Szimbolikus végrehajtás futásidejű hibadetektálásra és refaktoring tevékenységek vizsgálata egy új adatbázis alapján

Kádár István

Szoftverfejlesztés Tanszék
Szegedi Tudományegyetem

Szeged, 2017

Témavezető:

Dr. Ferenc Rudolf

Ph.D. értekezés tézisei

Szegedi Tudományegyetem

Informatika Doktori Iskola

Bevezetés

Nagyméretű, megbízható és robusztus szoftverrendszerek előállítása a szoftverfejlesztés egy nagy kihívása manapság. Az ipari szoftverfejlesztésben a fejlesztőknek tipikusan gyorsan kell jó megoldást adni egy problémára és a forráskód minősége gyakran csak másodlagos az időnyomás miatt. Másrészről a forráskód minősége egy nagyon fontos tényező, mert egy túl komplex és nehezen karbantartható kód több programhibához és nehezebben fejleszthető rendszerhez vezet. Az értekezés mögötti kutatási munkát az inspirálta, hogy hatékonyabban és könnyebben tudjunk magas minőségű szoftverrendszereket előállítani az ipari szoftverfejlesztésben, amivel a megrendelők és a végfelhasználók életét könnyebbé és hatékonyabbá tudjuk tenni.

A disszertáció két témát ölel fel: *szimbolikus végrehajtás felhasználását futásidejű hibák detektálása céljából*, valamint *refaktoring tevékenységek vizsgálatát a gyakorlatban*. Mindkét témakör a forráskód minőség területén helyezhető el.

A *szimbolikus végrehajtás* egy programelemzési technika, amely feltérképezi és végrehajtja a program lehetséges végrehajtási útvonalait úgy, hogy az input adatokat ismeretlen változókként, úgynevezett szimbolikus változókként kezeli. A szimbolikus végrehajtás fő felhasználási területei a magas lefedettséget eredményező tesztinput generálás, valamint olyan inputok generálása, ami programhibához vezet és annak leállását eredményezi. A módszer képes olyan programhibákat megtalálni, amiket rendkívül nehéz és költséges lenne teszteléssel detektálni, valamint exponenciálisan növekedne a javítás költsége is, amennyiben nem fedeznénk fel ezeket a problémákat a program futtatása előtt. Az értekezésben futásidejű programhibák detektálására fókuszálunk (mint például null pointer dereferencia, tömb túlindexelés, nullával való osztás, stb.) a kritikus végrehajtási útvonalak feltérképezésével, amelyet szimbolikus végrehajtással teszünk meg. Jelenleg az egyik legnagyobb kihívás a szimbolikus végrehajtás területén a lehetséges végrehajtási útvonalak óriási száma, ami exponenciálisan növekszik.

Az értekezésben a fenti problémára adunk lehetséges megoldást azzal, hogy a szimbolikus végrehajtást metódus szinten végezzük el, azaz a program minden metódusára külön-külön indítunk egy elemzést. Emellett megvizsgáltuk, hogy az állapoteret hogyan tudjuk optimálisan korlátozni, továbbá új bejárási algoritmusokat is fejlesztettünk. Hogy a potenciális futásidejű hibák detektálását hatékonyabbá tegyük, egy olyan új algoritmust javasolunk, amely követi a szimbolikus változók közötti összefüggéseket a szimbolikus végrehajtás alatt.

A *forráskód refaktoring* egy népszerű és hatékony technika programok belső minőségének javítására. A refaktoring fogalmát Fowler [10] vezette be. Eredetileg azt javasolta, hogy kódolási szabálysértések és gyanús kódrészletek (ún. code smell-ek) kellene, hogy mutassák a refaktoringgal javítandó konstrukciókat. Másrészről viszont kevés olyan tanulmány lelhető fel, amely azt vizsgálja, hogy mikor, miért és hogyan alkalmaznak a fejlesztők refaktoringot a mindennapi szoftverfejlesztésben, és mik a rövid és hosszú távú hatásai a karbantarthatóságra és a költségekre nézve. Ha megválaszolnánk ezeket a kérdéseket és megértenénk, hogy a fejlesztők hogyan alkalmaznak forráskód refaktoringot a gyakorlatban, akkor olyan módszereket és eszközöket fejleszthetnénk, amelyek hatékonyabbá tudnánk tenni az ipari szoftverfejlesztést. Hogy támogassuk a későbbi kutatását a forráskód refaktoringok gyakorlati alkalmazásának, elkészítettünk egy publikusan elérhető refaktoring adatbázist. Az adatbázis nyílt forrású Java rendszerekben talált refaktoringokat és forráskód metrikákat tartalmaz. Felhasználásával a refaktoringok forráskód metrikákra, valamint a karbantarthatóságra gyakorolt hatását vizsgáltuk, amik fontos minőségi jellemzők a szoftverfejlesztésben.

Az értekezés négy tézispontot tartalmaz, amelyeket két fő részbe szerveztünk. Jelen tézisfüzet egy-egy külön fejezetben foglalja össze az egyes tézispontok eredményeit .

I. rész
Új eredmények a szimbolikus végrehajtás alkalmazásában futásidejű hibák detektálása céljából

Szimbolikus végrehajtás

A szimbolikus végrehajtás [15] lényege, hogy a programot nem konkrét input adatokon futtatjuk, hanem ismeretlen változókként, úgynevezett szimbólumként kezeljük a bemenetet, a kimenet pedig ezen változók egy függvénye lesz. Egy szimbolikus változó egy konkrét változó lehetséges értékeinek halmaza, így egy szimbolikus programállapot is konkrét programállapotok halmaza lesz. Amikor a szimbolikus végrehajtás egy olyan feltételes vezérlési szerkezethez (például if utasításhoz) ér, amelyben a logikai kifejezés tartalmaz legalább egy szimbolikus változót, azt nem lehet kiértékelni, igaz és hamis egyaránt lehet, ezért a végrehajtás mindkét ágon továbbhalad. Ekképpen, elméletben szimulálni tudjuk a program összes lehetséges végrehajtási útvonalát.

Szimbolikus végrehajtás közben karban tartunk egy úgynevezett *path condition-t (PC)*. A path condition egy kvantormentes logikai formula, melynek iniciális értéke igaz, és a változói pedig a program szimbolikus változói. Amikor a végrehajtás elér egy feltételes utasításhoz, melyben szereplő logikai kifejezés tartalmaz legalább egy szimbolikus változót, akkor a kifejezést logikai ÉS művelettel hozzákapcsoljuk a meglévő PC-hez az igaz ágon, illetve annak negáltját a hamis ágon. A PC ilyen módon történő bővítésével, minden végrehajtású ághoz egy egyedi, szimbolikus változók feletti formulát rendelünk. Amellett, hogy a fenti módon bővítjük a PC-t, a szimbolikus végrehajtásban úgynevezett *constraint solver (korlátozás kielégítő)* programokat is használunk. A constraint solver a PC kielégítésére használatos. Olyan konkrét értékeket rendel a szimbolikus változókhoz, amelyeket behelyettesítve a logikai formula kielégül. Amennyiben egy ponton a PC kielégíthetetlen, egy ellentmondásos végrehajtási ágon vagyunk, amin nincs értelme továbbhaladni.

