

# **TDK-dolgozat**

Kádár István

**Szegedi Tudományegyetem  
Természettudományi és Informatikai Kar  
Szoftverfejlesztés Tanszék**

**Futásidejű kivételek detektálása Java programokban  
szimbolikus végrehajtás segítségével**

Tudományos diákköri dolgozat

Készítette:

**Kádár István**

III. évf. programtervező informatikus  
Bsc.

Témavezetők:

**Dr. Ferenc Rudolf**

egyetemi adjunktus

**Hegedűs Péter**

doktorandusz

Szeged  
2012

## Tartalomjegyzék

|                                             |    |
|---------------------------------------------|----|
| 1. Bevezetés.....                           | 4  |
| 2. Szimbolikus végrehajtás.....             | 5  |
| 2.1. Java PathFinder.....                   | 8  |
| 2.2. Symbolic PathFinder.....               | 11 |
| 3. A futásidejű kivételek detektálása ..... | 16 |
| 3.1. A futtatókörnyezet.....                | 17 |
| 3.2. Listener osztály megvalósítása.....    | 21 |
| 4. Eredmények.....                          | 27 |
| 4.1. Eredmények saját példákon.....         | 27 |
| 4.2. Eredmények valós rendszereken.....     | 31 |
| 4.3. Valódi találatok.....                  | 37 |
| 5. Kapcsolódó munkák.....                   | 42 |
| 6. Összefoglalás és további célok.....      | 46 |
| 7. Irodalomjegyzék.....                     | 47 |
| 8. Köszönetnyilvánítás.....                 | 49 |

## 1. Bevezetés

Nagyméretű, megbízható és robusztus szoftverrendszerek előállítása a szoftverfejlesztés egy nagy kihívást jelentő feladata manapság. A fejlesztési költségek nagyjából 40%-a tesztelési költség, továbbá az üzemeltetés során fellépő hibák megtalálása és kijavítása is jelentős erőforrásokat emészt fel és nagy költségekkel jár. Céлом egy olyan módszer és eszköz kidolgozása, amely a szoftverfejlesztés ezen fázisát támogatja azáltal, hogy a Java programokban képes futásidejű kivételeket felderíteni, valamint olyan veszélyes kódrészletekre rámutatni, amelyek mintegy időzített bombaként viselkednek a szoftver későbbi fejlesztéseinek folyamán. Mindezt anélkül szeretném elérni, hogy a programot valós környezetben futtatni kellene.

A Java programozási nyelv futásidejű kivételei a *java.lang.RuntimeException* osztály példányai, amelyek valamilyen futási hiba, például rossz típuskényszerítés, tömb túlindexelés, vagy nullával való osztás miatt keletkeznek. Ezek a kivételek azért veszélyesek, mert a program azonnali leállítását eredményezhetik, a programozó nem köteles kezelni őket.

A kivételek felderítése a szimbolikus végrehajtás módszerének felhasználásával történik. Szimbolikus végrehajtás során a vizsgálandó programot nem konkrét input adatokon futtatjuk, hanem szimbolikus változókként kezeljük a bemeneteket. Amikor a program végrehajtása egy a szimbolikus inputtól függő elágazáshoz ér, mindkét lefutási ág végrehajtódik. Ekképpen a vizsgált szoftver minden lefutási ága elméletben végrehajtásra kerül.

A Java PathFinder (JPF) egy a NASA Ames Research Center Robust Software Engineering részlegén kifejlesztett nyílt forrású modellellenőrző program, valójában egy Java virtuális gép, amely a Java bytecode speciális végrehajtására alkalmas. E program bővítménye a Symbolic PathFinder, amely a Java bytecode utasításokat úgy interpretálja, hogy a futtatás a szimbolikus végrehajtás elméletének eleget tegyen. Munkámhoz ezeket az eszközöket használom fel.

A dolgozatban kitérek arra, hogy hogyan valósítottam meg a Java programozási nyelv RuntimeException-jeinek detektálását a fent említett virtuális gép és a szimbolikus végrehajtás segítségével, meghatározva a kivételt dobó metódus azon paraméterezését is, amellyel az kivételre fut. Kifejtem a módszert, amellyel a lefutási ágak számát és az állapottér méretét csökkentem. Az elkészített alkalmazást a log4j, az ArgoUML és a jEdit nyílt forrású programokon teszteltem.

## 2. Szimbolikus végrehajtás

Minden program, végrehajtása során, adatokon végez műveleteket egy meghatározott sorrendben. A szimbolikus végrehajtás [9] alapgondolata az, hogy a program az aktuális, konkrét input adatok helyett szimbolikus változókon operáljon, az output pedig ezen szimbolikus változók egy függvénye legyen. Egy szimbolikus változó a program egy konkrét változójának lehetséges értékeinek halmaza, így a program egy szimbolikus állapota is konkrét állapotok egy halmazának feleltethető meg. Amennyiben a végrehajtás egy olyan szelekciós vezérlési szerkezethez ér, amelyben szereplő logikai kifejezés tartalmazza a szimbolikus változók egyikét, akkor azt nem lehet kiértékelni, igaz és hamis egyaránt lehet. Ennek megfelelően a végrehajtás mindkét ágon folytatódik. Ilyen módon a program összes lehetséges lefutása szimulálható.

Szimbolikus végrehajtás során karbantartunk egy ún. path conditiont (PC, útvonal feltétel). A path condition egy kvantormenetes logikai formula, melynek iniciális értéke *igaz*, változói a program szimbolikus változói lehetnek. Ha egy szimbolikus változótól függő elágazásra kerül a futás, a benne szereplő feltételt (logikai kifejezést) *logikai és* operációval hozzákapszoljuk a meglévő path conditionhoz, így jelezve az igaz ágat, majd ennek negáltját is hozzákapszoljuk szintén a már meglévő PC-hez, jelezve a hamis ágat. A PC ilyen módon történő bővítésével minden egyes végrehajtási ághoz egy egyedi, szimbolikus változók feletti formulát fogunk kapcsolni. A szimbolikus végrehajtó rendszerek amellet, hogy karbantartanak egy feltételrendszert, különböző eszközöket, úgynevezett constraint solver (korlátozás kielégítő) programokat is felhasználnak. Constraint solverek a path condition feltételrendszerének megoldására szolgálnak, azaz a benne szereplő szimbolikus változókhoz olyan értéket képesek rendelni, amelyek kielégítik a feltételrendszert. A path condition megoldása a szimbolikus végrehajtás tetszőleges pontján elvégezhető, a megoldások pedig gyakorlatilag olyan tesztinputként szolgálnak, amelyekre futtatva a programot a konkrét végrehajtás azt a lefutási ágat fogja követni, amelyhez tartozó PC-t megoldottuk.

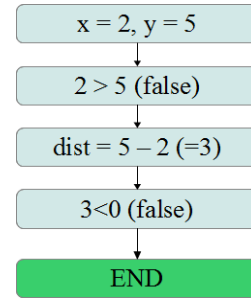
A lehetséges végrehajtási útvonalak egy összefüggő és körmentes irányított gráfot definiálnak, amelyet szimbolikus végrehajtási fának nevezünk. A fa egyes pontjai a program szimbolikus állapotainak felelnek meg. Egy szimbolikus állapot a szimbolikus változókat és az egyéb konkrétan kezelt változók értékeit, a szimbolikus változók feletti aktuális path conditiont, valamint egy utasítás számlálót (program counter) tartalmaz, amely azt mondja meg, hogy melyik a soron következő végrehajtandó utasítás. Az utasítás számláló definiálja a szimbolikus végrehajtási fa éleit.

```

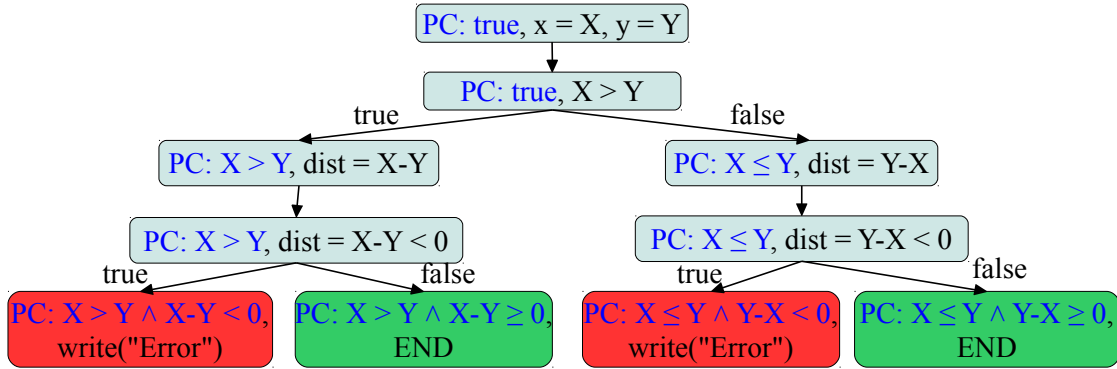
1. int x, y, dist;
2. ...
3. if (x > y) {
4.     dist = x - y;
5. } else {
6.     dist = y - x;
7. }
8. if (dist < 0)
9.     write("Error");

```

(a)



(b)



(c)

**2.1. ábra.** (a) Programrészlet két egész szám számegyenesen mért távolságának meghatározására. (b) A programrészlet egy lehetséges konkrét végrehajtása. (c) A programrészlet szimbolikus végrehajtási fája az  $x$  és  $y$  változók szimbolikus kezelése esetén.

Példaként tekintsük a 2.1-es ábrát. Az (a) része egy egyszerű forráskód részletet mutat, amely két egész típusú változó, az  $x$  és  $y$  számegyenesen mért távolságának meghatározására szolgál. A kódrészlet konkrét végrehajtása  $x = 2$  és  $y = 5$  értékek esetén a 2.1. (b) ábrán látható. A szimbolikus végrehajtást a 2.1. (c) ábra szemlélteti a szimbolikus végrehajtási fa megadásával. Az  $x$  és  $y$  változókat szimbolikusan kezeljük, a szimbólumok  $X$  és  $Y$ . A path condition kezdeti értéke  $true$ . Az első  $if$  utasításhoz érve (3.sor) két lehetőségünk van, a feltétel értéke vagy  $true$ , vagy  $false$ . Ettől függően elágazik a végrehajtás, és a PC-hez hozzáadódik a logikai kifejezés, illetve annak negáltja az alábbiak szerint.

$$true \wedge X > Y \Rightarrow X > Y,$$

valamint

$$true \wedge \neg(X > Y) \Rightarrow X \leq Y.$$

A  $dist$  változó értéke egy szimbolikus kifejezés, a  $true$  ágon  $X-Y$ , a  $false$  ágon  $Y-X$  lesz. A második  $if$  utasítás (8.sor) hatására szintén kétfelé halad tovább a vezérlés a megfelelő path conditionök bővítésével. Itt a  $true$  ágakon az alábbi PC-ket kapjuk.

$$X > Y \wedge X - Y < 0 \Rightarrow X > Y \wedge X < Y,$$

illetve

$$X \leq Y \wedge Y - X < 0 \Rightarrow X \leq Y \wedge X > Y.$$

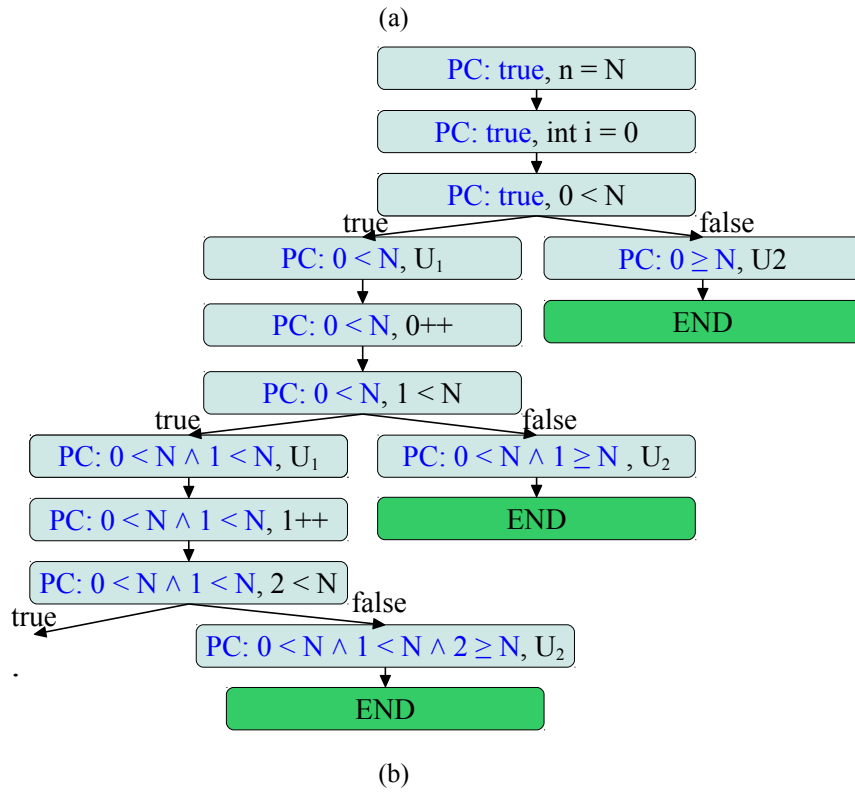
Nyilvánvaló, hogy mindkét formula kielégíthetetlen, nem lehet megadni olyan  $X$ -et és  $Y$ -t, amely eleget tenne a feltételeknek. Ez azt jelenti, hogy nincs olyan  $x$  és  $y$  input, melyeken ha elindítjuk a programot, a vezérlés elérné a `write("Error")` utasítást. Amennyiben egy állapothoz tartozó PC kielégíthetetlen, az innen induló részfa levágható, nincs értelme folytatni az elmentmondásos végrehajtást.

Az ismétléses vezérlési szerkezetek megvalósítása gyakran feltételes (ugró) utasításokkal van megvalósítva alacsonyabb szinten (például assembly, Java bytecode). A szelekciós vezérlési szerkezet szimbolikus végrehajtása kiterjeszthető az ismétléses vezérlési szerkezetekre.

```

1. int n;
2. ...
3. for (int i = 0; i < n; i++) {
4.     U1;
5. }
6. U2;

```



**2.2. ábra. (a)** Egyszerű for ciklus. **(b)** A ciklus szimbolikus végrehajtási fája az  $n$  változó szimbolikus kezelésével.

A 2.2. (a) ábra egy egyszerű for ciklust szemléltet. A ciklusmag az  $U_1$  utasítássorozat, a ciklusból való kilépés után pedig az  $U_2$  utasítássorozat kerül végrehajtásra. A 2.2. (b) ábra a ciklus szimbolikus végrehajtási fáját szemlélteti az  $n$  változó szimbolikus kezelése mellett. Az  $i$  inicializálása után az  $i < n$  feltétel feldolgozása következik, *true* eredmény esetén megkezd-

jük a ciklusmag végrehajtását, *false* esetén az  $U_2$ -vel folytatjuk. Ha beléptünk a ciklusmagba, akkor annak sikeres végrehajtása után  $i$  értéke eggyel megnő, majd  $i < n$  újbóli kiértékelése következik, ami újabb elágazáshoz vezet. A *true* ág végrehajtása esetén tehát mindig visszakerül a vezérlés a ciklus elejére, ami újabb és újabb útvonalak létrejöttét eredményezi. A szimbolikus végrehajtási fa egy elnyújtott, végtelen mélységű fa lesz. Végességet gyakorlatilag csak az jelenthet, hogy a ciklusváltozó eléri értékkészletének határát.

Egy program összes lehetséges állapotának felderítése, az összes lehetséges lefutás végrehajtása elfogadható időn belül lehetetlen. Egyfajta megoldás lehet például a fa mélységének, vagy a szimbolikus állapotok számának korlátozása, ami természetesen meggátolja az összes lehetséges állapot elérését. A következő alfejezetek rávilágítanak majd, hogy a Symbolic PathFinderben milyen módszerek állnak rendelkezésre a probléma kezelésére.

