

Szegedi Tudományegyetem
Informatikai Tanszékcsoport

DIPLOMAMUNKA

Kádár István

2014

**Szegedi Tudományegyetem
Informatikai Tanszékcsoporth**

**Szimbolikus végrehajtó keretrendszer kiegészítése
constraint építő mechanizmusokkal**

Diplomamunka

Készítette:

Kádár István

programtervező informatikus Msc.

Témavezető:

Dr. Ferenc Rudolf

egyetemi adjunktus

Szeged
2014

Feladatkiírás

A forráskód kézi ellenőrzése (code review, audit) és a szoftvertesztelés jelenleg a két elsődlegesen alkalmazott módszer a szoftver verifikációban. Ezekre a tevékenységekre rendkívül magas összegeket áldoznak a szoftverfejlesztő cégek (a teljes költség több mint felét), ugyanakkor e manuális tevékenységek részben kiválthatók automatikus eszközökkel, ami jelentős költségmegtakarítást hozhat. A Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén egy olyan eszköz fejlesztése indult be, amely a szimbolikus végrehajtás elveinek felhasználásával alkalmasak nagyméretű valós rendszerek forráskódjának auditálására és hibáinak felderítésére a rendszer valós környezetben történő futtatása nélkül. A modern szimbolikus végrehajtó rendszerek minden végrehajtási útvonalhoz karban tartanak egy feltételrendszert, aminek kiértékelésével az állapottér bejárása optimalizálható. A feladat olyan feltételrendszer (constraint) építési mechanizmusok kidolgozása és integrálása a Szoftverfejlesztés Tanszék szimbolikus végrehajtó motorjába, aminek köszönhetően az eddiginél pontosabb hibajelzések kaphatók.

Tartalmi összefoglaló

- ***téma megnevezése:***

Szimbolikus végrehajtó rendszer kiegészítése, hibatalálatainak pontosítása constraint (feltételrendszer) építő mechanizmusokkal.

- ***A megadott feladat megfogalmazása:***

A Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén fejlesztett szimbolikus végrehajtó motorjába olyan feltételrendszer építő mechanizmus integrálása, amely növeli annak hatékonyságát, azáltal, hogy kiküszöböli a hagyományos szimbolikus végrehajtás egy gyenge pontját, aminek köszönhetően olyan futásidejű hibák is detektálhatók, amelyeket egy hagyományos szimbolikus végrehajtó rendszer nem képes, emellett false positive hibatalálatok kiszűrését is lehetővé teszi.

- ***A megoldási mód:***

Minden végrehajtási ághoz tartozó feltételrendszer a program elágazásainál szereplő logikai kifejezésekből, illetve a program értékadásai által definiált egyenlőséget kifejező feltételekből épül fel. Az így kapott feltételrendszer partícionálásával, részleges megoldásával és a megoldás visszahelyettesítésével valósul meg a szimbolikus végrehajtó rendszer hibatalálatainak pontosítása.

- ***Alkalmazott eszközök, módszerek:***

Szimbolikus végrehajtás, Symbolic Checker, a Szoftverfejlesztés Tanszék szimbolikus végrehajtó motorja, Gecode, Google Test.

- ***Elért eredmények:***

A kidolgozott constraint építő mechanizmusokkal futtatott szimbolikus végrehajtó rendszer saját, kézzel összerakott példaprogramokban olyan hibatalálatokat képes felfedezni, amelyeket constraint építés nélkül nem tudott, és egy hagyományos constraint építési mechanizmust használó rendszer (például a Symbolic PathFinder) sem képes felfedezni, valamint hamis (false positive) hibatalálatok is kiszűrhetők a módszerrel. A nagy méretű valós rendszeren kapott eddigi eredmények igazolják, hogy a kidolgozott eljárás valóban alkalmas false positive hibatalálatok kiszűrésére.

- ***Kulcsszavak:***

Szoftverfejlesztés, szimbolikus végrehajtás, constraint építés, feltételrendszer építés.

Tartalomjegyzék

Feladatkiírás.....	3
Tartalmi összefoglaló.....	4
1. BEVEZETÉS.....	6
2. HÁTTÉR.....	7
2.1. Szimbolikus végrehajtás.....	7
2.2. A JPF Checker.....	11
3. SYMBOLIC CHECKER, A SZOFTVERFEJLESZTÉS TANSZÉK SZIMBOLIKUS VÉGREHAJTÓ MOTORJA.....	13
4. FELTÉTELRENDSZER ÉPÍTÉS.....	16
4.1. Alapelvek.....	16
4.2. Megvalósítás.....	19
4.3. A feltételrendszer összefüggő komponensekre bontása.....	26
4.4. Tesztelés.....	28
5. KIÉRTÉKELÉS.....	31
6. KAPCSOLÓDÓ MUNKÁK.....	38
7. ÖSSZEFOGLALÁS ÉS TOVÁBBI CÉLOK.....	42
Irodalomjegyzék.....	43
Nyilatkozat.....	45
Köszönetnyilvánítás.....	46

1. BEVEZETÉS

Nagyméretű, megbízható és robusztus szoftverrendszerek előállítása a szoftverfejlesztés egy nagy kihívást jelentő feladata manapság. A fejlesztési költségek nagyjából 40%-a tesztelési költség [1], továbbá az üzemeltetés során fellépő hibák megtalálása és kijavítása is jelentős erőforrásokat emészt fel és nagy költségekkel jár [2]. A Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén fejlesztett szimbolikus végrehajtó motor, a Symbolic Checker a szoftverfejlesztés ezen fázisát támogatja azáltal, hogy Java programokban képes futásidejű hibákat (például null pointer dereferencia, tömb túlindexelés, vagy nullával való osztás) felderíteni anélkül, hogy a programot valós környezetben futtatni kellene.

A szimbolikus végrehajtás elmélete szerint [4] a 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. Valamennyi elágazási pontnál a szereplő logikai kifejezést, illetve annak negáltját továbbvisszük az igaz, illetve hamis ágakon, így minden lefutási útvonalhoz egy egyedi feltételrendszer (constraint-rendszer) rendelhető, amelyeket kielégítve tesztinputok generálhatók az adott útvonalhoz.

A dolgozatban egy olyan feltételrendszer építési mechanizmust mutatok be, amely a szimbolikus végrehajtás alapelveitől eltérő módon alkalmazott feltételépítést használ úgy, hogy a programban szereplő értékadásokat is feltételként kezeli, így a szimbolikus változók függőségeit is nyomon követi, és adott szimbólumhoz egyértelmű értéket rendelő feltételek megléte esetén azt konkrét értékkel rendelkező változóvá alakítja. Ezáltal a szimbolikus végrehajtó motor futásidejű hibajelzései pontosíthatók, a false positive találatok kiszűrhetők. A feltétel rendszer felépítéséhez és annak kielégítéséhez az ingyenes Gecode korlátozás-kielégítő (constraint satisfaction) eszközkészletet [5] használtam fel.

A dolgozatban részletesen kifejtem, hogy hogyan valósítottam meg a fenti feltételrendszert felépítő algoritmust, hogyan integráltam a Szoftverfejlesztés Tanszék szimbolikus végrehajtó rendszerébe, és hogy annak hatékonyságát miként növeltem a kifejlesztett mechanizmusokkal. Az így kibővített szimbolikus végrehajtót összevetettem a NASA Ames Research Center által fejlesztett Symbolic PathFinder [7] alapú megoldással [3], valamint valós, nagyméretű rendszereken is futtattam.

2. HÁTTÉR

2.1. Szimbolikus végrehajtás

Minden program, végrehajtása során, adatokon végez műveleteket egy meghatározott sorrendben. A szimbolikus végrehajtás [4] 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 ágot, majd ennek negáltját szintén hozzákapszoljuk a már meglévő PC-hez, jelezve a hamis ágot. 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 ágot fogja követni, amelyhez tartozó PC-t megoldottuk.

A Szoftverfejlesztés Tanszék szimbolikus végrehajtó motorjának, a Symbolic Checkernek célja azonban nem tesztinputok generálása, hanem a programban lévő lehető legtöbb valószínű futásidejű hiba megtalálása. Ezen céllal összhangban a fentiekben leírt path condition építést változtatom meg, bővítem ki amit a 4. fejezetben fejtek ki részletesen. A feltételrendszer

kielégítésre természetesen ebben a megoldásban is szükség lesz, erre a Gecode constraint solver rendszert [5] használtam fel.

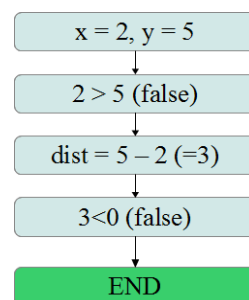
A lehetséges végrehajtási útvonalak egy összefüggő és körmentes irányított gráfot definiálnak, amelyet szimbolikus végrehajtási fának nevezünk. A fa egyes pontjai a program szimbolikus állapotainak felelnek meg. Egy szimbolikus állapot a szimbolikus változókat és az egyéb konkrétan kezelt változók értékeit, 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.

```

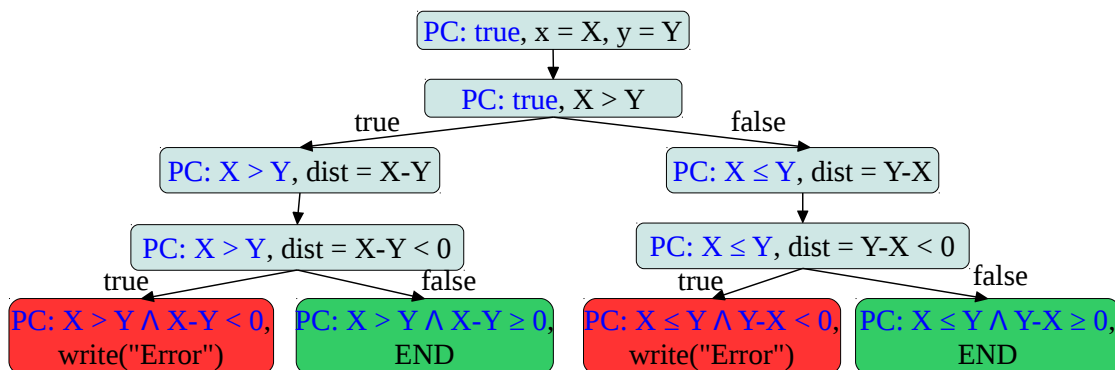
1. int x, y, dist;
2. ...
3. if (x > y) {
4.   dist = x - y;
5. } else {
6.   dist = y - x;
7. }
8. if (dist < 0)
9.   write("Error");

```

(a)



(b)



(c)

2.1.1. 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 2.1.1-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 2.1.1. (b) ábrán látható. A szimbolikus végrehajtást a 2.1.1. (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 felté-

tel 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 condition-ök bővítésével. Itt a *true* ágakon az alábbi PC-ket kapjuk a *dist* változó helyére a megfelelő szimbolikus kifejezést helyettesítve.

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

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 (az esetleges túl-, vagy alulcsordulási hibáktól tekintsünk el). Amennyiben egy állapothoz tartozó PC kielégíthetetlen, az innen induló részfa levágható, nincs értelme folytatni az ellentmondásos 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.

A 2.1.2. (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 2.1.2. (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 megkezdjük a ciklusmag végrehajtását, *false* esetén az U_2 -vel folytatjuk. Ha beléptünk a ciklusmagba, akkor annak sikeres végrehajtása után i értéke eggyel megnő, majd az $i < n$ feltétel újbóli kiértékelése következik, ami újabb elágazáshoz vezet. *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.

