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

# Szimbolikus végrehajtást használó tesztgeneráló eszközök egységes összehasonlítása

TDK DOLGOZAT

*Készítette*
Cseppentő Lajos

*Konzulens*
Dr. Micskei Zoltán Imre

2013. október 25.

# Kivonat

Napjaink szoftverei igen terjedelmesek és bonyolultak, amelyeknek a mélyreható manuális tesztelése sok erőforrást igényel. Épp ezért egyre nagyobb hangsúlyt kap az automatizált szoftvertesztelés témája ipari és akadémiai körökben is. Megjelentek olyan módszerek és eszközök, amelyek nem csak a tesztfolyamat automatizálására, hanem a tesztbemenetek és tesztesetek automatikus generálására is képesek. Aktív kutatás övezi a szimbolikus végrehajtást használó megoldásokat, amelyek a tesztelendő program struktúrája alapján választanak ki tesztbemeneteket.

Az elmúlt évtizedben több eszköz is készült, amely képes szimbolikus végrehajtáson alapuló algoritmusok segítségével tesztbemeneteket generálni. A tapasztalat viszont azt mutatja, hogy minden eszköznek megvannak a maga erősségei és gyengeségei, más-más bonyolultabb esettel (például karakterlánc vagy összetett objektum) nem tudnak megbirkózni. Az eszközök készítői különböző programokat használnak a saját esettanulmányaikban, a témát feldolgozó áttekintő munkák pedig komplex szoftverrendszereket használó kísérletekre koncentráltak eddig. Így jelenleg nincs pontos visszajelzés arról, hogy mi az a nyelvi szerkezet vagy program struktúra, amellyel egy-egy adott eszköz nem tud megbirkózni.

A dolgozatban bemutatott kutatás célja az volt, hogy egy nyelvfüggetlenül definiált szempontrendszert határozzak meg, amellyel a szimbolikus végrehajtást használó tesztgeneráló eszközök értékelhetők és összehasonlíthatók, majd ennek segítségével egy konkrét platform eszközeit össze is hasonlítsam. Korábban ilyen összehasonlítás nem készült, viszont ez egyaránt hasznos az eszközök fejlesztői és az eszközök felhasználói számára is: a fejlesztők visszajelzést kapnak az eszközről, a felhasználók pedig segítséget, hogy a legmegfelelőbb eszközt válasszák a saját problémájuk megoldására. Az elért eredményeim a következőképp foglalhatóak össze:

- Kidolgoztam egy szempontrendszert szimbolikus végrehajtást használó tesztgeneráló eszközök képességeinek vizsgálatára.
- Ezt leképeztem Java nyelvre, és előállítottam belőle egy futtatható tesztkészletet.
- A tesztkészlet segítségével megvizsgáltam több Java tesztgeneráló eszközt is, kiértékeltem és összehasonlítottam a kapott eredményeket.

# Abstract

Software systems are large and complex nowadays whose thorough manual testing requires a lot of resources. That is why automated software testing is getting significant attention in both industry and academia. Several methods and tools have been already published that do not only automate the testing process, but are also able to automatically generate test inputs and test cases. There is an active research in solutions using symbolic execution, which choose test inputs based on the structure of the software.

In the last decade several tools have been created, which are able to generate test inputs using algorithms based on symbolic execution. However, experience indicates that each tool has its strengths and weaknesses: they cannot cope with different complicated cases (for example string or complex object). The developers of the tools use different software in their case studies and the existing surveys concentrated on experiments with complex software systems. Thus, currently there is no accurate feedback available about the language elements and program structures that a given tool cannot cope with.

The purpose of the research presented in this paper was to define a language-independent system of criteria with which the test generation tools based on symbolic execution can be evaluated and compared to each other and then to compare the tools of one concrete platform. Previously no such comparison has been made, but this would be beneficial for both the tool developers and tool users. The developers receive feedback for their tool and the users receive guidance to choose the most suitable tool to solve their problem. The contributions of the paper can be summarized as follows:

- I have created a system of criteria to examine the abilities of test generation tools using symbolic execution.
- It has been implemented for Java and I have created a runnable test suite.
- With the test suite I have examined several Java test generation tools and I have evaluated and compared the results.

# Tartalomjegyzék

# 1. fejezet

# Bevezető és motiváció

A szoftvertesztelés az informatikának egy kiemeltem fontos részterülete. Egy szoftver fejlesztési költségének átlagosan a felét tesztelésre fordítják, ezért fontos, hogy a tesztelés hatékony és automatikus legyen.

Az elmúlt években több automatikus tesztbemenet-generáló eszköz is készült, ezek közül népszerűek a szimbolikus végrehajtás módszerét használó megoldások. A szimbolikus végrehajtás egy programanalízisre kidolgozott technika, amely során a programot nem konkrét bemeneti értékekkel, hanem szimbolikus változókkal futtatják, így a program futási útjai és a különböző futási utakat aktiváló bemeneti értékek felderíthetők. Azonban a módszer bonyolultsága és erőforrásigénye miatt az eszközök korántsem tökéletesek, ezért fontos a folyamatos vizsgálatuk és fejlesztésük.

## 1.1. Probléma kifejtése

A szimbolikus végrehajtás alapú tesztbemenet-generálás egy aktívan kutatott területe az informatikának. A szimbolikus végrehajtás ötlete az 1970–es években merült fel, azonban körülbelül csak egy évtizede jutott oda a technológia, hogy a módszert tesztgenerálásra is alkalmazni lehessen. Az elmúlt időszakban sok ilyen eszközt implementáltak többek közt C, .NET és Java platformokra, de létezik eszköz JavaScript-hez és x86-os gépi kódhoz is.

A módszer jelenleg több kihívással néz szembe, amelyeket meg kell oldani. A szimbolikus végrehajtás naiv implementációjának az erőforrásigénye óriási, így új módszereket kell kidolgozni a probléma enyhítésére. Az eszközök sokrétűsége miatt nem ritka, hogy egy eszköz egy adott esetet nem tud véges időn belül kiértékelni a rendelkezésre álló keretek között, míg egy másik igen, és fordítva. Ezért fontos az eszközök kiértékelése, összehasonlítása és esettanulmányokon való futtatása.

## 1.2. Korábbi eredmények

Amikor a fejlesztők publikálnak egy eszközt, ismertetik a saját futtatási eredményeiket. Ennek módszertana többféle lehet: vannak, akik saját példaprogramjaikon futtatják az eszközt, míg mások nyílt forráskódú szoftvereken végeznek esettanulmányokat. Azonban általában a felhasznált külső szoftverek is különbözők, így az egyes eszközök képességei

nehezen összemérhetőek. A publikációkban a szerzők mindig ismertetik az eszköz előnyeit és a fejlesztés során megvalósított innovációt, de kevesen térnek ki az eszközük korlátaira.

Emellett az évek során több áttekintő tanulmány is született, amelyekben főleg komplex szoftvereken, illetve szoftverrendszereken alkalmazták az eszközöket. Ezek a tanulmányok elsősorban kvantitatív jellegű eredményeket közölnek (például elért kódfedés) és arra mutatnak rá, hogy az eszközök éles helyzetben hogyan működnek és birkóznak meg a kihívásokkal. Azonban arról, hogy mely kódrészleteket tudja az adott eszköz kezelni és melyeket nem részletes visszajelzést nem adnak.

## 1.3. Célok

Amikor elkezdtünk a témával foglalkozni, több tesztbemenet-generáló eszközt is kipróbáltunk, és azt tapasztaltuk, hogy az eszközök a hozzájuk mellékelt példakódokra jól működnek, azonban több eszköz a saját kódunk bizonyos részeire hibát dobott. Volt, amikor számára ismeretlen vagy általa rosszul kezelt nyelvi elemmel találkozott, és volt, amikor a szimbolikus végrehajtás kihívásaiba ütközött bele.