A lehetséges végrehajtási útvonalak egy irányított körmentes gráfot definiálnak, amit *szimbolikus végrehajtási fának* nevezünk. A fa egyes pontjai egy-egy szimbolikus programállapotot reprezentálnak. Az 1. ábra a szimbolikus végrehajtás folyamatát demonstrálja.

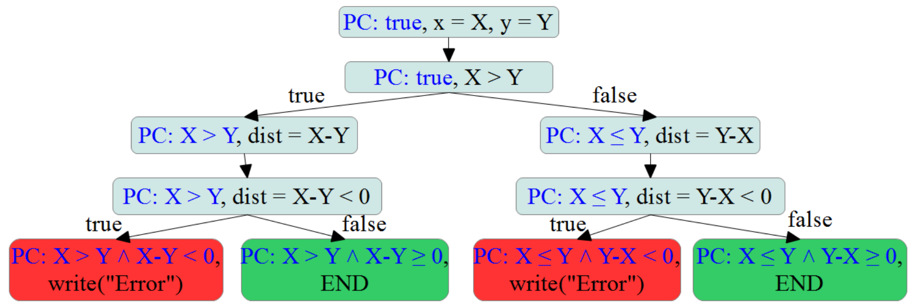
A szereplő kód szimbolikus végrehajtását a 1 (b) ábrán illusztráltuk a szimbolikus végrehajtási fa ábrázolásával. Az x és y változókat kezeljük szimbolikusan, a hozzájuk tartozó szimbólumok legyenek X és Y. Amikor a végrehajtás eléri az első if utasítást a 3. sorban, két lehetőség van: a logikai kifejezés lehet igaz és hamis is; ezért a végrehajtás elágazik és a logikai kifejezést illetve annak negáltja hozzákapcsoljuk a PC-hez az alábbi módon:

$$true \wedge X > Y \Rightarrow X > Y, \quad \text{és} \quad true \wedge \neg(X > Y) \Rightarrow X \leq Y$$

A 8. sorban a végrehajtás hasonlóan elágazik.

```
1. int x, y, dist;
2. ...
3. if (x > y) {
4.   dist = x - y;
5. } else {
6.   dist = y - x;
7. }
8. if (dist < 0)
9.   write("Error");
```

(a)



(b)

1. ábra. (a) Példakód, amely két egész szám szemegyenesen mért távolságát számítja ki (b) A példakód szimbolikus végrehajtási fája az x és y változók szimbolikus kezelésével

1. tézispont: Metódus szintű szimbolikus végrehajtás futásidejű hibák detektálására valós méretű szoftverrendszerekben

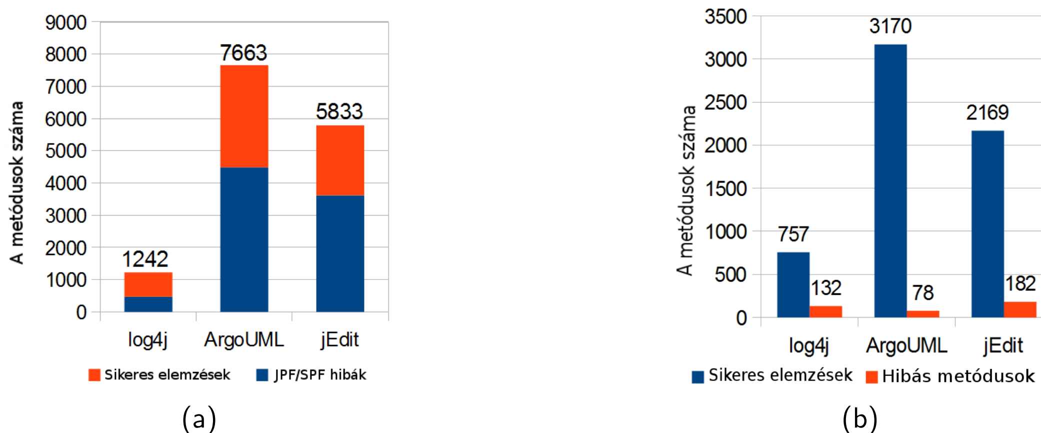
A futásidejű hibák többsége gyakran csak teszteléssel deríthető fel. A tesztelés azonban rendkívül költséges, illetve a karbantartási munkák, különösen a hibajavítás szintén jelentős költségeket emészt fel. A célunk az, hogy kifejlesszünk egy olyan módszert és eszközt, amely a szoftverfejlesztési folyamat fenti fázisait támogatja azáltal, hogy futásidejű kivételeket (mint például NullPointerException vagy ArithmeticException) detektál Java programokban, illetve olyan veszélyes kódrészletekre mutat rá, amelyek mintegy időzített bombaként viselkednek a fejlesztés későbbi szakaszaiban.

A szimbolikus végrehajtás programelemző technikát [15] használtuk fel arra, hogy megvalósítsuk ezt az eszközt. Szimbolikus végrehajtó motorként a Symbolic PathFinder-t (SPF) [21] használtuk fel, amely a Java PathFinder (JPF) [13] végrehajtási környezet (egy speciális Java virtuális gép) egy kiterjesztése, amelyet Java programok verifikálására készítettek.

Ahhoz, hogy a program végrehajtási útvonalaiból álló állapottér mérete kezelhető maradjon, egy olyan futtatókörnyezetet valósítottunk meg, amely tetszőleges Java program metódusait külön-külön, egyesével képes szimbolikus végrehajtással elemezni. Ha hagyományosan alkalmaznánk ezt a technikát, vagyis kizárólag a program belépési pontját jelentő *main()* metódusból indítanánk az elemzést, az állapottér kezelhetetlen méretűre nőne már az elemzés elején, ezért a programkód túlnyomó részét el sem érnénk.

A Symbolic PathFinder fejlesztőinek koncepciója az volt, hogy a programot hagyományosan kezdik végrehajtani, ahogy a valós végrehajtási környezetben is történne, majd a program komplexebb, valószínűleg több hibát tartalmazó részein átváltanak szimbolikus végrehajtásra [22]. A módszer előnye, hogy mivel normál módon kezdjük futtatni a programot a valós kontextusban, nagyobb eséllyel találunk igazi hibákat. Például, globális változók értéke már előre beállított lesz, de ha ezeket a változókat szimbolikusan kezeljük, olyan eseteket is megvizsgálunk, amelyek sosem fordulnak elő a valós futtatás során. Hátránya, hogy nehéz behatárolni a program problémás pontjait, ahol át kellene váltani szimbolikus végrehajtásra. Ehhez előzetes tudás, vagy munka szükséges. További hátrány, hogy a programot manuálisan kell futtatni úgy, hogy a vezérlés elérje azokat a metódusokat, amelyeket szimbolikusan elemezünk majd.