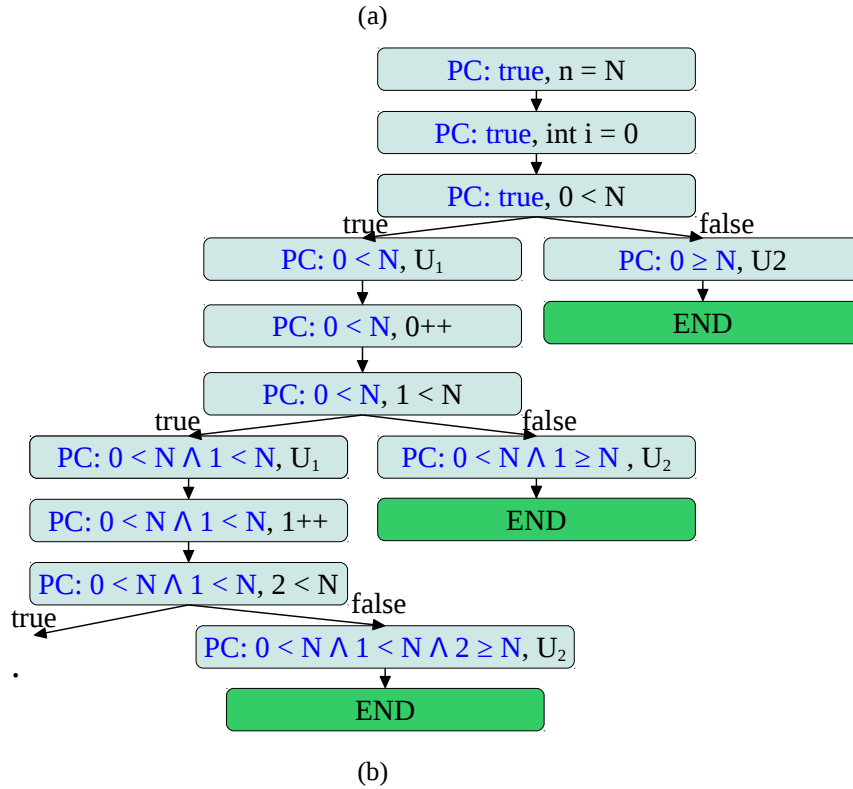
Egy program összes lehetséges állapotának felderítése, az összes lehetséges lefutás végrehajtása elfogadható időn belül lehetetlen. Egyfajta megoldás lehet a fa mélységének, a szim-

bolikus állapotok számának, vagy a path condition hosszának korlátozása, ami természetesen meggátolja az összes lehetséges állapot elérést.

```

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

```



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

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, ezek részleteire a dolgozat nem tér ki.

2.2. A JPF Checker

A 2012-ben beadott TDK dolgozatom témája egy olyan eszköz kifejlesztése volt, ami futás-idejű kivételek detektálását valósította meg Java programokban szimbolikus végrehajtással, és a felfedezett hibához vezető tesztinputot is előállított. A dolgozatban bemutatott munka alapján publikáció is készült 2013-ban [3]. Ebben az alfejezetben felvázolom ennek az eszköznek, a JPF Checkernek a működési alapelveit, hiszen a jelen dolgozatban bemutatott feltételrendszer építő mechanizmussal kiegészített elemzőt ezzel vetem össze az 5. fejezetben.

A JPF Checker a NASA Ames Research Center által kifejlesztett modellellenőrző rendszert a Java PathFindert (JPF) [6], illetve annak szimbolikus végrehajtást biztosító kiterjesztését a Symbolic PathFindert (SPF) [7], mint szimbolikus végrehajtó motort használta fel. A JPF egy olyan Java virtuális gép, ami Java bytecode-ot speciálisan interpretálja. Kiterjesztése az SPF pedig egy olyan Java bytecode utasításkészletet valósít meg, ami lehetővé teszi, hogy tetszőleges Java programot a szimbolikus végrehajtás elméleti alapelveinek megfelelő módon hajthassunk végre a JPF-fel.

A JPF Checker az adott Java program összes metódusát, egyesével külön-külön hajtatja végre szimbolikusán, amihez egy speciális futtatókörnyezet kialakítására volt szükség, hiszen a JPF/SPF egy Java virtuális gép, az a program belépési pontját jelentő *main()* metódustól kezdi el végrehajtást. Az SPF fejlesztőinek eredeti elképzelése az volt, hogy a program futása valós környezetben, konkrétan kezdődjön meg, majd bizonyos előre definiált pontokon (például a program komplexebb, nagy valószínűséggel hibát tartalmazó pontjain) váltson át a végrehajtás szimbolikusra [12]. Mindkét megközelítésnek léteznek előnyei és hátrányai is [3].

A vizsgált metódus input adatainak a paraméterei és a hivatkozott külső változók számításának. A JPF Checker ezeket az adatokat veszi fel szimbólumként egy metódus elemzésének megkezdése pillanatában.

A futásidejű kivételdetektálást a JPF listener mechanizmusának segítségével valósítottam meg. A JPF működésének számos eseményéről képes értesítő üzenetet küldeni például egy bytecode utasítás végrehajtásának befejezéséről, vagy új állapot eléréséről. Ezt a mechanizmust felhasználva a kivételek detektálásának módszere dióhéjban a következő. A megadott metódust szimbolikusán végrehajtva, annak valamennyi lefutási ága végrehajtott, így azok is, amelyek kivételt dobnak. Az *ATHROW* kivételdobó Java bytecode utasítás végrehajtása esetén a virtuális gép memóriaterületéről kinyerjük a dobott kivételobjektumot, eldöntjük, hogy a *java.lang.RuntimeException* osztály példánya-e, és amennyiben igen, megoldatjuk az

útvonalhoz tartozó path condition-t, és a megoldásokat hozzárendeljük az input szimbólumokhoz.

Az így elkészített elemző eszközközt több módon is teszteltem. Elsőként kézzel összerakott kisebb kivételt dobó példákon futtattam, majd valós, nyílt forrású rendszereket is elemeztem. Ezen nagy rendszerekben is számos futásidejű kivételt fedezett fel a JPF Checker, amelyek között vannak igazolhatóan valós hibát okozó találatok is, de sajnos több false positive találat is született.

A false positive találatok megjelenésének egyik oka az, hogy az ellenőrzés egy mester-séges végrehajtási környezetben történik, hiszen amikor egy metódus szimbolikus végrehajtását elindítjuk annak a valós futás során bekövetkező hívásainak körülményeiről semmilyen ismeretünk nincs. Egy másik ok az SPF-ben alkalmazott *lazy initialization* algoritmus [8] működésében keresendő. Ez az algoritmus a Java referencia típusú változóinak szimbolikus kezelésére is lehetőséget nyújt a következő módon. A szimbolikusan kezelendő, inicializálatlan, *T* osztályú referencia típusú változót nondeterminisztikusan 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 folytatódik tovább a végrehajtás. Az algoritmussal ugyan az összes lehetséges esetet végigkövethetjük, azonban az (1)-es inicializálással null értékek kerülnek a végrehajtásba olyan pontokon is, ahol ennek egyáltalán nem lenne létjogosultsága a futásidejű hibadetektálás szempontjából, és emiatt olyan *NullPointerException* kivételek kerülnek riportálásra, amelyek szinte biztosan false positive találatként vehetőek számba.

Egy másik problémakör az, hogy a JPF/SPF meglehetősen instabil. Például az ArgoUML modellező szoftver elemzése során a metódusok több mint felét nem tudtuk megvizsgálni a használt végrehajtó motor váratlan leállása miatt [3].

A fentiekben vázolt problémák miatt úgy döntöttünk, hogy egy saját szimbolikus végrehajtó motor fejlesztésébe kezdünk, aminek kimondott célja valós Java futásidejű hibák keresése a false positive találatok eliminálásával. Valós futásidejű hiba alatt olyan hibákat értünk, amelyekhez létezik olyan végrehajtási útvonal a programban, ami előidézi őket.

3. SYMBOLIC CHECKER, A SZOFTVERFEJLESZTÉS TANSZÉK SZIMBOLIKUS VÉGREHAJTÓ MOTORJA

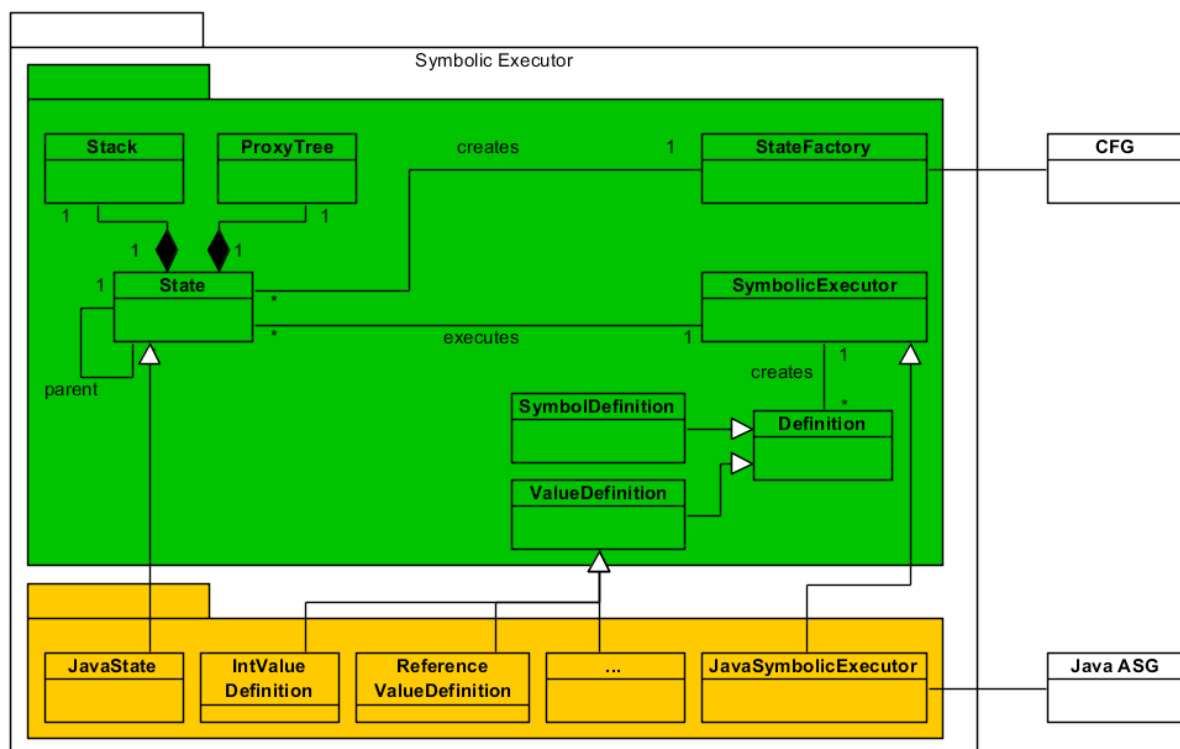
A JPF Checker elemzőeszköz kifejlesztése, majd az eredmények kiértékelése után a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén egy csapat, amelynek jómagam is része vagyok egy saját, a Symbolic Checker névvel felruházott szimbolikus végrehajtó motor fejlesztésébe kezdett, mivel a JPF/SPF alapú megvalósítás nem felelt meg maradéktalanul a céljainknak. A Symbolic Checker célja olyan valós futásidejű hibák detektálása, amelyeket a legtöbb auditáló eszköz nem képes megtalálni és amelyek felfedezése jobbára csak rengetek teszteléssel valósulhatna meg. Fontos számunkra, hogy a felfedett hibatalálatok a lehető legpontosabban legyen, vagyis az, hogy a false positive jelzéseket kivétel nélkül elimináljuk és minél több olyan true positive jelzést adjunk, amellyel nem csak a szoftverfejlesztők munkáját segítjük, hogy egy magasabb minőségű szoftvert adhassanak át tesztelésre, hanem a későbbi karbantartási munkákat is megkönnyítjük. A hibákra vezető tesztesetek generálása nem kitűzött cél, sokkal inkább azon végrehajtási útvonal szemléletes kijelölése, ami a hibához vezetett.

Jelen dolgozatnak nem témája a Symbolic Checker elemzőeszköz részletes bemutatása, azonban az általam kifejlesztett feltételrendszer építő mechanizmus megvalósításának megértéséhez elengedhetetlen az alapszintű megismerése. A rendszer C++-ban íródik, a fejlesztés jelenleg is folyamatban van. Jelenleg null pointer dereferencia, tömb túlindexelés, nullával való osztás és típuskényszerítési hibák (Java terminológiában *ClassCastException*) Java programokban történő detektálása implementált.

A JPF Checkerhez hasonlóan a Symbolic Checker is intraprocedurális elemzést hajt végre, vagyis a vizsgálandó rendszer valamennyi metódusának szimbolikus végrehajtásával végzi el az elemzést. A kezdeti szimbólumok a metódus paraméterei és a végrehajtás során hivatkozott, de nem inicializált változók lesznek. Fontos, hogy találatot csak akkor jelzünk, ha garantáltan a végrehajtás során konstanspropagálással megjelenik az az érték, ami problémát okoz. Például ha egy metódushívás garantáltan *null* értéket ad át, és a metódus azt az értéket garantáltan dereferálja, akkor jelzünk, de ha a dereferált változó szimbólum, akkor nem, mert annak értéke nem ismert, bizonytalan. A szimbolikus végrehajtási fa méretének korlátozására, annak maximális mélységének és az állapotok maximális számának megadása lehetséges. A tömbök, referencia típusú változók és a polimorfizmus kezelése megoldott, de a rendszer nem

támogatja még például a stringműveleteket.

