőkben ezen két perspektíva szerint fogom tárgyalni a kvantitív mérési eredményeket.

A mérnöki alkalmazhatóság alátámasztásához a *hatékonyság* mellett azonban a *használhatóság* mérésére is szükség van. Ilyen mérések kivitelezéséhez viszont részletesen, lépésről lépésre kidolgozott folyamatra van szükség, melyek végrehajtása több tíz résztvevő mérnökkel történik. Fraser és társai a tesztgeneráló eszközük használhatóságának mérésénél egy ilyen folyamatot dolgoztak ki és hajtottak végre [12]. Ez a folyamat a dolgozatom során tárgyalt vizualizációs eszköz mérnöki használhatóságának mérésére is alkalmas lehet, ám TDK munkám során ennek megvalósítására nem adódott lehetőségem.

Az alább tárgyalt méréseket állandó környezetben hajtottam végre, melynek tulajdonságai a következők: Intel Core i7 @ 3.1 GHz, 8 GB DDR3 @ 1333 MHz, SATA3 HDD @ 5400 rpm, Microsoft Windows 8.1 Pro, .NET v4.5.51641, Microsoft Visual Studio Ultimate 2013 v12.0.30501 Update 2, Microsoft Pex v0.94.51006.1.

5.3.1. Futási idő vizsgálata

A futási idő vizsgálata során azon kvantitatív mérési eredményeket mutatom be, melyek célja a vizualizáció és a szimbolikus végrehajtási futási idő kapcsolatának felderítése volt. A mérési folyamatot a következő két részre bontottam: 1.) mesterséges példák, 2.) valós példák. Így az elkészült eszköz és módszer két szempontból is vizsgálható, hiszen egyrészt felderíthető, hogy az eszköz megfelel-e a minimális követelményeinek (pl. alapstruktúrák vizualizációja hatékony-e), másrészről pedig, hogy komplex, valós példákon is megőrzi-e működőképességét (pl. nincs használhatóságot akadályozó mértékű futási idő növekedés).

A mérések során minden egyes metódus végrehajtásánál a következő attribútumokat vizsgáltam meg.

- futási idő (ezredmásodperc pontossággal)
- futási idő vizualizációval (ezredmásodperc pontossággal)
- bejárt csomópontok száma
- legmélyebb csomópont
- forráskód hossza (LOC – Line of Code – kódsorok száma)

A futási idő vizsgálatához a két mért időtartam közötti arányt érdemes tekinteni, hiszen ez mutatja meg a vizualizáció hatását.

A futási idők méréséhez elkészítettem egy olyan Pex komponenst, amely közvetlenül a felfedezés előtt elindít egy stoppert, közvetlenül utána pedig leállítja ezt. Ez a módszer kellően pontos időmérést tett lehetővé, szemben a Pex beépített órájával, ami több, számomra felesleges tevékenység idejét is beleszámolta a futásba (pl. naplófájl generálása).

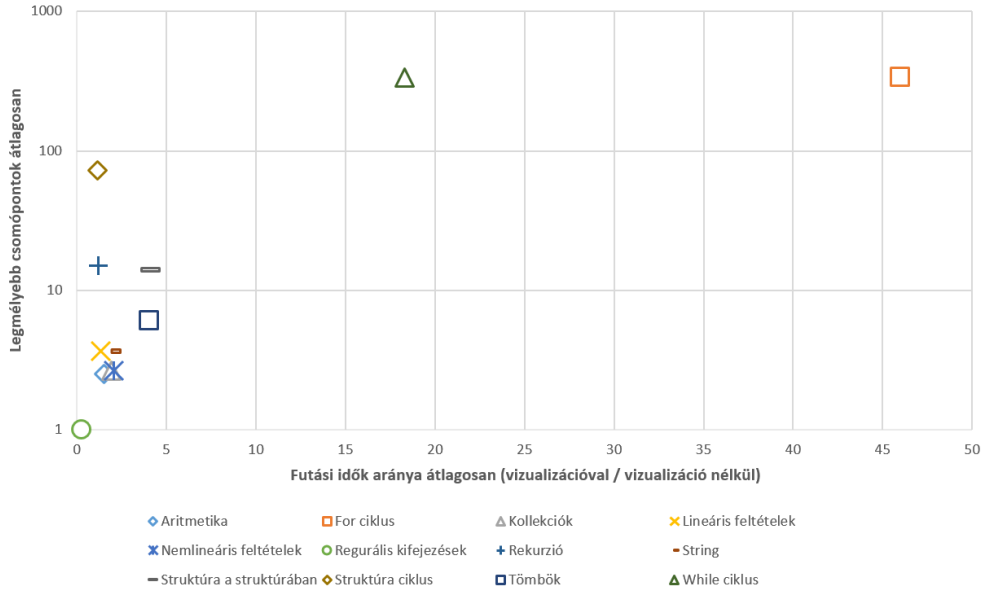
Mesterséges példák

A példakészlet egy, a szimbolikus végrehajtást alkalmazó eszközök szisztematikus vizsgálatára elkészített keretrendszerből való [21]. A vizsgálathoz kidolgozott metódusok így alkalmasak a vizualizáció működésének ellenőrzésére is, hiszen a legtöbb olyan kódrészlet megtalálható benne, melyek szimbolikus végrehajtása esetlegesen gondot jelenthet az eszközök számára. Az általam végzett mérések során ebből a példakészletből választottam

30 olyan metódust, melyek a sűrűn használatos programozási alapstruktúrák nagyobb részét lefedik, így ezeken keresztül történő vizsgálattal megállapítható a vizualizációs eszköz alapvető hatékonysága. A 30 metódust 11 csoportra bontottam funkciójuk alapján, a mérések eredményeit ezen csoportokra történő átlagolással tárgyalom. Az alábbiakban tehát a futási idők arányának kapcsolatát vizsgálom meg a következőkkel: legmélyebben fekvő csomópont, bejárt csomópontok száma, forráskód hossza. A kapott mérési eredményeket a következő táblázat tartalmazza (F: futási idő [másodperc], FV: futási idő vizualizációval [másodperc], A: két futási idő aránya, CS: bejárt csomópontok száma, M: legnagyobb csomópontmélység).