Ezzel szemben az általunk kifejlesztett eszköz a program összes metódusát egyesével elemzi szimbolikus végrehajtással. Előnye az, hogy felhasználónak nem kell semmiféle manuális futtatást elvégezni,



2. ábra. (a) Az összesen megvizsgált metódusok és a JPF/SPF hibák száma (b) A sikeresen elemzett metódusok és az ebből hibát tartalmazók száma

az egész folyamat automatizálható. Emellett a szimbolikus állapottér is korlátozott marad, mivel nem a teljes program szimbolikus végrehajtását indítjuk el, hanem annak részeit külön-külön. A mi módszerünk azt is lehetővé teszi, hogy osztálykönyvtárakat is elemezzünk, amelyek nem rendelkeznek *main()* metódussal, mint például a log4j naplózókönyvtár.

Egy ilyen végrehajtási környezet megvalósításához azt kell elérnünk, hogy a vezérlés sorban rákerüljön az összes metódusra. Azonban mivel az JPF egy virtuális gép, a program belépési pontját tartalmazó osztályt kell átadni neki, ahonnan kezdi a futtatást, nem tudjuk tetszőleges metódustól elindítani azt, kizárólag a main()-ből. A probléma megoldására a program minden metódusához generálunk egy vezérlő osztályt, amely tartalmaz egy main metódust, és az egyetlen feladata az, hogy átadja a vezérlést annak a metódusnak, amit elemezni szeretnénk és, hogy elvégezze az ezzel kapcsolatos összes teendőt.

A metódusokat szimbolikusan végrehajtva meghatározzuk azokat a végrehajtási útvonalakat, amelyek Java kivételt dobnak, továbbá az algoritmusunk tesztinputokat is generál ezekhez az útvonalakhoz, amelyek a normál futás során hibára vezetik a programot.

Amellett, hogy kisméretű példaprogramokon teszteltük a kifejlesztett eszközt, három nagyméretű nyílt forrású rendszeren is futtattuk, amelyekben számos futásidejű problémát találtunk. A három rendszer az ArgoUML modellező program, az log4j naplózókönyvtár és a jEdit szövegszerkesztő volt. A log4j naplózókönyvtárban talált hibák közül többet is megtaláltunk a rendszer hibakövető rendszerében, mint valós, bejelentett hibát.

A 2 (a) ábra az egyes programokban elemzett metódusok számát mutatja. A log4j 1242 metódusán futtattuk a hibadetektálást, amely közül csak 757-et tudunk sikeresen leelemezni, 474 esetben az elemzés a leállt a Java PathFinder (vagy a Symbolic PathFinder) hibája miatt. Az ArgoUML esetében is fennállt az, hogy számos metódust nem tudtunk leelemezni, a futtatások több, mint fele sikertelenül ért véget. A jEdit-nél is hasonló arányt tapasztaltunk. Sajnos általánosságban elmondható, hogy JPF gyakran váratlanul leállt különböző hibaüzenetek kíséretében.

JPF gyakori leállásnak ellenére a kifejlesztett eszköz számos futásidejű kivételt detektált mindhárom programban. A 2 (b) ábra kiemeli, hogy hány metódust elemeztünk le sikeresen, és hogy ezek közül hányban találtunk egy vagy több potenciális futásidejű hibát.

Fontos megjegyezni, hogy nem minden az eszköz által jelzett hiba jelent feltétlenül a valós futás

során is megjelenő hibát. Ennek az az oka, hogy az elemzés egy mesterségesen létrehozott környezetben történt, ami miatt hamis pozitív találatok jelenhetnek meg. A módszer és a hibatalálatok érvényességét azzal igazoltuk, hogy az elemzett programok hibakövető rendszereiben találtunk olyan bejelentéseket, amelyek arról számolnak be, hogy a program leállását a mi eszközünk által is megtalált futásidejű kivétel okozta.

Ezen bejelentések listája:

- Az első hibabejelentés szerint a log4j 1.2.14-es verzióját használó program egy NullPointerException miatt állt le.

  URL: https://issues.apache.org/bugzilla/show_bug.cgi?id=44038

- A következő probléma a log4j 1.2.25-ös verziójában lépett fel. A jelentés szerint a program leállást a *org.apache.log4j.NDC.remove()* metódus 377-edik soráról érkező NullPointerException okozta.

  URL: https://issues.apache.org/bugzilla/show_bug.cgi?id=45335

- Szintén a log4j 1.2.15-ös verziójában a *org.apache.log4j.net.SyslogAppender* osztály 312. soráról állapítottuk meg, hogy NullPointerException forrása lehet. URL: https://issues.apache.org/bugzilla/show_bug.cgi?id=46271

A szerző hozzájárulása az eredményekhez A szerző térképezte fel a szimbolikus végrehajtás módszerét és a Symbolic PathFinder végrehajtó motor lehetőségeit, illetve azt, hogy hogyan alkalmazható mindez futásidejű programhibák megtalálására. A metódus szintű szimbolikus végrehajtás kidolgozása és a futtatókörnyezet megvalósítása, ami elvégzi az elemzést teljes egészében a szerző munkája. Az ő érdeme a futásidejű hibák detektálása azzal, hogy megvalósított egy új modult a Symbolic PathFinderben, ami visszaadja a hibára vezető végrehajtási útvonalat és az elemzett metódusnak ehhez az útvonalhoz tartozó paraméterezését, amely tesztinputként felhasználható. A megtalált futásidejű kivételek átvizsgálása és azok érvényességének bizonyítása azzal, hogy megkereste őket az elemzett progoramok hibakövető rendszereiben szintén a szerző munkája volt. A szerző számos alkalommal felvette a kapcsolatot a Symbolic PathFinder fejlesztőivel, hogy olyan blokkoló hibákat jelentsen be, amelyek hátráltatták a kutatást.

2. tézispont: Új feltételrendszer építő mechanizmus szimbolikus végrehajtáshoz futásidejű hibadetektálás pontosítására

A Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén fejlesztett szimbolikus végrehajtó motor, a *Symbolic Checker* Java programokban képes futásidejű problémákat (mint például null pointer dereferencia tömb alul- és túlindexelés, nullával való osztás) detektálni anélkül, hogy a programot valós környezetben futtatni kellene. A szimbolikus végrehajtás elmélete szerint a programot nem konkrét input adatokon futtatjuk, hanem ismeretlenekként, ún. szimbolikus változókként kezeljük a bemenetet. Amikor a program végrehajtása egy olyan feltételes vezérlési szerkezethez ér, amelyben a logikai kifejezés tartalmaz szimbolikus változót, az igaz és a hamis ágon is folytatódni fog a végrehajtás. Ezeknél az elágazási pontoknál a vezérlési szerkezetben szereplő logikai kifejezést feljegyezzük és

továbbvisszük az igaz ágon, illetve annak negáltját a hamis ágakon. Ennek megfelelően az összes végrehajtási útvonalhoz egy egyedi, szimbolikus változók feletti formulát rendelünk, amelyet path condition-nek (PC) nevezünk.

Ebben a tézispontban bemutatunk egy a fent leírtaktól eltérő feltételrendszer építő mechanizmust, amely pontosítja a Symbolic Checker által adott futásidejű hibatalálatokat azzal, hogy a programban lévő értékadásokat is feltételként kezeli. Ennek köszönhetően követni tudjuk a szimbolikus változók közötti függőségeket kibővítve a path condition felépítésének eredeti koncepcióját.