A szimbolikus végrehajtás az elemzendő program nyelvfüggő absztrakt szemantikus gráján (ASG) [9] történik, a grájban lévő csúcspontok (node-ok) megfelelő sorrendben történő értelmezésével. Ezt a sorrendet a program nyelvfüggetlen vezérlési folyam grájja (CFG) [10] definiálja. Ez alapján a program ASG-je és CFG-je számítanak a Symbolic Checker inputjának, amelyek szintén a Szoftverfejlesztés Tanszéken készített elemzőeszközökkel állíthatók elő a program forráskódjából. A rendszer outputja egy olyan jelentés, amely a megtalált hibajelzéseket tartalmazza, feltüntetve azok típusát, a végrehajtási útvonalat, ami a szimbolikus végrehajtás belépési pontját jelentő metódustól a hiba helyéig vezet, valamint egy valószínűséget, ami azt becsüli, hogy a futásidejű hibára mekkora valószínűséggel fut rá a metódus.



3.1. ábra. A Symbolic Checker architektúrája a főbb komponensek feltüntetésével.

A Symbolic Checker architektúráját a főbb komponensek feltüntetésével a 3.1-es ábra szemlélteti. Inputja a vizsgálandó programot reprezentáló nyelvfüggő ASG és a program vezérlési folyam grájja. Maga a rendszer két jól elkülönülő részből tevődik össze: a nyelvfüggetlen részből, amely az általános szerkezeteket és működési elveket valósítja meg, illetve a nyelvfüggő, jelen esetben Java specifikus részből, ami többek közt a nyelv adattípusait, operátorainak viselkedését megvalósító részeket tartalmazza.

A CFG alapján a *StateFactory* hozza létre az állapotokat, amelyek a szimbolikus végre-

hajtási fát alkotják úgy, hogy a CFG-ben minden úgynevezett *basic block*hoz jön létre egy állapot. A *basic block* azoknak az elemi utasításoknak (valójában ASG node-oknak) a sorozata, amelyek biztosan egymás után következnek a végrehajtásban a CFG szerint. Minden állapothoz tartozik egy verem (*Stack*), amely az egyes műveletek operandusainak megfelelő sorrendben történő tárolásával biztosítja a művelet elvégzését, továbbá egy úgynevezett *ProxyTree*. A *ProxyTree* egy olyan speciális adatszerkezet, ami a változók hatékony tárolását teszi lehetővé a szimbolikus végrehajtási fában. Lényege, hogy csak a változásokat tárolja, a változatlan adatok a szülő állapot *ProxyTree*-jéből nyerhetők ki.

Az adott állapotba tartozó utasítások végrehajtása a *SymbolicExecutor* feladata. Végrehajtás közben *Definition* objektumokat hoz létre. A *Definition* egy átfogó elnevezése minden adatnak, ami a program szimbolikus végrehajtása során megjelenik. Ezek például a konkrét vagy szimbolikusan kezelt változók, konstansok, metódusok paraméterei, azok visszatérési értéke, és az egyes rész kifejezések eredményei is. Alapvetően kétféle típusú *Definition* létezik: *ValueDefinition* és *SymbolDefinition*. A *ValueDefinition* objektumok valamilyen konkrét értéket tárolnak, a *SymbolDefinition* példányok pedig az egyes szimbolikus változókat reprezentálják. Programozási nyelvtől függően a *ValueDefinition* osztály leszármaztatása szükséges a nyelvben megjelenő adattípusoknak megfelelően. Java esetén többek között a *ByteValueDefinition*, *ShortValueDefiniton*, *IntValueDefinition*, vagy akár a referenciák tárolására használt *ReferenceValueDefiniton* osztályokat hoztuk létre. A *Definition* osztályhierarchiában kapott helyet az egyes operátorok megvalósítása, vagyis például az, hogy egy *IntValueDefinition* hogyan kell összeadni egy *SymbolDefinition*nel. A *SymbolicExecutor* a létrehozott *Definition* objektumokat elhelyezheti a veremben, illetve kiveheti onnan, vagy éppen eltárolhatja a *ProxyTree*-ben azokat.

4. FELTÉTELRENDSZER ÉPÍTÉS

Munkám során egy olyan feltételrendszer építési mechanizmust dolgoztam ki, amely a Symbolic Checkerbe integrálódva lehetővé teszi, hogy az olyan futásidejű hibákat is megtaláljon, amelyre egy hagyományos szimbolikus végrehajtó rendszer nem képes. Ebben a fejezetben átfogóan, példákkal illusztrálva kifejtem, hogy ezt a feltételrendszer építési mechanizmust hogyan valósítottam meg, és felvázolom, hogy milyen előnyökkel jár.

4.1. Alapelvek

A 3. fejezetben leírtak szerint a Symbolic Checker csak akkor jelez hibát, ha garantáltan megjelenik olyan konkrét érték, amely azt előidézi. Szimbólumokra nem jelez, hiszen ha egy változó szimbólum az éppen azt jelenti, hogy annak értéke kétséges, nem ismert. Előfordulhat az, hogy szimbolikusan végrehajtva a programot a változók túlnyomó része szimbólummá válik, ami miatt a futásidejű hibák megtalálása ellehetetlenedik.

Az általam kifejlesztett feltételrendszer építési mechanizmus lényege, hogy amennyiben szimbolikus végrehajtás közben a program olyan feltételeket (constrainteket) ír elő, amelyek egy vagy több szimbólum értékét egyértelművé teszik, akkor ezeket a szimbólumokat értékke alakítva folytassuk tovább a végrehajtást az adott útvonalon. Mivel ezeket a változókat konkrét értékkel rendelkező adatként kezeljük tovább, olyan hibák megtalálása válik lehetővé, amelyeket egyébként nem lehetne detektálni. A szóban forgó feltételeket egyrészt a program, feltételtől függő elágazásai adják (*if*, *switch*, *while*, stb.), másrészt pedig a programban lévő értékadások fejezik ki, ide értve a Java nyelv inkrementáló és dekrementáló operátorainak (*++*, *--*) hatását is.

Összességben tehát a megvalósított feltételrendszer építés célja az, hogy minél több szimbólumot tudjunk konkretizálni, melyek révén több futásidejű hibát fedhetünk fel. Ehhez (1) olyan speciális *path conditiont* (PC-t) kell felépíteni, amely tartalmazza a szimbólumfüggőségeket is a program értékadásai által, valamint (2) amennyiben a feltételek bármely szimbólumra egyértelmű értéket írnak elő, akkor ezt az egyértelmű értéket felhasználva kell folytatni a végrehajtást. Mivel az ilyen módon kibővített *path condition* is tartalmazza valamilyen formában azokat a feltételeket, amelyek a hagyományos módon épített PC, ezért annak kielégíthetetlen volta miatt ugyanúgy kiszűrhetők azok a kódrészletek, amelyek nem kerülhetnek végrehajtásra. Ennek eredményeként false positive hibatalalatok szűrhetők ki.

A PC kibővítés alapötletének szemléltetéséhez tekintsük a szimbolikus végrehajtást bemutató fejezet 2.1.1-es ábráján lévő programrészletet. Ebben a példában azon végrehajtási útvonal hagyományos módon épített PC-je, amely a 3. sorban lévő *if*-nek az *igaz*, a 8. sorban lévőnél pedig a *hamis* ágán halad a következő volt:

$$X > Y \wedge \neg(X - Y < 0) \Rightarrow X > Y \wedge (X - Y \geq 0)$$

Ugyanezen útvonal kiterjesztett PC-je koncepcióm szerint pedig az alábbi:

$$X > Y \wedge \neg(dist < 0) \wedge dist = X - Y$$

Látható, hogy a *dist* változó, mint szimbólum is bekerült a feltételrendszerbe egyrészt úgy, hogy a 8. sorban lévő feltétel negáltját nem behelyettesítve, hanem közvetlen $\neg(dist < 0)$ formában szerepeltettük, másrészt pedig a 4. sor értékadását leíró $dist = X - Y$ feltételt is felvettük. Ennek köszönhetően a feltételrendszer a *dist* szimbólumról is hordoz információt, ami hasznosítható lehet a végrehajtás későbbi szakaszában. Az ilyen módon kapott új path conditon ebben a példában nem tartalmaz olyan feltételt, amely bármelyik változó értékét egyértelműsíteni tudná, így ennek hatása itt nem érvényesül, a 4.1.1. ábrán lévő kódrészlettel azonban egy ilyen esetet mutatok be.

```

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

4.1.1. ábra. Példakód, amely szimbólumok konkrét értékének konkretizálását írja elő, ami által olyan futásidejű hibát találunk meg, amelyet a hagyományos szimbolikus végrehajtás nem lenne képes.

A 4.1.1. ábrán látható példakódot szimbolikusan végrehajtva a *c* változó szimbolikus kezelésével a 7. sorban lévő ponton (szimbolikus állapotban) a következő feltételrendszert építjük fel:

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

Ez a feltételrendszer tartalmazza az *if* utasítások által előírt feltételeken túl az *a* szimbólum függőségeit is, vagyis azon értékadások által definiált feltételeket, amelyekkel az *a* változót definiálja a program. A feltételrendszert kielégítve azt kapjuk, hogy az *a* szimbólumra adott megoldás egyértelmű, kizárólag 9 lehet, ami magával vonja azt, hogy a *b* és *c* szimbólumok is egyértelműek: $b = 0.0$ és $c = -2$. Ilyen helyzetben a konkretizálódott szimbólumok egyértelmű értékével folytathatjuk a végrehajtást azon a végrehajtási útvonalon, amihez tarto-

zó feltételrendszert megoldottuk. A tekintett példában a $b = 0.0$ értékkel folytatva a végrehajtást a 8. sorban egy 0-val való osztás detektálható. Amennyiben a b szimbólum nem szerepelne a feltételrendszerben és egyértelműség esetén nem használnánk fel a konkrét értéket, akkor a nullával való osztás detektálása megghiúsulna.

A feltételrendszer építés algoritmusának lényegi lépési a 4.1.2. ábrán látható pszeudokód írja le. Az algoritmus akkor hajtódik végre, amikor a szimbolikus végrehajtási fában egy elágazás következik be.

```
1.  Constraint constraint;  
2.  set<Constraint> actualConstraints;  
3.  if (onTrueBranch()) {  
4.    constraint = constraintBuilder.createConstraint();  
5.  } else if (onFalseBranch()) {  
6.    constraint = constraintBuilder.createNegatedConstraint();  
7.  }  
8.  actualConstraints.insert(constraint);  
9.  actualConstraints.union(dependenciesOfSymbolsInConstraint);  
10. pathCondition.union(actualConstraints);  
11. constraintSolver.solve(pathCondition);  
12. if (pathCondition.hasSolution) {  
13.   if (!pathCondition.hasMoreSolution) {  
14.     buildBackSolutions();  
15.   }  
16. } else {  
17.   weight = 0.0;  
18. }
```

4.1.2. ábra. A feltételrendszer építés algoritmusának pszeudokódja.

Elsőként annak a meghatározása történik, hogy az igaz, vagy a hamis végrehajtási ágon bővítjük a feltételrendszert, és ettől függően a forráskódban szereplő logikai kifejezésből, vagy annak negáltjából hozunk létre egy feltételt, ez kerül a *constraint* változóba (3-7. sor). (A Java nyelv *switch* utasításának kezelését a pszeudokód nem szemlélteti.) Fontos, hogy pontosan abból a kifejezésből készül feltétel, ami a forráskódban is szerepel, a szimbólumoknak nem történik olyan fajta helyettesítése, mint amit a 2.1. fejezet 2.1.1-es példájánál láttunk a *dist* változó esetén. Ezután a létrehozott constraintet elhelyezzük az *actualConstraints* feltételhalmazban (8. sor), majd ebben a feltételben szereplő szimbólumok minden függőségét is beszúrjuk (9. sor). Ezek a függőségek az értékadások révén létrejött feltételek lesznek, létrehozásukról a későbbiekben lesz szó. A következő lépés az, hogy az így létrejött constraint halmazzal bővítjük az aktuális útvonalhoz tartozó path condition-t, ami szintén egy constraint halmaz (11. sor), majd megoldatjuk azt a constraint solverrel (12. sor). Amennyiben a path condition nem kielégíthető, az aktuális útvonal súlyát 0-ra állítjuk jelezve ezzel azt, hogy az útvonalon nem folytatódhat a végrehajtás az ellentmondó feltételek miatt (17. sor). Ha van

megoldás, megvizsgáljuk annak egyediségét, vagyis, hogy létezik-e másik olyan megoldás, ami kielégíti a feltételrendszert (13. sor), és amennyiben ez nem teljesül, az egyértelmű szimbólumokat értéként visszaépítjük a jelenlegi állapot adattárolásáért felelős *ProxyTree*-be (14. sor).

Hangsúlyozandó, hogy egy konkretizálódott szimbólumot kizárólag egyszer, és csak annak az állapotnak a *ProxyTree*-jébe építhetünk vissza, amelyhez tartozó feltételrendszer legelőször konkretizálta azt. Ha egy későbbi állapot *ProxyTree*-jébe is beépítenénk a konkrét értéket, az hibás működéshez vezetne olyan esetekben, amikor a kérdéses szimbólumot újradefiniálja a program. A probléma szemléltetéséhez, tekintsük a 4.1.3. ábrán lévő egyszerű kódrészletet. Az *A* szimbolikus állapotban a path condition $a=10$, az *a* szimbólumra egyértelmű megoldást kapunk, azt visszaépítjük. A *B* állapotban a PC $a=10 \wedge b=11$, *a*-ra és *b*-re is egyértelmű megoldást kapunk, de *a*-t már korábban visszaépítettük és ha itt is megtennénk az $a=10$ -es értékkel folytatódna a végrehajtás, holott az 5. sorban lévő értékadás miatt *a* értéke már 3 kell, hogy legyen.