Csoport	**Metódus**	**F [mp]**	**FV [mp]**	**A**	**CS**	**M**	**LOC**
Aritmetika	minMaxWithOrder	1,67	3,18	1,904	7	3	4
Aritmetika	powGuessExponent	2,186	2,384	1,091	5	3	5
For	complex	0,668	58,34	87,335	744	500	15
For	nestedLoop	2,53	120,58	47,66	991	500	8
For	withConditionAndLimit	0,092	0,28	3,04	23	11	6
Kollekciók	guessElementAndIndex	0,173	0,238	1,376	5	2	10
Kollekciók	guessListWithSize	0,172	0,23	1,337	6	3	4
Kollekciók	guessSizeAndElements	0,042	0,13	3,095	7	3	7
Lineáris	threeParamsDouble	1,97	2,77	1,406	10	5	7
Lineáris	threeParamsInt	0,16	0,25	1,563	7	3	7
Lineáris	threeParamsInt NoSolution	0,16	0,16	1	7	3	7
Nemlineáris	quadraticDouble NoSolution	1,763	3,32	1,883	3	2	6
Nemlineáris	quadraticInt	0,165	0,18	1,091	7	3	6
Nemlineáris	quadraticInt NoSolution	0,066	0,21	3,182	6	3	6
Regexp	regexCaseSensitive	0,459	0,14	0,305	3	1	7
Rekurzió	fibonacci	1,83	2,4	1,311	32	20	8
Rekurzió	simple	1,01	1,11	1,099	21	10	6
String	compareTo	0,137	0,143	1,044	3	1	10
String	equality	0,063	0,19	3,016	7	3	10
String	regionEquality	0,227	0,43	1,894	21	7	8
Struktúra egymásban	guess	4,55	17,5	3,846	559	24	10
Struktúra egymásban	guessParams	0,077	0,34	4,416	17	4	15
Struktúra ciklus	arrayOf StructuresParams	121,23	122,24	1,008	1201	133	16
Struktúra ciklus	with LimitParams	0,9	1,13	1,256	45	12	11
Tömbök	fromParamsWithIndex	0,175	1,06	6,057	12	6	7
Tömbök	guessOneArray WithLength	0,12	0,22	1,833	9	4	6
Tömbök	iterateWithFor	0,182	0,76	4,176	33	8	10
Tömbök	twoArrays	0,078	0,33	4,231	12	6	8
While	complex	2,19	58,91	26,9	744	500	18
While	nestedLoop	2,06	46,95	22,791	967	500	12
While	withConditionAndLimit	0,088	0,46	5,227	23	11	8

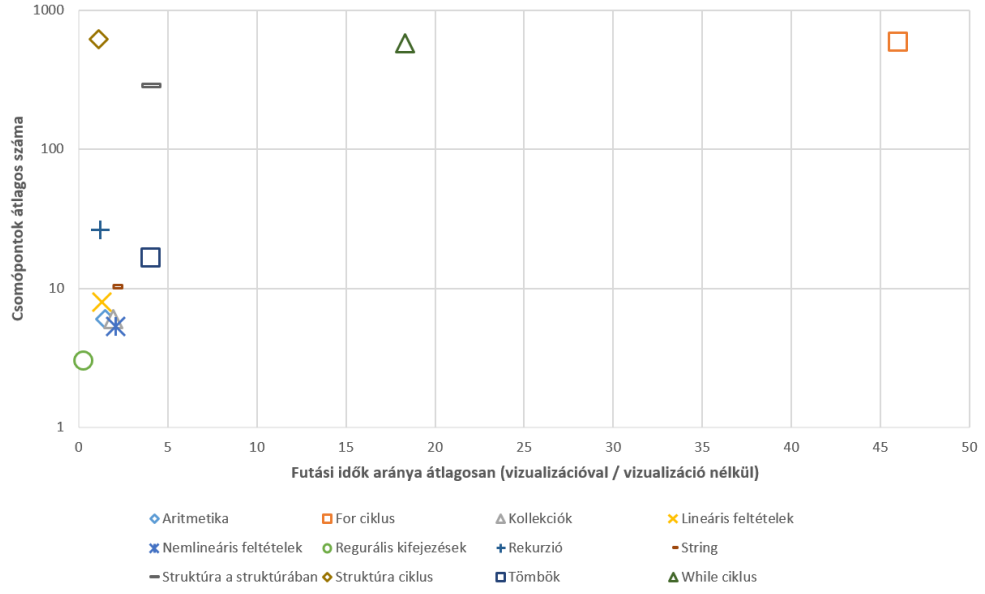
Tekintve az 5.7. ábrát látható, hogy két olyan pont van, melynek futási ideje vizualizációval kiugró arányban nagyobb, ha a legmélyebb csomópont is nagyobb mélységben található. A két pont a `while` és a `for` ciklusokat reprezentálja. Fontos kiemelni, hogy ezen két esetben a Pex futása mélységkorlátba (500 csomópont) ütközött, így valószínűleg a két futási idő aránya tovább nőtt volna mindkét esetben. A többi metóduscsoportot vizsgálva jól látható, hogy a mélységtől függetlenül legfeljebb ötszörös futási idő növekedés volt ta-



5.7. ábra. A legmélyebben fekvő csomópont és a futási idők arányának vizsgálata.

pasztalható, hiszen volt olyan eset, ahol még száz csomópont mélyen is ez alatt maradt a növekedés.

A csomópontok számának és a futási idők arányának vizsgálatánál (5.8. ábra) hasonló összefüggés állapítható meg, mint az előző esetben. Azaz tehát a két, ciklusokat tartalmazó vizsgálat során magas csomópontszám esetén kiugróan hosszabb volt a vizualizációval történő futási idő, egyéb esetben viszont itt is a csomópontok számától függetlenül legfeljebb ötszörös volt a futási idő növekedés, akár több száz, vagy közel ezer csomópont esetén is (pl. struktúra ciklus és struktúra a struktúrában csoportok).



5.8. ábra. A csomópontok számának és a futási idők arányának vizsgálata.

Érdemes megjegyezni, hogy már a puszta mérési eredmények alapján is jól látható, hogy a kódsorok száma nem áll összefüggésben a futási idő növekedésével, így ezt részletesen nem tárgyalom.

A mesterséges példákból kapott eredmények alapján következtetésként levonható, hogy kiugró futási idő növekedés vizualizáció mellett csak olyan ciklusok esetén tapasztalható, melyek nem korlátozottak. Ezek ugyanakkor a szimbolikus végrehajtás egyik alapvető kihívását is jelentik egyben, melyre a mérési eredmények összefoglalásakor még részletesebben is kitérek. Az ilyen ciklusokat tartalmazó eseteket kivéve viszont a futási idő legfeljebb ötszörös mértékben növekedett, ami a futási idők nagyságrendjét tekintve a *mérnöki gyakorlatban is kezelhető lehet.*