Mivel korábban nem készült részletes visszajelzés az eszközökről, ezért a kutatás célja egy általános módszertan kidolgozása volt, amelynek a segítségével a tesztbemenet-generáló eszközök egységesen összehasonlíthatók.

A módszer azon alapszik, hogy összegyűjtjük egy szempontrendszerbe a tesztbemenet-generáló eszközök által kezelendő funkciókat, majd minden szemponthoz azt jellemző kódrészlet(ek)et határozunk meg. Ezek a kódrészletek szolgálnak bemenetként az eszközök számára. Ezután utasítjuk az eszközöket, hogy generáljanak tesztbemeneteket ezekhez a kódrészletekhez. A generálás eredményei alapján lehet következtetni arra, hogy az adott eszköz megfelelően kezeli-e a kódrészlet által reprezentált esetet. Az összes kódrészlet használatával részletes visszajelzést kapunk egy eszköz képességeiről, így tulajdonképpen ezekkel a kódrészletekkel tesztelhetők a tesztgeneráló eszközök.

## 1.4. Eredmények

A TDK munka keretében a következő eredményeket értem el:

- Kidolgoztam egy, a fenti módszernek megfelelő általános szempontrendszert, amellyel a tesztbemenet-generáló eszközök képességei megvizsgálhatók és összehasonlíthatók.
- A szempontrendszerben definiált célok alapján elkészítettem egy tesztkészletet a Java platformra, valamint egy keretrendszert, amellyel a tesztfuttatás automatizálható.
- Az elkészült tesztkészletet lefuttattam három szimbolikus végrehajtással működő, valamint egy forráskód-alapú eszközre. A kapott eredményeket kiértékelve összehasonlítottam az eszközök képességeit.

Ezek az első kísérleti eredmények alapján is sikerült (i) az eszközök képességei közötti jelentős eltéréseket felderíteni, és (ii) azonosítani olyan funkciókat, amelyekkel egyik eszköz sem birkózik meg.

# 2. fejezet

# Háttérismeretek

## 2.1. A szimbolikus végrehajtás

A szimbolikus végrehajtás (symbolic execution, SE) egy szoftver olyan futtatási módja, amikor a bemeneteket nem konkrét értékekként, hanem szimbolikus változókként kezeljük. Így lehetőségünk nyílik arra, hogy analizáljuk, hogy adott bemenetre milyen eredményt ad a program. A szimbolikus végrehajtáshoz a lefordított kódot instrumentálni kell és a programot úgy kell futtatni, hogy összegyűjtjük az elágazások feltételeit (pl. az egyik paraméternek pozitívnak kell lennie, hogy az elágazás egyik ágát válasszuk). Az elágazási feltételeket rendszerint SMT[1] problémaként definiálják, így azok SMT megoldóval kielégíthetők és meghatározható a szimbolikus változó olyan értéke, amely használatával a feltétel igaz vagy hamis lesz. Így a program futását egy általunk meghatározott útra tudjuk irányítani. [7]

A szimbolikus végrehajtást gyakran használják automatikus tesztbemenet- és tesztgenerálásra (white-box testing). A program szimbolikus végrehajtásával előálló elágazási feltételek alapján meghatározható a bemenetek halmaza úgy, hogy a futtatások az összes lehetséges futási utat lefedjék. Ezt a halmazt általában érdemes minimalizálni. Az így előállt bemenetek segítségével automatikusan előállítható egy teljes fedettséget garantáló tesztkészlet.

## 2.2. Kihívások

A kihívásokat Chen és társai a „State of the art: Dynamic symbolic execution for automated test generation" [7] cikkben áttekinthetően összefoglalták, ezért ezeket itt csak tömören ismertetem.

### 2.2.1. Futási utak exponenciális növekedése

A szimbolikus végrehajtás egyik talán legnagyobb kihívása az, hogy ahogy nő a program mérete, a futási utak száma exponenciálisan nő (angolul *path explosion*-nek nevezik a jelenséget). Emellett az SMT probléma NP-teljes, így a problémára még nehezebben adható.

[1] Satisfiability Modulo Theories

Így ahhoz, hogy nagyobb szoftvereket is tudjunk egészében szimbolikus végrehajtás segítségével ellenőrizni, rengeteg erőforrásra van szükségünk. Amikor a területet az 1970-es években elkezdték kutatni, az erőforrások még szűkösek voltak, így ez igen jelentős korlátot szabott a SE alkalmazásának. Azonban azóta egy átlag számítógép erőforrásai jelentősen megnőttek. Emellett egyéb módszereket is kitaláltak a probléma kezelésére.

Több *intelligens keresési stratégiát* is alkalmaztak korábban. A mélységi keresés használatával kevesebb memória kell, viszont „beragadhat", így a problémát nem sikerült tökéletesen megoldani. A korlátozott mélységi keresés segítségével ezt kiküszöbölhetjük, de különösen fontos a mélységkorlát megfelelő kiválasztása. Manapság a 2008-ban megalkotott generációs keresés terjedt el. Emellett az eszközök gyakran használnak egyéb keresési technikákat, például lehet véletlenszerű vagy körökre osztott (round-robin). A probléma így kezelhetőbb, viszont még mindig jelentős ipari szoftverek esetén.

Másik megoldás az *összefoglaló (summary)*, amivel enyhíthetjük a problémát. Ennek a módszernek az a lényege, hogy a bonyolult függvényeket egyszerűbbekkel helyettesítjük, így a függvény futási útjainak a számát korlátozzuk (például rekurzív függvény esetén mélység korlátozása).

Az előzőek mellett *csökkenthetjük a keresési teret*, azaz a bemenetek lehetséges halmazát, ha például korlátozzuk a számok lehetséges értékeit vagy a tömbök maximális méretét. Azonban ilyenkor ügyelni kell arra, hogy ha külön futtatunk egy eszközt pozitív és negatív számokra, a futási eredmények uniója nem azonos azzal, ha egyszer futtattuk volna az eszközt az összes értéket figyelembe véve.

### 2.2.2. Komplex aritmetikai kifejezések és külső függvények

A szimbolikus végrehajtásnak korlátot szab az, hogy sokszor az SMT megoldók nem tudnak nemlineáris kifejezésekkel, hatványozással vagy hash-függvények kielégítésével megbirkózni. Továbbá az is probléma, hogy sokszor a külső függvények forrása nem áll rendelkezésre, így azokat instrumentálni nem tudjuk. Az egyik megoldás, hogy feketedobozként kezeljük ezeket a függvényeket. Másik megoldás az, hogy a nem instrumentálható részeket a szimbolikus végrehajtás idejére saját implementációval helyettesítjük, de ez például egy hash függvény implementációja esetén a futási utak exponenciális növekedéséhez vezet.

### 2.2.3. Lebegőpontos számítások

A programok gyakran használnak lebegőpontos számításokat, azonban ezek nem teljesen pontosak. Emiatt viszont egy SE használatával téves hibákat kaphatunk, mivel nem tudja tökéletesen kielégíteni a feltételeket. Ha ennek kezelésére felkészítjük az eszközt, akkor jóval bonyolultabb feltételeket kapunk, emellett a futási utak száma is jelentősen megnő. Azonban a teljes támogatáshoz az eszköz által támogatott SMT megoldónak is tudnia kell kezelnie a lebegőpontos számokat.

2010-ben Godefroid és társa publikált egy alternatív módszert [9], amellyel garantálható a memóriabiztonság lebegőpontos számítások során, és ehhez nincs szükség pontos szimbolikus kiértékelésre. Ez a módszer lokálisan értékeli ki az utakat és felülbecsli a lebegőpontos

műveleteket. Hatékonynak bizonyult, azonban a felülbecslés miatt elhanyagolhatatlan lesz a fals pozitívok száma.