Az kifejlesztett feltételrendszer építő mechanizmus lényege, hogy amennyiben az elemzés közben a program olyan feltételeket definiál, amelyek egyértelműen meghatározzák valamely szimbolikus változó értékét, akkor ezek a szimbólumok a konkrét értékké alakíthatók, és a szimbolikus végrehajtás úgy folytatódhat az aktuális útvonalon, hogy ezeket a konkretizált változókat használja. Ha a fenti módon konkretizáljuk bizonyos szimbolikus változók értékét, olyan hibákat is megtalálhatunk, amelyeket egy konvencionális szimbolikus elemző nem képes, hiszen szimbolikus végrehajtás folyamán csak akkor jelzünk hibát, ha azt egy konkrét érték váltotta ki. Ha egy változót szimbólumként kezelünk, az azt jelenti, hogy annak értéke kétséges, nem ismert. Tehát például, ha egy kifejezést elosztunk egy értékkel, amiről tudjuk, hogy nulla az aktuális végrehajtási útvonalon ($kifejezes/0$), akkor az eszköz hibát jelez, ha azonban ezt a kifejezést egy szimbólummal osztjuk ($kifejezes/szimbolum$), nincs hibajelzés.

A korábban említett feltételeket a vezérlési szerkezetekben lévő logikai kifejezésék és a program értékadási definiálják ide értve a Java nyelv inkrementáló és dekrementáló operátorait is ($++, --$).

Összességében a javasolt feltételrendszer építő mechanizmus célja az, hogy konkretizálni tudjuk a lehető legtöbb szimbolikus változót, aminek köszönhetően több futásidejű hibát találhatunk meg. Ahhoz, hogy ezt elérjük, (1) fel kell építeni egy speciális path condition-t a szimbolikus végrehajtás által bejárt útvonalakhoz, amely tartalmazza azokat a szimbolikus változók közötti függőségeket is, amiket a program értékadásai definiálnak, valamint (2) ha a PC-ben lévő feltételek egyértelműen meghatározzák bizonyos szimbolikus változók értékét, akkor a végrehajtást ezen szimbólumokat értékké alakítva kell folytatni az aktuális útvonalon. A felvázolt módon kibővített path condition is tartalmazza azokat a feltételeket, amelyek a konvencionális módszerrel épített PC-ben vannak, vagyis ha a PC kielégítetlen, az ugyanúgy azt jelenti, hogy ellentmondásos ágon vagyunk, és nincs értelme folytatni az elemzést ezen az útvonalon. Ezzel hamis pozitív hibatalálatok szűrhetőek ki.

Az 1. kódrészlet a kibővített PC előnyét szemlélteti.

```
1  // legyen c egy int szimbolum
2  double b = 2*c + 4;
3  int a = b + 9;
4  if (a > 8) {
5     ...
6     if (a < 10) {
7        // b konkretizalasa
8        int p = 1/b;
9     }
10 }
```

1. kódrészlet. Példakód, amin a kifejlesztett algoritmus szimbólumot konkretizál

A kódrészletet szimbolikusan végrehajtva a *c* változó szimbolikus kezelésével a következő feltételrendszert építi fel az új algoritmus a 7. soron állva:

$$a > 8 \land a = b + 9 \land b = 2 \cdot c + 4 \land a < 10.$$

A fenti feltételrendszer tartalmazza az if utasításokban megadott feltételeket, valamint az *a* szimbólum függőségeit, vagyis az *a* változót definiáló értékadásokat. A feltételrendszer kielégítése után azt kapjuk, hogy *a* értéke csakis 9 lehet, amiből következik, a *b* és *c* szimbólumok értéke is: $b = 0.0$ és $c = -2$. Ha a végrehajtást a $b = 0.0$ értékkel folytatjuk, a 8. sorban egy nullával való osztás detektálható. Amennyiben a *b* szimbólum nem szerepelt volna a PC-ben, és ha az egyértelmű értékeket nem használtuk volna fel, a nullával való osztás detektálása meghiúsult volna.

Valós méretű szoftverrendszerekben igazán nagy méretű feltételrendszerek keletkezhetnek, amelyek sok szimbólumra írhatnak fel különböző feltételeket. Könnyen belátható az, hogy egy nagyméretű feltételrendszert kiértékelve, kicsi annak az esélye, hogy annak pontosan egy megoldása legyen. A probléma megoldása érdekében a PC-t összefüggő komponensekre bontjuk, azaz olyan részfeltételhalmazokra, amelyek függetlenek abban az értelemben, hogy nem tartalmazzák ugyanazt a szimbolikus változót. Két feltétel akkor kerül ugyanabba a komponensbe, ha tartalmaznak legalább egy közös változót. Ezután az összefüggő komponensek külön-külön kielégíthetők, és ha közülük valamelyik egyértelműen meghatározza egy szimbólum értékét, akkor az a későbbiekben felhasználható. A fenti dekompozíció elvégezése után a path condition feltételhalmazok halmaza lesz.

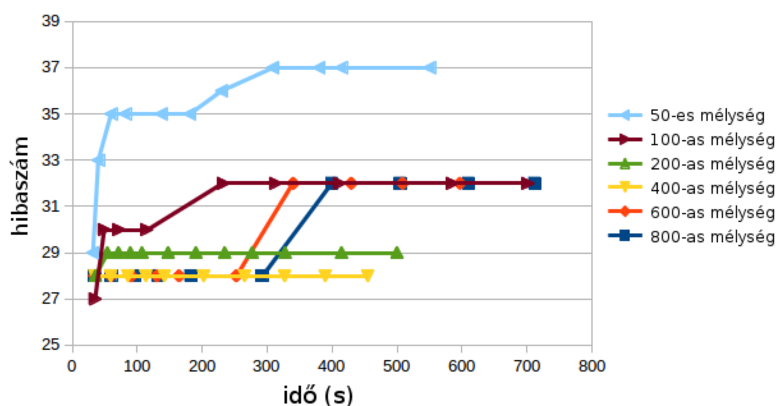
A feltételrendszer felépítéséhez és kielégítéséhez a nyílt forrású Gecode [11] korlátozás kielégítő eszközkészletet használtuk fel.

Az új feltételrendszer építő mechanizmus használatával olyan potenciális futásidejű hibákat is megtalálhatunk, amelyre egy konvencionális szimbolikus végrehajtó eszköz nem képes. Azzal, hogy szimbolikus változókat konkretizálunk, a szimbolikus végrehajtási fa mérete csökken, amivel az elemzéshez szükséges idő és memóriaigény is redukálódik. A kifejlesztett módszer előnyeit különböző példakódokon szemléltetjük, kihangsúlyozva a különbségeket egy konvencionális szimbolikus végrehajtó eszközhöz és a Symbolic Checker feltételrendszer építés nélküli verziójához hasonlítva a javasolt algoritmust. Valós, nagyméretű rendszerekben is találtunk olyan potenciális futásidejű hibákat, amik megtalálása a hagyományos úton nem lenne lehetséges.