Valós példák

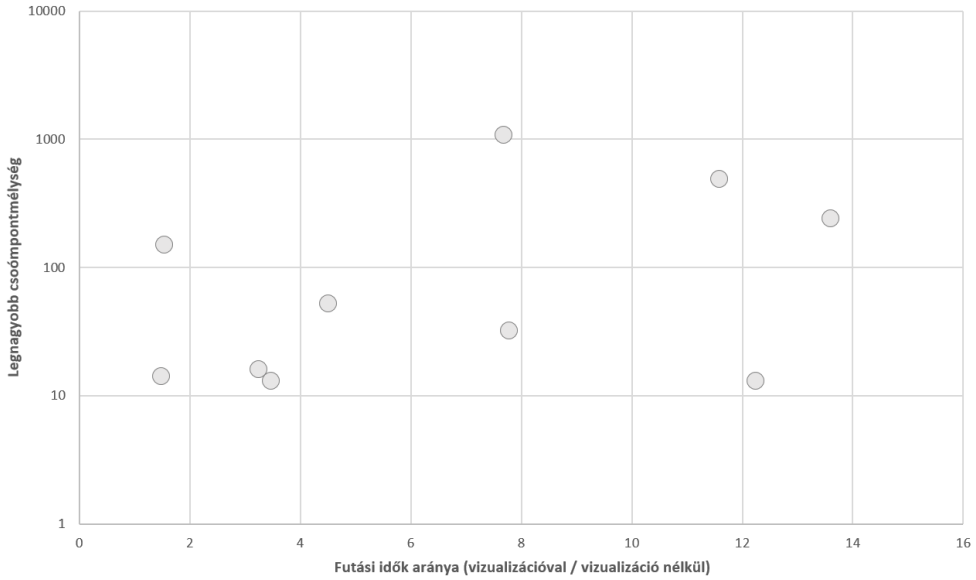
A valós példákkal történő mérésekhez 9 metódust választottam ki nyílt forráskódú projektekből, illetve egyet a korábbi tapasztalatok során említett ipari partnertől származó CMS rendszerből. A 9 metódus közül 5 olyan projektekből ([24],[11]) származik, melyeken a Microsoft Pex képességeit már korábban vizsgálták [31], így jó alapot adnak a vizualizáció képességeinek felmérésére is. A másik 4 metódus közül 2 egy biológiai használatra tervezett algoritmusokat és adatstruktúrákat tartalmazó projektből [25], míg a másik 2 metódus a Couchbase [18] NoSQL adatbázis-kezelő rendszer .NET-es változatából való. Az egyes metódusokat a projektekből pedig úgy állítottam össze, hogy a komplexitásuk változatos legyen, ám jelentsen kihívást is a vizualizációs eszköz számára (pl. mély bejárandó kód, több egymásba ágyazott ciklus).

A valós példák mérése során ugyanazokat a lépéseket követtem, mint a mesterségesek esetén, így tehát először összevetem a futási idők arányának kapcsolatát a legmélyebb csomóponttal, majd a csomópontszámmal, végül pedig a forráskód hosszával. Így megállapítható, hogy a mesterséges példákból történő következtetések megállják-e a helyüket valós környezetben is. Fontos kiemelni, hogy jelen vizsgálat során nem bontottam csoportokra a metódusokat, hiszen a megvalósított, valós funkcióik által nem voltak alkalmasak jól elkülönülő kategorizálásra (a projektjükön kívül). A vizsgált metódusok részletei és a mérésükből kapott eredmények tehát az alábbi táblázatban találhatók (a jelölések megegyeznek a mesterséges példánál alkalmazottakkal).

Projekt	Metódus	F [mp]	FV [mp]	A	CS	M	LOC
Couchbase	AppendData	11,994	54,082	4,509	676	52	49
Couchbase	Compare	6,102	47,527	7,789	989	32	54
CMS	CreateUser	9,991	135,954	13,61	2104	238	5
Bio	MapAcid	244,678	377,82	1,544	717	149	11
Bio	Compare	2,697	4,033	1,495	65	14	8
DSA	IsPrime	3,114	10,135	3,255	36	16	9
DSA	Avl.Remove	2,491	8,668	3,48	21	13	6
DSA	Heap.Remove	0,529	6,478	12,25	40	13	20
GitSharp	Merge	173,493	1334,22	7,69	7322	1073	62
GitSharp	MyersDiff	37,965	440,4	11,6	3197	488	23

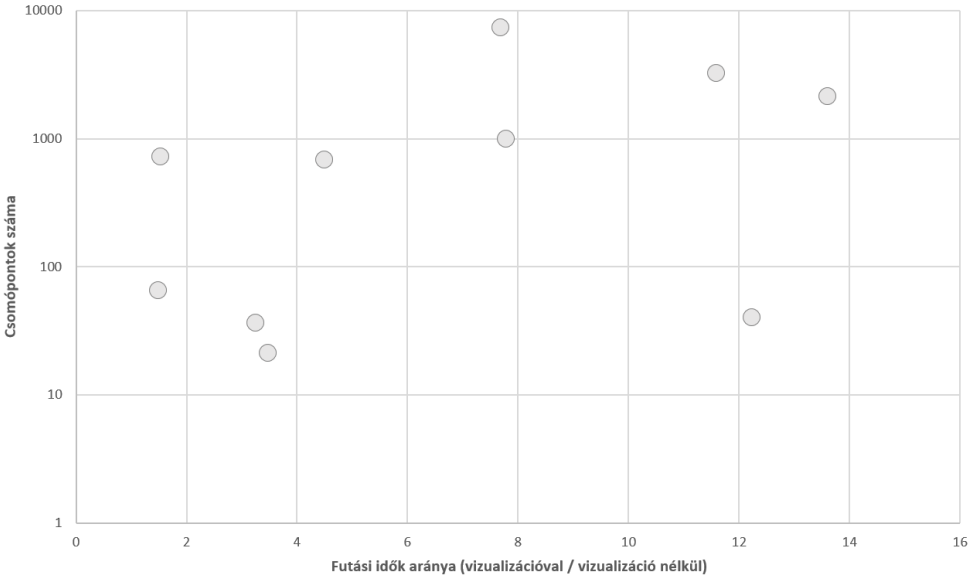
Tekintsük az 5.9. ábrát, mely a legmélyebb csomópont és a futási idők arányának vizsgálatát mutatja be. Megfigyelhető, hogy az előző vizsgálathoz képest itt a futási idő arányai jobban szétszóródnak, és öt feletti értékeket is felvesznek. Ennek oka, hogy a valós példák több és nagyobb iterációszámú ciklust tartalmaznak, ellenben korlát nélküli ciklust nem. Látható továbbá, hogy egyes felfedezések azonos legmélyebb csomópont esetén nagyobb arányú növekedést produkáltak, melynek magyarázata, hogy ezekben az esetekben – részletes vizsgálatok után megállapíthatóan – több hasonlóan mély ág is található a fában, amelyek többszöri bejárása lassítja a vizualizációt.

Érdemes megvizsgálni a csomópontok számának és a két futási idő arányának kapcsolatát is, melyet az 5.10. ábra mutat be. Az eredmények alapján megállapítható, hogy míg



5.9. ábra. A legmélyebben fekvő csomópont és a futási idők arányának vizsgálata valós példákon.

a csomópontok száma több nagyságrenddel nő pár tízről közel tízezerre, addig a futási idő mindössze duplájára növekszik a futások nagyrésze esetén. Ilyen futási idejű felfedezések esetén azonban ez még szintén elfogadható mérték lehet a mérnöki gyakorlatban. Szintén megfigyelhető, hogy egyes felfedezések esetén kisebb csomópontszám esetén nagyobb a futási idő növekedés. Ennek oka egyaránt a mélységgel kapcsolatos: több, viszonylagosan mély ág van, melyek lassítják a vizualizációt.



5.10. ábra. A csomópontszám és a futási idők arányának vizsgálata valós példákon.

A valós példák vizsgálatánál is érdemes megemlíteni, hogy – hasonlóan a mesterséges esetekhez – a forráskód hosszának és a futási idő növekedése között nem volt tapasztalható összefüggés, így itt sem tárgyalom ezt részletesen.