### 2.2.4. Mutató műveletek

A mutatókkal végzett műveletek külön problémához vezetnek, hiszen megeshet, hogy egy mutató változása függ a bemenettől. Előfordulhat, hogy így egy mutató érvénytelen helyre mutat a szimbolikus kiértékelés során, ami többek közt fals negatívhoz vezethet, esetleg az operációs rendszer leállíthatja az egész folyamatot. Elkarablieh és társai 2009-ben publikáltak egy új memóriamodellt [8], ami a memóriaterület pillanatképével dolgozik. Ez segít megoldani a problémát, azonban tovább növeli a futási utak számát, ráadásul jóval több memória szükséges a használatához.

### 2.2.5. Interakció a környezettel

A szoftverek általában függnek a környezettől, például használhatnak I/O és adatbázis műveleteket, futásuk eredménye függhet a felhasználó által megadott bemenettől, az időtől vagy akár az operációs rendszer aktuális eseményeitől is. Azonban szimbolikus végrehajtáskor ezen tényezők nehezen vagy egyáltalán nem befolyásolhatók. Az elmúlt években több módszert is javasoltak a probléma kiküszöbölésére, az adatbázis-kezelést már sikerült megoldani, valamint a felhasználói interakciókat (bemenet, kattintás stb.) is sikerült izolálni. A teljes operációs rendszer modellezése és izolációja rengeteg munkát igényel, de erre példa a KLEE nevű eszköz, ami az LLVM infrastruktúrára épül.

### 2.2.6. Többszálú alkalmazások

Habár Chen nem említi, de többszálú programok esetén problémát okoz az, hogy nem tudjuk pontosan szabályozni, mikor melyik szál fusson. Itt kénytelenek vagyunk más módszereket használni a szálbiztosság ellenőrzésére vagy kiterjeszteni a szimbolikus végrehajtást. A problémára megoldás az *általánosított szimbolikus végrehajtás (General Symbolic Execution)*, mellyel nem csak tesztbemenetet, hanem szálütemezést is előállít az eszköz. Erre a CUTE és jCUTE eszközök képesek [25].

## 2.3. Rendelkezésre álló eszközök

Az elmúlt évtizedek során sok szimbolikus végrehajtással működő tesztgeneráló eszközt alkottak meg. Az eszközöket a bemenetük szerint csoportosítva adom meg. Az eszközök főbb jellemzőit a 2.1 táblázat foglalja össze.

### 2.3.1. C-s eszközök

Az első eszközök egyike a DART [10] volt, ami részben szimbolikus végrehajtást is használt, hogy általánosítsa a konkrét végrehajtásokat. A CUTE [25] is konkolikus elven (concolic – konkrét és szimbolikus végrehajtással működő) eszköz, ami kerülte a redundáns teszteseteket és a téves hibajelzéseket. Az EXE [5] is konkolikus elven működik. Az EXE tovább-

fejlesztése a KLEE [4], ami már az LLVM-re épül [14]. Talán a KLEE volt az első eszköz, ami 90% feletti fedettséget produkált egy gyakran használt nyílt forráskódú szoftveren és több, mint 450 alkalmazást teszteltek vele a publikáció megszületéséig. A CREST-ben [3] egy új keresési heurisztikát fejlesztettek ki, mely hatékonyabbnak minősült a korábban alkalmazott mélységi keresésnél.

### 2.3.2. .NET-es eszközök

A Microsoft Research RiSE csoportja által fejlesztett Pex [26] egy kiválóan működő .NET-es eszköz. A Pex IL kódon (.NET bájtkód) dolgozik, így a C#, Visual Basic és F# nyelvekre is működik. Az eszközhöz készítettek egy online elérhető futtatót, a *Pex for fun* [15] oldalt. A teljes verzió akadémiai célokra elérhető és a Visual Studio-n belül használható.

### 2.3.3. Java-s eszközök

Az elmúlt évek során több szimbolikus végrehajtást használó eszköz készült a Java platformra. Ilyen többek közt a jCUTE [25], ami a CUTE Java-ra szánt változata. A PET/jPET [1] is a szimbolikus végrehajtás elvén működik, azonban az eszköz a Java bájtkódot először Prolog nyelvű CLP-vé (Constraint Logic Program) fordítja át, majd ennek a segítségével állít elő tesztbemeneteket. A Palus [27] egy hibrid elven működő eszköz: a véletlenszerű tesztgenerálást ötvözi a szimbolikus végrehajtással (a Randoop [20] eszközre építve), így képes metódushívások sorozatát is előállítani tesztesetként. A CATG [24] egy konkolikus elven működő tesztbemenet-generáló eszköz, akárcsak az LCT [12]. A NASA által fejlesztett Java PathFinder (JPF) eszközhöz készítettek egy Symbolic PathFinder (SPF) [22] kiterjesztést, ami a CATG-hez hasonlóan tesztbemeneteket generál, azonban a futtatáshoz magát a JPF-t használja.

Érdemes megemlíteni a CodePro AnalytiX eszközt [11], ami nem használ szimbolikus végrehajtást, de képes forráskód alapján automatikus tesztgenerálásra. A szoftvert a Google megvásárolta az Instantiations cégtől, majd azt elérhetővé tette.

**2.1. táblázat.** *A bemutatott tesztgeneráló eszközök összehasonlítása*

| Név | Platform | Hozzáférés | Megj. éve | Frissítve | Bemenet | Kimenet | Függőségek |
|---|---|---|---|---|---|---|---|
| CATG | Java | Nyílt forráskód | 2012 | 2013 | Bájtkód | Tesztbemenet | - |
| CodePro AnalytiX | Java | Zárt forráskód | 2001 | 2010 | Forráskód | JUnit tesztesetek | Eclipse, JUnit 4 |
| CREST | C | Nyílt forráskód | 2008 | 2012 | Forráskód | Tesztesetek | - |
| CUTE & jCUTE | C & Java | Nem elérhető | 2005 | 2008 | Forráskód/bájtkód | Tesztbemenet | - |
| DART | C | Nem elérhető | 2005 | 2005 | Forráskód | ? | - |
| EXE | C | Nem elérhető | 2006 | 2006 | Forráskód | ? | - |
| LCT | Java | Nyílt forráskód | 2010 | 2012 | Bbájtkód | Tesztbemenet | - |
| KLEE | C | Nyílt forráskód | 2008 | 2013 | LLVM bájtkód | Tesztbemenet | LLVM |
| Palus | Java | Nyílt forráskód | 2010 | 2011 | Bájtkód | Tesztesetek | Randoop |
| PET/jPET | Java | Nylt forráskód | 2009 | 2011 | Forráskód | Tesztesetek | Eclipse |
| PEX | .NET | Zárt forráskód | 2008 | 2013 | Bájtkód | Tesztesetek | Visual Studio |
| SPF | Java | Nyílt forráskód | 2012 | 2013 | Bájtkód | Tesztbemenet | Java PathFinder |

## 2.4. Korábbi összehasonlítások

Az elmúlt években több összehasonlító tanulmány is készült:

- Pasareanu és társa [21] a szakterületen elért korábbi kutatási eredményeket és aktuális problémákat foglalta össze. A tanulmányban részletesen kifejtették a szimbolikus végrehajtás alapelvét és több olyan nyelvi elemet gyűjtöttek össze, amelyekre a szimbolikus végrehajtás nem triviális (például tömb, karakterlánc). Külön kitértek az akkor ismert eszközökre, valamint az SMT megoldók közötti különbségekre is. Emellett a skálázhatóságot és a szimbolikus végrehajtás alkalmazásait is elemezték.

- Cadar és társai [6] több ismert és jól működő eszközt gyűjtöttek össze, és azoknak a képességeit és a tudományterületre gyakorolt hatásait összegezték. A bemutatott eszközök között szerepelt a CUTE/jCUTE, a KLEE, a Pex és az SPF is.