A szerző hozzájárulása az eredményekhez A szerző részt vett a Symbolic Checker szimbolikus végrehajtó motor fejlesztésében vezető fejlesztőként. Ő dolgozta ki a bemutatott feltételrendszer építő mechanizmust, megvalósította és integrálta azt a Symbolic Checker-be. A módszer kiértékelése és összehasonlítása a konvencionális megközelítéssel, valamint a példakódokon és a valós rendszereken történő tesztelemzések futtatása szintén a szerző munkája volt.

3. tézispont: Új bejárási algoritmusok és az állapottér korlátozások empirikus vizsgálata a szimbolikus végrehajtásban

A szimbolikus végrehajtó motor, amit a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén készítettünk komolyabb fejlesztéseken esett át, és az új *RTEHunter* nevet kapta. A szimbolikus végrehajtás elméletének megfelelően, az RTEHunter felépíti a lehetséges végrehajtási utak fáját, amit szimbolikus végrehajtási fának nevezünk. Futásidejű problémákat úgy detektál az eszköz, hogy bejárja ezt a fát végrehajtva a szóban forgó útvonalakat, és amennyiben egy bizonyos feltétel teljesül, hibát jelez. A végrehajtási utak száma azonban exponenciálisan növekszik az elágazási pontok számával,

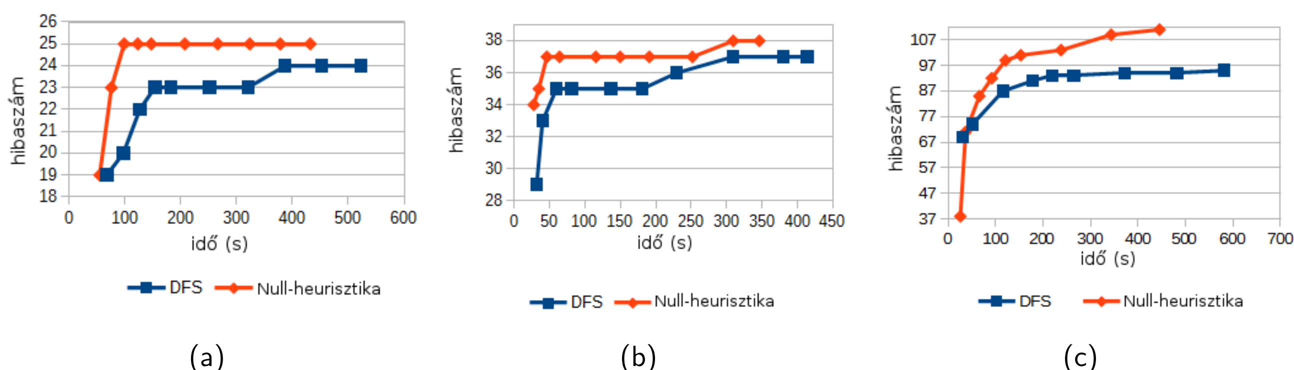


3. ábra. A megtalált hibák számának növekedése az elemzés idejének növekedésével különböző mélységkorlátokat alkalmazva.

vagyis a teljes végrehajtási fa bejárása lehetetlen a gyakorlatban. Ez a probléma azzal kezelhető, hogy különböző módokon korlátozzuk a fa méretét. Például a fában lévő szimbolikus programállapotok száma, a fa mélysége (állapot-mélység), vagy az elemzés időtartalma limitálható. Az RTEHunter-ben a maximális állapotok száma, a fa maximális mélysége, valamint a fabejáráshoz használt keresési algoritmus konfigurálható. A fent leírt álltoprobbanás miatt a szimbolikus végrehajtás állapotterének korlátozása releváns kutatási téma, ha a módszert a gyakorlatban is alkalmazni kívánjuk. A célunk az, hogy megtaláljuk az RTEHunter optimális paraméterezését, vagyis azt, hogy mi az maximális állapotszám, maximális fa mélység, és mi az a bejárási stratégia, amellyel a lehető legtöbb futásidejű hibát meg tudjuk találni minél rövidebb idő alatt.

A három nyílt forrású Java rendszeren (ArgoUML, JetSpeed és JFreeChart) végzett empirikus vizsgálatok azt mutatják, hogy a szimbolikus végrehajtási fában lévő állapotok maximális száma közvetlenül befolyásolja az elemzési időt, de a futásidejű hibajelzések számát nem. A bejárt és végrehajtott állapotok száma erősen korrelál az elemzési idővel. A Pearson korrelációs együtthatók kivétel nélkül 0.99 felett vannak, ami azt jelenti számunkra, hogy ha nagyra állítjuk a maximális állapotszámot, akkor az elemzés időtartama is hosszabb lesz. Az eredmények szignifikánsak a $p < 0.05$ szinten. Az végrehajtott állapotok száma és a talált futásidejű problémák közötti korreláció gyengébb, 0.3 és 0.8 közötti korrelációs együtthatókat kaptunk, és sok esetben ez nem is szignifikáns a $p < 0.05$ szinten. Ebből az következik, hogy a maximális állapotszám korlátozás a végrehajtási időt meghatározza, de a detektált potenciális futásidejű hibák száma más tényezőktől is függ.

Hogy megértsük a mélységkorlátotás szerepét, az RTEHunter-t különböző maximális mélység konfigurációkkal futtattuk. Ezt úgy tettük, hogy ugyanazt a maximális mélységkorlátot alkalmazva többször futtattuk az RTEHunter-t az egyes alkalmakkor megnövelve a maximális állapotszámot, vagyis egyre növelve az elemzés időtartamát. Ez alapján megvizsgáltuk azt, hogy adott mélységkorlátot alkalmazva, hogyan változik a detektált hibák száma az idő függvényében. Az 50, 100, 200, 400, 600 és 800-as mélységkorlátokat vizsgáltuk, a maximális állapotszámot pedig 200-tól 10.000-ig növeltük minden esetben. Az eredményeket olyan vonaldiagrammokon ábrázoltuk, amilyen a 3. ábrán látható. Mindhárom megvizsgált rendszeren különböző az optimális maximális mélységkorlátoka, de összességében azt a következtetést vonhatjuk le, hogy futásidejű problémák gyakrabban fordulnak elő a 0 és 60-as mélységszint között, mint a fa mélyebb részeiben, viszont ez nagyban függ a választott fabejárási stratégiától is.



4. ábra. A null-heurisztikus bejárási stratégia hatékonysága az alapértelmezett mélységi bejáráshoz hasonlítva az ArgoUML (a), a Jetspeed (b), és a JFreeChart (c) rendszereken

Kifejlesztettünk továbbá két olyan fabejárási stratégiát, melyek célja az, hogy a szimbolikus végrehajtás közben olyan útvonalakat válasszunk és azokon haladjunk tovább, amelyek a kód nagyobb valószínűséggel hibát tartalmazó részeihez vezetnek.