Az eredmények és a metódusok kódjának részletes elemzésével megállapítható, hogy a ciklusok a valós példák esetében is jelentős befolyásolják a futási idő növekedését vizualizációval kiegészített esetben. A végrehajtási fákat és a metódusok kódját megvizsgálva nagy

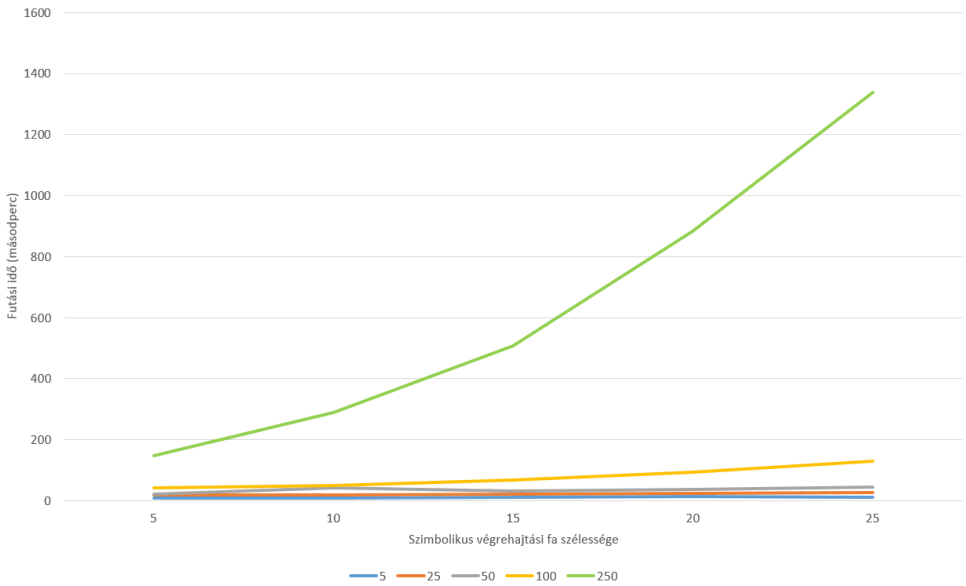
mélység két indokból következett be: 1) hosszú ciklusok, 2) hash függvények. Mindkettő azonban a szimbolikus végrehajtások kihívásai között említhető, így ezek megoldásával a vizualizáció futási idő növekedése is tovább csökkenthető.

Összességében elmondható, hogy kiugróan magas futási idő növekedés nem volt tapasztalható (mint például a mesterséges példák 40-szeres, vagy akár 80-szoros növekedése), hiszen a valós példák nem tartalmaznak korlátlan ciklusokat. Az eredmények azonban valamelyest nagyobb mértékű futási idő növekedést mutattak a valós esetben. Figyelembe véve ugyanakkor a futási idők nagyságrendjét, a növekedés mértéke még szintén elviselhető lehet a mérnöki gyakorlatot tekintve.

5.3.2. Skálázhatóság vizsgálata

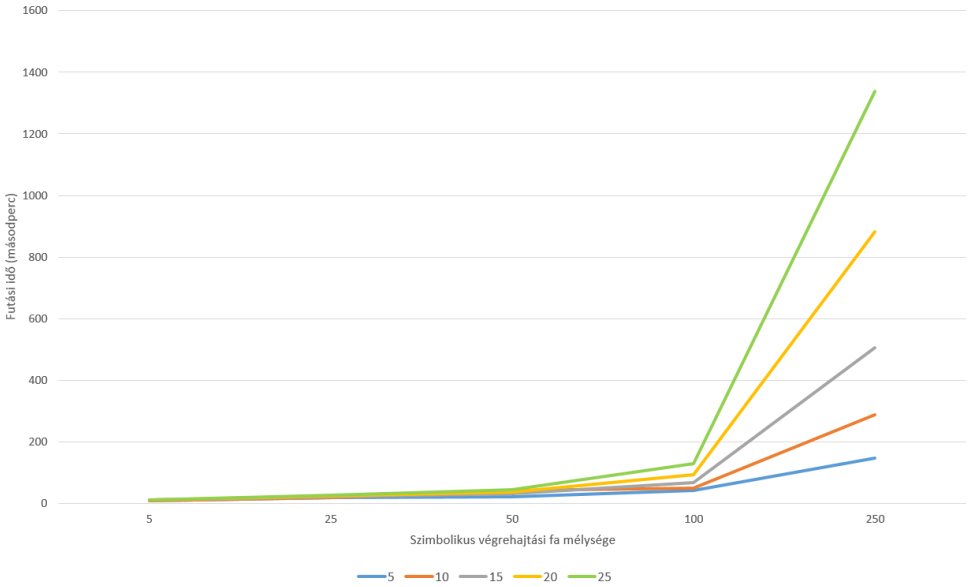
A skálázhatóság kvantitatív mérésére létrehoztam egy olyan paraméterezhető kódgenerátort, amelynek megadható, hogy a kód ellenőrzése során létrejövő szimbolikus végrehajtási fa mélysége és szélessége milyen értéket vegyen fel. Ezt a nagyfokú rugalmasságot a generált kódban lévő `switch` szerkezet (szélesség) és annak egyes ágaiban hívott `for` ciklus (mélység) segítségével értem el. Ezzel a struktúrával jól vizsgálható a skálázhatóság, hiszen a fa felépítési jellege így állandó marad, és csak a két említett tulajdonság változik. A vizsgálatok során tehát a következő paraméterek összefüggéseit vizsgáltam meg: fa mélysége, fa szélessége, futási idő vizualizációval, csomópontszám. Az alábbiakban a futási idők alatt a vizualizációval mért időket értem.

A mérésekhez a szimbolikus végrehajtási gráf mélységének értékeit 5 és 250 között választottam, míg a szélesség értékeit 5 és 25 között, melyek felosztásával összesen 25 mérést hajtottam végre. A két értéktartomány – a futási idő mérése során és a korábban tapasztaltak szerint is – átlagosan jól lefedi a valóságban előforduló méreteket, komplexitásokat.



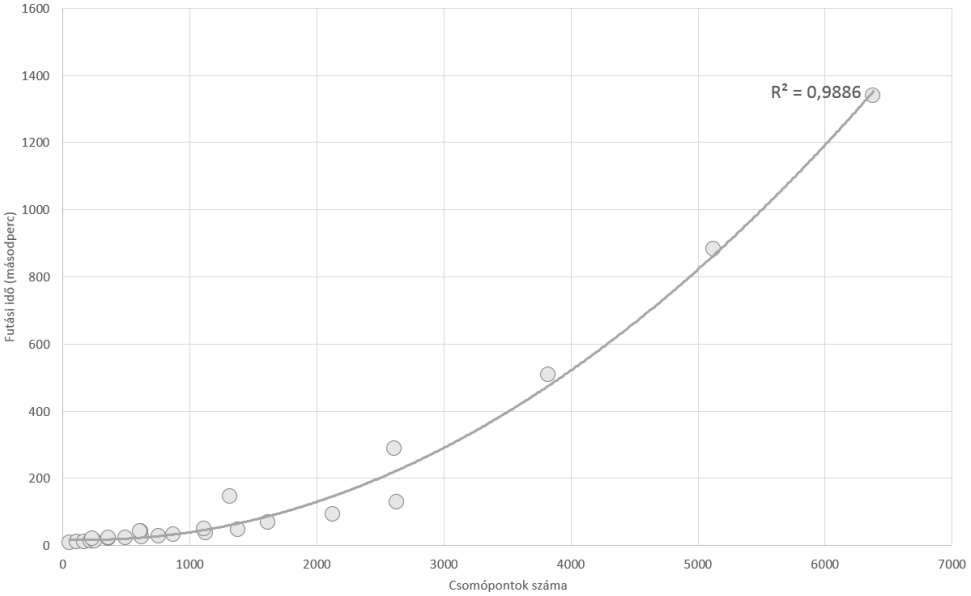
5.11. ábra. Futási idő és szélesség összefüggése különböző mélységű esetekben.