- Anand és társai készítettek egy átfogó tanulmányt [2], amelyben több tesztelési módszert részletesen elemeztek. Ezek közül egy a szimbolikus végrehajtás volt. A cikkben nemcsak a kihívásokat foglalták össze, hanem azoknak az alkalmazott és kutatott módszereit is részletezték. Több eszközt is összegyűjtöttek, azonban azokat nem hasonlították össze, inkább a kihívások kezelési módszereire koncentráltak.

A fenti tanulmányokon kívül két kísérleti adatokat tartalmazó cikket publikáltak:

- Lakhotia és társai [13] egy konkolikus (CUTE) és egy keresésalapú (AUSTIN) eszközt hasonlítottak össze. A két eszközt öt szoftverre futtatták le (`libogg`, `plot2d`, `time`, `vi` és `zile`), és kiértékelték futási eredményeket. Emellett részletesen összehasonlították a két eszköz közötti különbségeket, valamint részletezték az eszközök előnyeit és hátrányait is.

- Qu és társa [23] szimbolikus végrehajtás elvén működő tesztgeneráló eszközöket hasonlított össze (közöttük szerepelt a CUTE/jCUTE, a DART, a KLEE és a Pex is). A konkrét vizsgálatot a KLEE és a CREST eszközökre végezték el a `gcc` és a `linux` kernel forrásának, illetve az `ABB` cég egy saját komponensének a segítségével. Aggregált eredmények segítségével mutatták be a konlikus eszközök korlátait.

## 2.5. Összegzés

A fentiek alapján egyértelműen látszik, hogy a szimbolikus végrehajtással működő tesztbemenet-generálás aktívan kutatott területe az informatikának. Láthattuk, hogy a terület jelenleg több kihívással is küzd. Az eszközök sokrétűek mind a platformok, mind a megvalósítási módszerek tekintetében, azonban korábban még nem készült az eszközök képességeire koncentráló *egységes* összehasonlítás. Ez szolgált motivációként az általam végzett kutatáshoz.

# 3. fejezet

# A szempontrendszer tervezése

Ahhoz, hogy a különböző eszközöket könnyen össze tudjuk hasonlítani, fontos egy kiindulási alap és egy összehasonlítási módszer meghatározása. Egy általános *szempontrendszer* elkészítéséhez először az irányelveket kell meghatározni. A programozási nyelvek jelentősen eltérhetnek egymástól, gondoljunk csak a C és a C# közötti különbségekre. Ezért egy általános szempontrendszer mindenképpen fog olyan elemeket tartalmazni, amelyek egy bizonyos programozási nyelvben léteznek, egy másikban pedig nem. A szempontrendszer tervezése során elsősorban az imperatív programozási nyelvekre koncentráltam, mivel a korábban felsorolt eszközök ilyen nyelvekre készültek.

A szempontrendszer elemei mögött mindig egy cél van, mégpedig az, hogy kiderítsük egy adott funkció eszköz általi támogatottságát. A szempontok alapján példakódokat kell implementálni, amelyeket a vizsgált eszközökkel kell futtatni. Az előállt futási eredmények alapján valós összehasonlítást kapunk akár az egy eszköz által támogatott és nem támogatott funkciókról, akár az eszközök közti különbségekről.

## 3.1. Irányelvek

A szempontrendszer tervezése során a következő irányelveket tartottam szem előtt:

- *Törekvés a teljességre*[1]*:* a szempontrendszerrel igyekezzünk lefedni minél több nyelvi elemet és programszervezési struktúrát. A cél az, hogy alapszintű, de részletes visszajelzést nyújtsunk az eszközökről.

- *Egyértelműség:* a szempontrendszer legyen egyértelmű az összes programozási nyelvre nézve. Készüljünk fel arra, hogy egy fogalom két különböző nyelvben mást jelenthet (pl.: C++ referencia és Java referencia közötti eltérés). A szempontrendszerben használt fogalmakat a szempontrendszerrel együtt kell definiálni.

- *Strukturáltság:* törekedni kell arra, hogy a szempontrendszer jól strukturált legyen. Ez nem csak az áttekinthetőség miatt fontos, hanem a struktúrát követve ugyanúgy lesz strukturálva az összes későbbi munkafázis eredménye (implementáció, futtatási

[1] Fontos megjegyezni azonban, hogy a nagy bemeneti halmaz és a lehetőségek nagy száma miatt nem lehet reális cél a kimerítő tesztelés.

eredmények, kiértékelés). Azt se felejtsük el, hogy ha a szempontrendszer egy pontja módosításra kerül, akkor így sokkal könnyebb lesz a módosításokat a későbbi fázisokban elvégezni. A jól strukturáltságot faszerű rendszerezéssel érhetjük el.

- *Tömörség:* fontos, hogy a tesztkészlet ne legyen feleslegesen nagy, hiszen így a karbantartása, futtatása és az eredmények kiértékelése is több erőforrást vesz igénybe.

- *Függőségek minimalizálása:* a szempontrendszer elemei között kényszerűen lesznek függőségek. Például ahhoz, hogy egy elágazást használjunk elengedhetetlen követelmény, hogy a vizsgált eszköz az elágazásban használt típust tudja kezelni. Ügyelni kell arra, hogy különböző szempontok között csak egy irányban legyen függőség, valamint körkörös függőség se legyen. Emellett a függőségek számát érdemes alacsonyan tartani.

## 3.2. Nyelvi struktúrák összegyűjtése és kategorizálása

Ahhoz, hogy minél jobb szempontrendszert tervezzünk, fontos, hogy összegyűjtsük azokat a nyelvi és programszervezési struktúrákat, amelyek egy imperatív programozási nyelvben előfordulhatnak. A nyelvi elemeket és lehetőségeket a C++ [18], C# [16] és Java [19] hivatalos dokumentációiból gyűjtöttem össze, valamint saját programozói tapasztalatomra hagyatkoztam.

### 3.2.1. Alapvető nyelvi elemek

Egy programozási nyelv alapkövei általában a következő elemek:

- *Változók:* egy változó egy adott típusú értéket tárol. A tárolt érték a program futása során bármennyiszer kiolvasható, illetve módosítható.

- *Konstansok:* egy konstans egy adott típusú értéket tárol. A tárolt érték a program futása során bármennyiszer kiolvasható, de nem módosítható.[2]

- *Primitív (elemi) adattípusok:* azon adattípusok, melyek további részre nem bonthatók. Például: egész szám, lebegőpontos szám, logikai típus, karakter, mutató. Fontos megemlíteni, hogy a primitív adattípusok nyelvenként eltérőek lehetnek: C-ben nincs logikai típus, Java-ban közvetlenül nincs mutató.[3]

- *Tömbök:* a tömb egy olyan adattípus, ami több másik, ugyanolyan adattípust tartalmaz. A tömb elemeinek a száma a tömb életének során változatlan.[4]

[2]Modern programozási nyelvek lehetővé teszik, hogy ne a konstans létrehozásakor, hanem később adjunk neki értéket, azonban a konstans létrejötte után a konstans értékét kötelező beállítani.

[3]Már ezen látszik, hogy az implementáció két különböző nyelv esetén gyökeresen eltérhet.

[4]Fontos megemlíteni, hogy bizonyos programozási nyelvekben (pl.: Java és C#) a tömb objektum, sőt, bizonyos programozási nyelvekben a mérete is változhat. Ha a tömb objektum, figyelni kell arra, hogy ha egy eszköz nem tud objektumot kezelni, nem fog tudni tömböt sem kezelni, hacsak erre a speciális esetre külön fel van készítve.