```
1. // a és b int típusú szimbólumok
2. if (a == 10) {
3.     // A állapot, a konkretizálása itt
4.     ...
5.     a = 3;
6.     if (b == 11) {
7.         // B állapot, csak b-t építjük vissza
8.     }
9. }
```

4.1.3. ábra. Példakód a többszöri visszaépítés problémájának szemléltetésére.

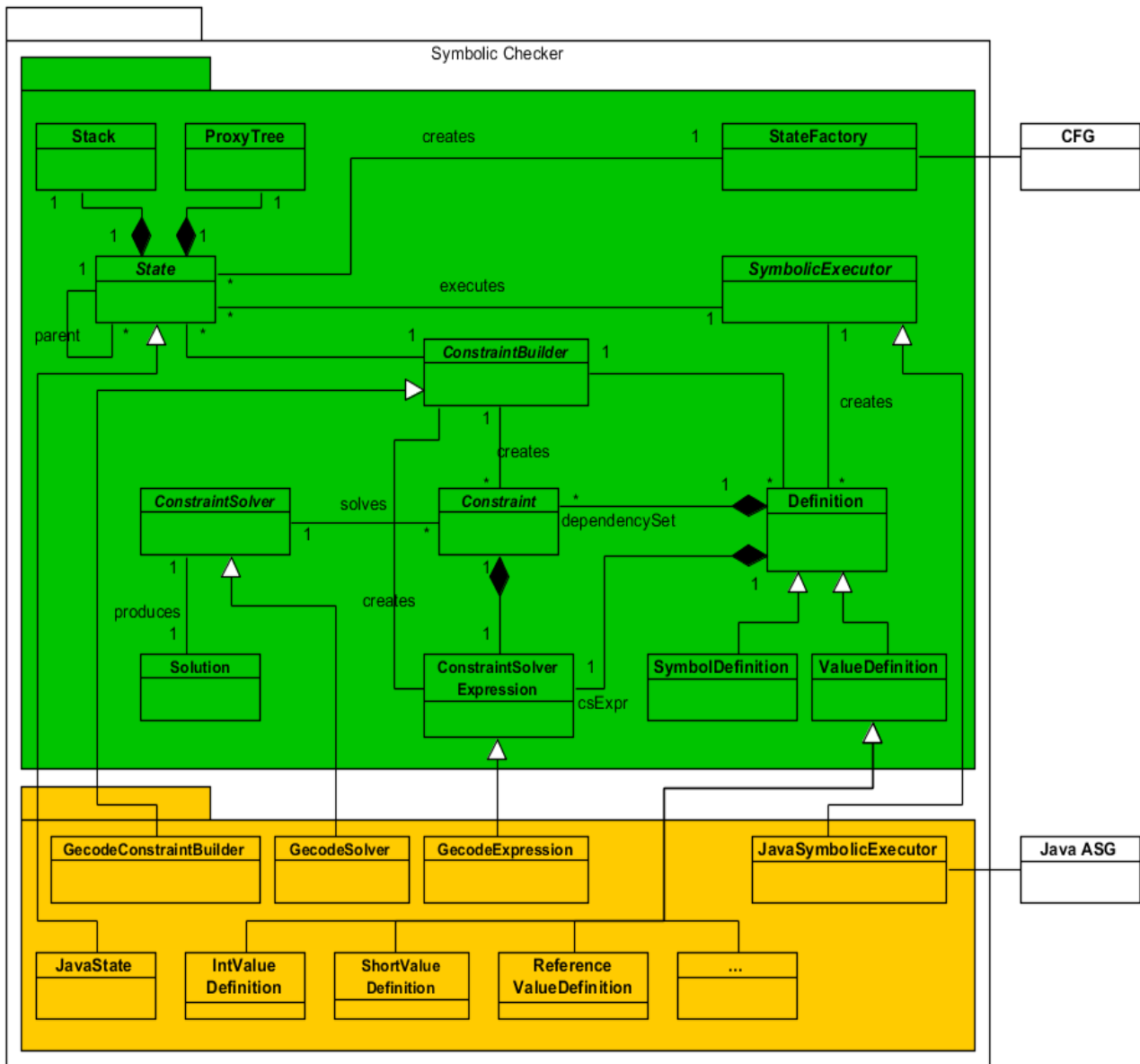
4.2. Megvalósítás

A feltételrendszer kielégítése és felépítése a Gecode constraint solver eszközkészlettel [5] történik. Alapvetően kétféle feltételről beszélhetünk: (1) az elemzett program szelekciós vezérlési szerkezeteiben szereplő feltételek, amelyeknél a szimbolikus végrehajtás szétágazik (ide értve a ciklusokban szereplő feltételeket is), valamint (2) ezen feltételekben szereplő szimbólumok függőségei. A következőkben azt írom le, hogy ezen feltételek építése, hogyan valósul meg a Symbolic Checkerbe integrálva.

A 3. fejezetben leírtak szerint a programban megjelenő minden adatra egy *Definition* objektum jön létre, legyen az egy változó, literál, vagy egy részkifejezés kiértékelésének eredménye. Tulajdonképpen a program végrehajtása ezen *Definition* objektumok propagálódását jelenti. A feladat az, hogy megvalósítsuk azt a nyomkövetést, ami megmondja, hogy egy Defi-

nition milyen más Definition-ökből és milyen műveletek által keletkezett. Így írjuk le azokat az összefüggéseket, amik az egyes szimbólumok között fennállnak és ezek szükségesek a feltételek felépítéséhez.

A feltételrendszer építés megvalósításához elkészült osztályokkal kiegészített osztálydiagram a 4.2.1. ábrán látható.



4.2.1. ábra. A Symbolic Checker osztálydiagramja kiegészítve a feltételrendszer építést megvalósító osztályokkal.

A megvalósításhoz minden Definition objektumba felveszek egy ún. *constraintSolver-Expression* adattagot, valamint egy *függőségi halmazt*, ami valójában egy feltételhalmaz.

Ezen attribútumok a Definition objektumokkal együtt propagálódnak a program szimbolikus végrehajtása folyamán.

A *constraintSolverExpression* a használt constraint solverrel, a Gecode-dal létrehozott kifejezésobjektum. A Gecode ezzel képes reprezentálni a kifejezések belső szerkezetét, vagyis azt, hogy milyen operandusokból, milyen műveletekkel hozták létre. A Gecode-os constraint-Solver kifejezés az absztrakt *ConstraintSolverExpression* osztályt mögé van rejtve, maga a Gecode megvalósítás a szimbolikus végrehajtó többi része elől ekképpen elrejtve marad. A *GecodeExpression* osztály egy wrapper osztályként funkcionál, amely a tényleges Gecode által felépített kifejezéseket csomagolja be. Három típusú Gecode-os kifejezés létezik: *IntVar*, *FloatVar* és *BoolVar*. Az elnevezések a reprezentált kifejezés típusára utalnak: egész, lebegőpontos és logikai. Ezekből az alaptípusokból származtatom a Java nyelv egyéb típusait. A Gecode-os terminológia ugyan változóknak (*Var*) nevezi ezeket a kifejezéseket, de valójában tetszőleges összetett kifejezést reprezentálhatnak. A *GecodeExpression* osztály azon túl, hogy becsomagol egy *IntVar*-t, *FloatVar*-t, vagy *BoolVar*-t azt is nyilvántartja, hogy a reprezentált kifejezésben, melyek az elemi szimbólumok, elemi Gecode-os változók. Ezt azért fontos nyilvántartani, mert amikor egy feltételrendszert megoldatunk, akkor meg kell mondani, hogy pontosan mely szimbólumokra, változókra szeretnénk megoldást kapni.

A *constraintSolverExpression*-ök úgy propagálódnak, hogy amikor műveletet végzünk a Definition objektumokon, vesszük a művelet operandusainak *constraintSolverExpression*-jeit, azokra is felírjuk a műveletet, s az így létrejött összetett kifejezést beállítjuk az eredmény Definition *constraintSolverExpression*-jének. ValueDefinition-ökon végzett műveletek esetén hatékonysági okokból a *constraintSolverExpression*-ökre nem írjuk fel a műveletet, hanem egyszerűen az eredmény értékhez létrehozunk egy olyan új gecode-os kifejezést, ami a kiszámolt értéket reprezentálja. Magán a Gecode-os kifejezésen túl a nyilvántartott elmei szimbólumok is propagálódnak, amelyekre megoldást kérünk a későbbiekben. Ez úgy történik, hogy az operandusok *ConstraintSolverExpression*-jeiben tárolt szimbólumokat egyesítjük, s ez az egyesített szimbólumhalmaz lesz az eredménykifejezés szimbólumhalmaza.

Az osztálydiagramon megjelenő *Constraint* osztály egy feltételt reprezentál. Egy logikai típusú (*BoolVar*) *ConstraintSolverExpression* objektumbot tárol és definiál a kielégítéshez szükséges interfészt. Ilyen *Constraint* objektumok halmaza lesz a Definition-ben tárolt másik fontos komponens, a *függőségi halmaz* (*dependency set*).

A *függőségi halmaz* tartalmazza a Definition-ben lévő *constraintSolverExpression*-ben

előforduló szimbólumok függőségét, amelyeket az értékadásokkal definiál a program. Erre azért van szükség, mert olyan értékadás esetén, ahol a jobb oldal egy SymbolDefinition, a bal oldalon szereplő változóra egy új, elemi szimbólumot veszünk fel, hogy a baloldali változó, mint különálló szimbólum megjelenjen. Ez az új szimbólum nem veszi át a jobb oldal constraintSolverExpression-jét, az értékadás jobb és bal oldalán szereplő SymbolDefinition objektumok közötti kapcsolatot úgy fejezzük ki, hogy felveszünk egy egyenlőséget kifejező feltételt (függőséget) a két oldal között. A függőségi halmaz a következő módon propagálódik. A műveletvégzés eredményeként létrejövő Definition függőségi halmaza az operandusok függőségi halmazának uniója lesz. Olyan értékadás esetén, amelynek jobb oldala SymbolDefinition, a bal oldalon keletkező új SymbolDefinition függőségi halmaza a jobb oldal függőségi halmaza kiegészítve a bal- és jobboldal constraintSolverExpression-jei közötti egyenlőséget definiáló feltétellel.

A szimbolikus végrehajtási fa elágazásait előíró feltételek is a programban szereplő kifejezések, ezért azok is Definition objektumként (valójában SymbolDefinition-ként) jelennek meg a Symbolic Checkerben. A 4.1.2. ábrán szemléltetett algoritmusban a *constraint* változó, ennek a Definition objektumnak a *constraintSolverExpression*-jéből képződik, a *dependenciesOfSymbolsInConstraint* feltételhalmazt pedig ennek a Definitionnek a függőségi halmaza alkotja. Ezek egyesítésével áll elő az *actualConstraints* halmaz, amellyel a path conditiont bővítjük.

A következőkben a 4.2.2. ábrán lévő kódrészleten mutatom be a feltételrendszer kialakulását a függőségi halmazok propagálódásának folyamatával. A *c* és a *d* változókat kezeljük szimbólumként, ezek olyan SymbolDefinition-ök, amelyek függőségi halmaza üres a constraintSolverExpression pedig egyetlen gecode-os ismeretlenből álló kifejezést jelent. A 2. sorban elsőként létrejön a 3-as egész literálra egy ValueDefinition, aminek a függőségi halmaza üres, majd a + művelet hatására létrejön a *d+3* SymbolDefinition. A *d+3* függőségi halmaza, a jobb- és baloldal függőségi halmazának uniója lesz, ami szintén üres halmaz:

$$\text{SymbolDef}(d+3).\text{depset} = \text{SymbolDef}(d).\text{depset} \cup \text{ValueDef}(3).\text{depset} = \emptyset$$

Az értékadás végrehajtásával a *b* szimbólum dependencia halmaza:

$$\text{SymbolDef}(b).\text{depset} = \text{SymbolDef}(d+3).\text{depset} \cup \{b=d+3\} = \{b=d+3\}$$

A 3. sorban létrejövő *c* szimbólum függőségi halmaza, a *b* szimbóluméhoz hasonló módon alakul:

$$\text{SymbolDef}(c).\text{depset} = \text{SymbolDef}(2 \cdot d).\text{depset} \cup \{c=2 \cdot d\} = \{c=2 \cdot d\}$$

A 4. sor értékadásának bal oldalán a $b-c$ SymbolDefinition függőségi halmaza a b és a c definíciók függőségi halmazának uniója:

$$\text{SymbolDef}(b-c).\text{depset} = \text{SymbolDef}(c).\text{depset} \cup \text{SymbolDef}(b).\text{depset} = \{b=d+3, c=2\cdot d\}$$

Majd az a függőségi halmaza:

$$\text{SymbolDef}(a).\text{depset} = \text{SymbolDef}(b-c).\text{depset} \cup \{a=b-c\} = \{b=d+3, c=2\cdot d, a=b-c\}$$

Az 5. sorban lévő $42 == a$ kifejezésre épülő SymbolDefinition függőségi halmaza megegyezik az a függőségi halmazával, amiből a path condition az *if* utasítás igaz ágán:

$$PC = \{42=a\} \cup \{b=d+3, c=2\cdot d, a=b-c\} = \{42=a, b=d+3, c=2\cdot d, a=b-c\}$$

A *Constraint* és a *ConstraintSolverExpression* objektumok létrehozásáért a *ConstraintBuilder* osztály felelős. Ez az osztály valójában szintén csak egy interfészt definiál, aminek Gecode-os megvalósítását a *GecodeConstraintBuilder* osztály adja. A 4.2.1. ábrán az is látható, hogy a *GecodeConstraintBuilder* a nyelvfüggő, Java rétegben kapott helyett. Erre azért volt szükség, mivel ez az osztály végzi a három Gecode-os alaptípus (int, float, boolean) leképezését a Java nyelv típusaira, ez állítja be az értéktartományok határait. Mindez szoros összefüggésben van a típus specifikus ValueDefinition-ökkel is, amelyek szintén a nyelvfüggő rétegekben vannak. A *ConstraintBuilder* gyártó objektumot alapvetően kétféle módon használjuk. Egyrészt a Definition objektumok konstruálásakor a megfelelő *constraintSolverExpression* létrehozására használják a Definition hierarchiában lévő osztályok. Másrészt a feltételek (*constraintek*) létrehozására használják az egyes értékadások esetén, valamint a *State* objektumok, a forráskódban lévő feltételek, illetve annak azok negáltjának létrehozására. A *ConstraintBuilder* osztály a deklarált metódusokkal a 4.2.3. ábrán szerepel.