Tekintsük az 5.11. ábrát, mely megmutatja, hogy különböző mélységek esetén milyen kapcsolat van a futási idő és a fa szélessége között: jól látható, hogy minél nagyobb a fa mélysége annál meredekebb növő az összefüggés. Ezt jól mutatja a 100 és a 250 mélységű esetek közötti lényeges különbség is. Érdemes megfigyelni, hogy 100 mélység esetén is még szélességtől függetlenül (így akár 2500 csomópont esetén is) bőven 200 másodperc alatt marad a futási idő.



5.12. ábra. Futási idő és mélység összefüggése különböző szélességű esetekben.

Az 5.12. ábrán látható, hogy a különböző szélességű esetekben milyen összefüggés van a futási idő és a fa mélysége között. Az eredmények alapján látható, hogy kisebb mélységek esetén hasonló a növekedés különböző szélességekkel, ám 250 mélységnél már nagy ugrás érzékelhető a futási időben, ahogy azt az előzőekben is megállapítottam. Az ábrán továbbá jól látható, hogy 250 mélység esetén már a szélesség is nagymértékben befolyásolja a futási időt, hiszen annyiszor kell bejárni az adott mélységet, ahány széles maga a fa. Ez alátamasztja a magyarázatát annak is, hogy a vizualizációs időtöbblet méréseinél a vizualizációs futási idő egyes esetekben miért nőtt jelentősen azonos csomópontszámnál és azonos legmélyebb csomópont esetén.



5.13. ábra. Futási idő és csomópontok számának összefüggése.

Az eddigi eredmények alapján jól látható, hogy mind a mélység, mind a szélesség befolyásoló tényező a futási idő tekintetében. Ezek alapján tehát érdemes lehet vizsgálni a

csomópontok számának és magának a futási időnek az összefüggését, amelyet az 5.13. ábra mutat be. A vizsgálat alapján elmondható, hogy a két változó értéke erősen összefügg, hiszen a fenti ábrán is látható négyzetesen polinomiális trendvonal közel teljesen illeszkedik az értékekre. A négyzetes összefüggés pedig jelen esetben egy jó skálázódási faktornak tekinthető.

5.3.3. Mérések értékelése

A mérések alapján megállapítható, hogy 1) a Pex futási idejét változó mértékben, de befolyásolja a vizualizációs komponens, illetve 2) a vizualizációs komponens jól skálázódik nagy gráfok megjelenítése esetén is.

A futási idő befolyásolását tekintve megállapítható, hogy zavaróan nagy mértékben csak problémás esetekben (pl. hosszú vagy korlátlan ciklusok, hash függvények) növekedett meg a végrehajtás ideje vizualizációval. Ennek kiküszöbölésére érdemes lehet az ilyen, potenciálisan túl mély ágakat detektálni és az adatgyűjtés és megjelenítés során absztrakcióval kezelni őket. Az esetek nagyobb részében tehát a futási idő növekedésének mértéke az eszköz mérnöki alkalmazását nem akadályozza.

A vizualizációs komponens skálázódását tekintve a négyzetes összefüggés megfelelő lehet a mérnöki gyakorlatban is, hiszen egységtesztelés során a kis méretű egységek ellenőrzésével 7000 csomópontnál nagyobb szimbolikus végrehajtási gráfok kis valószínűséggel keletkeznek. Ez alól kivételek természetesen a már említett problémás esetek, de azok absztrakciójával ez a skálázódási probléma is áthidalható.

A méréseket tekintve továbbá fontos szót ejteni az eredmények érvényességéről. Az alábbiakban felsorolok a felmerülő, érvényességet esetlegesen megkérdőjelező tényezőt és annak megoldására alkalmazott módszert.

- A méréseket manuálisan hajtottam végre, de az eredmények érvényessége érdekében minden mérést háromszor ismételtem meg, és a kapott eredmények átlagát használtam fel az elemzésekhez.
- A futási idő mérése során alkalmazott mesterséges példák egy másik kutatásból származnak, így az esetlegesen bennük lévő hibákat – melyek befolyásolhatták volna a mérési eredményeket – a megfelelő metódusok kiválasztásával és a forráskódjuk részletes átvizsgálásával védtem ki.
- A valós példákon keresztül történő futási idő mérései magukban hordozhatják azt a problémát, hogy – a körültekintő válogatás ellenére is – valós, ipari környezetben ritkán vagy egyáltalán nem fordulnak elő, így nem felelnek meg céljuknak. Ezért a metódusok közé olyanokat is választottam, amelyeken már a Pex képességeit más kutatások során is vizsgálták és a belőluk kapott eredményeket relevánsnak tekintették.
- Az aránylag kis számú valós példa által lehetséges, hogy a kiválasztott példák nem fedik le megfelelő mértékben a valós viszonyokat, így más metódushalmazt véve eltérő eredményeket kaptam volna. Ennek kivédésére a metódusok közé a Pex korábbi kutatásaiból származó projektek mellett több, eltérő jellegű nyílt forráskódú projektből emeltem ki metódusokat a vizsgálatokhoz.

5.4. A módszer és az eszköz korlátai

Az alábbiakban a kidolgozott vizualizációs módszer és a hozzá elkészült eszköz korlátait tárgyalom, mellyel behatárolhatók a javítandó, fejlesztendő területek.

A bemutatott vizualizációs módszer a leírt tulajdonságaival nem képes biztosítani **ciklusok, rekurziók, ismétlődő műveletek** esetén azok felismerését és absztrakcióját. Ennek következményeképp a szimbolikus végrehajtási gráf (ahogy a mérések során elő is fordult) nagyon nagy mélységekbe eljuthat, amely átláthatatlanná teszi a munkát és lassítja a működést. Ennek megoldására a következő alfejezetben említek egy kutatást, melynek során pontosan ezen kérdéssel foglalkoztak.

A vizualizációs technika **tesztelési folyamatba történő integrálása** nem triviális feladat, hiszen az elkészült eszköz kevésbé hatékony felhasználása akár lassíthatja is a munkát annak támogatása helyett. Ehhez a következő fejezetben mutatok be egy lehetséges megoldást, amely segíti az eszközt felhasználó mérnököket a szimbolikus végrehajtás vizualizációjának hatékony felhasználásában a tesztek generálása során.

A nem eszköz-specifikus korlátok között említhetők végül magából a **szimbolikus végrehajtásból eredők**, melyek közül az alábbiak a legfontosabbak [7]:

- Többszálúság kezelése
- Környezettel való kommunikáció
- Külső hívások kezelése
- Lebegőpontos számok kezelése
- Állapottér robbanása
- Kényszermegoldók optimalizációja

Az elkészült implementációra vonatkozóan elsőként említhető az **optimalizáció problémája**. A mérések alapján látható, hogy a vizualizációja befolyásolja a futási időt, ám ennek megoldása további részletes vizsgálatokat igényel.