- *Strukturák:* a struktúra egy olyan adattípus, ami több (nem feltétlenül különböző) adattípusból épül fel. Egy struktúra építőeleme bármilyen típus lehet, akár struktúra is.[5]

- *Alapműveletek:* gyakorlatilag az operátorok, melyek az operandusokon műveletet végeznek. Nyelvenként ez is eltérő lehet.

A nyelvi elemeket önmagában nem lehet szimbolikusan tesztelni, mivel ehhez egy futtatható programra van szükség.

### 3.2.2. Programszervezési szerkezetek

Egy program során a számítógép utasításokat hajt végre. Az utasítások megadására többféle szervezési mód és vezérlési szerkezet áll rendelkezésre, hogy az igényeket kielégítsük:

- *Utasítások sorozata:* a legegyszerűbb programszervezési módszer az utasítások sorozata, amikor a számítógép az utasításokat egymás után, sorrendben hajtja végre.

- *Feltételes elágazások:* a számítógép egy feltétel (igazság)értéke alapján eltérő helyen folytatja a program futtatását. Gyakori típusai: `if-else`, `if-elseif-else`, `switch`, `goto`.

- *Ciklusok:* a számítógép egy feltétel igazságértéke alapján vagy a ciklusmagot futtatja, vagy a ciklusmag utáni programrészt. Két főbb fajtája van: elől- és hátultesztelő ciklus. Gyakori típusai: `while`, `for`, `do-while`. A ciklusmagban általában lehetőség van arra, hogy a ciklusmag végére lépjünk (`continue`) vagy hogy az egész ciklusból kilépjünk (`break`). Modern programozási nyelvek azt is lehetővé teszik, hogy egymásba ágyazott ciklusok esetén meghatározzuk, hogy a belső ciklusból külső ciklusmagra vonatkozóan adjunk ki ugrási utasítást (pl.: Java címkézés).

- *Függvények:* a függvényeknek tetszőleges számú bemeneti paramétert adhatunk, de maximum egy visszatérési értékük lehet. Egy függvényt akármennyiszer meghívhatunk.[6]

- *Kivételek:* a kivételek segítségével az esetek túlnyomó részében sokkal könnyebben kezelhetjük a speciális eseteket egy programban, mint a visszatérési értékek (hibakódok) ellenőrzésével. Ügyelni kell arra, hogy a kivételek használatának módja nyelvenként elég eltérő lehet, például:

  - *C++*: bármilyen típus lehet kivétel, de nincs `finally` blokk.

[5] Olyan nyelv esetén, amely közvetlenül nem támogatja a struktúrákat, de támogatja az objektumokat, a struktúra olyan objektumnak feleltethető meg, ami nem rendelkezik metódussal, valamint minden adattagja publikus. Fontos megemlíteni, hogy C# esetén a `struct` és a `class` is struktúra, ha ezeknek a feltételeknek megfelel (itt a különbség abban rejlik, hogy érték szerinti vagy cím szerinti a változó kezelése).

[6] Ezen konvenció szerint az objektumorientált nyelvek osztályainak a statikus metódusai is függvényeknek minősülnek.

– *Java*: minden kivétel objektum, adott közös őstől kell származniuk. A kivételek dobására fel kell készülni, kivéve azokat, melyek a `RuntimeException`-től származnak.
– *C#*: minden kivétel objektum, adott közös őstől kell származniuk. A kivételek dobására nem kötelező felkészülni.

Fontos megemlíteni, hogy a fenti platformok esetén ha egy kivétel a program belépési függvényén kívül jut, akkor az a szál, amin a kivétel érkezik, leállításra kerül.

A fent ismertetett szerkezetek segítségével építhetjük fel a programunkat. Fontos megjegyezni, hogy vannak olyan gyakori „származtatott" programszervezési szerkezetek, melyekre érdemes külön figyelmet fordítani. Ilyen például a végtelen ciklus vagy a rekurzív függvény (utóbbi is lehet végtelen, csak ilyenkor a hívási verem szokott megtelni).

### 3.2.3. Objektumok és kapcsolataik

Mivel az objektumorientált nyelvek manapság már eléggé elterjedtek, nem lehet figyelmen kívül hagyni az objektumokat. A korábbiakat kiegészítve az objektum egy olyan típus, amely egyéb típusokat és metódusokat tartalmazhat. A *metódusok* hasonlóak a függvényekhez: lehet több bemeneti paraméterük és maximum egy visszatérési értékük, valamint az adott példány által tartalmazott adattagokat tudják kezelni. Emellett a metódusok az objektum belső állapotát is megváltoztathatják. A metódusok és tagváltozók láthatósággal is rendelkeznek. A láthatóság lehetséges értékei általában nyelvenként eltérőek.

Fontos még megemlíteni az *interfész* fogalmát, amely csak függvénydefiníciókat, esetleg konstansokat tartalmazhat, de implementációt nem[7]. Az *absztrakt osztály* olyan osztály, aminek van legalább egy olyan metódusa, melyhez nem tartozik implementáció.

Egységteszteknél az objektum függőségeket érdemes izolálni. Egy függvény vagy metódus is használhat objektumot, valamint az objektumok más objektumokat is tartalmazhatnak. Ezt szem előtt kell tartani a szempontrendszer tervezésekor. Emellett érdekes lehet, hogy ha nem egy példányosítható objektumot, hanem interfészt vagy absztrakt osztályt kap egy függvény vagy metódus paraméterként, azzal megbirkózik-e a vizsgált eszköz.

Míg a struktúrák között csak tartalmazási kapcsolat lehetett, az objektumok esetén a tartalmazás mellett öröklés vagy interfész-implementáció is előfordulhat. Örökléskor külön oda kell figyelni a metódusok túlterhelésére. Java-ban minden metódus virtuális (polimorfizmus), de C++-ban ezt explicit kell megadni. C#-ban felüldefiniáláskor két kulcsszót is használhatunk (`override` és `new`). C++/C#-ban így sokszor nem mindegy, hogy mi a statikus („fordításkori") és a dinamikus („futtatáskori") típus. Itt érdekes lehet, hogy a szimbolikus végrehajtást használó eszközök a megfelelő metódust használják-e.

[7] C++-ban akkor beszélünk interfészről, amikor csak „pure virtual", azaz implementáció nélküli virtuális metódus van az osztályban.

### 3.2.4. Generikus függvények és adatszerkezetek

A modern objektumorientált nyelvek egy része valamilyen módon támogatja a generikus adatszerkezeteket. Ezt az adott nyelv programozói gyakran használják, hiszen könnyíti a munkát és sok típuskonverziós hibalehetőségtől véd minket. Azonban ennek az implementációjában a C++, C# és Java nyelvek is eléggé eltérnek:

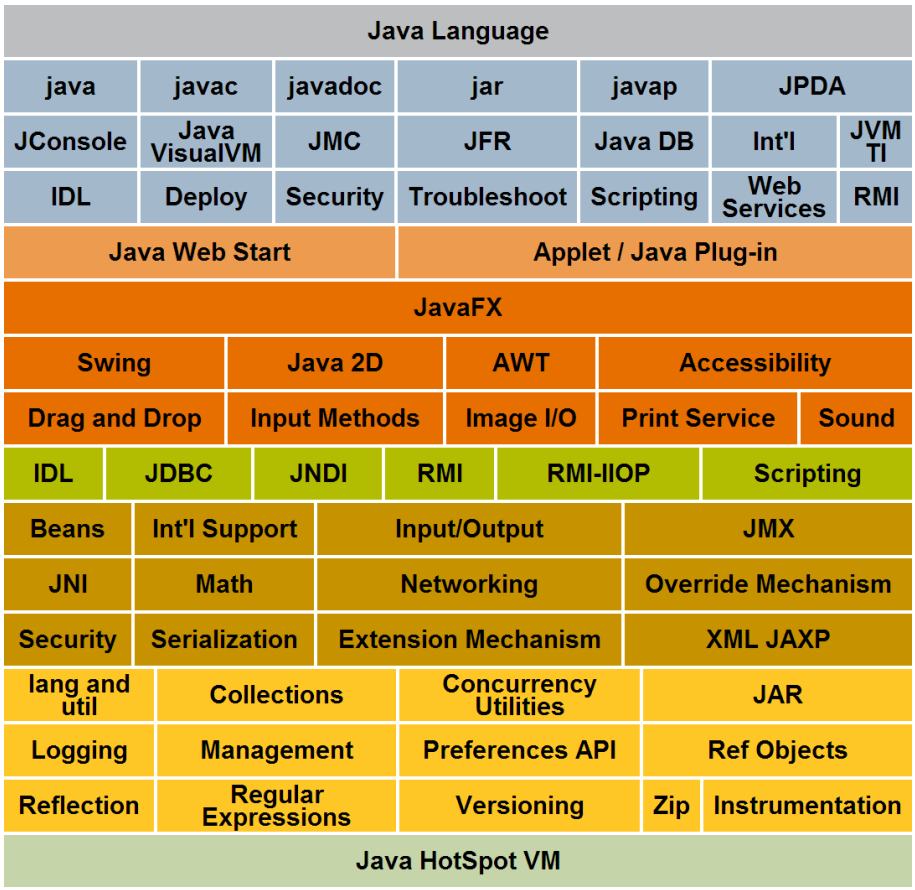
- *C++:* sablonok létrehozására van lehetőség, fordítási időben a sablonok kiértékelődnek. A sablonok a forráskód nélkül korábban nem használt típussal nem használhatók.
- *C#:* fordítási időben jönnek létre a valós osztályok, újrahasznosítható.
- *Java:* futási időben már nem áll rendelkezésre a generikus paraméter típusa, ugyanis a bájtkódban már csak ősobjektumként jelenik meg, a fordító pedig típuskonverziókat helyezett el a bájtkódban.

A generikus osztályoknál a generikus típusparaméterekre is használhatunk megkötéseket, valamint generikus osztálytól lehet úgy örökölni, hogy a típusparamétert lekötjük.

### 3.2.5. Nyelvhez tartozó osztálykönyvtár használata

Több – főleg újabb – programozási nyelvhez beépített osztálykönyvtár tartozik. Egy nyelvhez azért készítenek függvénykönyvtárat, hogy a gyakori feladatokat ne kelljen mindig újraimplementálni, hanem azok már a nyelvvel együtt rendelkezésre álljanak. Így egy tesztbemenet-generáló eszköztől reális elvárás, hogy azokat a programrészeket is kezelje, melyek a nyelvhez tartozó osztálykönyvtárat használják. A beépített függvénykönyvtárakban általában a következők találhatók meg:

- *Matematikai könyvtár*
- *Karakterlánc és az azt kezelő osztályok és függvények*
- *Csomagoló osztályok*
- *Kollekciók*
- *Algoritmusok*
- *I/O kezelést segítő osztályok és függvények*
- *Naplózás*
- *Hálózati kapcsolatok kezelése*
- *Többnyelvűsítés, lokalizáció (`i18n` & `l10n`)*
- *2D és 3D megjelenítés*
- *Grafikus felület*
- *Adatbázis-kezelés*
- *Stb.*




**3.1. ábra.** *A Java API teljes lefedése már önmagában is kihívás [19]*

### 3.2.6. Egyéb

Számos egyéb, eddig nem kategorizált nyelvi elem áll még rendelkezésünkre, a teljesség igénye nélkül felsorolok párat a Java, illetve a C# nyelvekből:

- *C# tulajdonság, delegált, esemény,* `var`*,* `Linq`*:* fordításkor kiértékelődik, az IL kódban már nem lesz önálló nyelvi elem.
- *Változó számú paraméterek:* a bájtkódban már tömbként jelenik meg.
- *Enumeráció*: gyakorlatilag egy egész számként kerül leképezésre.
- *Névtelen és beágyazott osztályok:* a bájtkódban ezek külön jelennek meg, mint a többi osztály, csak az elnevezésük más.
- *Harmadik fél által készített függvénykönyvtár használata:*
- *Többszálúsítás:* itt külön figyelni kell a szálak közötti kommunikációra és a zárakra.
- *C#* `yield`*, C# aszinkron metódushívás*

## 3.3. A tesztkészlet magas szintű leírása

A leírás során a legfelső csoportosítási egység a *modul*. Egy modul általában az összes előző modulra épít, de kivételek persze lehetnek (pl. a tömb támogatása nélkül még lehet, hogy tesztelhetünk tömbmentes objektumokat). A modulok definiálása során lefedtem az összes korábban felsorolt nyelvi struktúrát. Értelemszerűen amikor egy adott nyelvre implementáljuk a tesztkészletet, a nyelvben nem megtalálható elemekre meghatározott eseteket ki kell hagyni.

### 3.3.1. Alapműveletek és alaptípusok

Egy eszköz vizsgálatakor érdemes legelőször azt megtudni, hogy mely primitív típusokat, operátorokat és alapvető nyelvi elemeket támogat. Ehhez viszonylag sok rövid tesztesetre van szükség, azonban így nagyon részletes visszajelzést kapunk.

**B1** *Alaptípusokkal műveletet végző függvények:* az összes nyelvben megtalálható alaptípust érinteni kell, valamint használni kell az összes nyelvben megtalálható alapműveletet (operátort), beleértve a bitműveleteket is. Külön meg kell nézni az egy- ill. többparaméteres függvényeket, valamint a konstansokat.
*Cél:* támogatott típusok és műveletek felderítése.

**B2** *Alaptípusokkal dolgozó, elágazást tartalmazó függvények:* az összes feltételes elágazási szerkezetet tesztelni kell, valamint fel kell mérni, hogy az eszköz milyen bonyolultságú feltételeket tud kielégíteni (célszerűen lineáris és nemlineáris feltételek)[8].
*Cél:* elágazások támogatottságának és a kényszerkielégítési korlátoknak a felderítése.

**B3** *Alaptípusokkal dolgozó, ciklust tartalmazó függvények:* az összes ciklust megvalósító szerkezetet tesztelni kell, valamint a ciklus belsejéből kilépő utasításokat is. Ezek mellett teszteljük az is, ha olyan kódrész van a ciklusban, ami sose fut le, valamint a végtelen ciklust is.
*Cél:* ciklusok támogatottságának a felderítése, path explosion-re és végtelen ciklusra adott reakció vizsgálata.

**B4** *Alaptípusok tömbjeivel műveletet végző függvények:* tesztelni kell, hogy a tömböket tudja-e kezelni az eszköz. Fontos, hogy legyenek olyan feltételek, amelyek a tömb elemeitől, illetve hosszától függenek, ugyanis így fogunk valós visszajelzést kapni.
*Cél:* tömbök támogatottságának felderítése.

**B5** *Függvényt hívó függvények, rekurzív függvények:* fontos ellenőrizni, hogy az eszköz függvényhíváskor a hívott függvényt is szimbolikusan hajtja-e végre, valamint azt, hogy hogyan reagál a rekurzív függvényekre.
*Cél:* függvényhívási korlátok felderítése.

[8] Az eredmények kiértékelésekor figyelembe kell venni, hogy milyen SMT megoldót használ az eszköz, ugyanis ha egy feltételt nem tud kielégíteni, az az eszköz vagy az SMT megoldó korlátja is lehet.

**B6** *Kivételek:* ha többféle kivételtípus van, mindet le kell tesztelni, valamint külön kell tesztelni a mindig és a csak feltételesen kivételt dobó függvényt. Le kell tesztelni az elkapás-újradobás műveletet, valamint a *finally* blokkot. Saját kivétel típusra is le kell tesztelni a működést.
*Cél:* annak megállapítása, hogy az eszköz milyen mértékig képes megbirkózni a kivételekkel.