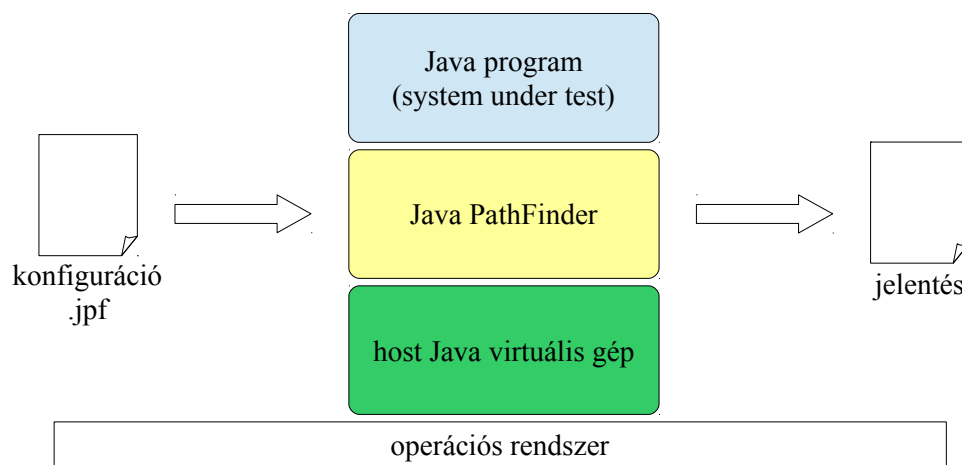
Az eddigiek során csak olyan példát láttunk, ahol egész típusú szimbolikus változók szerepeltek. Manapság egy programban azonban nem csak egész típusú adatok vannak, a valós, a karakter és egyéb primitív típusok mellett összetett adattípusokkal is találkozunk, mint például a tömb, a string, vagy akár a programozó által definiált saját típusok. Ez utóbbiak szimbolikus kezelése, valamint olyan constraint solver készítése, amely meg tud oldani összetett típusú szimbolikus változók feletti formulát, nem triviális. Különböző szimbolikus végrehajtó rendszerek különféle technikákkal vannak felvértezve, a Symbolic PathFinder képességeire szintén a következő alfejezetek térnek ki.

## 2.1. Java PathFinder

A Java PathFinder (JPF) egy nagymértékben testre szabható végrehajtási környezet, ami Java programok verifikációját célozza meg. Fejlesztése 1999-ben a NASA Ames Research Center Robust Software Engineering részlegén kezdődött meg a különböző NASA missziókhöz kapcsolódó szoftverek megbízhatóságának és robusztusságának növelése érdekében. A fejlesztés változó intenzitással azóta is folyik. 2005-ben vált nyíltforrásúvá, NOSA 1.3 licenc alatt szabadon letölthető a projekt weboldaláról [1]. Felhasználása elsősorban a NASA belső projektjeiben és egyetemi kutatásokban számottevő, ugyanakkor a fejlesztők magas letöltési számokról számolnak be anonim felhasználóktól is. A rendszer több díj tulajdonosa, elnyerte például a Turning Goals into Reality díjat 2003-ban és az Outstanding Technology Development díjat 2009-ben.

A JPF valójában nem más, mint egy Java virtuális gép (JVM). Ez azt jelenti, hogy Java programok végrehajtására képes. Inputja Java bytecode (.class kiterjesztésű) fájlok egy hal-

maza, azaz egy lefordított Java program, az a program, amit ellenőrizni szeretnénk (system under test, SUT). A JPF maga is Java-ban van megvalósítva, tehát szintén egy virtuális gépen kell futnia. A különbség a JPF-et futtató virtuális gép és JPF között az, hogy míg a standard Java virtuális gép az Oracle Java Virtuális Gép Specifikációt követi a bytecode utasítások értelmezésében, normál módon végrehajtva a Java programokat, addig a JPF minden bytecode utasítást ettől eltérően, speciálisan interpretál, annak megfelelően, hogy a verifikációt, bizonyos tulajdonságok és hibák meglétének ellenőrzését el tudja végezni. A JPF tehát egy Java nyelven megírt, Java virtuális gép, amely Java programok verifikációjára alkalmas, tulajdonképpen egy köztes réteg a bytecode és a valós JVM között. A rendszer architektúrája a 2.1.1. ábrán látható.



**2.1.1. ábra.** A Java PathFinder, mint egy virtuális gép maga is egy Java virtuális gépen fut, miközben egy Java program verifikációját végzi.

Az, hogy a JPF milyen hibák megtalálását, és milyen tulajdonságok ellenőrzését célozza meg, nehezen meghatározható, elsősorban attól függ, hogy hogyan konfiguráljuk. A rendszert a kezdetektől könnyen bővíthetőre és a lehető legjobban konfigurálhatóra tervezték. Az alábbiakban néhány általános jellemvonását mutatom be.

A JPF képes a programban olyan pontokat azonosítani, ahonnan a program végrehajtása különbözőképpen tud folytatódni (execution choices). Az, hogy ezek az elágazási pontok mik legyenek beállításfüggő, tipikusan különféle szálütemezési sorrendek, vagy egy véletlen szám generálás pontja lehet. A pontok és azokat érintő végrehajtási útvonalak egy gráfot definiálnak, melynek szisztematikus bejárásával ellenőrzi a programot a JPF. A pontok és a kiinduló élek úgynevezett *ChoiceGenerator* osztályokkal definiálhatóak, és a gráf bejárasi stratégiája is megválasztható (a mélységi és a szélességi bejárás előre megvalósított), sőt saját heurisztikákat is megadhatunk.

Az állapotok és végrehajtási útvonalak száma gyakran órási méreteket ölt. Ezt nevezzük

az állapotrobbanás problémájának. A JPF, amint egy elágazási ponthoz ér, megvizsgálja, hogy a pont volt-e már érintve a bejárás során (state matching). Amennyiben ez az állapot már érintve volt, a keresés nem folytatódik, hanem visszalépünk egy korábbi állapothoz, amelyből még indulnak ki bejáratlan élek (backtrack). Annak az eldöntése, hogy két állapot mikor egyezik meg, összetett kérdés. A JPF minden állapotban az egyes programszálakról, statikus és dinamikus változókról tart karban információkat. Az egyenlőség vizsgálatot ezekre az információkra vezeti vissza. További állapottér redukáló módszer a programszeletelés, mely során a JPF kiszűri azokat az utasításokat, amelyek nem relevánsak az ellenőrzendő tulajdonság szempontjából. Ezen utasítások mellőzése az állapotszám csökkenését idézi elő. A JPF-ben alkalmazott hasonló technikákat részletesebben Willem Visser és mások munkája tárgyalja [2].

Az ellenőrzendő tulajdonságokat a felhasználó maga definiálhatja. Ez a JPF-ben úgynevezett listenerek segítségével tehető meg. A listenerek olyan monitorozó osztályok, amelyek segítségével a JPF minden tevékenysége figyelemmel kísérhető. Definiálható például, hogy mi történjen az egyes utasítások végrehajtásakor, objektumok létrehozásakor, vagy megszűnésekor, állapotváltáskor, új állapot elérésekor, vagy visszalépéskor. JPF-ben holtpontok (deadlock) és adatversenyek (data race) detektálásához szükséges listenerek vannak megvalósítva, és ezen tulajdonságok ellenőrzése szerepel is az alapértelmezett konfigurációban.

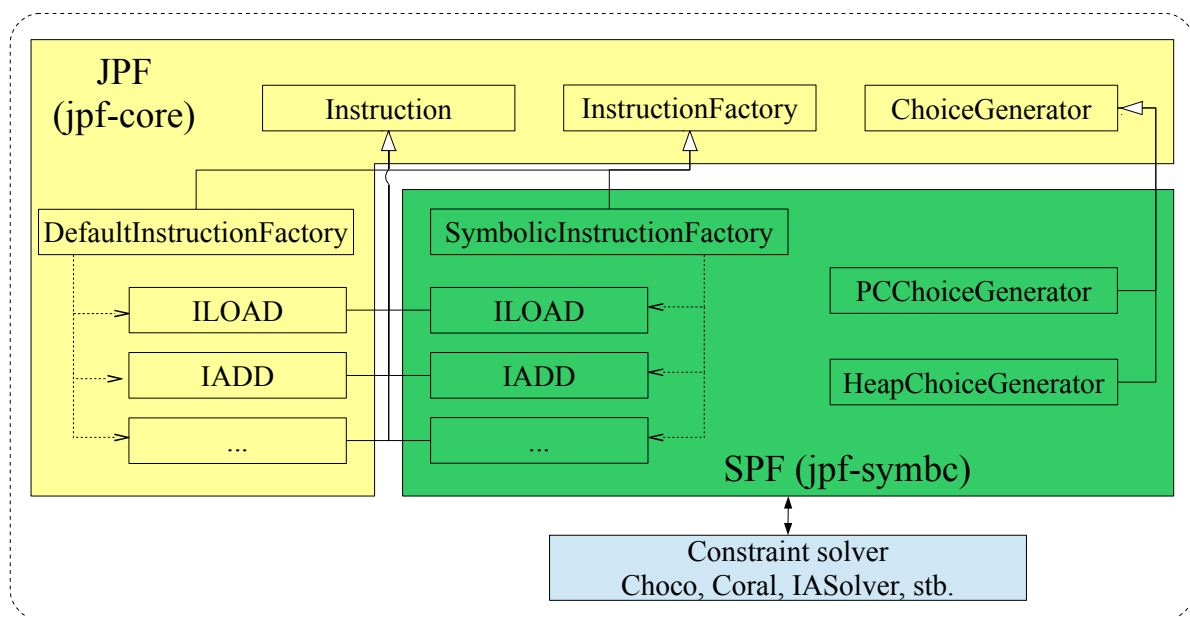
A JPF konfigurálása több szinten történik kulcs-érték párok megadásával. A projekt rendelkezik egy alapértelmezett szöveges konfigurációs állománnyal, ez a `jpf.properties` fájl. Ez a virtuális gép és a verifikálás alapvető beállításait tartalmazza. Itt szerepelnek például az ellenőrzendő tulajdonságok (*search.properties*), a bejárési stratégiát definiáló osztály (*search.class*), de az is, hogy a virtuális gép futtassa-e a szemétgyűjtőt (*vm.gc*). Egy másik szinten érdemes definiálni az ellenőrzendő program sajátosságait. Ezek azok az információk, amiket a standard Java virtuális gépnek is meg kell adni egy Java program elindítása előtt. A legfontosabb a program belépési pontját jelentő `main` függvényt tartalmazó osztály megadása (*target*), valamint az az útvonal, amelyen a JPF a program által felhasznált osztályokat, osztálykönyvtárakat keresi (*classpath*). Ezeket a beállításokat gyakran egy külön `.jpf` kiterjesztéssel rendelkező fájlba írjuk, de parancssori argumentumként is megadhatjuk a JPF indításakor. Az, hogy melyik kulcs-érték párt hol, melyik szinten adjuk meg, lényegtelen, a JPF egyszerre olvassa be az összes konfigurációs állományt és egy közös *Config* típusú objektummal reprezentálja a beállításokat.

A JPF a futtatott elemzések eredményeit jelentések formájában adja vissza. A jelentés formátumát, tartalmát, a kimenet helyét magunk definiálhatjuk a konfigurációban, illetve a

listener osztályokban.

Már több ízben láttuk, hogy egy könnyen testre szabható és bővíthető rendszerről van szó. Ezt támasztja alá az is, hogy a JPF-hez több kiegészítő projekt is létrejött, amelyek mintegy futásidejű modulként kapcsolódnak a Java PathFinder magjához. Mindegyikük saját végrehajtási modellt és tulajdonságokat definiálhat, külön beállításokkal rendelkezhet, de mindegyikhez szükséges a JPF megléte. Az eddigiekben ismertettet JPF a `jpf-core` projektnevet viseli. A `jpf-numeric` projekt például egy olyan bytecode utasításkészletet valósít meg, amely numerikus tulajdonságok és hibák (például túlsordulás) ellenőrzésére alkalmas, a `jpf-awt` célja, hogy lehetővé tegye a modell ellenőrzést GUI-val rendelkező, pontosabban a `java.awt` és a `javax.swing` könyvtárakat felhasználó programokban, a `jpf-statechart` pedig UML állapotdiagramok verifikációját és végrehajtását tudja elvégezni. A szimbolikus végrehajtási környezetet a `jpf-symbc` projekt biztosítja, képességeit a következő alfejezetben fejtem ki. A JPF-re épülő összes projekt listája, és rövid leírása megtalálható a JPF honlapján [1].

## 2.2. Symbolic PathFinder



2.2.1. ábra A Symbolic PathFinder és a Java PathFinder kapcsolata.

A Symbolic PathFinder (SPF, `jpf-symbc` projekt) [3] a JPF olyan kiterjesztése, amely Java programok szimbolikus végrehajtását teszi lehetővé. A logikai, egész és valós típusú változók szimbolikus ábrázolása mellett képes komplex típusú objektumok kezelésére is, valamint többszálú programok szimbolikus végrehajtását is lehetővé teszi. A projekt korábbi megvalósítása [4] a Java bytecode utasítások speciális interpretációja helyett először a bytecode instrumentálását végezte el, majd az instrumentált programot hajtotta végre. Ez az implementáció

tömbök és *String* objektumok szimbolikus kezelésére is képes volt, azonban a legfrissebb, speciális bytecode-értelmezésen alapuló SPF erre még nem képes, csak kezdetleges állapotot érhető el ezen adattípusok támogatásának.

A Symbolic PathFinder lényeges egységei és kapcsolata a JPF-fel a 2.2.1. ábrán látható. Az SPF alapvetően három ponton bővíti a rendszer magját: új utasításkészletet definiál, lecseréli az egyes utasítások példányosításáért felelős *DefaultInstructionFactory*-t, valamint egyéni *ChoiceGenerator*-okat hoz létre a megfelelő elágazási pontok azonosítása érdekében.

```
public class ISUB extends Instruction {
    public Instruction execute(..., ThreadInfo th) {
1.      int v1 = th.pop();
2.      int v2 = th.pop();
3.      th.push(v2 - v1);
4.      return getNext(th); // következő utasítás
    }}
```

(a)

```
public class ISUB extends ...bytecode.ISUB {
    public Instruction execute(..., ThreadInfo th) {
1.      IntegerExpression sym_v1, sym_v2;
2.      sym_v1 = ... .getOperandAttr(0);
3.      sym_v2 = ... .getOperandAttr(1);
4.      if(sym_v1==null && sym_v2==null)
5.          // mindkét szám konkrét, konkrét végrehajtás
6.          return super.execute(..., th);
7.      else {
8.          int v1 = th.pop();
9.          int v2 = th.pop();
10.         // a konkrét érték nem számít, valmit
11.         th.push(0);
12.         IntegerExpression result = null;
13.         if (sym_v2!=null) {
14.             if (sym_v1!=null)
15.                 result = sym_v2._minus(sym_v1);
16.             else
17.                 result = sym_v2._minus(v1);
18.         } else if (sym_v1!=null)
19.             result = sym_v1._minus_reverse(v2);
20.         ... .setOperandAttr(result);
21.         return getNext(th);
    }
}}
```

(b)

**2.2.2. ábra (a)** Az ISUB bytecode utasítás konkrét megvalósítása. **(b)** Az ISUB bytecode utasítás szimbolikus megvalósítása.