Eszköz-specifikusan fontos kiemelni, hogy az adatgyűjtő komponens (*PexExtensions*) a **Pex által rendelkezésre bocsájtott adatokat ismeri csak** és tudja felhasználni, amelyből több hátránya is származik: 1) A Pex kiajánlásaitól függ, hogy egyáltalán milyen adat kapható meg belőle, lévén nem nyílt forráskódú rendszer, 2) a kiajánlások által korlátozottan fér csak hozzá a bejárt forráskódhoz, így nem mindig a lehető legpontosabb a leképezése a csomópontoknak , 3) a Pex újabb verziójának megjelenésekor nem garantálható, hogy a komponens továbbra is működni fog egy esetleges interfésztörés[3] esetén.

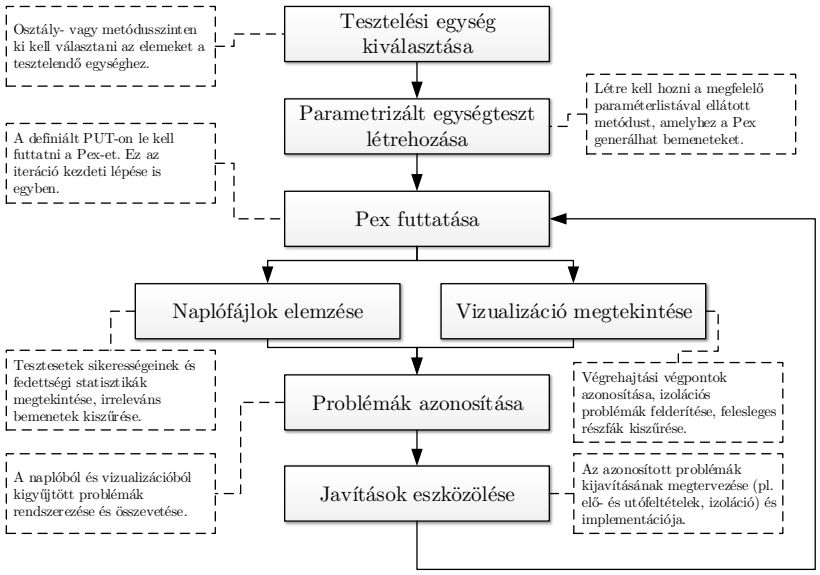
5.5. Tesztelési hatékonyság növelése

Az alábbiakban bemutatok egy módszert, amellyel lehetővé válik az egységtesztelés hatékonyságának növelése a Microsoft Pex és a szimbolikus végrehajtás vizualizációjának segítségével. A módszer lényege az iteratív és inkrementális megközelítés, mellyel minden lépésben egyre közelebb jut a tesztelést végző mérnök a teljes blokklefedettséghez (ami azonban nem feltétlenül elérhető). A követendő lépéseket tehát a 5.14. ábra mutatja be.

A folyamatot a saját tapasztalataim alapján építettem fel, hiszen korábbi munkáim során sikerült azonosítanom több olyan kihívást is, amelyre ez a követendő módszer megoldást adhat.

A módszert a mérések során már említett Couchbase rendszer `Compare` metódus vizsgálata során alkalmaztam, melyben három iteráció után a kezdeti 52% blokklefedettségből a második iterációban már 64% lett, a harmadikban pedig 82%-osat sikerült előállítanom, ami a komplexitást tekintve jónak mondható. A fedettség növekedése meglehetősen gördülékenyen elérhető volt, köszönhetően a rendelkezésre álló adatoknak mind a naplófájlokból,

[3]Egy korábbi verzióban használt interfész megváltozik, így a régire építő alkalmazások nem képesek együttműködni az új verzióval.



5.14. ábra. Ajánlott tesztelési lépések a Pex és vizualizációjának használatával.

mind pedig a vizualizációból. A második iteráció végén a nagyobb fedettségi ugrást a vizualizációból kifejtett hiányzó részfa segítségével értem el.

Összességében elmondható, hogy a két adatforrás jól kiegészíti egymást, így a mérnöki gyakorlatban a kettőt együtt alkalmazva érdemes a tesztelést végezni.

5.6. Kapcsolódó irodalom

A szimbolikus végrehajtás vizualizációját több kutatás is megcélozta már, azonban azok esetén eltérő volt a motiváció. Hentschel és társai a nem korlátos szimbolikus végrehajtás vizualizációjának problémájára adnak megoldást [16] a végrehajtási fák ágainak absztrakciójával korlátlan ciklusok és rekurzív hívások esetén. Az általuk készített eszköz (SED) egy szimbolikus végrehajtás alapú *debugger*, ami vizuálisan képes megjeleníteni a nyomkövetés közbeni lépéseket. A kutatásukban említést tesznek a vezérlési-folyam gráfok (CFG – Control Flow Graph) és a szimbolikus végrehajtási fák közötti kapcsolatról. A kutatás célja azonban a debugger eszköz használhatóságának növelése volt.

Egy másik hasonló, de korábbi kutatás [15] szintén a nyomkövetés témakörétől közelíti meg a szimbolikus végrehajtás vizualizációját. Hähnle és társai hozzám hasonlóan konkrét leírást adtak a vizualizációról és a csomópontok jelentéséről, ugyanakkor itt is a debugger eszköz továbbfejlesztése volt a fő motiváció, míg esetemben a tesztbemenet-generálás hatékonyságának növelése volt a cél.

Elmondható tehát, hogy a szimbolikus végrehajtás vizualizációjának kutatása egy aktív terület. Ugyan a vizualizáció motivációja eltérő lehet, a különböző nézőpontból született ötletek, eredmények azonban segíthetik a további kutatásokat, így növelve a szimbolikus végrehajtás alkalmazhatóságát a mérnöki gyakorlatban.

6. fejezet

Kitekintés: automatikus izoláció

Az egységtesztelés egyik legnagyobb kihívása a tesztelt egységek izolációjának megfelelő megvalósítása. A munkám során kidolgozott módszer és a hozzá tartozó eszköz támogatja ugyan az egységek izolálását az egységből történő kilépés megjelölésével, de a hatékony munkához további lépések szükségesek. Ennek egyik lehetséges megoldásaképp alkottam meg az automatikus izoláció absztrakciós technikáját.

6.1. Motiváció

Programok kis egységeinek ellenőrzése jelenleg is aktív kutatási terület. Vanoverberghe és szerzőtársa a problémát a kompozit szimbolikus végrehajtás felől közelítette meg [30], melynek lényege, hogy az egyes metódusokat külön-külön, de akár párhuzamosítva ellenőrzik, majd az eredmények kompozíciója adja meg a végső eredményt. Ez azonban nem triviális művelet, hiszen könnyedén találhatók olyan esetek, amikor a metódusok egyes függései által sosem elérhető egy utasítás, ám ez csak az adott metódust vizsgálva nem kideríthető. A megoldásra magát a kompozit végrehajtást tranzíciós rendszerként modellezték, amellyel egy formális bizonyítást adtak a kidolgozott módszerük pontosságára és előrehaladási képességére.