### 3.3.2. Struktúrák

A primitív típusoknál egy fokozattal összetettebb típus a struktúra, így sorrendileg ennek a támogatottságát érdemes ellenőrizni a primitív típusok tesztelése után.

**S1** *Struktúrával műveletet végző egyszerű függvény:* először azt érdemes kipróbálni, hogy egy eszköz egyáltalán támogatja-e a struktúra használatát, tudja-e kezelni, ha struktúra a paraméter vagy a visszatérési érték.
*Cél:* struktúra alapvető támogatottságának a felderítése.

**S2** *Struktúrával műveletet végző, elágazást használó függvény:* fontos annak a tesztelése, hogy a vizsgált eszköz képes-e olyan feltételeket kielégíteni, melyekben legalább egy adattagja szerepel a struktúrának.
*Cél:* megnézni, hogy struktúrákat használó kényszereket tud-e az eszköz kezelni.

**S3** *Struktúrával műveletet végző, ciklust használó függvény:* azt is ellenőrizni kell, hogy az eszköz tudja-e a struktúrákat ciklusban is használni.
*Cél:* megnézni, hogy struktúrákat használó ciklusokat tud-e az eszköz kezelni.

**S4** *Struktúrát tartalmazó struktúrával műveletet végző függvény:* azt is ki kell próbálni, hogy az eszköz tudja-e kezelni a struktúrát tartalmazó struktúrákat.
*Cél:* struktúrát tartalmazó struktúra támogatottságának a felderítése.

### 3.3.3. Objektumok és kapcsolataik

**O1** *Egyszerű objektum:* a struktúrához hasonlóan itt is azt érdemes először letesztelni, hogy függőség nélküli objektum használatakor jól működik-e a vizsgált eszköz.
*Cél:* objektum alapvető támogatottságának a felderítése.

**O2** *Más objektumot használó objektum:* azt is tesztelni kell, hogy az objektumok közötti interakciót és a tartalmazási függőségeket is tudja-e kezelni az adott eszköz, ugyanis ennek a támogatottsága nem triviális.
*Cél:* annak felderítése, hogy az eszköz tudja-e kezelni az objektumok közötti kapcsolatokat.

**O3** *Öröklés, interfész, absztrakt osztály:* fontos ellenőrizni, hogy az adott eszköz tudja-e kezelni az objektumok közötti öröklést, valamint tudja-e használni az interfészeket és az absztrakt osztályokat. Fontos ellenőrizni az egyszerű objektumok közötti öröklést valamint az interfésztől és az absztrakt osztálytól való öröklést is. Olyan függvényeket

is kell készíteni, melyek nem példányosítható objektumot várnak paraméterként.
*Cél:* öröklés, interfész és absztrakt osztály támogatottságának a felderítése.

**O4** *Túlterhelés:* azt is ki kell próbálni, hogy a túlterheléseket megfelelően támogatja-e az eszköz. Ha az adott nyelvben lehetséges az operátorok túlterhelése, azt is ellenőrizni kell.
*Cél:* túlterhelés támogatottságának a felderítése.

### 3.3.4. Generikus függvények és adatszerkezetek

**G1** *Generikus függvények:* paraméter és visszatérési érték tekintetében is ellenőrizni kell a generikus függvények támogatottságát. Fontos, hogy teszteljünk egy és több generikus paraméterrel is, valamint olyan generikus paraméterrel, melyre megkötéseket készítünk (amennyiben ez az adott nyelvben lehetséges).
*Cél:* támogatottság felderítése generikus paraméterrel rendelkező függvények esetében.

**G2** *Generikus osztályok:* tesztelnünk a generikus osztályokat hasonló síkokon, mint a generikus függvényeket. Tesztelni kell az öröklést is: ilyenkor meg kell különböztetni azt az esetet, amikor a gyermekosztály meghagyja a generikus paramétert attól az esettől, amikor a gyermekosztály a szülő valamely generikus paraméterét leköti. Emellett azt is tesztelnünk kell, hogy ha a gyermekosztály új generikus paramétert vezet be. Emellett fontos olyan függvények kipróbálása, melyek generikus osztályt várnak paraméterül.
*Cél:* támogatottság felderítése generikus osztályok esetében.

### 3.3.5. A nyelvhez tartozó osztálykönyvtár használata

Fontos, hogy olyan teszteseteink is legyenek, melyek a nyelvhez tartozó osztálykönyvtár bizonyos részeit használják. A gyakrabban használt elemek nagyobb prioritást élveznek. Emellett külön ki kell térni az osztálykönyvtári osztályból való származtatásra is. A fő kérdés itt, hogy a nyelvhez tartozó osztálykönyvtár használatakor működőképes-e az eszköz, valamint hogy annak bájtkódját is végre tudja-e szimbolikusan hajtani. A nyelvhez tartozó függvénykönyvtár implementációja általában tartalmaz natív kódot is, amely gyakran a szimbolikus végrehajtás végső korlátja lehet.

A különösen fontos részek a következők:

**L1** *Aritmetikai függvények:* a szimbolikus végrehajtás kihívásai közé tartoznak, így feltétlenül tesztelnünk kell.

**L2** *Karakterláncok:* azért fontos az ellenőrzése, mert valós szoftverekben gyakran előfordulnak.

**L3** *Csomagoló osztályok:* amennyiben az adott platform tartalmaz csomagoló osztályokat, mindenképpen tesztelni kell a működésüket.

**L4** *Kollekciók:* ez is gyakran használt alapfunkció.

### 3.3.6. Egyebek

Fontos a korábban felsorolt egyéb nyelvi elemek és lehetőségek tesztelése is. Különösen fontos, hogy kipróbáljuk a harmadik fél által készített függvénykönyvtár használatát is, hogy azt is tudja-e szimbolikusan tesztelni a vizsgált eszköz, hiszen a szoftverfejlesztés során gyakran használnak külső függőségeket.

## 3.4. A szempontrendszer és a tesztesetek értékelése

Egy ilyen összehasonlító szempontrendszer és a hozzátartozó tesztkészlet értékelése során fontos szempont annak teljessége. Ezt jelen esetben egyrészt a nyelvi elemek lefedését, másrészt a szimbolikus végrehajtás általános kihívásainak lefedését jelenti.

### 3.4.1. Nyelvi elemek lefedése

Minden korábban bemutatott nyelvi elem legalább egyszer szerepel valahol a szempontrendszerben, kivéve az egyéb csoport elemeit. Azonban, hogy a tesztkészlet ne legyen túl nagy, a tesztesetek közti redundanciát minimalizálni kell. Például miután minden primitív típust kipróbáltunk, a későbbi modulokban nem fontos az összes primitív típus használata, hiszen tudjuk a korábbi tapasztalatok alapján, hogy melyekre működik az eszköz. A ciklusok belső vezérlési szerkezeteit sem kell minden ciklusnál kipróbálni, valamint bonyolultabb típusok esetén már nem kell minden vezérlési szerkezetet sem kipróbálni.

### 3.4.2. Szimbolikus végrehajtás korlátainak lefedése

A teljesség másik fontos eleme, hogy a 2.2. fejezetben bemutatott általános kihívásokkal kapcsolatos viselkedést mennyire tudjuk a tesztkészlet elemeinek segítségével vizsgálni. Az elkészített tesztkészlet elemei a következő módon fedik le a kihívásokat:

- *Futási utak exponenciális növekedése:* ezt a jelenséget ciklussal vagy rekurzióval tudjuk előidézni (**B3** és **B5**).
- *Komplex aritmetikai kifejezések:* ezt nemlineáris feltételekkel és aritmetikai függvényekkel tudjuk ellenőrizni (**B2**, **L1**).
- *Külső függvények:* ezt külső függvénykönyvtár használatával tudjuk tesztelni (egyéb).
- *Lebegőpontos számítások:* ezt a lebegőpontos számokra megfogalmazott feltételeket tartalmazó elágazásokkal tudjuk tesztelni (**B2**).
- *Műveletek mutatóval:* ezt külön kell ellenőrizni (egyéb).
- *Interakció a környezettel:* osztálykönyvtári hívásokat használó tesztesetre van szükség (**L**).
- *Többszálú alkalmazások:* külön teszteseteket kell létrehozni (egyéb).