```
class ConstraintBuilder {
public:
    ConstraintBuilder(void);
    virtual ~ConstraintBuilder(void);

    virtual ConstraintSolverExpr* makeIntValue(const int value) = 0;
    virtual ConstraintSolverExpr* makeLongValue(const long value) = 0;
    virtual ConstraintSolverExpr* makeShortValue(const short value) = 0;
    virtual ConstraintSolverExpr* makeByteValue(const char value) = 0;
    virtual ConstraintSolverExpr* makeCharValue(const wchar_t value) = 0;
    virtual ConstraintSolverExpr* makeBoolValue(const bool value) = 0;
    virtual ConstraintSolverExpr* makeFloatValue(const float value) = 0;
    virtual ConstraintSolverExpr* makeDoubleValue(const double value) = 0;
    virtual ConstraintSolverExpr* makeReferenceValue(const unsigned long value) = 0;
```

```

virtual ConstraintSolverExpr* makeIntVar() = 0;
virtual ConstraintSolverExpr* makeLongVar() = 0;
virtual ConstraintSolverExpr* makeShortVar() = 0;
virtual ConstraintSolverExpr* makeByteVar() = 0;
virtual ConstraintSolverExpr* makeCharVar() = 0;
virtual ConstraintSolverExpr* makeBoolVar() = 0;
virtual ConstraintSolverExpr* makeFloatVar() = 0;
virtual ConstraintSolverExpr* makeDoubleVar() = 0;
virtual ConstraintSolverExpr* makeReferenceVar() = 0;
...

virtual ConstraintSolverExpr* makeNumericExpr(const Definition& left,
                                              const Definition& right,
                                              const InfixOperatorKind op) = 0;
virtual ConstraintSolverExpr* makeNumericExpr(const Definition& operand,
                                              const PrefixOperatorKind op) = 0;
virtual ConstraintSolverExpr* makeBoolExpr(const Definition& left,
                                           const Definition& right,
                                           const InfixOperatorKind op) = 0;
virtual ConstraintSolverExpr* makeBoolExpr(const Definition& operand,
                                           const PrefixOperatorKind op) = 0;

virtual Constraint* makeConstraint(const ConstraintSolverExpr &expr) = 0;
virtual Constraint* makeNegatedConstraint(const ConstraintSolverExpr &expr) = 0;
}

```

4.2.3. ábra. A *ConstraintBuilder* osztály.

A függvények első csoportjával konkrét értéket ábrázoló *ConstraintSolverExpression*-ök hozhatók létre. Ezek a *makeIntValue()*, *makeByteValue()*, *makeBoolValue()*, stb. Ezen metódusok paramétere egy olyan változó, amivel a Symbolic Checker ábrázolja az egyes *ValueDefinition*-ökben lévő adatokat. A Java nyelv referenciáit egész számokkal reprezentáljuk, ennek megfelelően a *makeReferenceValue()* egy *unsigned long* típusú értéket vár.

A *ConstraintBuilder* függvényeinek második csoportja elemi szimbólumokat hoznak létre, a melyeket szintén egy *ConstraintSolverExpression*-be csomagolva adnak vissza: *makeIntVar()*, *makeByteVar()*, *makeBoolVar()*. Ahogy az korábban említésre került a Gecode három változótípust biztosít: egész, lebegőpontos, logikai. Ezek felhasználásával kell létrehozni a Java nyelvben lévő változótípusokat. A referencia típus egész típusból származtatható, a *boolean* típus pedig a Gecode által is definiált, így azon átalakítást nem kell elvégeznünk. A Java nyelv egész és lebegőpontos típusainak ábrázolásához a Gecode-os *IntVar*-t és *FloatVar*-t úgy használtam fel, hogy ezen változók értéktartományát a Java nyelv által definiált értéktartományoknak megfelelően állítottam be. A 4.2.4. ábrán lévő táblázat a Java nyelv egész és lebegőpontos típusainak értéktartományát jeleníti meg. Ezen értékkeszletek beállításával a Java nyelv változóit megfelelően reprezentálhatjuk.

Típus	Értékkészlet
long	-9223372036854775808 ... 9223372036854775807
int	2147483648...2147483647L
short	-32768...32767
byte	-128...127
char	0...65535
float	1.40129846432481707e-45 ... 3.40282346638528860e+38
double	4.94065645841246544e-324 ... 1.79769313486231570e+308

4.2.4. ábra. A Java nyelv primitív típusainak értékkészlete.

A *makeNumericExpression()* és *makeBoolExpression()* függvények arra szolgálnak, hogy egy művelet eredményét adják vissza *ConstraintSolverExpression*ként. A metódusok megkapják a művelet operandusait, amelyek *Definition* objektumok, és a magát a műveletet, amelyet el kell végezni. Ez alapján metódusok elkészítik az összetett kifejezést reprezentáló Gecode-os kifejezést. A Gecode számos műveletet támogat, amely a Java nyelvben előfordul, azonban van néhány kivétel. A Java programozási nyelv bitenkénti operátorai (*|*, *&*, *^*, *<<*, *>>*, *>>>*) sajnos nem támogatottak. Ilyen esetekben nem lehetséges elvégezni a műveleteket, az összetett kifejezést nem tudjuk felírni. A problémát úgy hidaltam át, hogy ekkor egyszerűen egy megfelelő típusú szimbólumot adok vissza. Ezzel ugyan romlik a pontosság, mert az adott kifejezés kialakulásáról semmit nem fogunk tudni. Hamis információ ezáltal azonban nem kerül a rendszerbe, az épített feltételrendszer konzisztenciája nem sérül. A Gecode biztosít lehetőséget arra, hogy saját adattípusokat és műveleteket definiálhassunk, ezen lehetőség kiaknázása szerepel a jövőbeli tervek között.

A *makeConstraint()* függvény a kapott *ConstraintSolverExpression*ből egy *Constraint*et konstruál. A paraméter *ConstraintSolverExpression*nek logikai típusúnak kell lennie ellenkező esetben kivétel dobódik. A *makeNegatedConstraint()* függvény hasonlóan működik, de a kapott logikai kifejezést először negálja. Erre akkor van szükség, amikor egy elágazás hamis ágán halad tovább a szimbolikus végrehajtás.

A *ConstraintSolver* absztrakt osztály egyetlen *solve()* függvényt tartalmaz, amely paraméterben egy *Constraint* objektumokból álló halmazt (vagyis egy feltételhalmazt) kap és fel-

adata ezen feltételhalmazt kielégíteni. A visszatérési értéke egy *Solution* típusú objektum, amely a kapott eredménye, illetve a megoldással kapcsolatos egyéb metainformációkat tartalmaz, mint például, hogy van-e megoldás, ha van akkor több megoldás van-e, időkorláton belül sikerült-e kielégíteni a feltételrendszert. A *GecodeSolver* osztály a *Gecode* osztálykönyvtár megfelelően részeinek felhasználásával implementálja a *solve()* függvényt és előállítja a *Solution* objektumot, mint eredményt. A *GecodeSolver* a nyelvspecifikus rétegben szerepel, mivel egy szimbólumra kapott megoldáshoz megfelelő típusú *ValueDefinition*-t hoz létre, a konkrét *ValueDefinition* osztályok viszont már a nyelvspecifikus rétegben vannak.

4.3. A feltételrendszer összefüggő komponensekre bontása

Mit ahogy azt a 4.1.2. ábrán bemutatott algoritmus szemlélteti, a kibővített path condition megoldás után megvizsgáljuk, hogy a megoldás egyértelmű-e, és amennyiben ez teljesül, a megoldásokat visszaépítjük az aktuális állapot *ProxyTree*-jébe. Egy valós programban igen nagy feltételrendszerek épülhetnek, amelyek sok szimbólumot tartalmaznak. Könnyen belátható, hogy csekély annak az esélye, hogy egy ilyen nagy formulahalmazt kiértékelve, annak pontosan egy megoldását kapjuk.

A 4.3.1. ábrán látható kódrészlet rávilágít a szóban forgó problémára. Mindkét *if* utasítás igaz ágán haladva a path condition $b > 0 \wedge a = 0$ lesz. Ennek a formulának viszont végtelen sok megoldása van, mert a $b > 0$ feltételt bármely pozitív b kielégíti, ugyanakkor a értéke kizárólag 0 lehet, amit előnyös lenne felhasználni a végrehajtás további részében.