Az SPF a Java bytecode utasítások konkrét végrehajtási szemantikáját olyan szemantikára cseréli, amely a szimbolikus végrehajtás elméletének megfelelően működik szimbolikus változókon és kifejezéseken. Példaként tekintsük rögtön a 2.2.2-es ábrát, amelyen az ISUB

utasítás konkrét és szimbolikus megvalósítása szerepel. Az ISUB utasítás két int típusú számot levesz a veremről (2.2.2. (a) 1. és 2. sor), majd a különbségüket visszatesszi (3. sor). Az `execute` metódus visszatérési értéke a következő végrehajtandó utasítás lesz. Szimbolikus esetben azonban már nem ilyen egyszerű az implementáció. Először megvizsgáljuk, hogy a két operandus szimbolikus-e (2.2.2. (b) 4. sor). Amennyiben `sym_v1` és `sym_v2` is `null` az operandusok konkrét értékek, a kivonás megtehető a konkrét szemantika szerint (6. sor). Ha azonban legalább az egyikük különbözik `null`-tól, azaz legalább az egyik operandus szimbolikus az ISUB művelet eredménye is szimbolikus lesz. Az eredmény szimbolikus kifejezést a `_minus()`, illetve a `_minus_reverse()` metódusok állítják elő (15., 17. és 19. sor). Természetesen ebben az esetben is gondoskodnunk kell a verem tartalmáról, két int típusú adatot le kell venni, majd egyet visszatenni (8-11. sor). A veremre visszarakott szám tetszőleges lehet, hiszen az eredmény nem egy konkrét érték, hanem egy szimbolikus kifejezés. A konkrét érték gyakorlatilag eldobható.

Amennyiben a szimbolikus végrehajtás egy feltételes utasításhoz (például IFEQ, IFGT) ér, a `true` és a `false` ág végrehajtása is szükséges. A nemdeterminisztikus elágaztatást a `PCC-choiceGenerator` végzi. A két új útvonalnak megfelelően a szereplő logikai kifejezés, illetve annak negáltja hozzáadódik a karban tartott path condition formulához, majd azt az SPF egy külső constraint solvernek továbbítja. Ha a formula kielégíthető, a keresés folytatódik, különben visszalépés (backtrack) történik.

Az SPF több constraint solver-t is támogat, és egy általános interface-t definiál a velük történő kommunikációra. A `cvc3` lineáris formulák kielégítésére alkalmazható, a `choco` már nemlineáris formulákat is kezel, az `IASolver` pedig intervallum-aritmetikai módszereket alkalmaz a path condition kielégítésére. A támogatott kielégítő rendszerek közül a komplex matematikai feltételek kielégítésére kiélezett `CORAL` bizonyul a leghatékonyabbnak mind a megoldott példák száma, mind a megoldás időigénye tekintetében [5].

Az SPF a Java referencia típusú változóinak szimbolikus kezelésére is lehetőséget nyújt. A módszer a lusta inicializálás (lazy initialization) [6]. Ez az algoritmus a szimbolikusan kezelendő, inicializálatlan, `T` osztályú referencia típusú változót, nemdeterminisztikusan inicializálja (1) null-ra, (2) egy új `T` osztályú objektumra inicializálatlan adattagokkal, és (3) egy a szimbolikus végrehajtás során már korábban inicializált `T` típusú objektumpéldányra. Amennyiben a (2) vagy (3) esetben létrehozott objektum valamely inicializálatlan adattagjára hivatkozás történik a programban, akkor, ha az primitív típusú, egy neki megfelelő szimbolikus változó jön létre, ha referencia típusú, akkor a fent leírt háromirányú elágazás mentén folyta-

tódik tovább a végrehajtás. Ezen választási pontok létrehozása a *HeapChoiceGenerator* feladata. Az SPF nemrég megvalósított funkciója a polimorfizmus támogatása. Ekkor a (2) lépésben a változó nemcsak  $T$ , hanem  $T$  összes leszármazottjának megfelelő osztályú új példányra is inicializálódik, valamint ehhez hasonlóan a (3) lépésben a változóhoz már korábban létrehozott  $T$ , és  $T$ -ből származó osztályú objektumokat rendelünk.

Az szimbolikus állapottér csökkentésére az SPF több lehetőséget is kínál. Megadható a szimbolikus végrehajtási fa maximális mélysége, valamint a path conditionben lévő elemi feltételek száma is korlátozható. További lehetőség, hogy a *symbolic.minint*, *symbolic.maxint*, *symbolic.minreal* és *symbolic.maxreal* opciókkal az egész és a valós adattípusok értéktartományát korlátozhatjuk. Az értékek megfelelő megválasztásával az állapottér és a végrehajtás időigénye jelentősen csökkenthető.

```
1. if (x >= 0) {  
2.   if (y >= f(x)) {  
3.     ...  
4.   }  
5. }
```

**2.2.3. ábra.** Példakód metódushívást tartalmazó feltétellel rendelkező szelekciós utasításra.

A szimbolikus végrehajtás tárgyalása során még nem merült fel a kérdés, hogy a 2.2.3. ábrán látható helyzetet, ahol is a 2. sorban lévő feltételben egy metódushívás szerepel, hogyan kezeljük (az  $x$  és  $y$  változók szimbolikus kezelése mellett). Az SPF háromféle megoldást ad a problémára. A legkézenfekvőbb az, hogy az  $f()$  függvényt is hajtsuk végre szimbolikusan, a visszatérési értéke egy szimbolikus kifejezés legyen. Ez a módszer egyszerűbb esetekben működik, azonban könnyű olyan kódot írni, ahol visszatérési érték nem adható meg egy szimbolikus kifejezéssel. Egy másik megoldás a JPF *native peer* mechanizmusa, amelyet natív könyvtárak és nehezen kezelhető programrészek modellezésére dolgoztak ki. Az SPF a technikát úgy alkalmazza, hogy az  $f(x)$  metódushívást nem hajtja végre, hanem a kódban szereplő formájában beépíti a  $y \geq f(x)$  feltételt a path conditionbe, és így továbbítja azt valamelyik korlátozás kielégítő programnak, amelyet természetesen fel kell készíteni az  $f()$  függvény kezelésére. Az SPF valamennyi a *java.lang.Math* könyvtárban található matematikai függvény esetén így jár el. A harmadik megoldás az úgynevezett concolic (**concrete-symbolic**) végrehajtás [7]. Ennek lényege, hogy két path conditiont tartunk karban. Az egyik tartalmazza azokat a feltételeket, amelyekben nem szerepel metódushívás, a másik az úgynevezett komplex PC, ebben szerepelnek a függvényhívást is tartalmazó feltételek. Az SPF először megoldatja az első PC-t, ami példánkban  $x \geq 0$  (1. sor), a megoldás lehet például  $x = 1$ , majd ezen eredményt felhasználva konkrétan végrehajtja az  $f(1)$  metódushívást. A metódus visszatérési értéke egy

konkrét érték lesz, amely most már konstansként szerepelhet a feltételekben.

A fejezet zárásaként összefoglalom a Java PathFinder és a Symbolic PathFinder legfontosabb, munkám során is felhasznált opcióit (amelyeket a .jpf konfigurációs állományban tudunk használni):

- *target=<classname>*

Annak az osztálynak a nevét kell itt szerepeltetni, amely a program belépési pontját jelentő main metódust tartalmazza.

- *classpath=<classpath>*

Az ellenőrzendő program által felhasznált osztályok és osztálykönyvtárak vesszővel elválasztott keresési útvonala.

- *native\_classpath=<classpath>*

A JPF és az SPF által felhasznált osztályok útvonalának definiálására szolgál. Tipikusan itt szokás megadni a saját listener osztályokat tartalmazó könyvtárat.

- *sourcepath=<sourcepath>*

Az ellenőrzendő program forrásállományainak helye.

- *listener=<classnames>*

Itt a használni kívánt monitorozó listener osztályok neve sorolható fel vesszővel elválasztva.

- *search.depth\_limit=<depth>*

A szimbolikus fa maximális mélységét megadó jellemző.

- *symbolic.method=<methodname>(sym#con#...)*

Ezzel az opcióval definiálhatjuk, hogy mely metódusokat hajtsa végre az SPF szimbolikusan. A metódus teljes nevének magadása után zárójelek között meg kell adnunk, hogy melyik paramétert kezelje szimbolikusán és melyiket konkrétan az SPF egy „#” karakterrel elválasztva. Például *org.apache.log4j.FileAppender.setFile(sym#con#sym#sym)*.

- *symbolic.class=<classname>*

- *symbolic.fields=instance/static/instance,static*

Ezzel a párral definiálhatjuk, hogy mely osztály adattagjait szeretnénk szimbolikusán kezelni. Az osztály- és példánytagok szimbolikus voltát különválaszthatjuk.

- *symbolic.lazy=true/false*

A lusta inicializálás alkalmazásának ki-be kapcsolását tehetjük meg ezzel az opcióval.

- *symbolic.lazy.subtypes=true/false*

A polimorfizmus kezelésének ki-be kapcsolására szolgáló opció.

### 3. A futásidejű kivételek detektálása

Munkám során egy olyan eszközt dolgoztam ki, amely egy tetszőleges Java nyelvű rendszer átvizsgálását képes elvégezni futásidejű kivételek felderítése céljából. Ebben a fejezetben átfogóan, a részletekre kitérve kifejtem, hogy ezt az ellenőrző programot, a Jpf Checkert miként valósítottam meg.

Egy rendszer teljes átvizsgálása a szimbolikus végrehajtás módszerét alkalmazva, a Symbolic PathFinder felhasználásával történik. Az átvizsgálást azonban nem úgy teszem meg, hogy elindítom az egész program szimbolikus végrehajtását, hogy annak minden lehetséges lefutását megvizsgáljam, hanem úgy, hogy a rendszer összes metódusát egyesével, külön-külön hajtatom végre a Symbolic PathFinderrel. Ezáltal a teljes program automatikus, részegységenkénti elemzését írom elő, amely a szimbolikus végrehajtás állapottérének jelentős csökkenését eredményezi.

Fontos kérdés, hogy mit, mely változókat kezeljünk szimbolikusan. Egy metódus lefutásának menete több mindentől függ, elsősorban annak paraméterezésétől és a hivatkozott külső változók értékétől. Ezek az adatok a metódus inputjai, amiket célszerű szimbolikusan kezelni a metódus általános vizsgálata szempontjából. Jelenleg a vizsgált metódus paramétereit és osztályának adattagjait kezelem szimbolikusan, a külső osztályok tagváltozóinak szimbolikus-sá tétele egyelőre nem megoldott. A metódus végrehajtását befolyásolhatják még véletlen változók, valamilyen időzítés és egyéb, programon kívüli tényezők, jelenleg ezeket sem veszem figyelembe.

Céлом nemcsak az, hogy detektáljam azokat a futásidejű kivételeket, amelyeket egy metódus dobhat, hanem az is, hogy meghatározzam a metódus azon paraméterezését, amellyel azt meghívva a vezérlés a jelzett kivételt okozó utasításra fog kerülni. Egy metódus számos más metódust hívhat meg, és a meghívottak is hívhatnak továbbiakat. A kivétel ezekben a metódusokban is keletkezhet több metódushívási mélységben. Amellett, hogy a legfelső szinten lévő metódus egy lehetséges paraméterezését megadjam, amellyel az kivételre fut, a kivételhez tartozó stack trace-ben szereplő összes metódusnak a hibához vezető paraméterezését is meg kívánom határozni. Fontos, hogy csak a legfelső szinten lévő metódus paramétereit kezeljünk szimbolikusan, a további hívások során, egy metódus paramétere, csak abban az esetben lesz szimbolikus, ha annak valamilyen szimbolikus változó, vagy kifejezés lett átadva a meghíváskor.

Munkám alapvetően két részre tagolható. Elsőként szükséges egy olyan futtatókörnye-

zetet, ami az ellenőrzendő program összes metódusán végigiterálva, egyesével el tudja indítani mindegyik szimbolikus végrehajtását, valamint szükséges egy a Java PathFinder listener mechanizmusára épülő egység megvalósítása, amely az aktuálisan vizsgált metódus szimbolikus végrehajtásának monitorozásával képes a lehetséges futásidejű kivételek felderítésére és a kivételre vezető paraméterek meghatározására.

### 3.1. A futtatókörnyezet

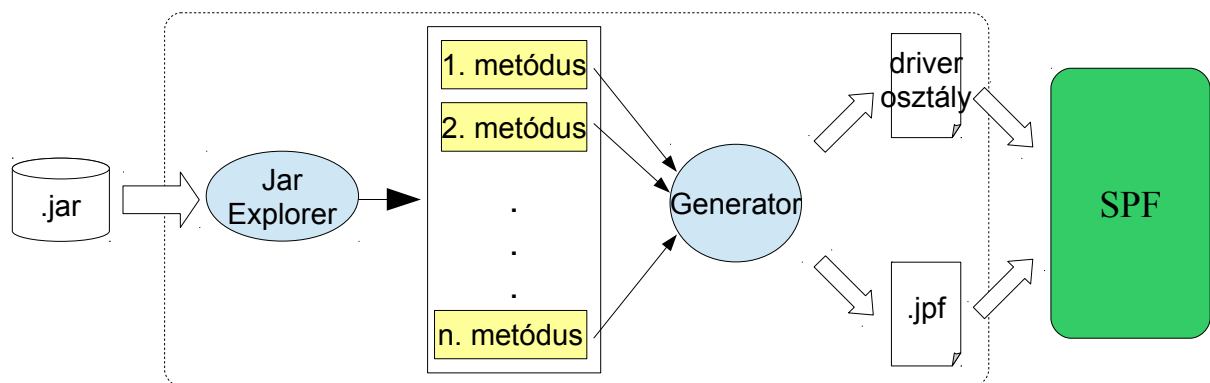
Symbolic PathFinder fejlesztőinek elképzelése az volt, hogy a program futtatása valós környezetben, konkrétan kezdődjön meg, majd egy adott ponton, például a program komplexebb, nagy valószínűséggel hibát tartalmazó részein váltson át a végrehajtás szimbolikusra [8]. Ennek megfelelően működnek az SPF opciói is. A *symbolic.method*, illetve a *symbolic.class* definiálja azokat a metódusokat és osztályokat, melyeket szimbolikusán fog a rendszer kezelni, de a program többi része konkrétan kerül végrehajtásra. A megközelítés előnye az, hogy mivel valós a kontextus, valódi hibák megtalálására nagyobb az esély. Például az egyes globális változók értéke már előre beállított lesz, míg ha azokat is szimbolikusán kezeljük olyan eseteket is végigkövetünk, amely valós futás során soha nem fordulhat elő. Hátránya, hogy a program problémás pontjainak előzetes felderítése nehéz feladat, előismeretet, vagy előzetes munkát igényel. További hátrány, hogy az ellenőrzendő programot manuálisan kell futtatnunk, méghozzá úgy, hogy a vezérlés elérje azokat a pontokat, amelyeket az SPF szimbolikusán fog majd végrehajtatni. Ez például egy komplex grafikus felhasználói felülettel rendelkező program esetén, előzetes fejlesztői ismeretek nélkül időigényes és fáradtságos munka lehet.

Mindezzel szemben az általam kifejlesztett eszköz előzetes konkrét végrehajtás nélkül képes egy megadott metódust, vagy akár a program összes metódusát szimbolikusán végrehajtatni az SPF-fel. A megközelítés előnye, hogy nem szükséges semmilyen manuális futtatás, a teljes folyamat automatizálható, továbbá a szimbolikus végrehajtás állapotterének korlátozottságát is megőrizzük, hiszen nem az egész programot futtatjuk szimbolikusán, hanem csak annak részegységeit külön-külön. A módszer azt is lehetővé teszi, hogy a *main* metódussal nem rendelkező osztálykönyvtárakat is elemezzünk, mint amilyen például a log4j. A hátrányok közé sorolható, hogy a valós végrehajtási környezettől elszakadunk, ami a tényleges programra nem jellemző, „false positive” hibatalálatok megjelenéséhez vezethet.

Egy ilyen végrehajtási környezet megvalósításához mesterségesen elő kell idézni, hogy a vezérlés a megadott, ellenőrizni kívánt metódusra kerüljön. A JPF virtuális gép természete

miatt azonban egy program belépési pontot vár, ami alap esetben a *main* metódus, illetve az azt tartalmazó osztály. Az egyik lehetőség az, hogy ezt megváltoztatjuk úgy, hogy egy másik metódus legyen a belépési pont, az, amit rögtön szimbolikusan végre szeretnék hajtatni. Erre a JPF bizonyos szinten fel van készítve, a belépési pont megváltoztatható, esetünkben viszont mindannyiszor más és más belépési pontra lenne szükség. Ehhez alaposan felparaméterezhető, általános belépési pontot kellene létrehozni, valamint azt is lehetővé kellene tenni, hogy példány metódusok indítására is lehetőség legyen, amihez elengedhetetlen a metódus osztályának példányosítása, ami további bonyodalmakhoz vezet. A JPF ilyen mértékű módosítása nem várt mellékhatásokat eredményezhet.