A null-heurisztikus bejárási stratégia arra törekszik, hogy úgy irányítsa a fabejárást, hogy több null pointer dereferencia hibát találjunk meg. Ehhez, minden szimbolikus állapotban összesítettük az elemzett programban azokat referencia típusú értéket (változóértékeket, literálokat, függvény visszatérési értékeket, stb.), amelyek *null* értékűek. A bejárás során minden egyes döntési pontban azzal az állapottal folytatjuk a szimbolikus végrehajtást, amelyben magasabb a null értékek száma, azt feltételezve, hogy ahol több null érték van a programban, ott nagyobb valószínűséggel fogunk null pointer dereferenciát találni. A null-heurisztikus bejárási stratégiával 16%-kal több potenciális futásidejű hibát detektáltunk ugyanannyi idő alatt, mint az alapértelmezett mélységi bejárás használatával. A kapott eredmények a 4. ábrán szerepelnek.

A null-heurisztikus bejárási stratégia mellett kifejlesztettünk egy olyan heurisztikát, amely nem csak null pointer dereferenciák megtalálást támogatja, hanem mind a négy típusú probléma megtalálását, amelyet az RTEHunter detektálni képes. A heurisztika megvalósításához egy lineáris regressziós modellt használtuk, ami a fabejárás aktuális levél állapotaihoz egy-egy pontszámot rendel. Mindig azzal az állapottal folytatjuk a bejárást, amihez a legmagasabb pontszámot rendeltük. Ez a pontszám az állapotból elérhető részfában detektálható hibák számának becslése.

A lineáris regresszió tanulóadatbázisa az RTEHunter által korábban már bejárt állapotokhoz tartalmaz egy-egy tanulópéldát. Minden példaállapot címkéje (supervisory signal) az állapotból elérhető részfában egy korábbi elemzés során megtalált hibák száma lesz.

Minden állapotra öt prediktor attribútumot definiáltunk:

1. *Az állapot mélysége a szimbolikus végrehajtási fában.* Amennyiben a hibák jelentős része jellemzően a fa bizonyos mélységében található, ezt az információt belekódoljuk a modellbe.
2. *Az állapotban lévő null értékek száma.*
3. *Az állapotban lévő numerikus nulla értékek (változóértékek, literálok, függvény visszatérési értékek, stb.) számának és az állapotból a vezérléi folyam gráf szerint 15 basic block mélységig elérhető osztás operátorok számának összege.*
4. *Az állapothoz tartozó metódus logikai programsorainak száma (LLOC).*
5. *Az állapothoz tartozó metódus ciklomatikus komplexitása [17].*

		500 max állapot	1000 max állapot	1500 max állapot
ArgoUML	DFS	22	23	23
(max mélység: 100)	LR alapú	51	54	55
Jetspeed	DFS	33	35	35
(max mélység: 50)	LR alapú	66	74	73
JFreeChart	DFS	91	93	94
(max mélység: 400)	LR alapú	122	131	138

1. táblázat. A detektált hibák száma a lineáris regresszió alapú bejárási stratégiát használva (LR alapú) összehasonlítva az alapértelmezett mélységi bejárással (DFS)

A lineáris regresszió alapú bejárási stratégia kiértékeléséhez 10-szeres keresztvalidációt használtunk mindhárom vizsgált rendszeren. Ez a heurisztika is jobban teljesít mint a mélységi bejárás. Több, mint kétszer annyi potenciális hibát detektáltunk vele az ArgoUML-ben és a Jetspeed-ben, ahogy ez a 1. táblázatban látható.

A szerző hozzájárulása az eredményekhez A szimbolikus végrehajtási fa maximális állapotszáma, az elemzés időtartama és talált futásidejű hibajelzések száma közötti kapcsolat empirikus úton történő megtalálása az RTEHunter számtalan futtatásával a vizsgált rendszereken a szerző munkája. Ő végzett el számos kísérleti elemzést annak érdekében, hogy megtaláljuk azt az optimális mélységkorlátot a vizsgált rendszereken, ahol a futásidejű hibák többsége detektálható. A bemutatott két fabejárási stratégia ötlete, azok megvalósítása és kiértékelése szintén a szerző érdeme.

II. rész

Refaktoring tevékenységek vizsgálata egy új refaktoring adatbázis alapján

4. tézispont: Refaktoring tevékenységek vizsgálata osztály és metódus szinten egy új publikus adatbázis felhasználásával

A forráskód refaktoring egy népszerű és hatékony technika a forráskód belső minőségének javítására. A refaktoring fogalmát Fowler [10] vezette be. Napjainkban a forráskód refaktorálása lényeges szerepet tölt be a szoftverfejlesztésben az IT szakemberek egybehangzó véleménye szerint. Annak ellenére, hogy a refaktoring használata széles körben elfogadott az ipari szoftverfejlesztésben, több tanulmány is kimutatta [8, 19, 23], hogy a fejlesztők a Fowler által javasolt módszertől eltérően alkalmaznak refaktoring műveleteket a gyakorlatban. Emellett kevés olyan tanulmány lelhető fel, amely azt vizsgálja, hogy mik a refaktoring műveletek valós hatásai, és hogy a fejlesztők hogyan alkalmazzák azokat a gyakorlatban.

Létrehoztunk egy olyan adatbázist, amely 7 nyílt forrású Java rendszer verzióira tartalmaz forráskód metrikákat és verziók között elvégzett refaktoring műveleteket. Az adatbázis célja, hogy támogassa a refaktoring tevékenységek kutatását. Az adatbázis a Ref-Finder nyílt forrású refaktoring detektáló

eszközzel [14] megtalált refaktoringokról tartalmaz pontos információkat, valamint több mint 50 féle forráskód metrikát 7 nyílt forrású Java rendszer 37 verziójára osztály és metódus szinten. A forráskódmetrikák mellett az adatbázis a kódelemek (osztályok és metódusok) relatív karbantarthatósági indexét (RMI) is tartalmazza, amelyet a *QualityGate* eszközzel számítottunk ki, ami a *ColumbusQM minőségmodell* egy implementációja [9]. Az adatbázis előállításához továbbfejlesztettük a Ref-Finder eszközt, hogy az a verziókezelő rendszer alapján több egymást követő verzió között automatikusan is képes legyen a refaktoringok meghatározásra. Azt is megvalósítottuk benne, hogy a refaktoringokat a pontos forráskód pozíciójukkal együtt meghatározott struktúrájú CSV (Comma Separated Values) fájlokba exportálja.

Az adatbázis elemzésével a refaktoringok forráskód metrikákra és karbantarthatóságra gyakorolt hatását vizsgáltuk. Az alábbi kutatási kérdésekre kerestünk választ:

KK1. *Több refaktoringot alkalmaznak az alacsonyabb karbantarthatóságú kódelemeken a gyakorlatban?*

KK2. *Mely minőségi attribútumokra (forráskódmetrikákra) van legnagyobb hatással a refaktoring és milyen irányba változtatja azokat?*

A KK1-es kutatási kérdés megválaszolásához, a kódelemek RMI értékei és az őket érintő refaktoring műveletek száma közötti korrelációt vizsgáltuk. Az x_i-edik verzióban lévő kódelemek RMI értékeit és az x_{i+1}-edik verzió kódelemit érintő refaktoring darabszámokat vettük ahhoz, hogy felmérjük, hogy vajon az alacsonyabb karbantarthatóságú kódelemeket intenzívebben refaktorálták-e a két verzió között, mint a magasabb belső minőséggel rendelkezőket. Mivel semmilyen információval nem rendelkeztünk a karbantarthatósági indexek és a refaktoring darabszámok eloszlásáról, Spearman-féle rangkorrelációt alkalmaztunk.