```

1. //      a      és      b
   szimbólumok
2. if (b > 0) {
3.   ...
4.   if (a == 0) {
5.     // visszaépítés?
6.     ...
7.   }
8. }
```

4.3.1. Kódrészlet, ami rámutat arra, hogy a path condition-nek több megoldásai is lehet, mégis konkretizál változót.

A problémára azt a megoldást dolgoztam ki, hogy a path condition feltételhalmazt összefüggő komponenseire bontom, vagyis olyan feltétel halmazokra, amelyek egymástól függetlenek, így azok külön-külön megoldathatók és egyértelműség esetén az eredmények visszaépíthetőek. Két formula akkor kerül egy komponensbe, ha azokban szerepel ugyanaz a szimbólum. Az összefüggő komponensekre bontás után a path condition egy feltétel halmazoz-

kat tároló halmaz lesz.

A path condition erősen összefüggő komponenseire bontása kapcsán megállapítható, hogy a forráskód által előírt elágazási feltétel és annak függőségi halmaza biztosan egy komponensbe kerülnek, vagyis a 4.2. ábrán szereplő algoritmusban az *actualConstraints* halmazt biztosan nem kell szétbontani. Ezt a gondolatmenetet folytatva a path condition ezen *actualConstraints* halmazok halmaza lesz, és az összefüggő komponensekre bontás tulajdonképpen azt jelenti, hogy ezen halmazok közül össze kell vonni azokat, amelyekben van olyan feltétel, amikben megjelenik ugyanaz a szimbólum.

A path condition összefüggő komponensekre bontásának algoritmusát a 4.3.2. ábrán látható pszeudokód írja le. Az algoritmus inputja az *actualConstraints* halmazok halmazaként előálló path condition, outputja pedig az összefüggő komponensek halmaza. Tulajdonképpen szélességi bejárásokat hajt végre egy olyan irányítatlan gráfon, amelynek csúcsai a path condition halmazai, két halmaz között pedig akkor van él, ha van bennük olyan feltétel, amely ugyanarra a szimbólumra fogalmaz meg megszorítást.

```
1.  set< set<Constraint> > components;
2.
3.  while (!pathConition.empty()) {
4.      queue< set<Constraint> > reached;
5.      set<Constraint> > first = pathCondition.begin();
6.      reached.push(first);
7.      pathConition.erase(first);
8.      set<Constraint> component;
9.      while (!reached.empty()) {
10.         set<Constraint> constSet = reached.popFront();
11.         component.union(constSet);
12.         foreach (set<Constraint> neighbour : constSet.neighbourhood()) {
13.             reached.push(*neighbourIt);
14.             pathConition.erase(neighbour);
15.         }
16.     }
17.     components.insert(component);
18. }
19.
20. return components;
```

4.3.2. ábra. A path condition feltételrendszer összefüggő komponensekre bontásának algoritmus.

Elsőként kiválasztjuk a path condition egy elemét (akár véletlenszerűen), azt elhelyezük a *reached* sorban, majd töröljük ezt az elemet a PC-ből (5-7. sor). Ezután megkezdjük a bejárást ebből az elemből indulva a 9. sorban lévő ciklussal. A *reached* sorból kivett elemet egyesítjük a *component* halmazzal, ami az aktuálisan meghatározandó összefüggő komponens

(11. sor), majd ennek az elemnek a még a PC-ben lévő szomszédait beszúrjuk a *reached* sorba, töröljük a PC-ből (12-16. sor) és ezt a műveletsort megismételjük a *reached* következő elemével. A bejárással elért halmazok unióját, a *component* halmazt beszúrjuk a *components* halmazba, majd amennyiben még van elem a PC-ben egy újabb összefüggő komponens keresését indítja el a külső *while* ciklus (3. sor)

Végül a feltételrendszer építés összefüggő komponensekre bontással kiegészített algoritmus a 4.3.3. ábrán szerepel. A módosítás lényege az, hogy a path condition minden összefüggő komponensét külön-külön megoldjuk, és amennyiben egyértelmű megoldást kapunk, azt visszaépítjük.

4.4. Tesztelés

A Symbolic Checkerhez a fejlesztés alatt folyamatosan készültek unit tesztek, és természetesen a feltételrendszer építéshez is készítettem tesztek. A projekt a *Google C++ Testing Framework*-öt [13], röviden a *Google Test*et használja, ezzel összhangban én is ezt a keretrendszert használtam tesztek elkészítéséhez és futtatásához. Tesztesetek a fejlesztés alatt folyamatosan készültek amelyeknek a gyakorlatban nagy hasznát vettem. Az új funkciók és képességek implementálása után rendszeresen újravittam a meglévő tesztek (regressziós teszt), így számos olyan hibára, hiányosságra fény derülhetett, amire a fejlesztés alatt nem gondoltam, illetve ha egy fejlesztés esetleg elrontott egy korábban jól működő funkciót akár a szimbolikus végrehajtó alap működésében, arra is hamar fény derülhetett. A feltételrendszer építéséhez ezidáig 110 darab tesztet készült el.

A tesztesetek inputja egy Java program, illetve az abból előálló ASG és CFG. Egy tesztet elsőként lefuttatja a Symbolic Checkert a kapott programon, majd elvégzi a definiált ellenőrzést, az elvárt outputhoz történő összehasonlítást. Többnyire ezek a vizsgálatok a felépített szimbolikus végrehajtási fához, az egyes állapotokhoz, illetve az állapotok által tárolt változókhoz kötődnek.

A tesztesetek létrehozásához a *Google Test*ben elsőként egy úgynevezett *test fixture*-t hoztam létre, ami arra szolgál, hogy ugyanazt a környezetet, konfigurációt használjam több tesztet is. Esetemben az azonos környezet azt jelenti, hogy minden esetben le kell futtatni a Symbolic Checkert egy adott Java programon és el kell végezni az ezzel kapcsolatos összes teendőt. Egy *test fixture* létrehozásához a *Google Test* keretrendszerben definiált `::testing::Test` osztályból kell származtatni egy saját osztályt, aminek `SetUp()` függvényével inicia-

lizálható és felépíthető a kívánt környezet, a *TearDown()* függvény pedig elvégezhetőek a tesztelés lefutása utáni teendők, például felszabadíthatók azok az erőforrások, amelyeket a *SetUp()*-ban allokalunk. A *test fixture*-höz tesztelések *TEST_F* makróval definiálhatóak, amelyekben pedig ASSERT kezdetű makrókkal fogalmazhatjuk meg az elvárt és a valójában kapott kimenet között fennálló elvárásokat.

A 4.4.1. ábra (a) részén a feltételrendszer építéshez létrehozott *test fixture* váza és egy tesztelés látható. Az ábra (b) része az a Java programot ábrázolja, melyen a Symbolic Checker futását teszteli a szemléltetett tesztelés.

```

1.  class SymbolTrackerTest : public ::testing::Test {
2.  protected:
3.      java::asg::ASGFactory *m_asgFactory;
4.      cfg::CFGFactory *m_cfgFactory;
5.      ...
6.
7.      virtual void SetUp() { ... }
8.      virtual void TearDown() { ... }
9.      void setupFactory(string jsi) { ... }
10.     void runSymbolicTest(string method) { ... }
11.     ...
12.
13. };

14. TEST_F(SymbolTrackerTest, SymbolTrackerTestBuildBack12) {
15.     setupFactory("SymbolTrackerTestBuildBack12");
16.     runSymbolicTest("action");
17.
18.     State *state0 = getStateByNodeName("thisState0");
19.     SmartDefinition def = getSmartDefinitionInStateByNodeId(state0,
20.                                                             getNodeIdByName("a"));
21.     if (def->isValueDefinition()) {
22.         int expectedValue_a = 2;
23.         int value = boost::dynamic_pointer_cast<IntValueDefinition>(def)->getValue();
24.         ASSERT_TRUE(value == expectedValue_a) << "expected: " << expectedValue_a <<
25.                                                             "reality: " << value;
26.     } else {
27.         ASSERT_TRUE(false) << "value definition not found for a";
28.     }
29.
30.     State *state1 = getStateByNodeName("thisState1");
31.     def = getSmartDefinitionInStateByNodeId(state1, getNodeIdByName("a"));
32.     ASSERT_TRUE(def->isSymbolDefinition());
33. }

```

(a)

```

1. public class SymbolTrackerTestBuildBack12 {
2.     void action(int a, int b) {
3.         if (a > 0) {
4.             if (a <= 1) {
5.                 a = 2;
6.                 if (b == 10) {
7.                     int thisState0 = 0;
8.                 }
9.             }
10.        } else {
11.            int thisState1 = 0;
12.        }
13.    }
14. }

```

(b)

4.4.1. ábra. (a) A feltételrendszer építéshez teszteléséhez létrehozott *test fixture* váza és egy teszteset. **(b)** A Java program, amelyen a Symbolic Checker futását teszteljük.

A *SymbolTrackerTest* nevű *test fixture* a *SetUp()* és *TearDown()* függvényeken kívül számos egyéb segédfüggvényeket is definiál, amelyek itt nincsenek feltüntetve. A *setUpFactory()* végzi el az adott nevű Java fájl beolvasását majd abból az ASG és a CFG felépítését, *runSymbolicTest()* pedig elindítja a Symbolic Checkert. A teszteset, amely a *SymbolTrackerTestBuildBack12* nevet viseli elsőként elvégzi a beolvasást, majd elindítja a szimbolikus végrehajtást (15-16. sor). Ezután a 18. sorban lekéri a végrehajtási fa egy állapotát, mégpedig azt, amelyikben a *thisState0* változót létrejön. A (b) ábrára tekintve ez a 7. sorban keletkező állapot lesz. A Java kódban a *thisState0* és *thisState1* változók csupán állapotkiejelölő funkciót töltenek be. A 19. sor a *getSmartDefinitionInStateByNodeId()* az *a* nevű változóhoz tartozó Definition objektumot kéri le, majd a tesz további részen ezen a definition objektumon jat végre ellenőrzéseket. Megvizsgálja, hogy ValueDefinition-e (természetesen annak kell lennie a (b) ábra 5. sorában található értékadás miatt), ha igen, lekéri az értékét és összeveti az *expexctedValue_a* változóban tárolt elvárt értékkel az *ASSERT_TRUE* makró felhasználásával. Ha ez nem teljesül, a teszteset hibásan fut le. Amennyiben a lekért állapotban (*state0*) az a változó nem ValueDefinition, szintén hibával tér vissza a teszteset (27. sor). Az (a) ábra 30. és 31. sorában a *thisState1* változóhoz tartozó állapotban lévő a változót kérjük le és a 32. sorban lévő *ASSERT_TRUE* azt vizsgálja, hogy az *a* változó egy szimbólum-e.

5. KIÉRTÉKELÉS

A feltételrendszer építéssel kiegészített Smybolic Checker elemzőeszközt többféle módon teszteltem, a fejezet ezen futtatások eredményeit taglalja. Elsőként néhány kisebb méretű példaprogram elemzését mutatom be, kiemelve miben különböznek a kapott eredmények a JPF Checkerrel, és a feltételrendszer építés nélkül futtatott Symbolic Checkerrel kapott eredményekhez viszonyítva, demonstrálva ezzel a feltételrendszer építés előnyeit, majd az eddigi valós rendszereken történő futtatás tapasztalatairól írok.

Elsőként tekintsük a 5.1. ábrán lévő Java forráskódot. A *Property* osztály *merge()* metódusától indul az elemzés. A metódus egy *other* nevű szintén *Property* típusú paraméterrel rendelkezik, amit szimbólumként kezelünk, továbbá mivel egy példánymetódusról van szó, a *this* objektum *name* és *repeated* adattagjai is szimbólumként viselkednek mind a JPF Checker, mind a Symbolic Checker esetén. A metódus első sorában rögtön a *convertName()* statikus metódust hívja meg önmagát átadva paraméterként (28. sor). A *convertName()* először megvizsgálja, hogy a *p* *Property* *repeated* adattagja hamis-e, és amennyiben igen, visszatér, különben pedig a 41. sortól folytatja a végrehajtást. A 42. sorban ismét egy *if* utasítás következik, ha a *p* *name* adattagja null-tól különböző akkor az igaz ág végrehajtása folytatódik. Érdekesebb azonban a hamis ágot vizsgálni, ahol a következő path condition-t kapjuk:

$$\neg \neg p.repeated \wedge name = p.name \wedge \neg (name \neq null) \\ \Rightarrow \\ p.repeated \wedge name = p.name \wedge name = null$$

Ez a feltételrendszer azon túl, hogy előírja, hogy a *repeated* adattagnak igaznak kell lennie (amit valójában már korábban a 41. sortól tudunk), a *name* lokális változóra, s ezen keresztül a *p* objektum *name* tagjára is egyértelmű értéket, null-t ír elő. A null érték beépül az éppen aktuális állapot változói közé, s folytatódik a végrehajtás, a *convertName()* üres stringgel visszatér a hívó *merge()* metódushoz. A 29. sor *if* utasításának a hamis ágán haladva tovább a *Property* osztály 12. sorában található konstruktora hívódik meg. Ez a konstruktor a 15. sorban hivatkozik a *p1* paraméterére, és annak *name* adattagját dereferálja. Azonban ez a *name* változó az, amelyre az előzőekben tárgyalt feltételrendszer kiértékelése után visszaépítettük a null referenciát, ami itt egy *NullPointerException* megtalálását eredményezi.

A Symbolic Checker egy olyan végrehajtási útvonalat talált tehát, ami feltételezi és kezeli azt, hogy a *name* attribútum a null referencia, később viszont ezt az ellenőrzést elmulaszt-

ja, ami egy null pointer dereferencia futásidejű programhibához vezet. Az 5.1. ábrán bemutatott példa teljesen életszerű helyzetet szemléltet. Számos esetben előfordulhat, hogy egy program az adott referencia ellenőrzését a legtöbb helyen dereferálás előtt elvégzi, néhány helyen azonban ez elmarad, ami a program leállásához vezethet.