A JPF módosítása helyett kisebb osztályokat, úgynevezett driver osztályokat hozok létre, amelyek tartalmazzak egy *main* metódust és egyetlen feladatuk, hogy a vezérlést átadják a szimbolikusan végrehajtani kívánt metódusnak és elvégezzék az ezzel kapcsolatos összes teendőt. A különböző metódusok meghívásához más és más driver osztály szükséges, ezért minden metódus elemzése előtt egy új driver osztályt kell létrehozni. A létrehozás az osztály forráskódjának generálásával történik. Az osztály létrehozása mellett egy megfelelő JPF konfigurációs fájlt is generálni kell, melyben megadjuk többek között a program mesterségesen létrehozott belépési pontját és a szimbolikusan végrehajtani kívánt metódust. Az állományok létrehozása után még hátra van a generált Java osztály lefordítása, a műveletsort pedig a Symbolic Pathfinder elindítása zárja.



**3.1.1. ábra.** A futatókörnyezet architektúrája.

A rendszer architektúráját a 3.1.1. ábra szemlélteti. Az input .jar fájlt a *JarExplorer* dolgozza föl, kiolvassa a fájlban található összes osztály összes metódusát, melyekből egy listát állít elő. A metóduslista elemeit egyesével veszi át a Generator, amely egy JPF konfigurációs fájlt és egy driver osztályt generál, az adott metódushoz. A fájlok létrehozása után megkezdődhet az elemzés a Symbolic Pathfinder elindításával.

A Java programozási nyelv lehetőséget ad láthatósági opciók megadására. Az iménti megoldás tárgyalásakor átsiklottunk afölött a tény fölött, hogy egy külső osztály számára a program nem publikus osztályai és metódusai láthatatlanok, így azokat példányosítani, illetve meghívni nem lehet. Nem lehet a szokványos körülmények között. A Java „reflection” mechanizmusa azonban lehetőséget biztosít erre. A Reflection API segítségével a forráskód ismerete nélkül, a .class állományokból lekérhető egy osztály szerkezete, a metódusai és a konstruktorai is. A lekért osztály metódusa meg is hívható, az osztály példányosítható. Mindez azonban még nem elég. Ahhoz, hogy a nem *public* módosítóval ellátott osztályokat és operációkat is elérjük a *Constructor* és a *Method* osztály *setAccessible(true)* metódusát kell meghívni. Ezáltal tetszőleges láthatóságú reflectionnel lekért osztály akkor is példányosítható lesz, ha nem rendelkezik publikus konstruktorral, és tetszőleges láthatóságú metódus meghívása is lehetővé válik a driver osztályban.

```
1. import java.lang.reflect.Method;
2. import java.lang.reflect.Constructor;
3. public class GeneratedMain {
4.     public static void main(String[] args) throws Exception {
5.         Class<?> c = Class.forName("org.apache.log4j.Category");
6.         Constructor<?> cTor =
7.             c.getDeclaredConstructor(Class.forName("java.lang.String"));
8.         cTor.setAccessible(true);
9.         Object[] cTorParameters = { null };
10.        Object anObject = cTor.newInstance(cTorParameters);
11.        Method m = c.getMethod("removeAppender",
12.            Class.forName("org.apache.log4j.Appender"));
13.        m.setAccessible(true);
14.        Object[] parameters = { null };
15.        m.invoke(c.cast(anObject), parameters);
16.    }
17. }
```

**3.1.2. ábra.** Egy driver osztály, amely az *org.apache.log4j.Category.removeAppender(org.apache.log4j.Appender)* metódust hívja meg reflectionnel.

A 3.1.2. ábrán egy driver osztály generált forráskódja látható, amely reflectionnel példányosítja az *org.apache.log4j.Category* osztályt, majd meghívja a *removeAppender()* metódusát. A *Class* objektumot az osztály neve alapján hozzuk létre az 5. sorban, a példányosítás a 10., a példánymetódus meghívása pedig a 15. sorban történik. A két *setAccessible(true)* metódushívás a 8. és a 13. sorban található. A konstruktor és a meghívott metódus aktuális paramétereinek mindig nulla-szerű értékeket adunk meg, boolean esetén *false*-t, char esetén a 0 kódú karaktert, egész és valós változó esetén 0-t, referencia típus esetén pedig *null*-t. A megadott érték gyakorlatilag tetszőleges, szimbolikus végrehajtás során az SPF nem veszi figyelembe a konkrét paramétereket. Az ábrán látható példában a konstruktor egy *String*, a metódus pedig

egy *org.apache.log4j.Appender* referencia típusú paraméterrel rendelkezik.

A *log4j org.apache.log4j.Category.removeAppender()* metódusának szimbolikus végrehajtásához a 3.1.3. ábrán szereplő konfigurációs fájlt szükséges létrehozni. A *target* opció a generált belépési pont, a *GeneratedMain* osztály lesz. A *native\_classpath*-szel láthatóvá teszszük az SPF számára a projekt összes osztályát, így a megvalósított listener osztályt is. A megalkotott ellenőrző eszköz a *jpf\_checker* projektnevet viseli, a két generált állomány pedig a *generated* alkönyvtárban kap helyet. Ennek megfelelően a classpath-ban meg kell adni az ellenőrzendő program mellett a *generated* könyvtárt is. Az 5. és a 6. sorban az SPF lusta inicializálás algoritmusát kapcsoljuk be, a 7-9. sorok pedig a szimbolikusan kezelendő programrészeket definiálják. A szimbolikusan végrehajtandó metódus minden paraméterét és a metódust tartalmazó osztály valamennyi adattagját minden esetben szimbolikusan kezeljük. A 10-11. sorok az állapottér korlátozására szolgálnak, a 12. sorban pedig a constraint solvert definiáljuk, ami jelen esetben a *choco*. A Java futásidejű kivételeinek detektálása az *ExceptionListener* osztályban van megvalósítva, ennek megadása a konfigurációs fájl utolsó sorában található.

```
1. target = GeneratedMain
2. native_classpath = ... /jpf_checker/bin
3. classpath = ... /jpf_checker/generated, ... /log4j-1.2.15.jar
4. symbolic.lazy = true
5. symbolic.lazy.subtypes = true
6. symbolic.class = org.apache.log4j.Category
7. symbolic.fields = instance,static
8. symbolic.method = org.apache.log4j.Category.removeAppender(sym)
9. search.depth_limit = 1000
10. symbolic.minint = -999
11. symbolic.maxint = 999
12. symbolic.dp = choco
13. listener = columbus.jpf.listener.ExceptionListener
```

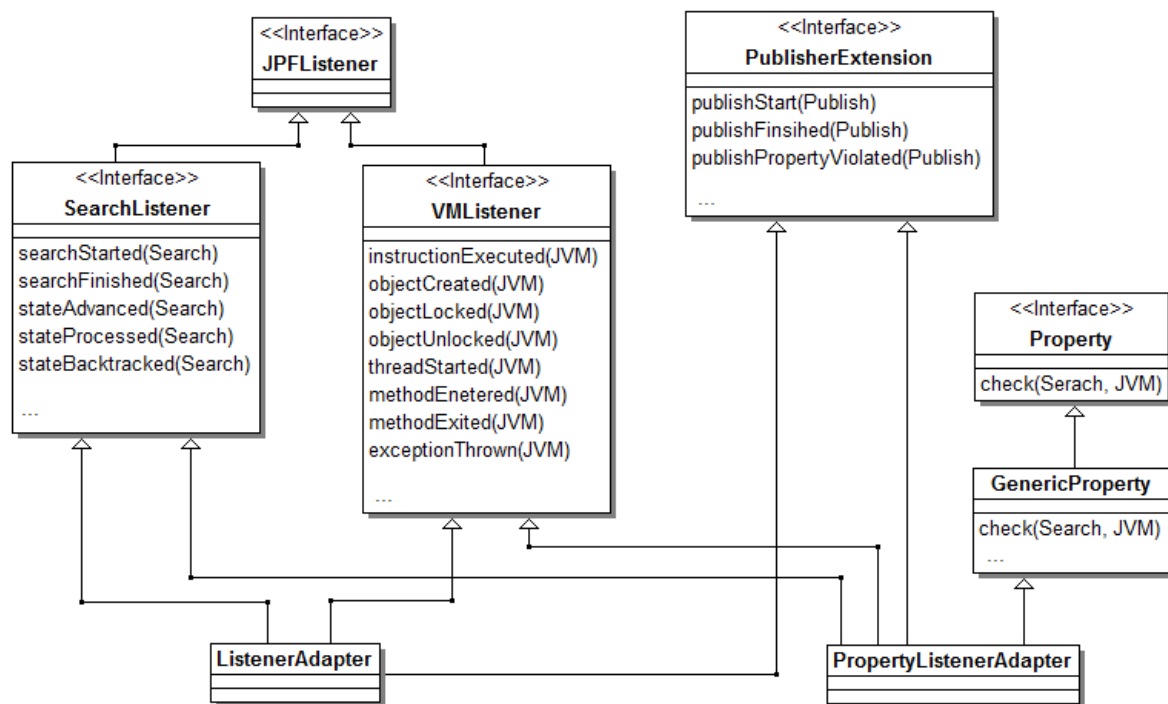
**3.1.3. ábra.** Egy JPF konfigurációs fájl a *org.apache.log4j.Category.removeAppender(org.apache.log4j.Appender)* metódus szimbolikus végrehajtásához.

A Jpf Checker indításához elegendő megadni a vizsgálandó programot, ez egy tetszőleges .jar fájl lehet, és opcionális paraméterként megadható egy teljes metódusnév is. Ha ezt megadja a felhasználó csak ezt az egy metódus vizsgálatát végzi el a program. Amennyiben a megadott nevű metódusból több is szerepel az osztályban, a felhasználó egy listát kap a meg-egyező nevű metódusokról pontos szignatúrával, amelyek közül az egyiket kiválasztva meg-kezdődhet a futásidejű kivételek felderítése. A program parancssori futtatása az alábbi módon történhet:

```
jpf_checker ../log4j-1.2.15.jar org.apache.log4j.Category.removeAppender
```

### 3.2. Listener osztály megvalósítása

A JPF listener mechanizmusa az observer tervezési mintára épül. A beregisztrált megfigyelő objektumok értesülnek bizonyos eseményekről, amelyekre valamilyen módon reagálhatnak. A JPF működésének szinte minden részletéről képes értesítés küldeni, a beregisztrált listenerek számára. Vannak egészen alacsony szintű események, mint például egy bytecode utasítás végrehajtása, de vannak magas szintű események is, mint amilyen az állapottérben való keresés elkezdése vagy befejezése. Alapvetően két típusú listener interface áll rendelkezésre: a *SearchListener* és a *VMLListener* interface. Míg előbbi az állapottérben való keresés eseményeit tartalmazza, addig az utóbbi a JPF virtuális gép részének értesítéseit közvetíti. Mivel mindkét interface hozzávetőlegesen nagyméretű, valamint a konkrét listener osztályok gyakran mindkét interface-t implementálják, a JPF fejlesztői adapter osztályokat készítettek, amelyek üres metódustörzsekkel implementálják ezen interface-ek metódusait. Ennélfogva egy saját listener létrehozásakor elegendő valamely adapterből leszármazott osztályt létrehozni, és csak a szükséges metódusokat kifejteni.



3.2.1. ábra. A JPF listener interface-ének osztálydiagramja.

A JPF listener interface-ének osztálydiagramja a 3.2.1 ábrán található. Az osztálydiagramon két adapter osztály szerepel, a *ListenerAdapter* és a *PropertyListenerAdapter*. Látható, hogy a *SearchListener* és a *VMLListener* mellett mindkettő implementálja a *PublisherExtension* interface-t is, amely jelentések létrehozására használható metódusokat deklarál. Ezen túl, a

*PropertyListenerAdapter* egy *check()* metódussal is rendelkezik, melynek *false* visszatérési értéke jelzi a JPF felé, hogy egy tulajdonság megsérült (például a program holtpontra került), a keresés leállítható. Én a *ListenerAdapter* osztályt származtatom a futásidejű kivételeket detektáló listener elkészítéséhez.

Megoldásom gondolatmenetét, az algoritmust az alábbiakban röviden összefoglalom. A kérdéses metódust szimbolikusan végrehajtva annak valamennyi lefutási ága végrehajtódik, így azok is, amelyek kivételt dobnak. Amennyiben kivétel dobódik egy lefutás során, elsőként el kell dönteni, hogy futásidejű kivételről van-e szó, azaz, hogy a dobott kivételobjektum a *java.lang.RuntimeException* Java osztály példánya-e. Ha igen, az SPF-től elkérjük az aktuális úthoz tartozó path conditiont, amelyet ezután megoldatunk valamelyik korlátozás kielégítő rendszerre. Az eredményül kapott értékeket hozzárendeljük a metódus paramétereikhez, hiszen ezek a megoldások éppen azt a paraméterezést, azt a tesztinputot adják, mellyel a kivételhez jutunk. A hibára vezető paraméterezés több metódushívási szintre történő megadásához, más tényezőket is figyelembe kell venni, ezeket a későbbiekben részletesen kifejtem.

A kivételek felderítésének kulcsa a Java PathFinder *exceptionThrown* eseménye. A *ListenerAdapter*ből származó saját listener osztály ennek az eseménynek megfelelő metódust implementálja a fentiekben leírt algoritmus lépését követve. A némileg egyszerűsített forráskód a 3.2.2. ábrán tekinthető meg.

```
1. public void exceptionThrown(JVM vm) {
2.     if (methodInvokeInfo != null) {
3.         invocationChain = buildInvocationChain(vm);
4.         ...
5.         ExceptionInfo exInfo = vm.getPendingException();
6.         ElementInfo exception = exInfo.getException();
7.         if (!ONLY_RUNTIME || isInstanceOfRuntimeException(exception)) {
8.             // get the current PathConditions
9.             PathCondition pc = ListenerUtils.getPC(vm.getSystemState());
10.            PathCondition heapPc = ListenerUtils.getHeapPC(vm);
11.            if (pc != null || heapPc != null) {
12.                // create summary
13.                FoundExceptionSummary summary = new FoundExceptionSummary();
14.                summary.setException(exception.getClassInfo().getName());
15.                summary.setThrownFrom(...);
16.                summary.setParametrization(parsePC(pc, heapPc,
17.                                                    methodInvokeInfo));
18.                ArrayList<String> stackTrace = new ArrayList<String>(4);
19.                for (MethodInvocationInfo mii : invocationChain) {
20.                    String result = parsePC(pc, heapPc, mii);
21.                    stackTrace.add(mii.getStackFrame().getStackTraceInfo()
22.                                   + " " + result);
23.                }
24.                summary.setStackTrace(stackTrace);
25.                ...
26.                summaries.add(summary);
            }
        }
    }
}
```

```

27.      }
28.    }
29.    ...
30.  }}

```

**3.2.2. ábra.** Az *exceptionThrown* eseményre reagáló kódrészlet.

Valahányszor a végrehajtás során kivételt dob a program (azaz a JPF egy *ATHROW* bytecode utasításra fut), mindannyiszor lefut az *exceptionThrown* metódus. Számunkra azonban csak azok a kivételek lényegesek, amelyek a vizsgált, szimbolikusan végrehajtott metódusból dobódnak (először a driver osztály végrehajtása kezdődik meg, sőt azt is megelőzik a virtuális gép inicializálási lépési, az ott fellépő esetleges kivételekre nem akarunk reagálni). A forráskódrészlet 2. sorában egy ezzel kapcsolatos ellenőrzést találunk. A *methodInvokeInfo* az osztály egy *MethodInvocationInfo* típusú adattagja. A *MethodInvocationInfo* osztályt egy metódus meghívásával kapcsolatos információk összefogására hoztam létre, többek között magában foglalja a metódushoz tartozó vermet, az aktuális paraméterek értékét, és a metódust JPF-ben reprezentáló objektumot. A *methodInvokeInfo* tagváltozó iniciális értéke *null*, ami azt jelzi, hogy még nem léptünk be a vizsgálandó metódusba. Értékének beállítására a *methodEntered* eseményt használjuk. Amennyiben a végrehajtás folyamán beléptünk abba a metódusba, amelyet a konfigurációban szimbolikusra definiáltunk, a *methodInvokeInfo*-t beállítjuk, biztosítva ezzel, hogy a csak olyan kivételekre reagáljunk, ami a metódusból dobódott.