A KK2-es kutatási kérdés megválaszolásához először kiszámoltuk a metrika különbségeket az egymást követő verziók között. A legtöbb esetben a negatív különbség jelenti azt, hogy a metrikaérték javult a két verzió között, mivel az alacsonyabb érték (például alacsony komplexitás) jelent jobb minőségi mutatót. A Mann-Whitney U próbát [16] futtattuk ahhoz, hogy meghatározzuk, hogy van-e szignifikáns eltérés a refaktoring műveletek által érintett és egyáltalán nem érintett kódelemek metrikakülönbsége (javulása) között. A Mann-Whitney U próba egy nem-parametrikus statisztikai próba, ami azt vizsgálja, hogy két csoport értékeinek eloszlása szignifikánsan eltérő-e. A teszt p-értéke adja meg számunkra, hogy van-e szignifikáns különbség a refaktoring által érintett és nem érintett kódelemeket metrikaértékeinek javulásában.

Ennek a tesztnek az eredménye azt adja meg számunkra, hogy melyek azok a metrikák, amelyek szignifikánsan javultak refaktoring hatására. Hogy megbecsüljük a metrikák változásának mértékét is, meghatároztuk a Cliff's delta (δ) hatás mértéket (effect size) is [12]. Cliff's δ érték azt méri, hogy az egyik csoportban lévő értékek hányszor nagyobbak, mint a másikban lévők. Egyszerűen fogalmazva, ha a Cliff's δ pozitív, akkor a metrika érték különbségek (vagyis a javulás mértéke) a refaktoring által érintet kódelemek csoportjában nagyobb, míg ha negatív, a metrika érték különbségek a refaktoring által nem érintet kódelemek csoportjában a jelentősebb.

Eredményül azt kaptuk, hogy az alacsony karbantarthatósági mutatóval rendelkező osztályokat többször refaktorálták a gyakorlatban, mint jobb minőségi mutatóval rendelkezőket. A forráskód metrikákat tekintve a klónok száma, a komplexitás és a csatoltságot jellemző metrikák javultak refaktorálás hatására, a kommentezettséget mérő metrikák azonban romlottak. Ezen felül, a méret alapú metrikák is jelentős csökkenést mutatnak. A metódusok szintjén szintén azt állapíthatjuk meg, hogy

az alacsonyabb karbantarthatóságú metódusokat jellemzően több refaktoring érinti a gyakorlatban. A klón lefedettség, a méret, és a metódusokból kimenő hívások száma csökkent a legintenzívebben a refaktoring által érintett metódusokban, ami arra utal, hogy a refaktoring ténylegesen mérsékli a negatív mutatójú forráskódjellemzőket, és karbantarthatóbb szoftvert eredményeznek a gyakorlatban.

A Ref-Finder szerzői szerint, az eszköz pontossága 79%-os [20], azonban a mi vizsgálatink azt mutatták, hogy az eszközzel kinyert refaktoring adathalmaz pontossága ennél alacsony a hamis pozitív refaktoring példányok miatt. Emiatt elkészítettük a refaktoring adatbázis manuálisan validált részhalmazát. Az új adatbázis a Java rendszerek egy-egy verziójára tartalmaz validált refactoringokat, amelyek pontosabb adatokat biztosítanak a vizsgálatainkhoz. Az eredeti adatbázisban lévő összes, valamint a manuálisan validált refaktoring műveletek számát, pontosság információval kiegészítve a 2 táblázat tartalmazza.

Rendszer	# Összes	Verzió (Release)	# Kiértékelt	TP	FP	Prec.
antlr4	269	30/06/2013 [3468a5f]	112	50	62	44.64%
junit	1,080	08/04/2010 [a30e87b]	29	14	15	48.28%
mapdb	4,547	30/07/2014 [967d502]	171	4	167	2.34%
mcMMO	448	11/07/2013 [4a5307f]	63	6	57	9.52%
mct	716	27/09/2013 [f2cdf00]	97	28	69	28.87%
oryx	123	11/04/2014 [0734897]	71	25	46	35.21%
titan	3,661	13/02/2015 [fb74209]	84	18	66	21.43%
Összesen	**10,844**	–	**627**	**145**	**482**	**23.13%**

2. táblázat. Az összes és a manuálisan validált metódusok száma pontosság információval (Prec.) az egyes rendszerekben

Habár a manuálisan validált adatbázis létrehozása és kiadása önmagában jelentős kontribúció, megismételtük rajta az eredeti adatbázison elvégzett vizsgálatokat is. A validált adatbázison végzett vizsgálatok azt mutatják, hogy a refaktoring által érintett forráskód elemek átlagos karbantarthatósága sokkal alacsonyabb volt a refaktorálás előtti verzióban, mint a refaktoring utáni kiadásban azokhoz az osztályokhoz és metódusokhoz képest, amiket egyáltalán nem érintettek refaktoring műveletek. Ez erősen azt mutatja, hogy jellemzően az alacsonyabb minőségű kódrészeken végeznek refaktoringot a fejlesztők a gyakorlatban, függetlenül attól, hogy ezt tudatosan teszik-e vagy nem. Újra megvizsgáltuk azt is, hogy a tipikus forráskód metrikák eloszlása hogy alakul a refaktorált és refaktoring által nem érintett forráskódelemek csoportjában. Az új adatbázis szerint a méret, a komplexitás és a csatolás metrikák eloszlása mutatja a legnagyobb különbséget a refaktorált és a nem refaktorált elemek között. Ezen felül azt is meg tudjuk erősíteni, hogy a fejlesztők nem célzottan ezen metrikák alapján választják ki a refaktorálandó kódrészleteket, de igyekeznek kontrollálni és csökkenteni azok értékét, hiszen ezek a metrikák sokkal lassabban nőttek (vagy akár csökkentek) a refaktoring által érintett kódrészekben.

A bemutatott refaktoring adatbázisok felhasználásában rejlő lehetőségek jelentősen túlmutatnak az általunk elvégzett vizsgálatokon, emiatt publikusan elérhetővé tettük azokat a PROMISE adattárházban [18] támogatva ezzel komplexebb összefüggések felfedezésére irányuló jövőbeli kutatásokat.

A szerző hozzájárulása az eredményekhez A szerző fejlesztette tovább a Ref-Finder refaktoring kereső eszközt, hogy az a verziókezelő rendszer alapján számos egymást követő verzió között automatikusan is képes legyen a refaktoringok meghatározásra, valamint arra, hogy a megtalált refaktoringokat a pontos pozícióinformációval együtt megfelelő struktúrában CSV fájlokba exportálja. A szerző

az egyes refaktoring példányok kódelemekhez rendelésével is hozzájárult az adatbázis létrejöttéhez, valamint a manuális validáció felét is ő végezte el. A szerző részt vett az elemzési módszertan kidolgozásában és az eredmények kiértékelésében.