Godefroid az úgynevezett *Micro execution* technikát dolgozta ki a kis egységek automatikus végrehajtásának problémájára [13]. Kutatásában egy olyan módszert alkotott, ami biztosítja a program bármely utasításának egyedüli lefuttatását, hasonlóan egy debuggerhez, ugyanakkor itt a programot bármelyik pontjától lehet indítani. Az izoláció megvalósítása memória szintjén történik, és az egyes szükséges memóriahelyeket kitöltő adatokat különböző algoritmusok generálják. Godefroid kitér a technika egységek verifikációjára történő használatára is, megemlítve a kompozit szimbolikus végrehajtást, mint használandó módszert.

Munkám során én is megtapasztaltam az automatikus tesztelés során az izolációs probléma nehézségét mind a már említett Petri-háló modellező, mind a tartalomkezelő-rendszer esetén. A megfelelő függések manuális megtalálása és kitöltése számomra is időigényes feladat volt ezen komplex szoftverekben. Ez nehezítheti és lassíthatja is a tesztelés folyamatát, amely a mérnöki gyakorlatban nagymértékű hátrányt jelenthet.

6.2. Izolációs algoritmus

A mérnöki gyakorlatban egységtesztelés során tehát gyakran felmerülő probléma az izoláció megvalósítása. Ebben segít a függőségek megjelölése a szimbolikus végrehajtási gráfon, azonban további lépésként megvalósítható egy olyan technika, ami képes a szimbolikus végrehajtás során felismerni és automatikusan izolálni azokat. Ez a technika a vizualizációs

komponens kiegészítéseként megvalósítható. A hívási absztrakció ötletét a következő (5.) algoritmus írja le.

Algoritmus 5 A hívási absztrakciókat bevezető algoritmus

```
1: // Minden olyan hívás esetén hívjuk, amikor kilépünk a vizsgált egységből.
2: function GenerateCallAbstraction(method, putParamList)
3:     if HasReturnValue(method) then                    ▷ ha van visszatérés
4:         r := TypeOf(ReturnValueOf(m));                ▷ elkérjük a típusát
5:         putParamList.Add(r);                          ▷ kiegészítjük a PUT-ot
6:         // Minden m hívás esetén ezt adjuk vissza
7:         for each call to m := { return putParamList[r]; };
8:     end if
9: end function
```

Ez a hívási absztrakció a definiált vizualizációs technikába ott illeszkedik bele, amikor az egységből való kilépést észleljük (4.2.5.). Ekkor a szimbolikus végrehajtást le kell állítani, majd kiegészíteni a parametrizált egységteszt paraméterlistáját a megfelelő változóval és újraindítani a szimbolikus végrehajtást. Mindezt úgy, hogy közben egy absztrakciót vezetünk be, hogy minden, az adott izolált metódus felé történő hívás felé a paraméterlistáról kapott érték adódjon vissza.

A megoldás segítségével megkönnyíthető a szimbolikus végrehajtás alkalmazása a mérnökök számára, hiszen nem kell feltérképezniük az adott metódus függőségeit. Ezt követően a vizualizáció megjelenítésekor a *mérnök számára jelezni kell* az absztrahált hívásokból visszaadott értékeket, hogy eldönthesse azok tekinthetők-e valósaknak vagy sem. Továbbá, ezen értékek később felhasználhatók az adott függőség egységtesztelése során is.

A technika segítségével tehát áthidalhatók a nem megfelelően izolált komponensekből eredő generálási hiányosságok, így tehát megválaszoltam a kitűzött kutatási kérdések közül a negyediket is. Az algoritmus működésének demonstrálására tekintsük az alábbi példát.

8. példa. *Legyen a példakód a már korábban, a 7. példában említett* `Current.Foo` *és* `External.Bar` *metódusok. A vizsgálni kívánt egység itt is álljon egyedül a* `Current.Foo` *metódusból. Ennek parametrizált egységtesztje a legegyszerűbb esetben a következő.*

```
public int FooTest(Current target, int a, bool b)
{
    target.Foo(a,b);
}
```

A szimbolikus végrehajtás vizualizációval támogatva az első lefutás során felfedezi az egységből való kilépést a `Bar` *metódus hívásakor. Ekkor lefut az 5. algoritmus, amely előállítja az izolációs logikát és kiegészíti a paraméterlistát a parametrizált tesztmetódusban, amely így a következőképp változik meg. A szimbolikus végrehajtást ezek után újra kell indítani.*

```
public int FooTest(Current target, int a, bool b, bool bar)
{
    // A Moles izolációs keretrendszer szintaktikájával
    External.AllInstances.Bar = (parameter1) => { return bar; };
    target.Foo(a,b);
}
```

A kódrészletben a Moles izolációs keretrendszer [22] szintaktikáját használtam fel annak kifejezésére, hogy az `External` *osztály minden példányának* `Bar` *metódusának hívásakor a megadott absztrakciós kódrészlet induljon el.*

6.3. Korlátok

A módszer korlátai között biztosan említhetők az alábbiak, azonban fontos megemlíteni, hogy a megoldáshoz jelenleg még nincs kísérleti implementáció, így nem zárható ki, hogy további korlátok, hátrányok is felmerülnek a valós használat során.

- A kompozit szimbolikus végrehajtás során említett probléma itt is felmerül, hiszen nem deríthető ki, hogy az absztrakció miatt behozott paraméter által biztosított **visszatérési érték valójában előfordulhat-e** a program izoláció nélkül futásakor.
- A módszer számára erős korlátot jelent a szimbolikus **végrehajtás újraindítása** minden egyes, újonnan felfedezett függőség esetén. A módszer kijavítható például egy előzetes statikus analízissel a külső függőségek felderítésének céljából.
- Problémát jelenthet a **konstruktorok hívása**, hiszen ilyenkor példányt kell visszaadni. Ennek megoldása azonban már inkább tekinthető implementációs részletnek, így részletesen nem tárgyalom.
- Szintén implementációs szintű korlátot jelenthet a **referencia típusok** kezelése. Lehetséges olyan metódushívás mely nem ad vissza értéket, viszont a metódus törzsében a referencián keresztül mégis módosít értéket.

Összességében azonban elmondható, hogy a módszer jó kiegészítése lehet a szimbolikus végrehajtás vizualizációjának és segítheti a tesztbemenet-generálást végző mérnököt a hatékony munkában az imént említett, jelenlegi korlátai ellenére is.

7. fejezet

Összefoglalás

A fejezetben összefoglalom a munkám során elért eredményeket, illetve ismertetem a továbbfejlesztési lehetőségeket.

7.1. Eredmények

Dolgozatom során az automatikus tesztbemenet-generálás, azon belül pedig a szimbolikus végrehajtás mérnöki alkalmazhatóságának problémáját vizsgáltam meg. A kutatásom fő kérdése a következő volt: **Hogyan támogatható a szimbolikus végrehajtás alapú tesztbemenet-generálás mérnöki alkalmazása?** Az alábbiakban ezen kérdésre, illetve a belőle fakadó alkérdésekre adott válaszokat foglalom össze.