Ahhoz, hogy a hibára vezető paraméterezést minden metódushívási szintre meghatározzuk, elsőként fel kell építeni a stack trace-t, azaz a felső szinten lévő eredetileg vizsgálni kívánt metódus és a kivételt közvetlen dobó metódus közötti hívási láncot. Ezt a 3. sorban meghívott *buildInvocationChain()* operáció fogja elvégezni, melynek visszatérési értéke egy *MethodInvocationInfo* objektumokat tartalmazó lista lesz.

Az 5. sorban elkérjük a JPF-től a dobott kivételt egy *ExceptionInfo* objektum formájában, következő sor pedig egy *ElementInfo* osztályú objektumot hoz létre, ami magát a kivétel-objektumot hivatott tárolni. A JPF-ben az *ElementInfo* kis eltéréssel ugyan, de a Java nyelv *Object* osztályának felel meg.

A továbbiakban annak a vizsgálata szükséges, hogy a dobódott kivétel futásidejű kivétel-e, azaz hogy a *RuntimeException* Java kivételosztály leszármazottjáról van-e szó, vagy sem. A kérdés eldöntésére az *isInstanceOfRuntimeException* metódust használom. A metódusban a kapott *MethodInfo* paraméter ősosztályait vizsgálom meg az öröklési hierarchiában felfelé haladva. Amennyiben a *RuntimeException* osztály itt szerepel, futásidejű kivételről beszélünk, ha azonban egy *Object* típusú osztály a soron következő ős, a keresés befejezhető, a ki-

vétel nem futásidejű. Az *ONLY\_RUNTIME* konstans értékét *false*-ra állítva ez viselkedés megváltoztatható, ekkor minden dobott kivételre reagálni fog a listener.

A 9. és 10. sorban az aktuális path conditiont kérjük el az SPF-től. Látható, hogy két *PathCondition* objektumról van szó. Az objektumreferencia összehasonlításokat (például *obj1 == obj2*, *obj1 != null*, *obj2 == null*) az SPF külön, a numerikus PC-től szeparáltan, a *HeapPC*-ben tartja karban. Ha legalább az egyik *PathCondition* objektumot már inicializálta a rendszer (vagyis már elindult a szimbolikus végrehajtás), megkezdődhet a path condition megoldása és a megoldások hozzárendelése a metódusok paramétereikhez.

Ezeket a feladatokat a *parsePC()* metódus végzi el. A metódus a két *PathCondition* és egy *MethodInvocationInfo* osztályú objektumot vár paraméterként, visszatérési értéke pedig egy *String*, amely a *MethodInvocationInfo* objektumhoz tartozó, hibára vezető konkrét paraméterlista szöveges reprezentációja lesz. A normál PC és a *HeapPC* megoldása után az eredmények a *PathCondition* objektumok belső szerkezetében tárolódnak. Az, hogy melyik eredmény melyik szimbolikus változóhoz tartozik legegyszerűbben név alapján kereshető vissza. A szimbolikus változónevek a metódushoz tartozó verem megfelelő indexein található szimbolikus változók szöveges reprezentációjából nyerhető ki. Terjedelmi okok miatt ennek technikai részleteit nem tárgyalom. Ha megszereztük a szimbolikus változóneveket a PC megoldásból visszakeresett értéket hozzáfűzzük az eredmény paraméterlistához.

Ezzel azonban még nem vagyunk kész. Nem feltétlenül szerepel minden szimbolikus változó a PC-ben. Ha egy paraméterre a forráskód nem ír elő semmilyen megszorítást a kivételhez vezető útvonalon, akkor az a változó semmilyen formában nem szerepel a PC-ben, ennélfogva megoldás sem fog tartozni hozzá. Ez azt jelenti, hogy akármilyen érték is szereplejen e paraméter helyén az aktuális paraméterlistán az nem befolyásolja lefutást, a metódus mindenképpen kivételre fog futni a többi paraméter értéke mellett. Ilyen esetekben a *parsePC()* metódus a „don't care” szöveget fűzi a paraméter helyére.

Az is előfordulhat, hogy egy paraméter konkrét. Pillantsunk az 3.2.3. ábrára. Tegyük fel, hogy az *x()* metódust hajtjuk végre szimbolikusan. Az *x()* meghívja az *y()* metódust az *a* és a *b* paraméterekkel (3.sor), amelyek közül az *a* szimbolikus a *b* viszont konkrét. Ekkor, ha *y()* dob egy kivételt, és *x()* paraméterezésén túl a hívási láncban szereplő *y()*-ra is meg szeretnénk határozni a kivételt reprodukáló értékeket, a *b* változónak a konkrét értékét kell figyelembe vennünk, ami jelen esetben 42. A konkrét érték ilyen esetben a szintén a metódushoz tartozó veremből kapható meg.

```

1. void x(int a) {
2.   short b = 42;
3.   y(a, b);
4. }

```

```

5. void y(int a, short b) {
6.   ...
7.   throw new NullPointerException();
8.   ...
9. }

```

**3.2.3. ábra.** Példakód, melyben az *y()* metódus meghívásakor konkrét és szimbolikus paramétereket is átadunk.

Referencia típusú szimbolikus változók esetén a path condition azok primitív típusú adattagjaira tartalmaz elemi megszorításokat. Például, ha van egy *t* objektum, amely rendelkezik egy *a* int típusú tagváltozóval, a path conditionben *t.a > 0* alakú korlátozások szerepelnek. Ezzel kapcsolatban sajnos az a tapasztalatom, hogy az SPF hibásan kezeli az ilyen neveket. Ha például egy korlátozás *t1.t1.t3.a > 0* alakú lenne, azaz ha a *.* operátor többszöri alkalmazása szerepel, az SPF ezt tévesen a *t1.a* szimbolikus azonosítóval szerepelteti. Emiatt referencia típusú szimbolikus változók esetén a constraint solver megoldásainak eredményét adattagokra nem keresem vissza a *parsePC()* metódusban, hanem csak azt az információt adom vissza, hogy a referencia típusú változó null-e vagy sem azon az útvonalon, amelyen a metódus kivételre fut. Sok esetben ennyi információ is hasznosnak bizonyul.

Egy megtalált kivételről minden információt egy *FoundExceptionSummary* típusú objektumban tárolok el. Egy *FoundExceptionSummary* objektum tartalmazza a megtalált kivétel osztályának pontos nevét, egy a *parsePC()* metódus által visszaadott lehetséges paraméterezést, amellyel a metódus erre a kivételre ráfut, és még a kivétel stack trace-ét is, melynek elemeihez tartozó paraméterezés is szerepel. A *setThrownFrom()* metódussal megadható az a forráskódsor, amely a kivételt dobja. Ez utóbbihoz meg kell adni a JPF konfigurációs állományban a *sourcepath* opciót is, amely a program forráskódjának helyét jelöli.

A megtalált kivételeket összefoglaló jelentés elkészítéséhez a *PublisherExtension* interface *publishFinsihed()* metódusát használom, amely a szimbolikus végrehajtás végeztével elkészít egy összefoglalót.

JavaPathfinder v6.0 - (C) RIACS/NASA Ames Research Center

```

===== system under test
application: columbus/jpf/test/Example.java

===== search started: 2012.04.10.
===== Found Exception Summary:
                                           callRun(int,int,java.lang.Object)

Exception:
  java.lang.ArrayIndexOutOfBoundsException
Exception thrown from:
  columbus/jpf/test/Example.java:26 ( tomb[101] = i; )
Parametrization:
  callRun(7, 11, !null)

```

```

Stack trace:
  columbus.jpjf.test.Example.callRun(Example5.java:14)    callRun(7, 11, !null)
  columbus.jpjf.test.Example.run(Example5.java:26)      run(7, 11)

Exception:
  java.lang.NullPointerException
Exception thrown from:
  columbus.jpjf.test.Example.java:15  ( i.byteValue(); )
Parametrization:
  callRun(7, don't care, null)
Stack trace:
  columbus.jpjf.test.Example.callRun(Example5.java:15) callRun(7, don't care,null)
===== statistics
elapsed time:      00:00:00
states:           new=3, visited=0, backtracked=3, end=0
search:           maxDepth=2, constraints hit=0
choice generators: thread=1 (signal=0, lock=1, shared ref=0), data=1
heap:             new=410, released=4, max live=369, gc-cycles=1
instructions:     3026
max memory:       117MB
loaded code:      classes=93, methods=1099

===== search finished: 2012.04.10. 17:00

```

### 3.2.4. ábra. Egy jelentés a felderített kivételekről.

A 3.2.4. ábrán két futásidejű kivételt felderítő elemzés jelentése látható. A megtalált kivételek listája a Found Exception Summary részlegben szerepel. Az ellenőrzött metódus a *callRun(int,int,java.lang.Object)*, amely egy *ArrayIndexOutOfBoundsException* és egy *NullPointerException* kivételt dob. Az Exception thrown from rész mutatja meg, hogy a kivétel pontosan melyik forrásfájl, hanyadik soráról érkezett, és zárójelben szerepel a problémás forráskódsor is. A Parameterization alatt egy aktuális paraméterlista szerepel, mellyel meghívva a metódust az ráfut a jelentésben szereplő kivételre. Az ábrán lévő első kivétel esetén az látható, hogy ha a 7 és 11 paraméterekkel futtatjuk a metódust, továbbá a harmadik paraméter egy nem null referencia, az egy *ArrayIndexOutOfBoundsException* kivételt fog dobni. A jelentésben szerepel még egy stack trace is, amely azt mutatja, hogy pontosan honnan dobódott a kivétel. A stack trace minden eleme mellett szerepel a lehetséges problémás paraméterezés is. Jelentés végén a JPF által elkészített statisztika kap helyet.

## 4. Eredmények

Az elkészített alkalmazást többféle módon teszteltem, a fejezet ezen futtatások eredményeit taglalja. Elsőként magam írt, futásidejű kivételeket dobó kisebb példakódokat vizsgáltam, majd valós, nyíltforrású programok elemezését végeztem el, amivel igazolom, hogy nemcsak mesterségesen előállított programokban képes az alkalmazás futásidejű kivételeket felfedezni. Tesztalanyom volt a log4j naplózórendszer, az ArgoUML modellezőszoftver és a jEdit szövegszerkesztő program is. Ezekben a rendszerekben talált kivételek valódiságát azzal bizonyítottam, hogy a projektekhez tartozó hibakövető rendszerekben találtam olyan hibákat, illetve hibajelentéseket, melyek oka egy a fejlesztett eszköz által is jelentett RuntimeException. Először néhány egyszerűbb és kisebb példakódot és a hozzájuk tartozó jelentéseket tárgyalom, majd az említett nagyobb rendszerek elemezésének eredményit szemléltetem, végül a hibakövető rendszerekben is megtalált esetekről írok.

### 4.1. Eredmények saját példákön

```
public class Example5 {  
    ...  
8.   public void callRun(int x, int y) {  
9.       Integer i = null;  
10.      if (x > 6) {  
11.          int b = 9;  
12.          run(b, y);  
13.          i = Integer.valueOf(b);  
14.          System.out.println(i);  
15.      } else {  
16.          i = Integer.valueOf(3);  
17.          System.out.println(i);  
18.      }  
19.  }  
20.  public void run(int x, int y) {  
21.      if (y > 10) {  
22.          int[] tomb = new int[5];  
23.          for (int i = 0; i < x; i++) {  
24.              tomb[i] = i;  
25.          }  
26.      } else {  
27.          Integer i = null;  
28.          if (y < 5) {  
29.              i = Integer.valueOf(4);  
30.              i.floatValue();  
31.          } else {  
32.              System.out.println(i.floatValue());  
33.          }  
34.      }  
35.  }}
```

(a)

```

JavaPathfinder v6.0 (rev 648+) - (C) RIACS/NASA Ames Research Center
===== system under test
application: columbus/jpf/test/Example5.java
===== search started: 04.11. 11:51
===== Found Exception Summary:
                                             callRun(int,int)

Exception:
    java.lang.ArrayIndexOutOfBoundsException
Exception thrown from:
    columbus/jpf/test/Example5.java:24 ( tomb[i] = i; )
Parametrization:
    callRun(7, 11)
Stack trace:
    columbus.jpf.test.Example5.callRun(Example5.java:12)    callRun(7, 11)
    columbus.jpf.test.Example5.run(Example5.java:24)      run(9, 11)

Exception:
    java.lang.NullPointerException
Exception thrown from:
    columbus/jpf/test/Example5.java:32 ( System.out.println(i.floatValue()); )
Parametrization:
    callRun(7, 5)
Stack trace:
    columbus.jpf.test.Example5.callRun(Example5.java:12)    callRun(7, 5)
    columbus.jpf.test.Example5.run(Example5.java:32)      run(9, 5)

===== statistics
elapsed time:      00:00:00
states:           new=7, visited=0, backtracked=7, end=1
...

===== search finished:04.11. 11:51

```

(b)

**4.1.1. ábra (a)** Példaprogram, melynek *callRun()* metódusát vizsgáljuk. **(b)** Jelentés a futtatás eredményéről.

Az 4.1.1. (a) ábrán szereplő program az első példa. A vizsgált metódus a *callRun()*, amely a forráskód 12. sorában hívja az osztály *run()* metódusát. Az ábra (b) részén az elemzésről készült jelentés látható. Láthatjuk, hogy két *RuntimeException* találat született, az első egy *ArrayIndexOutOfBoundsException*, a második pedig egy *NullPointerException*. Az első kivételt a *run()* metódus, a kód 24. sora dobta, melyhez a *callRun(7, 11)* paraméterezés vezetett. Látható, hogy a *run()* metódus csakis akkor hívódik meg, ha  $x > 6$  (10.sor), ezt a 7 kielégíti. A *run()* metódus meghívása a 9-es konkrét értékű *b*-vel és a szimbolikus *y* változóval történik, *y*-ra ekkor még nincs semmilyen korlátozás. A *run()* metódus kizárólag akkor futtat rá a 24. sorra ha  $y > 10$ , a jelzett 11-es paraméterértéket ezen megszorítás kielégítésével kapható meg. A kivétel létrejöttének oka az, hogy 22. sorban egy 5 elemű tömböt hozunk létre, és az azt követő ciklus 0-tól *x*-ig fut. Az *x* változó értéke ezen a ponton 9, amely a tömb túlindexeléséhez vezet. A stack trace mellett látható, hogy az egyes metódusok mely paraméterezése vezetett a tömbtúlindexeléshez. A második kivétel esetén is hasonló a gondolatmenet. Itt a prob-

léma az, hogy a 27. sorban létrehozott *i* változót csak a 29. sorban inicializálódik egy *null*-tól különböző értékre, az else blokkban nem, ezért a 32. sor egy *NullPointerException* fog dobni. Ehhez az szükséges, hogy a *run()* metódusban *y* értéke ne legyen nagyobb, mint 10 és ne legyen kisebb, mint 5 (másként fogalmazva egy 5 és 10 közötti szám kell, hogy legyen). Ezeket a korlátozásokat kielégíti az 5, a hibára vezető paraméterezésben a 7-es érték pedig a már tárgyalt módon azért szükséges, hogy a *run()* metódus egyáltalán meghívódjon. A statisztika első sorában az elemzés időtartama szerepel. Látható, hogy kevesebb mint egy másodperc alatt véget ért.

```

...
3. public class Example3 {
    ...
8.   public void run(int n,
        boolean check, A a) {
9.       boolean b = check && n >= 0;
10.      int max = Integer.MIN_VALUE;
11.      if (b) {
12.          if (a != null) {
13.              int l = n;
14.              int r = 2*n + 1;
15.              if (a.getMember() > 120) {
16.                  if (l <= a.getMember()) {
17.                      max = a.getMember();
18.                  } else {
19.                      max = l;
20.                  }
21.                  if (r > max) {
22.                      max = r;
23.                  }
24.                  while (max < n) {
25.                      max = doit(n, 0);
26.                  }
27.              }
28.          }
29.      } else {
30.          max = doit(n, 0);
31.      }
32.      System.out.println("Maximum"
33.          + value: " + max);
34.  }

35. private int doit(int n, int m) {
36.     double res = count(m);
37.     if (res > n) {
38.         do {
39.             res = n / res;
40.             res -= 2;
41.         } while (res >= 0);
42.         return n + m;
43.     } else {
44.         return (int)res;
45.     }
46. }
47.
48. private int count(int l) {
49.     int count = 1;
50.     for (int i=100; i>0; i--) {
51.         if (i % 3 == 0) {
52.             count++;
53.         }
54.     }
55.     return count;
56. }
57.
58. }

1. public class A extends Letter {
2.     ...
3.     public int member;
4.
5.     public int getMember() {
6.         return member;
7.     }
8.     ...
9. }