Összefoglalás

A disszertáció két témát ölel fel: szimbolikus végrehajtást, valamint refaktoring tevékenységek vizsgálatát.

A szimbolikus végrehajtás területén elvégzett kutatás az exponenciálisan növekvő állapottér kezelésére és a hatékonyabb futásidejű hibadetektálásra fókuszál. Az eredmények összefoglalásaként azt mondhatjuk, hogy sikeresen alkalmaztuk a szimbolikus végrehajtást valós méretű Java nyelvű rendszerek elemzésére a javasolt metódus szintű végrehajtással, hatékonyabbá tettük a hibakeresést egy új feltételrendszer építő mechanizmussal, valamint megvizsgáltuk az állapottér korlátozásának lehetőségeit és új bejárási algoritmusokat fejlesztettünk. Mindezen kutatási munka eredménye, hogy kifejlesztettünk egy olyan eszközt, ami ma már része egy kereskedelmi forráskód elemző eszközkészletnek.

A refaktoring tevékenységek vizsgálatát tekintve felfedtük a Java rendszerekben alkalmazott refaktoringok és a szoftver karbantarthatóság kapcsolatát, és azt is megvizsgáltuk, hogy a refaktoring hogyan befolyásolja a különböző forráskód metrikákat osztály és metódus szinten egyaránt. Jelenleg nem tudunk hasonló tanulmányról, amely metódus szinten vizsgálná ezeket a kérdéseket. A vizsgálatokat elvégzéséhez felépítettünk egy refaktoring adatbázist, amely részletes információkat tartalmaz 7 Java nyelvű rendszerben elvégzett refaktoringokról. Ezeket a refaktoring információkat több, mint 50 fajta forráskódmetrikával együtt osztályokhoz és metódusokhoz rendeltük. Az adatbázist és annak manuálisan validált részhalmazát publikusan hozzáférhetővé tettük. Úgy gondoljuk, hogy ezen refaktoring adatbázis felhasználásában rejlő lehetőségek jelentősen túlmutatnak az általunk elvégzett vizsgálatokon, ezért arra buzdítjuk a terület kutatóit, hogy használják az adatbázist más összefüggések felfedezésére is a jövőben.

A 3. táblázatban összefoglaljuk a tézispontokat alátámasztó publikációkat.

$\mathcal{N}o.$	[4]	[2]	[5]	[1]	[6]	[7]	[3]
1.	•	•					
2.			•				
3.				•			
4.					•	•	•

3. táblázat. A tézispontokat alátámasztó publikációk

Köszönetnyilvánítás

Mindenekelőtt szeretnék köszönetet mondani a témavezetőmnek, Dr. Ferenc Rudolfnak, hogy végig vezetett és támogatott a PhD tanulmányaim alatt, valamint a programozáshoz és a kutatáshoz való pozitív hozzáállásával mindig motivált engem. Nélküle valószínűleg soha nem vettem volna részt semmilyen kutatási munkában. Nem is kívánhatnék jobb mentort, mint amilyen ő. Külön szeretnék köszönetet mondani Dr. Hegedűs Péternek, akit a második számú mentoromnak és témavezetőmnek tartok. Mindig segített, bármivel is fordultam hozzá. Hálás vagyok a tőle kapott támogatásért, iránymutatásért és bátorításért. Szintén szeretném megköszönni Dr. Gyimóthy Tibornak, a Szoftverfejlesztés Tanszék vezetőjének a kutatómunkám során nyújtott támogatását. Sok-sok köszönettel tartozom Dr. Bakota Tibornak, akitől rengeteget tanultam, valamit Dr. Jász Judit és Perényi Tamás kollégáimnak is, akik munkájukkal segítették a kutatásaimat. Köszönettel tartozom még a publikációim névtelen bírálóinak hasznos megjegyzéseikért és javaslataikért, valamint Kintses-Fejér Andreának a disszertációm angol nyelvű változatának lektorálásáért. Végül, de nem utolsósorban, szeretném kifejezni hálámat szüleimnek, hogy végig támogató hátteret nyújtottak a tanulmányaimhoz, és hogy végig biztattak az idáig vezető úton.

Kádár István, 2017. október

A szerző publikációi

[1] **István Kádár**. The optimization of a symbolic execution engine for detecting runtime errors. *Acta Cybernetica*, 23(2):573–597, 2017.

[2] **István Kádár**, Péter Hegedűs, and Rudolf Ferenc. Runtime exception detection in java programs using symbolic execution. *Acta Cybernetica*, 21(3):331–352, 2014.

[3] **István Kádár**, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, pages 10:1–10:4, New York, NY, USA, 2016. ACM.

[4] **István Kádár**, Péter Hegedűs, and Rudolf Ferenc. Runtime exception detection in java programs using symbolic execution. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools, SPLST'13*, page 215–229, Szeged, 2013. University of Szeged, University of Szeged.

[5] **István Kádár**, Péter Hegedűs, and Rudolf Ferenc. Adding constraint building mechanisms to a symbolic execution engine developed for detecting runtime errors. In *International Conference on Computational Science and Its Applications*, pages 20–35. Springer, 2015.

[6] **István Kádár**, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 599–603. IEEE, March 2016.

[7] **István Kádár**, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. *Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods*, chapter Computational Science and Its Applications – ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part IV, pages 610–624. Springer International Publishing, 2016.

Hivatkozások

[8] Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. Understanding the Longevity of Code Smells: Preliminary Results of an Explanatory Survey. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, pages 33–36, New York, NY, USA, 2011. ACM.

[9] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy. A Probabilistic Software Quality Model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252, Sept. 2011.

[10] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] Gecode Tool-set. `http://http://www.gecode.org/`.

[12] Melinda R Hess and Jeffrey D Kromrey. Robust Confidence Intervals for Effect Sizes: a Comparative Study of Cohen's d and Cliff's delta Under Non-normality and Heterogeneous Variances. In *Annual Meeting of the American Educational Research Association*, pages 1–30, 2004.

[13] Java PathFinder Tool-set. `http://babelfish.arc.nasa.gov/trac/jpf`.

[14] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-Finder: a Refactoring Reconstruction Tool Based on Logic Query Templates. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering (FSE'10)*, pages 371–372, 2010.

[15] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[16] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[17] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[18] T. Menzies, R. Krishna, and D. Pryor. The Promise Repository of Empirical Software Engineering Data, 2015.

[19] R. Peters and A. Zaidman. Evaluating the Lifespan of Code Smells using Software Repository Mining. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 411–416, March 2012.

[20] K. Prete, N. Rachatasumrit, N. Sudan, and K. Miryung. Template-based Reconstruction of Complex Refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10, Sept 2010.

[21] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[22] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[23] Aiko Fallas Yamashita and Leon Moonen. Do Developers Care about Code Smells? An Exploratory Survey. In *Proceedings of the 2013 Working Conference on Reverse Engineering, WCRE*, volume 13, pages 242–251, 2013.