```

1.  import java.util.Date;
2.
3.  public class Property {
4.      private String name;
5.      private boolean repeated;
6.
7.      public Property(String name, boolean repeated) {
8.          this.name = name;
9.          this.repeated = repeated;
10.     }
11.
12.     public Property(Property p1, Property p2, String newName) {
13.         repeated = p1.isRepeated();
14.         name = newName + "(";
15.         if (p1.getName().length() >= p2.getName().length()) {
16.             name += p1.getName() + "," + p2.getName();
17.         } else {
18.             name += p2.getName() + "," + p1.getName();
19.         }
20.         name += ")";
21.     }
22.
23.     public String getName() { return name; }
24.
25.     public boolean isRepeated() { return repeated; }
26.
27.     public Property merge(Property other) {
28.         String newName = convertName(this);
29.         if (other.getRepeated()) {
30.             name = newName;
31.             return this;
32.         } else {
33.             return new Property(this, other, newName);
34.         }
35.     }
36.
37.     public static String convertName(Property p) {
38.         if (!p.isRepeated()) {
39.             return p.getName();
40.         }
41.         String name = p.getName();
42.         if (name != null) {
43.             if (name.length() == 0) {
44.                 return "";
45.             } else {
46.                 return name.replaceAll("REP", "_");
47.             }
48.         } else {
49.             return "";
50.         }
51.     }
52.
53. }

```

5.1. ábra. *Property* osztály, melynek *merge()* metódusát futtatva *NullPointerException* detektálható a feltétel-

rendszer építés alkalmazásával.

A feltételrendszer építés nélkül futtatott Symbolic Checker ezt a hibát nem képes felfedezni. Ugyan bejárja ugyanazt a végrehajtási útvonalat, amelyet a feltételrendszer építéssel kiegészített Symbolic Checker, de a 42. sorban található *if* hamis ágán nem használja fel az egyértelmű értéket. A JPF Checkert futtatva szintén nem találjuk meg a 15. sorban lévő null dereferenciát, ellenben a 29. sorra egy lehetséges *NullPointerException* dobást detektál, mivel az *other* paramétert null-ra is inicializálja a *lazy initialization* algoritmust alkalmazva. Ez a találat azonban false positive-nak tekinthető.

Az 5.2. ábrán feltüntetett példaprogramban egy tömbtúlindexelési hibát detektál a feltételrendszer építéssel kiegészített Symbolic Checker a *run()* metódusban.

Elsőként végigkövetjük, hogy ezt mi okozza, majd azt vizsgáljuk meg, hogy hogyan detektálható. Az 5. sorban egy *max* hosszú tömb jön létre, ami az *arr* nevet viseli. Amennyiben a paraméterben kapott *n* nagyobb, mint 0 (6. sor), egy műveletsor végrehajtása kezdődik meg ezen a tömbön, ami két összeget hivatott kiszámolni a tömb tartalma alapján. Elsőként feltöltjük a tömböt (7-9. sor), majd az *n* változótól, mint indextől indulva összegezzük minden páros indexű elemének *member* adattagját a *sum1* változóba (11-16. sor). Ezt követően *n*-nel és az *arr* 0-dik elemével meghívjuk a *gcd()* metódust (18. sor), és a visszatérési érték (-1)-szeresétől indulva összegezzük a tömb elemeit (20-22. sor). A *gcd()* metódus két szám legnagyobb közös osztóját számolja ki, eredménye biztosan egy pozitív szám lesz *n*-nel és *arr[0]*-val hívva. Emiatt a *negOfGcd* változó garantáltan negatív, ami a 21. sorban egy *ArrayIndexOutOfBoundsException* futásidejű kivétel megjelenésével a program leállítását eredményezi.

A *run()* metódus elemzésének elindításkor az *n* paraméter az egyedüli szimbólum. A *max* változó konkrét, az *arr* tömb is konkrétan példányosítódik, és az elemei is mind konkrét értékek lesznek. A 11. sorban lévő ciklus végrehajtása azonban már *n*-től függően történik. A hamis ágon a 16. sortól folytatódik a végrehajtás, az igaz ágon pedig belépünk a ciklusmagba, majd annak végrehajtása után ismét elágazunk a 11. sorban lévő feltételtől függően. Ez a szimbolikus végrehajtási fa maximális mélységig folytatódik. Amikor a végrehajtási útvonal legalább egyszer belépett a ciklusmagba, majd onnan kilépett, a kibővített path conditionnek biztosan része az alábbi formula:

$$\begin{aligned} n_{prev} < max \wedge n = n_{prev} + 1 \wedge \neg(n < max) \\ \Rightarrow \\ n_{prev} < max \wedge n = n_{prev} + 1 \wedge n \geq max \end{aligned}$$

Ebben a formulában n_{prev} az *n* szimbólum ciklusba való belépéskori példányát jelenti.

Ekkor az $n_{prev} < max$ feltétel írható fel. A ciklusmagban a 15. sor inkrementálása hatására az n változóhoz egy új szimbólum keletkezik (mivel újradefiniálás történt), majd felvesszük az $n = n_{prev} + 1$ feltételt a létrejött új szimbólumra. Mivel a következő iterációban már nem lépünk be, hanem a hamis ágon folytatjuk a $\neg(n < max)$ feltétel felvétele szükséges. A fenti részformulát kiértékelve az n szimbólumra egyértelmű megoldást kapunk, ami a max értékével egyezik meg. Mindez azt jelenti, hogy amennyiben kilépünk a 11. sor *while* ciklusából n értéke csakis max lehet.

A kapott egyértelmű értéket visszaépítve a *gcd()* metódus egyik paramétere sem szimbólum már, ezért az konkrétan hajtódik végre, és visszatérési értéke is egy konkrét szám lesz. Ez a szám, ahogy láttuk, negatív, ami rossz tömbindexeléshez vezet a 21. sorban.

```

1. class Example {
2.
3.     public void run(int n) {
4.         int max = getCharPos('w');
5.         A[] arr = new A[max];
6.         if (n > 0) {
7.             for (int i = 0; i < max; ++i) {
8.                 arr[i] = new A(max - i);
9.             }
10.            int sum1 = 0;
11.            while (n < max) {
12.                if (n % 2 == 0) {
13.                    sum1 += arr[n].getMember();
14.                }
15.                n++;
16.            }
17.            System.out.println("Sum1: " +
18.                sum1);
19.            int negOfGcd = -gcd(n, arr[0]);
20.            int sum2 = 0;
21.            while (negOfGcd < max) {
22.                sum2 += arr[negOfGcd+
23.                +].getMember();
24.            }
25.            System.out.println("Sum2: " +
26.                sum2);
27.        }
28.    }
29.
30.     public int getCharPos(char c) {
31.         return c - 'a' + 1;
32.     }
33.
34.     private int gcd(int x, int y) {
35.         while (y != 0) {
36.             int m = x % y;
37.             x = y;
38.             y = m;
39.         }
40.         return x;
41.     }
42.
43.     class A {
44.         private int member;
45.
46.         public A(int member) {
47.             this.member = member;
48.         }
49.
50.         public int getMember() {
51.             return member;
52.         }
53.     }
54. }

```

5.2. ábra. Példaprogram, melynek *run()* metódusát vizsgáljuk.

A fentiekben részletezett példa arra világít rá, hogy egy konkretizálódott változó miként képes a végrehajtás akár jelentős részét is konkrétá tenni. A szimbólumok továbbgyűrése mérséklődik, vagyis kevesebb változót kell ismeretlen adatnak számító szimbolikus változóként kezelni. Ennek köszönhetően az elemzés gyorsabbá válik, hiszen kevesebb útvonalat kell megvizsgálni. Példánkban az n változó konkretizálása nélkül a *gcd()* metódus teljes végrehajtási fáját is be kellene járni, ami viszonylag költségesnek számít a benne lévő ciklus miatt.

A bemutatott tömbtúlinindexelési hibát sem a JPF Checker, sem a feltételrendszer építés nélkül futtatott Symbolic Checker nem képes megtalálni.

Az utolsó példában egy valós kódrészleten, a log4j naplózó könyvtár 1.2.11-es verziójának `org.apache.log4j.net.SMTPAppender.sendBuffer()` metódusán mutatom be, hogy a feltételrendszer építéssel kiegészített Symbolic Checker miként képes false positive hibatalálathoz kiszűrni.

```
// SMTPAppender.java
public class SMTPAppender extends
    AppenderSkeleton {
    ...
    protected Layout layout;
    protected CyclicBuffer cb =
        new CyclicBuffer(bufferSize);
    ...
    protected
    void sendBuffer() {
        ...
224.     int len = cb.length();
225.     for(int i = 0; i < len; i++) {
226.
227.         LoggingEvent event = cb.get();
228.         sbuf.append(layout.format(event));
229.         if(layout.ignoreThrowable()) {
230.             String[] s =
                event.getThrowableStrRep();
231.             if (s != null) {
232.                 for(int j = 0; j < s.length; j++) {
233.                     sbuf.append(s[j]);
234.                 }
235.             }
236.         }
237.     }
    ...
}

public class CyclicBuffer {
    int numElems;
    ...
101. public
102. LoggingEvent get() {
103.     LoggingEvent r = null;
104.     if(numElems > 0) {
105.         numElems--;
106.         r = ea[first];
107.         ea[first] = null;
108.         if(++first == maxSize)
109.             first = 0;
110.     }
111.     return r;
112. }
    ...
119. public
120. int length() {
121.     return numElems;
122. }
    ...
}

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

5.3. ábra. Az `org.apache.log4j.net.SMTPAppender.sendBuffer()` metódus és környezete.

A `sendBuffer()` metódus 228. sorában a `CyclicBuffer get()` metódusa hívódik, amely egy `LoggingEvent` referenciával tér vissza. A `get()` metódus elsőként az `r` referenciát inicializálja `null`-ra (103. sor), majd amennyiben az osztály `numElems` adattagja nagyobb mint 0, az `r` változó új értéket kap. A hamis ágon azonban a metódus a `null`-ra inicializált `r` változóval tér

vissza. Ez utóbbi végrehajtási ágat követve az *SMTPAppender.sendBuffer()* metódus 227. sorában az *event* változó értéke null lesz, amit paraméterként átad a 228. sorban lévő *SimpleLayout.format()* metódusnak, amely pedig a 60. sorában dereferálja azt.

Ez azonban egy false positive hibatalálat lenne, a *format()* 60. sorára nem kerülhet null érték. A *sendBuffer()* 224. sorában a *cb* objektum *numElems* adtagját kérjük le, amelyre a 225. sorban a *for* ciklus az első iterációba belépve egy feltételt ír fel, a PC a következő módon alakul:

$$0 < len \wedge len = cb.numElems$$

A 227. sorban hívott *CyclicBuffer.get()* metódus azonban csak a hamis ágon tér vissza null-lal, ahol a *numElems > 0* feltétel nem teljesül, a path condition tehát az alábbi lesz:

$$0 < len \wedge len = cb.numElems \wedge \neg(numElems > 0)$$

Ez a formula azonban kielégíthetetlen, ami azt jelenti, hogy ezen a lefutási ágon nem folytatható a végrehajtás, a *sendBuffer()* 227. sorában az *event* változó nem kapja meg a null értéket, így a *SimpleLayout* osztály *format()* metódusa sem dereferálhatja azt. Mindez tulajdonképpen azt jelenti, hogy a *sendBuffer()* 225. sorában lévő *for* ciklusába csak akkor lép be a vezérlés, ha a *len* változó legalább 1, ekkor azonban a *get()* metódus sem ad vissza null-t a 104. sorban található *if* utasítás hamis ágán.