```

(a)

```

JavaPathfinder v6.0 - (C) RIACS/NASA Ames Research Center
===== system under test
application: columbus/jpf/test/Example3.java
===== search started: 04.11. 14:55
===== Found Exception Summary:
run(int,boolean,A)

Exception:

```

```

    java.lang.ArithmeticException
Exception thrown from:
    columbus/jpf/test/Example3.java:39  ( res = n / res; )
Parametrization:
    run(-999, true, don't care)
Stack trace:
    columbus.jpf.test.Example3.run(Example3.java:30) run(-999, true, don't care)
    columbus.jpf.test.Example3.doit(Example3.java:39)  doit(-999, 0)

===== statistics
elapsed time:      00:00:00
states:           new=15, visited=9, backtracked=23, end=4
...
===== search finished: 04.11. 14:55

```

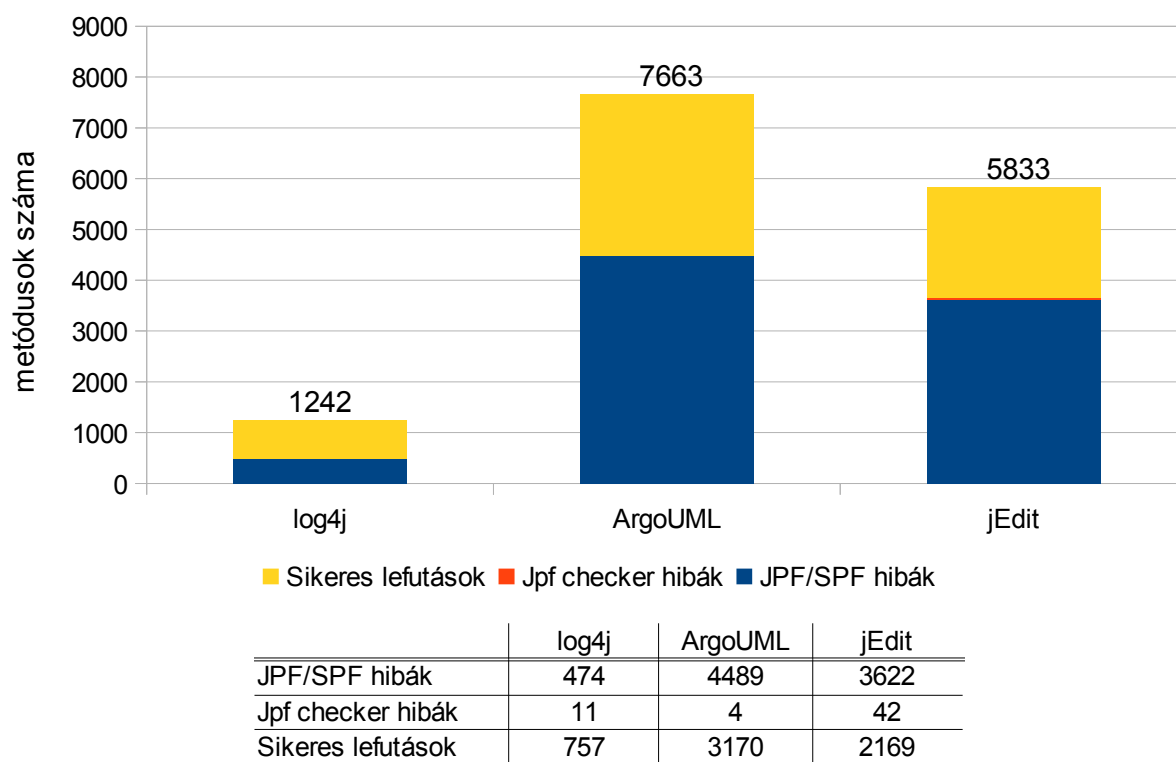
(b)

**4.1.2. ábra. (a)** Példaprogram, melynek *run()* metódusát vizsgáljuk. **(b)** Jelentés a futtatás eredményéről.

A második példaprogram és a hozzá tartozó jelentés az 4.1.2. ábrán szerepel. A jelentés egy *ArithmeticException*-ről számol be, ami a 39. sorról dobódik. Már a jelentésben látható, hogy a sor egy osztásműveletet tartalmaz, ami arra enged következtetni, hogy valószínűleg egy 0-val való osztás történt. A stack trace-ből kiderül, hogy a *run()* metódus 30. sorában meghívott *doit()* metódussal van gond. A 30. sorra kizárólag akkor kerülhet a vezérlés, hogyha a 11. sorban megvizsgált *b* változó értéke *false*. A *b* változó értéke a *check* és az *n* paraméterektől függ. Amennyiben például az *n* -999, a *check* pedig *true* értékkel bír, ahogy ez a jelentésben szereplő paraméterlistában szerepel, a *b* hamis lesz és az else blokkban a *doit()* metódushívásra kerül a vezérlés. A *doit()* első paramétere a szimbolikusan kezelt *n*, második paramétere pedig egy konkrét érték. A 36. sorban a *res* változó egy konkrét értéket fog felvenni, mert a *count()* metódus végrehajtása is konkrétan történik meg. Belátható, hogy értéke mindenképpen egy nemnegatív szám lesz, ennél fogva a 37. sorban lévő feltétel *n=-999* esetén igaz értéket fog felvenni. A ciklus végrehajtás megkezdődik, és bizonyos számú iteráció lefutása után a *res* 0-ra csökken, ami egy 0-val való osztáshoz vezet. A jelentésben szereplő paraméterezésen a harmadik, referencia típusú paraméter helyén a „don't care” szöveg szerepel. Ennek az az oka, hogy a paraméter nem játszik szerepet abban, hogy a metódus ráfut-e tárgyalt kivételre vagy sem, bármilyen értéket felvehet, ha -999 és *true* a megelőző két paraméter, a metódus egy *ArithmeticException*-t fog dobni. A *run()* metódus 25. sorában is szerepel a *doit(n, 0)* függvényhívás, ide tartozó kivétel azonban nem szerepel a jelentésben. Első gondolatunk az lehet, hogy ennek oka az hogy, a 37. sorban lévő feltétel nem teljesül, és nem éri el a vezérlés a do-while ciklust, valójában azonban a 13-23. sorokban lévő utasításoknak köszönhetően a 24. sor feltétele soha nem fog teljesülni, így be sem lépünk a ciklusmagba.

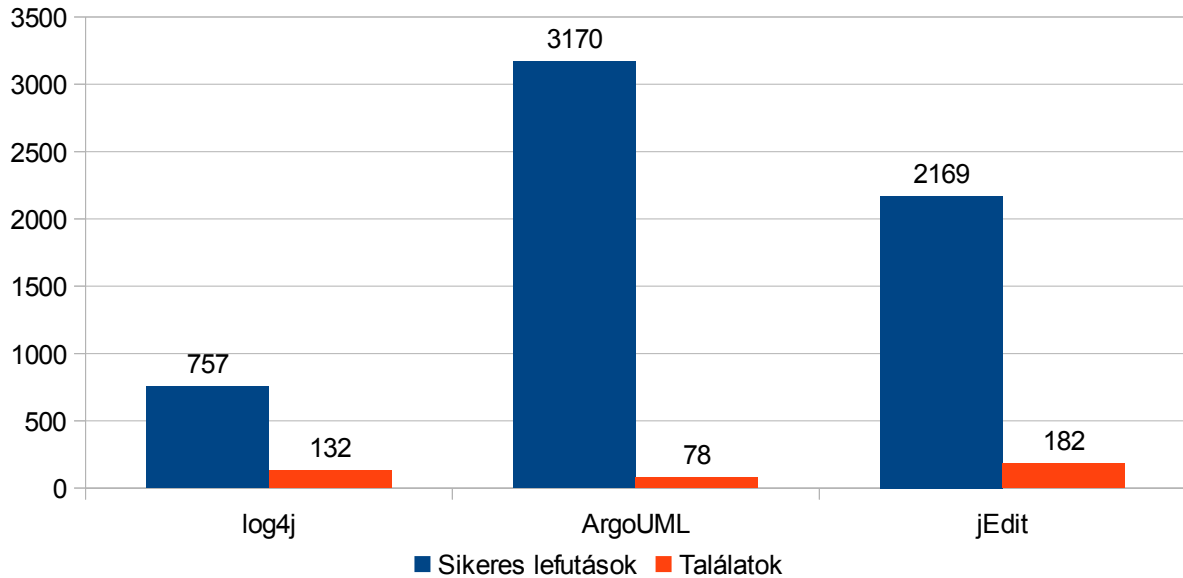
## 4.2. Eredmények valós rendszereken

Az ArgoUML, a log4j és a jEdit programok elemzését egy asztali számítógépen végeztem el, amely egy Intel Core i5-540M 2,53 GHz processzorral és 8 GB memóriával rendelkezik. A log4j 1.2.15-ös, az ArgoUML 0.28-as, a jEditnek pedig a 4.4.2-es kiadása a tesztalany. Az elemzés mindhárom esetben a programokat reprezentáló .jar állományban lévő összes metódus szimbolikus végrehajtásával történt.



**4.2.1. ábra.** Az elemzett programokban vizsgált metódusok száma a hibásan véget ért futások feltüntetésével.

Az 4.2.1. ábra a programok elemzése során megvizsgált metódusok számát jeleníti meg egy halmozott oszlopdiagramon. A log4j-ben összesen 1242 metódus vizsgálatát indítottam el, amelyből csupán 757 futott le sikeresen, 474 esetben a JPF (vagy az SPF) működése hiba miatt leállt. Az ArgoUML metódusai között is rengeteg olyan van, amellyel a JPF nem tudott megbirkózni, az elindított 7663 metódusvizsgálat több, mint fele sikertelenül zárult, és a jEdit esetén is hasonló arány mutatkozik. Az elkészített eszköz, a Jpf Checker miatti hibás lefutások száma lényegesen kisebb. Sajnos általánosságban elmondható, hogy a JPF nagyon sok esetben különféle hibaüzenetek kíséretében váratlanul leáll.



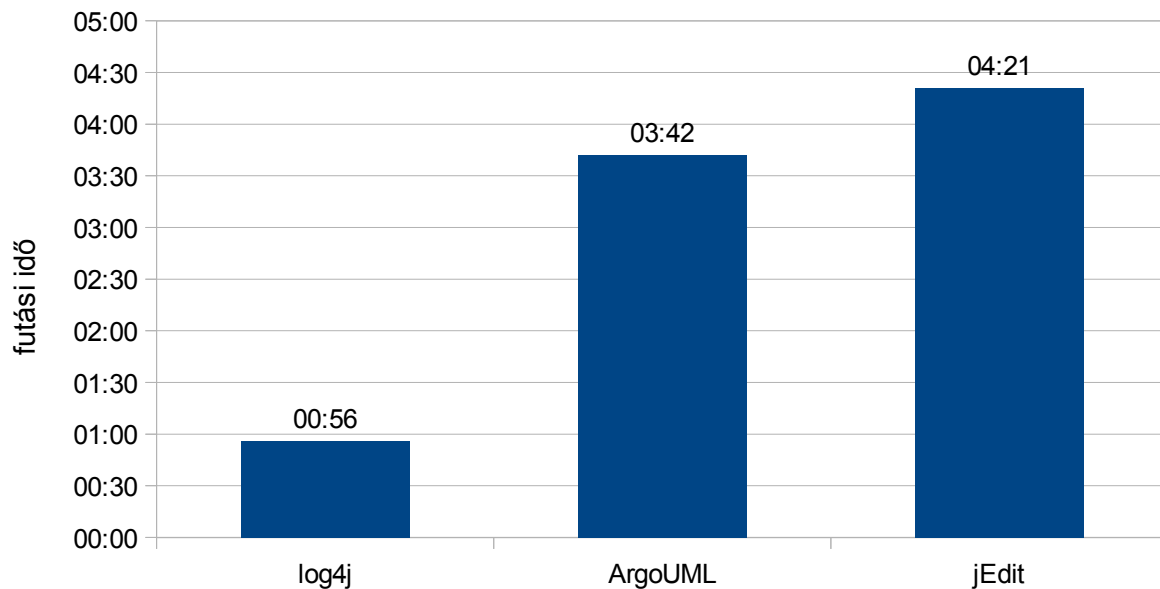
**4.2.2. ábra.** A sikeresen elemzett, és az ezek közül kivételt dobó metódusok száma.

A JPF gyakori hibás működésének ellenére a Jpf Checker szép számmal talál futásidejű kivételeket mindhárom programban. Az 4.2.2. ábra azt mutatja, hogy a sikeresen leelemzett metódusok közül hányban találtunk egy, vagy több futásidejű kivételt. A találatok aránya a log4j-ben a legmagasabb, az ArgoUML-ben pedig a magas metódusszám ellenére viszonylag kevés kivételt dobóra találtunk rá.

Fontos megemlíteni, hogy nem minden felderített kivétel jelent valós hibát. Az ellenőrzés egy mesterséges végrehajtási környezetben történik, ami „false positive” találatok megjelenését eredményezheti. Amikor egy metódus szimbolikus végrehajtását elindítjuk annak a valós futtatás során bekövetkező meghívásainak körülményeiről semmilyen ismeretünk nincs. Minden paramétert, és adatot szimbolikusán kezelünk, azaz úgy tekintjük, hogy értékük bármi lehet, holott lehetséges, hogy valós környezetben egy konkrét állapot, egy adott változóérték nem fordulhat elő.

Annak ellenére, hogy nem minden jelentett kivétel képvisel valódi hibát, mindenképpen valódi veszélyforrást jelent. A szoftver módosításkor óhatatlanul előfordul, hogy olyan változtatások kerülnek a forráskódba, amelyek valamilyen hibát eredményeznek. Ezen hibák gyakran valamilyen futásidejű kivétel formájában jelennek meg. Például publikus osztályok publikus metódusait a program tetszőleges pontján, tetszőleges aktuális paraméterlistával meghívhat bárki, nem sejtve, hogy ez egy *RuntimeException* kiváltója lehet, amely a program azonnali terminálását eredményezheti. A jelentett kivételek többsége tehát nem valódi hibáról számol be, hanem potenciális veszélyforrásokról értesít. Minden találat intő jel lehet, amelyre ér-

demes kitüntetett figyelmet fordítani.



**4.2.3. ábra.** Az elemzések időtartama a három program esetén.

Az elemzések időtartama a 4.2.3. diagramon szerepel. A log4j elemzése egy órán belül befejeződött, a több mint 7500 metódust tartalmazó ArgoUML vizsgálata pedig 3 óra 42 percig tartott. A jEdit, bár kevesebb metódust tartalmaz, mint az ArgoUML, elemzése több időt vett igénybe. A vizsgálatok időigénye elfogadhatónak mondható különösen annak tükrében, hogy az elemzés egy hétköznapi számítógépen, és nem egy nagy teljesítményű szerveren történt, ugyanakkor feltételezhető, hogy amennyiben kevesebb hibás lefutás lenne, az elemzéshez szükséges idő is megnőne.