Kihívások azonosítása

Munkám első fázisában a korábbi tapasztalatok alapján és a kapcsolódó kutatások alapján azonosítottam az automatikus tesztbemenet-genárálás kihívásait, melyek közül napjainkban az egyik legnagyobb jelentőségű a mérnöki alkalmazhatóság kérdése. A lehetséges, támogató technikák közül pedig kiválasztottam egyet, a vizualizációt, melyet részletesen is kidolgoztam.

Vizualizációs technika definiálása

A kutatásom kérdései közül háromra maga a szimbolikus végrehajtás vizualizációjának definiálása ad választ. Ez a három kérdés a következő volt.

- A lefutások milyen jellegű vizualizációja segíti a megértést?
- Milyen jellegű információk nyerhetők ki?
- Milyen információkat érdemes megjeleníteni?

A vizualizáció definiálása során mindhárom kérdésre sikerült választ adnom. A megjelenítés jellegét tekintve egy gráf alapú reprezentációt választottam, melynek külső megjelenését részletes leírással adtam meg. A szimbolikus végrehajtás során kinyerhető információk alapján, az áttekinthetőséget is figyelembe véve, megadtam azokat a metaadatokat, amelyek fontosak a tesztelést végző mérnökök számára a használhatóság megkönnyítésének szempontjából. Ezen információk megjelenését pedig részletesen tárgyaltam a definiált gráfreprezentációra vonatkozóan.

Vizualizációs eszköz a definált alapján

Munkám során elkészítettem a definiált vizualizációs technikához illeszkedő eszközt, a *DSEVisualizer*-t. Az eszköz segítségével a gyakorlatban példákon keresztül is bemutattam a működését, illetve mérések segítségével mutattam meg az eszköz hatékonyságát gyakorlati alkalmazás során is. Dolgozatomban mindemellett röviden bemutattam a szimbolikus végrehajtás vizualizációjához kötődő kutatási eredményeket.

Alkalmazási lehetőségek kidolgozása

Dolgozatomban bemutattam a munkám során kidolgozott vizualizációs technika lehetséges gyakorlati alkalmazásait is. Egyrészt kidolgoztam egy esetlegesen követendő tesztelési metodikát, melyben a vizualizáció által javítható a Pex segítségével történő automatikus tesztbemenet-generálás hatékonysága. Másrészt pedig, dolgozatom végén az utolsó feltett kutatási kérdésre is választ adtam: Hogyan hidalhatók át a nem megfelelően izolált komponensekből eredő generálási hiányosságok? A probléma megoldásaként kidolgoztam egy automatikus hívási absztrakciót a vizualizációs technikát alapul véve. A módszer a tesztelt egységek függőségeinek automatikus leválasztásában vállal szerepet, amellyel így segíthet a mérnökök számára a tesztelési folyamat felgyorsításában

7.2. Továbbfejlesztési lehetőségek

A szimbolikus végrehajtás vizualizációja sokat segítheti a mérnöki alkalmazás megkönnyítésében, azonban a munkám során több korlátot vagy lehetőséget is felfedtem, amelyek új irányokat vetítenek elő. A mérnöki alkamazás hatékonyságát a munkám következő továbbfejlesztési irányai növelhetik.

- Ciklusok, rekurziók, mély futások kezelése a végrehajtási fában, mellyel ilyen esetekben áttekinthetőbbé válik a megjelenített gráf és optimalizálható a futás.
- Az adatgyűjtési algoritmusok optimializációjával minimálisra lehetne csökkenteni a vizualizáció beavatkozását a tesztbemenet-generálás működésébe.
- További részletes, ipari környezetben való használhatóságot alátámasztó esettanulmányok kidolgozása.
- A Pex-en kívül más szimbolikus végrehajtást alkalmazó eszköz csatlakoztatása a vizualizációs komponenshez.
- Az automatikus hívás absztrakció ötlete ötvözhető a kompozit szimbolikus végrehajtással és a vizualizációval, amelyekkel együttesen a tesztelő mérnökök válláról további feladatok vehetők el, így megkönnyítve és gyorsítva munkájukat.

Ábrák jegyzéke

Irodalomjegyzék

[1] IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-1993*, pages i–, 1994.

[2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High–coverage tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[5] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM.

[6] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[7] Ting Chen, Xiao song Zhang, Shi ze Guo, Hong yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758 – 1773, 2013.

[8] Honfi Dávid, Micskei Zoltán Imre, and Vörös András. Struktúra alapú tesztelés vizsgálata egy komplex szoftverrendszerben, szakdolgozat, BME. 2013.

[9] Jonathan de Halleux and Nikolai Tillmann. Parameterized Unit Testing with Pex. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 171–181. Springer Berlin Heidelberg, 2008.

[10] Leonardo de Moura and Nikolaj Bjorner. Z3: An Efficient SMT Solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[11] eqqon GmbH. GitSharp. http://www.eqqon.com/index.php/GitSharp, 2013.

[12] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 291–301, New York, NY, USA, 2013. ACM.

[13] Patrice Godefroid. Micro Execution. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 539–549, New York, NY, USA, 2014. ACM.

[14] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, January 2012.

[15] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A Visual Interactive Debugger Based on Symbolic Execution. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 143–146, New York, NY, USA, 2010. ACM.

[16] Martin Hentschel, Reiner Hähnle, and Richard Bubel. Visualizing Unbounded Symbolic Execution. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs*, volume 8570 of *Lecture Notes in Computer Science*, pages 82–98. Springer International Publishing, 2014.

[17] IEEE. IEEE Standard for Software and System Test Documentation. *IEEE Std 829-2008*, 2008.

[18] Couchbase Inc. Couchbase Lite .NET. https://github.com/couchbase/couchbase-lite-net, 2014.

[19] ISTQB. Foundation Level Syllabus, 2011.

[20] John Ellson and Emden Gansner and Lefteris Koutsofios and Stephen North and Gordon Woodhull and Short Description and Lucent Technologies. Graphviz - open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[21] Cseppentő Lajos and Micskei Zoltán Imre. Szimbolikus végrehajtást használó tesztgeneráló eszközök összehasonlítása, szakdolgozat, BME. 2013.

[22] Microsoft Research. Unit Testing with Microsoft Moles . http://research.microsoft.com/en-us/projects/pex/molestutorial.pdf, 2010.

[23] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[24] CodePlex project. Data Structures and Algorithms. https://dsa.codeplex.com/, 2008.

[25] CodePlex project. .NET Bio. http://bio.codeplex.com/, 2013.

[26] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[27] Xiao Qu and B. Robinson. A Case Study of Concolic Testing Tools and their Limitations. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 117–126, Sept 2011.

[28] Nikolai Tillmann and Jonathan de Halleux. Pex–White Box Test Generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin Heidelberg, 2008.

[29] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 385–396, New York, NY, USA, 2014. ACM.

[30] Dries Vanoverberghe and Frank Piessens. Theoretical Aspects of Compositional Symbolic Execution. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software Engineering*, volume 6603 of *Lecture Notes in Computer Science*, pages 247–261. Springer Berlin Heidelberg, 2011.

[31] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proc. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, November 2013.