A tárgyalt false positive hibatalálat kiszűrése feltételrendszer építés nélkül futtatott Symbolic Checkerrel nem valósulhatott volna meg, azonban a JPF Checkerrel igen, mert itt szimbólum konkretizálódás nem történt és a path condition ellentmondásosságát a JPF/SPF-en alapuló megoldás is vizsgálja.

A feltételrendszer építéssel kiegészített Symbolic Checker elemzőeszközt nagy rendszerekben is lefuttattam, a kapott eredmények kiértékelése azonban még nem ért véget. A hibatalálatok egyesével történő manuális átnézése jelentős időigényű, a belépési pont és a hibatalálat közötti hosszú végrehajtási útvonalak végigkövetése és értelmezése miatt. Ami az eddigi eredmények alapján látszik, az az, hogy lényegesen kevesebb hibatalálatot kapunk a Symbolic Checker-t futtatva a JPF Checker találatszámához viszonyítva. Sok false positive találat kiszűrődik, amin a bemutatott feltételrendszer építés tovább javít. Ez nem jelenti azt, hogy a feltételrendszer építéssel kiegészített Symbolic Checker találati listájában nincsenek false positive találatok, de jelentős részük valós hibákra, potenciális hibaforrásokra hívja fel a figyelmet.

Az elemzések időtartamát tekintve a feltételrendszer építéssel futtatott Symbolic Checker futásideje a JPF Checker futásideje alatt marad, azonban ez az időtartam nagyjából két-

szerese a feltételrendszer építés nélkül futtatott elemzésének. Erre természetesen számítottunk, hiszen a feltételrendszer építése, annak összefüggő komponensekre bontása és főként ki-elégítése adott esetben nagy mennyiségű számítást igényelhet.

6. KAPCSOLÓDÓ MUNKÁK

Ebben a fejezetben olyan munkák és publikációk kerülnek bemutatásra, amelyek valamilyen módon kapcsolódnak a dolgozat témájához. Először bemutatom, hogy milyen módszerek és technikák léteznek futásidejű hibák megtalálására főként Java programokban, majd rámutatok arra, hogy milyen problémák megoldására alkalmaztak szimbolikus végrehajtást, végül olyan munkákat mutatok be, amik a szimbolikus végrehajtás technikáját kiegészítették, vagy módosítva használták fel.

Biztonság-kritikus rendszerek biztonságos kivételkezelésének a bizonyításával foglalkozik Weimer és Necula [23]. Olyan teszteseket generálnak, amely a nyelv valamilyen szabályának megsértésével kivételhez vezet. A tesztinputok generálása itt nem a szimbólumok feletti feltételrendszer megoldásával, hanem a program vezérlési folyam 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 [24] Java osztályok halmazát várja inputként. Az osztályok típusvizsgálata után egy Java programot hoz létre, amely példányosítja a megadott osztályokat és az osztályok publikus metódusait véletlenszerű paraméterezéssel futtatja. Ezzel olyan hibák kerülhetnek napvilágra, amik a program azonnali leállításához vezetnek, így például kezeletlen futásidejű kivételek is. A rendszer alkalmas JUnit tesztesetek generálására, továbbá az Eclipse fejlesztőkörnyezetbe is integrálható. JCrasher is egy speciális driver környezetet 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.

A DART [16] 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 véletlenszerű, 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 szimbolikus végrehajtás alapötlete nem új keletű, az első publikációk és végrehajtó

eszközök az 1970-es években jelentek meg. Az első alpmű a témában [4], 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 szimbolikusan 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 [14], 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) [15] 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 számára a forráskódban előre meg kell jelölni a szimbolikusan kezelendő változókat, míg a Symbolic Checker esetén a forráskód előzetes szerkesztése nem szükséges.

A fent említett és további rendszerek összefoglaló leírását és összehasonlítását [17] és [18] tartalmazza.

Hálózati protokoll implementációk verifikációjára is alkalmazták a szimbolikus végrehajtást [19]. 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 [20] egy SQL injection támadási lehetőségeket 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

vel 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 (JPF) [6], illetve a szimbolikus végrehajtó kiterjesztése, a Symbolic PathFinder (SPF) [7] fő alkalmazási területe a NASA belső projektjeinek verifikációja. Például a Symbolic PathFinder használatát írja le [21] 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 mellett dinamikus módszerekkel is történhet. A program manuális futtatása közben valamilyen monitorozó szoftverrel figyelhetik a végrehajtást, a lefutási ágak alapján következtetnek tervezési minták meglétére. von Detten [22] 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 MIX [25] rendszer a program szimbolikus végrehajtását statikus típusellenőrzés alapú technikákkal vegyíti. A programban típusos és szimbolikus blokkok kijelölésével határozza meg, hogy mely kódrészleteken kell szimbolikus végrehajtást és melyeken statikus típusellenőrzést végrehajtani. Ezen blokkok határán úgynevezett mix rule-okat használ az információ továbbításához. A rendszer célja, hogy egy kompromisszumos megoldást adjon a precíz, de erőforrásigényes szimbolikus végrehajtás és a kevésbé precíz, de gyorsabb típusellenőrzés között.

Shannon és mások [26] egy absztrakciós réteget építenek a Java stringkezelés fölé véges állapotú automaták felhasználásával. A megvalósítás a *java.lang.String* osztályon túl a *StringBuilder* és *StringBuffer* osztályokat is magában foglalja. Ennek köszönhetően a rendszer olyan constrainteket is képes kezelni, amelyekben stringek, stringműveletek is szerepelnek, ezáltal olyan programokra is alkalmazható, amik bonyolultabb stringekkel, például SQL lekérdezésekkel dolgoznak. A Symbolic Checker jelenleg nem kezel string-constrainteket, ennek megvalósítását későbbi fejlesztésekben tervezzük.

Programokat szimbolikusan végrehajtva előfordulhat, hogy a path conditionbe olyan feltételek kerülnek, amelyek függvényhívást tartalmaznak, például $if(y \geq f(x))$. Erre ad egy le-

hetséges megoldást az úgynevezett concolic (*concrete-symbolic*) végrehajtás [11], amely speciális constraintépítést alkalmaz. Ennek lényege, hogy két path conditiont tartunk karban. Az egyik tartalmazza azokat a feltételeket, amelyekben nem szerepel metódushívás, a másik az úgynevezett komplex PC, amiben a függvényhívást is tartalmazó feltételek szerepelnek. Az algoritmus elsőként kielégíti az egyszerű PC-t értékeket rendelve a szimbólumokhoz, majd ezeket az értékeket felhasználva konkrétan végrehajtja azokat a függvényhívásokat, amelyek lefutása függött a szimbólumoktól, amelyekhez immár konkrét értékeket rendeltünk. Az itt alkalmazott módszer annyiban hasonlít a dolgozatban bemutatott eljáráshoz, hogy a szimbólumok konkretizálódását igyekszik kihasználni a precízebb és hatékonyabb elemzés érdekében.

7. ÖSSZEFOGLALÁS ÉS TOVÁBBI CÉLOK

A szimbolikus végrehajtás alapelvei már a évtizedek óta ismertek és számos eszköz készült a módszer lehetőségeinek kiaknázására. A Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén fejlesztett szimbolikus végrehajtó elemező a Symbolic Checker a legtöbb ilyen eszköztől eltérő módon nem tesztelési vagy tesztinput generálással kapcsolatos célokat tűzött ki, hanem azt, hogy a lehető legpontosabban detektáljon olyan végrehajtási útvonalakat, amik futásidejű hibákra és veszélyes kódrészletekre vezetnek. Ezen cél elérése érdekében dolgoztam ki egy a hagyományos szimbolikus végrehajtás alapelveitől eltérő feltételrendszer építési mechanizmust és integráltam azt a Symbolic Checkerbe. A bemutatott megközelítés minden végrehajtási útvonalhoz egy olyan feltételrendszert épít, ami a szimbolikus inputoktól függő változókra is megfogalmaz megszorításokat és egyértelműség esetén fel is használja a kiszámolt értékeket. Ennek köszönhetően olyan hibák is detektálhatók, amelyeket a hagyományos módszerrel nem, ahogyan azt a Symbolic PathFinder alapú megvalósítással összevetve bemutatam. A szimbólumok konkretizálásával a szimbolikus végrehajtási fa mérete redukálható, amivel a performancia is javul, valamint az alaptan false positive találatok kiszűrésének képességét is megőrzi a rendszer az ellenmondást tartalmazó útvonalak végrehajtásának mellőzésével.

Az eddigi eredmények biztatóak, a fejlesztés folytatódni fog. Elsőként a valós rendszeren kapott eredmények átvizsgálása és kiértékelése fog megtörténni, ami alapján meghatározzuk a további feladatokat. Tervek között szerepel az egész szimbolikus végrehajtó keretrendszer optimalizálása a feltétel rendszer építést is beleértve, valamint további olyan módszerek és technikák kidolgozása, amelyekkel a futásidejű hibatalálatok még pontosabban detektálhatók.

Irodalomjegyzék

- [1] Pressman S. Roger. Software Engineering: A Practitioner's Approach. McGraw-Hill Science/Engineering/Math. 2001.
- [2] Tassey, G. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology. 2002.
- [3] István Kádár, Péter Hegedűs, Rudolf Ferenc. Runtime Exception Detection in Java Programs Using Symbolic Execution. Proceedings of the 3th Symposium on Programming Languages and Software Tools, pages 215-229, 2013.
- [4] James C. King. Symbolic Execution and Program Testing. Commun. ACM, pages 385-394, 1976.
- [5] Gecode Tool-set, <http://www.gecode.org/>
- [6] Java PathFinder Tool-set, <http://babelfish.arc.nasa.gov/trac/jpf>.
- [7] 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.
- [8] 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.
- [9] Rudolf Ferenc and Árpád Beszédes and Mikko Tarkiainen and Tibor Gyimóthy. Columbus - Reverse Engineering Tool and Schema for C++. Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002), pages 172-181. 2002.
- [10] Frances E. Allen. Control Flow Analysis. SIGPLAN Not., pages 1-19, 1970.
- [11] 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.
- [12] Corina S. Păsăreanu, Peter C. Mehltz, 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.
- [13] Google C++ Testing Framework, <https://code.google.com/p/googletest/>
- [14] 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.

- [15] 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.
- [16] 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.
- [17] Coward, P. David. Symbolic execution systems - a review. Softw. Eng. J., pages 229-239, 1988.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] David Bushnell, Dimitra Giannakopoulou, Peter Mehrlitz, Russ Paielli, Corina Păsăreanu. Verification and Validation of Air Traffic Systems: Tactical Separation Assurance. Aerospace conference, IEEE, pages 1-10, 2009.
- [22] 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.
- [23] 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.
- [24] Christoph Csallner and Yannis Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. Softw. Pract. Exper., pages 1025-1050, 2004.
- [25] Khoo Yit Phang and Chang Bor-Yuh Evan and Jeffrey S. Foster. Mixing Type Checking and Symbolic Execution. PLDI, pages 436-447, 2010.
- [26] Daryl Shannon and Sukant Hajra and Alison Lee and Daiqian Zhan and Sarfraz Khurshid. Abstracting Symbolic Execution with String Analysis. Academic and Industrial Conference Practice and Research Techniques, page 13-22, 2007.

Nyilatkozat

Alulírott programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport Szoftverfejlesztés Tanszékén készítettem, programtervező informatikus Msc. diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy diplomamunkámat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

2014. május 13.

Kádár István

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőimnek Dr. Ferenc Rudolfnak a program és a dolgozat elkészítésében nyújtott sok segítségért. Ezen túl szeretném megköszönni Hegedűs Péter, Dr. Bakota Tibor és Dr. Jász Judit munkáját, kinek hasznos észrevételeit és tanácsait figyelembe vehettem a program készítése során. Végül de nem utolsósorban szeretnék köszönetet mondani Perényi Tamásnak, aki a SymbolicChecker fejlesztőjeként a feltételrendszer építés integrálásában segített nekem hozzájárulva a dolgozat létrejöttéhez.