A következőkben néhány érdekesebb találatot fogok megmutatni. Az első példa a log4j *org.apache.log4j.SimpleLayout.format()* metódus, amely az 4.2.4. ábrán szerepel.

```
public class SimpleLayout extends Layout {
    ...
    58. public String format(LoggingEvent event) {
    59.
    60.     sbuf.setLength(0);
    61.     sbuf.append(event.getLevel().toString());
    62.     sbuf.append(" - ");
    63.     sbuf.append(event.getRenderedMessage());
    64.     sbuf.append(LINE_SEP);
    65.     return sbuf.toString();
    66. }
    ...
}

public class LoggingEvent implements java.io.Serializable {
    ...
    transient public Priority level;
```

```

    ...
255.   public Level getLevel() {
256.       return (Level) level;
257.   }
    }

    public class Level extends Priority implements Serializable{
        ...
    }

```

(a)

```

===== Found Exception Summary:
format(LoggingEvent)

```

```

Exception:
  java.lang.NullPointerException
Exception thrown from:
  null
Parametrization:
  format(null)
Stack trace:
  org.apache.log4j.SimpleLayout.format(SimpleLayout.java:61) format(null)

```

```

Exception:
  java.lang.NullPointerException
Exception thrown from:
  null
Parametrization:
  format(!null)
Stack trace:
  org.apache.log4j.SimpleLayout.format(SimpleLayout.java:61) format(!null)

```

```

Exception:
  java.lang.ClassCastException
Exception thrown from:
  null
Parametrization:
  format(!null)
Stack trace:
  org.apache.log4j.SimpleLayout.format(SimpleLayout.java:61) format(!null)
  org.apache.log4j.spi.LoggingEvent.getLevel(LoggingEvent.java:256) getLevel()

```

(b)

**4.2.4. ábra. (a)** Az *org.apache.log4j.SimpleLayout.format()* metódus és környezte. **(b)** A metódus elemzésének eredménye.

A metódusban három lehetséges futásidejű kivételt is talált a Jpf Checker. Az első kettő egy-egy *NullPointerException*, mindkettőt a 61. kódsor dobja, azonban míg az első akkor dobódik, ha az egyetlen paraméter *null*, a második akkor, mikor ez a paraméter különbözik a null értéktől. Ha a paraméter értéke *null* az *event.getLevel()* kifejezés okozza a kivételt, hiszen egy *null* referencia metódusát nem lehet meghívni. Ha az *event* nem *null*, ez minden további nélkül megtehető. Ekkor elkérjük az *event* objektum level adattagját, melynek a *toString()* függvénye hívódik meg. A második *NullPointerException* az okozza, hogy az elkért adattag

is lehet *null*, így a *.* operátor használata illegális lesz. A harmadik kivétel egy *ClassCastException*. Látható, hogy a *LoggingEvent* osztály 256. sorában egy típuskényszerítés szerepel, a *Priority* típusú *level* tagváltozó konverziója történik egy *Level* típusú objektummá. Az (a) ábra alján szereplő kódrészlet tanulsága szerint azonban a *Level* a *Priority* osztály leszármazottja, tehát a 256. sor egy lefelé irányú konverziót ír elő, amely hibás abban az esetben, ha a *level* adattag nem *Level* típusú. A jelentésben az Exception thrown from szekcióban a null szöveg szerepel, mert az elemzés futtatásakor nem adtam meg a log4j forrásfájljait tartalmazó könyvtárat a konfigurációban, a stack trace azonban tartalmazza a visszakereséshez szükséges információt.

A következő példában az ArgoUML *org.argouml.ui.PredicateMType.create()* metódusát vesszük szemügyre.

```

    public class FindDialog extends ArgoDialog ... {
        ...
    }

    class PredicateMType extends PredicateType {
        ...
727.    public static PredicateType create(Object c0, Object c1, Object c2) {
728.        Class[] classes = new Class[3];
729.        classes[0] = (Class) c0;
730.        classes[1] = (Class) c1;
731.        classes[2] = (Class) c2;
732.        return new PredicateMType(classes);
733.    }
        ...
    }

```

(a)

```

===== Found Exception Summary:
                                create(Object,Object,Object)

```

Exception:

java.lang.ClassCastException

Exception thrown from:

null

Parametrization:

create(null, null, !null)

Stack trace:

org.argouml.ui.PredicateMType.create(FindDialog.java:731)

create(null, null, !null)

Exception:

java.lang.ClassCastException

Exception thrown from:

null

Parametrization:

create(null, !null, don't care)

Stack trace:

org.argouml.ui.PredicateMType.create(FindDialog.java:730)

create(null, !null, don't care)

```

Exception:
  java.lang.ClassCastException
Exception thrown from:
  null
Parametrization:
  create(!null, don't care, don't care)
Stack trace:
  org.argouml.ui.PredicateMType.create(FindDialog.java:729)
                                     create(!null, don't care, don't care)
                                     (b)

```

**4.2.5. ábra (a)** Az *org.argouml.ui.PredicateMType.create()* metódus. **(b)** A metódus elemzésének eredménye.

Az 4.2.5. ábra (b) részén látható jelentés három felderített lehetséges *ClassCastException*ról számol be, melyek a 729., 730. és 731. sorról érkeznek. A forráskód ezen sorai típusellenőrzés nélkül konvertálják az *Object* osztályú paramétereket *Class* típusúra, amely semmiképpen sem mondható elegánsnak. A jelentés első bejegyzése szerint a *create(null, null, !null)* paraméterezés vezet a 731. sor által dobott hibához. Ha a *c0* és a *c1* paraméter is *null* a 729. és a 730. sor gond nélkül végrehajtódik, mert a *null* referencia castolása bármilyen osztályra megengedett. Fontos, hogy nem arról van szó, hogy a *c0*-nak és *c1*-nek is feltétlenül *null*-nak kell lennie, a jelentés egy olyan lehetséges paraméterezést ad meg melynek hatására kivétel keletkezik. Ha a harmadik, *Object* osztályú paraméter nem a *null* referencia *ClassCastException* dobódhat a 731. sorról. Ehhez nyilvánvalóan az is szükséges, hogy a paraméter típusa ne *Class* legyen. Ez az információ a jelentésben nem szerepel. A 730. sor lehetséges hibájára a *create(null, !null, „don't care”)* paraméterezés visz rá. A gondolatmenet az előzőhöz hasonló, amennyiben *c0* *null*, *c1* pedig egy nem *null* (és persze nem a *Class* osztály példánya) referencia, a kivétel létrejön. A harmadik paraméter értéke teljesen lényegtelen. Az utolsó esetben *c1* és *c2* értéke sem számít.

Az utolsó példa egy apró metódus, a 4.2.6. ábrán látható *org.apache.log4j.lf5.viewer.configure.MRUFileManager.getFile()*.

```

public class MRUFileManager {
    ...
    private LinkedList _mruFileList;
    ...
    public int size() {
        return _mruFileList.size();
    }
97.    public Object getFile(int index) {
98.        if (index < size()) {
99.            return _mruFileList.get(index);
100.        }
101.    }

```

```

102.    return null;
103.    }
    ...
    }

```

(a)

```

===== Found Exception Summary:
                                           getFile(int)

Exception:
    java.lang.IndexOutOfBoundsException
Exception thrown from:
    null
Parametrization:
    getFile(-999)
Stack trace:
    org.apache.log4j.lf5.viewer.configure.MRUFileManager
        .getFile(MRUFileManager.java:99)    getFile(-999)
    java.util.LinkedList.get(LinkedList.java:315)    get(-999)
    java.util.LinkedList.entry(LinkedList.java:365)    entry(-999)

```

(b)

**4.2.6. ábra (a)** Az *org.apache.log4j.lf5.viewer.configure.MRUFileManager.getFile()* metódus. **(b)** A metódus elemzésének eredménye.

A *getFile()* metódus a 98. sorban azt ellenőrzi, hogy a paraméterül kapott index változó kisebb-e, mint a *\_mruFileList* láncolt lista mérete. Ha igen, a visszatérési érték a láncolt lista megfelelő indexű eleme, különben *null*. Az 4.8. ábra (b) részén lévő jelentés arra világít rá, hogy a paraméter akár negatív is lehet. Ez az eset nincs levédve, a *LinkedList* osztály *get()* metódusa *java.lang.IndexOutOfBoundsException* típusú kivételt dob, ha például -999-et adtunk át híváskor. A metódus meghívása egy negatív számmal ésszerűtlennek tűnhet, és nyilvánvalóan az is, de megtehető.

### 4.3. Valódi találatok

Ebben az alfejezetben a programok hibakövető rendszereiben talált olyan hibákat írok le, amelyeket futásidejű kivételek okoznak, és amelyeket a kifejlesztett detektálóeszköz is megtalál.

Az egyik ilyen hibajelentés egy a log4j 1.2.14-es verziójában fellépő *NullPointerException*-ről számol be, ami a naplózórendszert használó program leállításához vezetett. A bejelentő az *ThrowableInformation.java* fájl 59. soráról, a *org.apache.log4j.spi.getThrowableStrRep()* metódusból kapta a kivételt, a leírt stack trace tanulsága szerint. A problémás kódrészlet és az elemzés futtatása után kapott jelentés a 4.3.1. ábrán látható.

```

...
public class ThrowableInformation implements java.io.Serializable {
    private transient Throwable throwable;

```

```

...
54. public String[] getThrowableStrRep() {
55.     if(rep != null) {
56.         return (String[]) rep.clone();
57.     } else {
58.         VectorWriter vw = new VectorWriter();
59.         throwable.printStackTrace(vw);
60.         rep = vw.toStringArray();
61.         return rep;
62.     }
63. }
...
}

```

(a)

```

===== Found Exception Summary:
getThrowableStrRep()

Exception:
    java.lang.NullPointerException
Exception thrown from:
    null
Parametrization:
    getThrowableStrRep()
Stack trace:
    org.apache.log4j.spi.ThrowableInformation
    .getThrowableStrRep(ThrowableInformation.java:59)    getThrowableStrRep()

```

(b)

**4.3.1. ábra. (a)** A hibajelentésben szereplő *org.apache.log4j.spi.ThrowableInformation.getThrowableStrRep()* metódus forráskódja. **(b)** Az elemzés eredménye.

Itt a probléma abban áll, hogy a *ThrowableInformation* osztály *throwable* adattagjának inicializálása elmarad, értéke *null*, ami egy *NullPointerException* futásidejű kivétel keletkezését eredményezi az 59. sorban. A hibajelentés az alábbi webcímen található meg:

[https://issues.apache.org/bugzilla/show\\_bug.cgi?id=44038](https://issues.apache.org/bugzilla/show_bug.cgi?id=44038).

A következő kivétel szintén egy *NullPointerException*, ami az 1.2.15-ös *log4j*-ben jelentkezett. A hibabejelentés ebben az esetben arról számol be, hogy az *org.apache.log4j.NDC.remove()* metódusból, pontosan a tartalmazó forrásfájl 377. soráról érkezik a program leállítását okozó futásidejű kivétel. A forráskódrészlet és a metódus elemzésének eredménye a 4.3.2. ábrán látható.

```

...
public class NDC {
    ...
    static Hashtable ht = new Hashtable();
    ...
374. static
375. public
376. void remove() {
377.     ht.remove(Thread.currentThread());
378. }

```

```

379.    // Lazily remove dead-thread references in ht.
380.    lazyRemove();
381.    }
    ...
}

```

(a)

```

===== Found Exception Summary:
remove()

Exception:
    java.lang.NullPointerException
Exception thrown from:
    null
Parametrization:
    remove()
Stack trace:
    org.apache.log4j.NDC.remove(NDC.java:377)    remove()

```

(b)

**4.3.2. ábra. (a)** A hibajelentésben szereplő *org.apache.log4j.NDC.remove()* metódus forráskódja. **(b)** Az elemzés eredménye.

A hiba oka itt az, hogy a *ht* statikus adattag *null*. Az (a) ábrán látható, hogy ugyan az inicializálása megtörténik, de előfordulhat, hogy a végrehajtás közben az értéke visszaáll *null*-ra. A hibajelentés tanulsága szerint ez meg is történik. A bejelentő a bejegyzésben azt is megemlíti, hogy észrevételei szerint, az *NDC* osztály több metódusa is, amely hivatkozik a *ht* változóra, először leellenőrzi azt, hogy *null*-e vagy sem, a *remove()* metódusban azonban ez nem történik meg. A bejelentés a [https://issues.apache.org/bugzilla/show\\_bug.cgi?id=45335](https://issues.apache.org/bugzilla/show_bug.cgi?id=45335) webcímen tekinthető meg.

Utolsóként szintén a log4j 1.2.15-ös verziójában megtalált valós hiba leírása következik. A hiba az *org.apache.log4j.net.SyslogAppender* osztályát definiáló forrásfájl 312. sorában van, amely az *append()* metódushoz tartozik. Itt ismét egy *NullPointerException*-ről van szó. A forráskódrészlet és a jelentés a 4.3.3. ábrán szerepel.

```

public abstract class AppenderSkeleton {
    protected Layout layout;
    ...
}
public class SyslogAppender extends AppenderSkeleton {
    SyslogQuietWriter sqw;
    private boolean layoutHeaderChecked = false;
    ...
291.    public
292.    void append(LoggingEvent event) {
293.
294.        if(!isAsSevereAsThreshold(event.getLevel()))
295.            return;
296.
297.        // We must not attempt to append if sqw is null.

```

```

298.     if(sqw == null) {
299.         errorHandler.error("No syslog host is set for SyslogAppender"
300.                             + named " + this.name + ".");
301.         return;
302.     }
303.
304.     if (!layoutHeaderChecked) {
305.         if (layout != null && layout.getHeader() != null) {
306.             sendLayoutMessage(layout.getHeader());
307.         }
308.         layoutHeaderChecked = true;
309.     }
310.
311.
312.     String packet = layout.format(event);
313.     String hdr = getPacketHeader(event.timeStamp);
314.
315.     if(facilityPrinting || hdr.length() > 0) {
316.         StringBuffer buf = new StringBuffer(hdr);
317.         if(facilityPrinting) {
318.             buf.append(facilityStr);
319.         }
320.         buf.append(packet);
321.         packet = buf.toString();
322.     }
    ...
    }}
}

```

(a)

```

===== Found Exception Summary:
= append(LoggingEvent)
Exception:
    java.lang.NullPointerException
    org.apache.log4j.net.SyslogAppender.append(SyslogAppender.java:294)
                                     append(null)
Exception:
    java.lang.ClassCastException
    org.apache.log4j.net.SyslogAppender.append(SyslogAppender.java:294)
                                     append(!null)
...
Exception:
    java.lang.NullPointerException
Exception thrown from:
    null
Parametrization:
    append(!null)
Stack trace:
    org.apache.log4j.net.SyslogAppender.append(SyslogAppender.java:312)
                                     append(!null)

```

(b)

**4.3.3. ábra. (a)** A hibajelentésben szereplő *org.apache.log4j.net.SyslogAppender.append()* metódus forráskódja. **(b)** Az elemzés eredménye.

A Jpf Checker az *append()* metódusban három lehetséges kivételt derít fel. Az első kettő

rövidítve szerepel az ábra (b) részén lévő jelentésben. Az első *NullPointerException* akkor keletkezhet, ha a metódus *event* paramétere *null*, a *ClassCastException* pedig a *LoggingEvent* osztály *getLevel()* metódus dobhatja az előző alfejezet 4.2.2. ábrájánál már ismertett módon. A jelnetés harmadik bejegyzése tartozik a hibakövető rendszerbe bejelentett kivételhez. A kivétel a 312. sorról dobódik, melynek oka az inicializálatlan *layout* adattag, amelyet a *SyslogAppender* osztály *AppenderSkeleton* őstől örököl.

A hibabejelentés a [https://issues.apache.org/bugzilla/show\\_bug.cgi?id=46271](https://issues.apache.org/bugzilla/show_bug.cgi?id=46271) weboldalon szerepel.

## 5. Kapcsolódó munkák

Ebben a fejezetben olyan munkák és publikációk kerülnek bemutatásra, amelyek valamilyen módon kapcsolódnak a dolgozat témájához. Először különféle szimbolikus végrehajtó rendszereket mutatok be röviden, majd kitérek arra, hogy milyen problémák megoldására alkalmasnak szimbolikus végrehajtást. Rámutatok, hogy az általam is használt Symbolic PathFinder milyen feladatok megoldására használták már fel, végül kitérek arra, hogy milyen módszerek és technikák léteznek futásidejű kivételek megtalálására.

A szimbolikus végrehajtás alapötlete nem új keletű, az első publikációk és végrehajtó eszközök az 1970-es években jelentek meg. Az első alpmű a témában [9], amely lefekteti a szimbolikus végrehajtás elméleti alapjait, valamint a PL/I nyelv szimbolikus végrehajtására képes rendszert, az EFFIGY-t mutatja be. Az EFFIGY bár csak egész típusú változókat tud szimbolikusán kezelni, egy olyan interaktív rendszer, amelyben a felhasználó töréspontok elhelyezésével, az állapotok elmentésével és visszatöltésével vizsgálhatja a szimbolikus végrehajtás menetét. Szintén az 1970-es években látott napvilágot egy hasonló rendszer, a SELECT [10], amely a LISP nyelvcsalád nyelvein megírt programok szimbolikus végrehajtására alkalmas. A felhasználó feltételeket definiálhat az egyes változókra és a visszatérési értékre, amelyek teljesülését, illetve nem teljesülését visszakapja outputként. A program teszt-input generálásra is használható, továbbá minden útvonalhoz a szimbolikus változók feletti path conditiont is vissza adja.

Az elmúlt évtizedben újra megnőtt az érdeklődés a technika iránt, több olyan program is született, amely elsősorban a dinamikus tesztinput generálást tűzte ki célul a szimbolikus végrehajtás technikájának felhasználásával. Az EXE (EXecution generated Executions) [11] egy hibakereső eszköz, olyan inputok generálására készítették a Stanford Egyetem kutatói, amelyre futtatva a programot az hibával azonnal terminál. Az inputgenerálás a beépített constraint solver, az STP végzi a hibára ráfutó végrehajtási útvonal path conditionjének megoldásával. Az EXE valós rendszereken futtatva ígéretesnek bizonyult. Hibákat talált a BSD és a Linux csomagszűrő implementációiban, az udhcpd DHCP szerverben és különféle Linux fájlrendszerekben. A dolgozatban bemutatott eszköz az EXE-hez hasonló elven működik, szintén a PC megoldásával generálunk tesztinputot. Az alapvető különbség az, hogy az EXE számára a forráskódban előre meg kell jelölni a szimbolikusán kezelendő változókat, míg a Jpf Checker esetén a forráskód előzetes szerkesztése nem szükséges.

A DART [12] vagyis a Directed Automata Random Testing a szimbolikus végrehajtás

azon hiányosságait próbálja kiküszöbölni, amikor az egy feltételt nem tud kezelni, például annak nem lineáris volta miatt. A DART a program végrehajtását random, vagy előre megadott konkrét inputokon végzi, és feltételes utasításokhoz érve feljegyzi a feltételek által az input változókra vonatkozó megszorításokat. A következő iterációkban ezeket a feltételeket figyelembe véve olyan konkrét adatokkal futtat, hogy a program végrehajtását egy másik lefutás felé terelje. A cél, hogy a program összes lehetséges és elérhető ágát végrehajtsa megadva az ághoz tartozó tesztinputot. A CUTE és jCUTE [13] rendszerek a DART továbbfejlesztései többszálúság és dinamikus adatszerkezetek kezelésével. Ezen rendszerek előnye, hogy a konkrét végrehajtás miatt a komplex matematikai feltételek kezelését megoldják. Ez azonban a Jpf Checkerben az SPF concolic végrehajtásának felhasználásával is megoldható, ugyanakkor a szimbolikus végrehajtás a kód alaposabb átvizsgálását teszi lehetővé.

A fent említett és további rendszerek összefoglaló leírását és összehasonlítását [14] és [15] tartalmazza.

Hálózati protokoll implementációk verifikációjára is alkalmazták a szimbolikus végrehajtást [16]. A SymNV névre hallgató rendszer hálózati csomagsorozatok előállítását végzi, amelyekkel magas lefedettség érhető el a vizsgálandó hálózati démon forráskódjában, ezáltal a protokoll specifikációjából származó potenciális szabálysértések kerülhetnek napvilágra.

A SAFELI [17] egy SQL injection (beszúrásos támadás) detektáló rendszer. Webes Java alkalmazások vizsgálatára készítették. Elsőként elvégzi a bytecode instrumentálását, majd az instrumentált kódot szimbolikusan futtatja. Amint egy SQL lekérdezéshez ér a végrehajtás, egy string egyenletet készít a webes beviteli komponensek iniciális tartalma és beépített SQL injection támadási minták alapján. Ha az egyenlet megoldása sikeres, a megoldások tesztinputként szolgálnak, amiket a rendszer az automatikus GUI tesztelő modulja segítségével leellenőriz. A tesztinputokat HTML fájlba írja és elküldi az oldalt a szervernek. A visszakapott válasz alapján dönti el, hogy a megtalált input valódi támadás lehet-e.

A Java PathFinder, illetve a szimbolikus végrehajtó kiterjesztésének fő alkalmazási területe a NASA belső projektjeinek verifikációja. Például a Symbolic PathFinder használatát írja le [18] a TSAFE (Tactical Separation Assisted Flight Environment) légi forgalom irányító és ütközés előrejelző szoftverkomponens ellenőrzésére. Az elsődleges cél hasznos tesztesetek automatikus generálása TSAFE-hez, ami többek között a szél, a különböző radarkép és repülőgép-menetrend adatokat magában foglaló matematikai formulákkal teletűzdelt forráskóddal rendelkezik.

A reverse engineeringben tervezési minták felismerése a statikus programvizsgálat mel-

let dinamikus módszerekkel is történhet. A program manuális futtatása közben valamilyen monitorozó szoftverrel figyelhetik a végrehajtást, a lefutási ágak alapján következtetnek tervezési minták meglétére. Markus von Detten [20] a manuális végrehajtás mellett szimbolikus végrehajtást is eszközöl a Symbolic PathFinderrel. Így sokkal több lehetséges lefutás vizsgálható, és a hagyományos módszerekkel megtalált viselkedési minták pontosíthatók, a hibás találatok kiszűrhetők.

A JPF egy érdekes felhasználását mutatja be Petri Ihantola munkája [19]. Az oktatásban a diákok programjainak ellenőrzésére automatikus tesztinput generálást hajtanak végre. A leírt megközelítés az, hogy azon tesztesetek, amelyeket a feladat specifikációja alapján készítenek, és azok eredménye (sikeres vagy sikertelen) nem biztos, hogy elegendő és elég tanulságos példa a tanulók számára. A specifikáció mellett a beadott program alapján szimbolikus végrehajtással kinyert tesztpéldák is szükségesek. Így a diákok olyan automatikus visszajelzést is kaphatnak, hogy például „a program rosszul működik, ha az  $a$  változó nagyobb, mint a  $b$  változó plusz  $10$ ”.

Java futásidejű hibák lokalizációjával foglalkozik Saurabh Sinha és mások [21]. A bemutatott módszer a meglévő hiba kijavításnak megkönnyítését célozza meg. A dobódott kivétel stack trace-éből kinyeri azt az utasítást ahonnan a kivétel jött, majd ettől a ponttól visszafelé irányuló adatfolyam elemzéssel megkeresi azokat az utasításokat, amelyek ténylegesen hibásak lehetnek és felelősek a kivétel keletkezéséért.

Biztonság-kritikus rendszerek biztonságos kivételkezelésének a bizonyításával foglalkozik Westley Weimer és George C. Necula [22]. Olyan teszteseket generálnak, amely a nyelv valamilyen szabályának megsértésével kivételhez vezet. A Jpf Checkerrel ellentétben a tesztinputok generálása itt nem a szimbolikus felhasználásával, hanem a program vezérlési folyamat gráfja (CFG) feletti globális optimalizálási probléma megoldásával történik az alapján, hogy egy adott tesztinput mennyire áll közel ahhoz, hogy a végrehajtás elérje valamilyen kivétel keletkezésének helyét.

A JCrasher [23] Java osztályok halmazát várja inputként. Az osztályok típusvizsgálata után egy Java programot hoz létre, amely példányosítja a megadott osztályokat és az osztályok publikus metódusait random paraméterezéssel futtatja. Ezzel olyan hibák kerülhetnek napvilágra, amik a program azonnali leállásához vezetnek, így például kezeletlen futásidejű kivételek is. A rendszer alkalmas JUnit tesztesetek generálására, továbbá az Eclipse fejlesztőkörnyezetbe is integrálható. A Jpf Checkerhez hasonlóan tehát a JCrasher is egy speciális driverkörnyezetet hoz létre, azonban csak a publikus metódusok vizsgálatát tudja elvégezni, és

szimbolikus végrehajtás helyett véletlen tesztadatokat generál, amely nyilvánvalóan nem alkalmas minden lehetséges lefutás megvizsgálására.

## 6. Összefoglalás és további célok

A kidolgozott módszer és eszköz mellett, hogy kézzel összeállított, kisebb példákon jól működik, valódi rendszerekben is számos kivételt képes felderíteni, amelyek között igazolhatóan szereplenek a használat során ténylegesen fellépő, valós problémát okozók is. A detektált futásidejű kivételek többsége azonban nem feltétlenül jelent a valós futás során előálló hibát. Sok olyan találat van, amely minden bizonnyal nem fordul elő a program valós környezetben történő futtatása során. Ezen jelzések jelentőségét sem szabad lebecsülni, minden találat olyan potenciális hibaforrásra mutat rá, amely a forráskód módosítása esetén bármikor igazi problémává válhat. Figyelembe vételük a fejlesztési és karbantartási munkát eredményesebbé teheti, ezáltal egy magasabb minőségű szoftver adható át tesztelésre, majd pedig a megrendelőnek is. Mint láttuk, nagyobb rendszerek vizsgálata is elfogadható idő alatt véget ér, tehát gyakorlatban is használható a megoldás.

A felhasznált eszköz, a Java PathFinder és bővítménye, a Symbolic PathFinder sajnos sok szoftverhibát tartalmaz. Ez a fejlesztést jelentősen megnehezítette, azonban a NASA fejlesztői rendkívül segítőkészek. Nagyjából 30 levélváltás történt köztünk, a kérdésekre gyorsan válaszolnak, és számos hibát kifejezetten a kérésemre javítottak ki.

Az eredmények mindenképpen biztatóak, a fejlesztés folytatódni fog. A jövőbeli tervek között szerepel a csekély jelentőséggel bíró, alaptalan találatok kiszűrése mellett az is, hogy egy felderített kivétel kapcsán a metódus környezetéről teljesebb, átfogóbb információt adjunk. Jelen pillanatban a referencia típusú paraméterekről csak azt az alapvető információt közöli a Jpf Checker, hogy null-e vagy sem, illetve a metódus osztályának adattagjainak azon értékéről sem mondunk semmit, amely szerepet játszik a kivétel létrejöttében. A jövőben nem csak kivételek felderítését szeretnénk megvalósítani. A szimbolikus végrehajtás lehetőségének kiaknázásával más jellegű hibák és szabálysértések megtalálása is szerepel a célok között. Tervek vannak egy a végtelen ciklusok bizonyos típusainak detektálását megvalósító, és SQL injection támadások lehetőségeire rámutató eszköz kifejlesztésére is.

## 7. Irodalomjegyzék

- [1] Java PathFinder Tool-set. <http://babelfish.arc.nasa.gov/trac/jpf>.
- [2] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. Proceedings of the 15th IEEE international conference on Automated software engineering, pages 203-232, 2000.
- [3] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. Proceedings of the IEEE/ACM international conference on Automated software engineering, pages 179-180, 2010.
- [4] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 134-138, 2007.
- [5] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. CORAL: Solving Complex Constraints for Symbolic Pathfinder. Proceedings of the Third International Conference on NASA Formal Methods, pages 359-374, 2011.
- [6] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of systems, pages 553-568, 2003.
- [7] Corina S. Păsăreanu, Neha Rungta, and Willem Visser. Symbolic Execution with Mixed Concrete-Symbolic Solving. Proceedings of the 2011 International Symposium on Software Testing and Analysis, pages 34-44, 2011.
- [8] Corina S. Păsăreanu, Peter C. Mehltitz, 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. Proceedings of the 2008 international symposium on Software testing and analysis, pages 15-26, 2008.
- [9] James C. King. Symbolic Execution and Program Testing. Commun. ACM, pages 385-394, 1976.
- [10] Boyer, Robert S. and Elspas, Bernard and Levitt, Karl N. SELECT - a formal system for testing and debugging programs by symbolic execution. Proceedings of the International Conference on Reliable Software, pages 234-245, 1975.
- [11] Cadar, Cristian and Ganesh, Vijay and Pawlowski, Peter M. and Dill, David L. and Engler, Dawson R. EXE: Automatically Generating Inputs of Death. Proceedings of the 13th ACM Conference on Computer and Communications Security, pages 322-335, 2006.
- [12] Godefroid, Patrice and Klarlund, Nils and Sen, Koushik. DART: Directed Automated

- Random Testing. Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 213-223, 2005.
- [13] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. CAV, pages 419-423, 2006.
  - [14] Coward, P. David. Symbolic execution systems - a review. Softw. Eng. J., pages 229-239, 1988.
  - [15] Cadar, Cristian and Godefroid, Patrice and Khurshid, Sarfraz and Păsăreanu, Corina S. and Sen, Koushik and Tillmann, Nikolai and Visser, Willem. Symbolic Execution for Software Testing in Practice: Preliminary Sssessment. Proceedings of the 33rd International Conference on Software Engineering, pages 1066-1071, 2011.
  - [16] JaeSeung Song and Tiejun Ma and Cristian Cadar and Peter Pietzuch. Rule-based Verification of Network Protocol Implementations using Symbolic Execution. 20th IEEE International Conference on Computer Communications and Networks, 2011.
  - [17] Fu, Xiang and Qian, Kai. SAFELI: SQL Injection Scanner using Symbolic Execution. Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, pages 34-39, 2008.
  - [18] David Bushnell, Dimitra Giannakopoulou, Peter Mehlitz, Russ Paielli, Corina Păsăreanu. Verification and Validation of Air Traffic Systems: Tactical Separation Assurance. Aerospace conference, IEEE, pages 1-10, 2009.
  - [19] Petri Ihantola. Test Data Generation for Programming Exercises with Symbolic Execution in Java PathFinder. Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006, pages 87-94, 2006.
  - [20] Markus von Detten. Towards Systematic, Comprehensive Trace Generation for Behavioral Pattern Detection through Symbolic Execution. Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, pages 17-20, 2011.
  - [21] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault Localization and Repair for Java Runtime Exceptions. Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pages 153-164, 2009.
  - [22] Westley Weimer and George C. Necula. Finding and Preventing Run-time Error Handling Mistakes. Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pages 419-431, 2004.
  - [23] Christoph Csallner and Yannis Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. Softw. Pract. Exper., pages 1025-1050, 2004.

## **8. Köszönetnyilvánítás**

Ezúton szeretnék köszönetet mondani témavezetőimnek, Hegedűs Péternek és Dr. Ferenc Rudolfnak a program és a dolgozat elkészítésében nyújtott sok segítségért. Ezen túl szeretném megköszönni Bakota Tibor munkáját, kinek hasznos észrevételeit és tanácsait figyelembe vehettem a program készítése során. Végül, de nem utolsó sorban köszönetet mondok a Symbolic PathFinder egyik fejlesztőjének, Corina S. Păsăreanunak, aki a legtöbb SPF-fel kapcsolatos levelemre segítőkészen válaszolva hozzájárult a dolgozat létrejöttéhez.