(Az egyéb kategóriába sorolt kihívásokkal a szempontrendszer nem foglalkozik részletesen, ez továbbfejlesztési lehetőség.)

A szempontrendszer definiálása után a következő lépés az itt meghatározott célok és tesztesetek implementációja egy platformra.

# 4. fejezet

# A tesztkészlet implementációja

A tesztkészletet *Java* nyelvre implementáltam. Azért esett erre a platformra a választás, mert több eszköz is rendelkezésre állt, így érdemi összehasonlítást lehetett végezni. Mivel a nyelv lehetőséget ad rá, hogy csomagokba szervezzük a kódot, ezért minden modul külön csomagba került, valamint a modulokon belüli részek is külön csomagba kerültek. Hat főcsomag szükséges, a szempontoknak megfelelően: `_1_basic`, `_2_structures`, `_3_objects`, `_4_generics`, `_5_api`, `_6_other`. A következő szakaszokban először tisztázom az implementációs kérdéseket, majd kifejtem, hogy mely csomagokban pontosan milyen tesztesetekre van szükségünk.

## 4.1. Implementációs kérdések és problémák

A szempontrendszerben kétségkívül használok olyan fogalmakat, melyek implementációja nem egyértelmű. Ezek a következők:

- *Konstans:* publikus, statikus, nem módosítható tagváltozó.
- *Függvény:* statikus osztálymetódus.
- *Struktúra:* olyan csak publikus adattaggal, alapértelmezett és másoló konstruktorral rendelkező osztály, amelytől nem lehet örökölni.

Továbbá probléma, hogy Java-ban a kivételek is objektumok és az ősobjektum és őskivétel az osztálykönyvtár része[1]. Az a tervezői döntés született, hogy ennek ellenére a kivételeket és az objektumokat az osztálykönyvtár tesztelése előtt tesztelem. Ugyanez a helyzet a generikus adatszerkezettel, ahol az `Integer` csomagoló osztályt használtam.

A külső függvénykönyvtár esetében csak a bájtkód áll rendelkezésre, az viszont mindegy, hogy `.class` vagy `.jar` fájlban. Futtatáskor az értelmezőnek a *classpath*-ot kell megadni, amely egyaránt lehet könyvtár vagy JAR állomány.

[1] A Java nyelvet már az osztálykönyvtárral együtt tervezték.

## 4.2. Tesztesetek megtervezése

### 4.2.1. `_1_basic`: Alapműveletek és alaptípusok

Nagyon fontos, hogy az alapvető nyelvi elemeket lefedjük, így itt viszonylag sok, azonban rövid tesztesetre van szükségünk. A **B1** pont alapján minden primitív típushoz kell legalább három teszteset: egy- és kétargumentumú, valamint egy konstanssal visszatérő. Emellett az összes aritmetikai, relációs és bitművelet operátorra meg kell határozni 1-1 tesztesetet. Itt teszteltem a `?:` operátort is, amely valójában egy feltételes kifejezést valósít meg.

Az elágazásnak (**B2**) két fajtája van: `if-else` és `switch`. Az első esetén két-két tesztesetet határoztam meg az `int`, `double` és `boolean` bemeneti típusokra. A `switch` feltételt csak az `int` típusra használtam[2]. A teszteseteket úgy határoztam meg, hogy az egyik egyszerűbb (minden ág könnyen érinthető), míg a másik bonyolultabb és van benne olyan ág, melyet sose érinthetünk futás során. Emellett külön készítettem teszteseteket, melyek lineáris és nem-lineáris feltételt használnak. Ezekben az esetekben a korábbi szám típusokon kívül a `float` is releváns, így erre a típusra is alkottam teszteseteket. Van olyan eset is, aminek egész esetén csak túlcsordulással lehet megoldása[3].

**4.1. lista.** *Elágazásokat és nemlineáris kényszerkielégítést tesztelő tesztesetek*

```
public final class B2a_IfElse {
    public static int int2(final int x, final int y) {
        if (x > 0 && y > 0) {
            return 1;
        } else if (x < 0 && y > 0) {
            return 2;
        } else if (x < 0 && y < 0) {
            return 3;
        } else if (x > 0 && y < 0) {
            return 4;
        } else if (x > 0 && y < 0) {
            // impossible branch, because the previous is the same
            return -1;
        } else if (x == 0 || y == 0) {
            return 0;
        } else {
            // impossible branch
            return -2;
        }
    }
}

public final class B2c_NonLinear {
    // [...]
    public static boolean int1NoSolution(final int a, final int b) {
        // {a,b} = {1, 1.5} -> 1.5 is not an integer
        // no overflow solution
        if (2 * a * a - 5 * a + 3 == 0 && 2 * b * b - 5 * b + 3 == 0 && a != b) {
            return true;
        } else {
            return false;
```

[2] Java-ban még `enum`-ra és karakterláncra lehet, azonban ezek a szempontrendszerben jóval később vannak.

[3] Ez érdekes eset, hogy az adott eszköz megtalálja-e. A Pex ezt is kezeli.

```
        }
    }
    // [...]
}
```

A ciklusok (**B3**) esetén több teszteset van egy ciklustípusra, ugyanis ebben már lehet feltétel is vagy ugrási utasítás. Öt különböző tesztesetet határoztam meg:

- a ciklusnak maximalizálva van az iterációk száma,
- a ciklusban van feltétel,
- a ciklusban ugrási utasítás van,
- a ciklusban bemeneti paramétertől függő ugrási utasítás van és
- végtelen ciklus.

**4.2. lista.** *Ugrási feltételekkel rendelkező hátultesztelő ciklus tesztje*

```
public final class B3c_DoWhile {
    // [...]
    public static int test3(final int x) {
        // 1+2+4+5+7+8+...[+98+100]
        int i = 1;
        int sum = 0;

        do {
            if (i % 3 == 0) {
                continue;
            }
            if (i > 100) {
                break;
            }

            sum += i;
            i++;
        } while (i <= x);

        return sum;
    }
    // [...]
}
```

Az első azért kell, hogy nagy bemeneti paraméter esetén se legyen túl sok futási utunk, a negyedik pedig már teljesen engedi a futási utak számának az óriási ütemben való növekedését.

A tömbök tesztelése (**B4**) nem bonyolult, itt minden tesztesetet kétszer készítettem el: az egyik típus nem dobhat kivételt, a másik viszont igen (például túlindexelés).

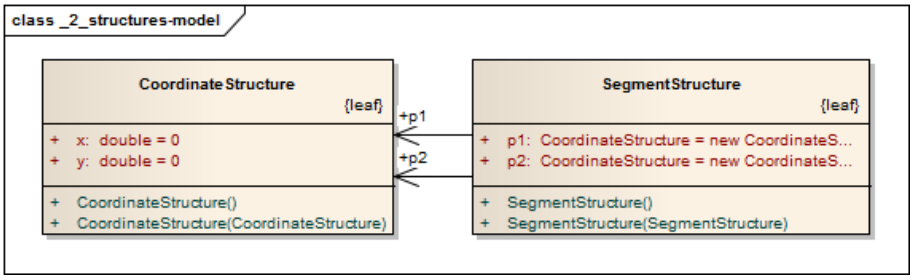
A függvényhívásoknál (**B5**) külön esetnek vettem, amikor publikus és privát függvényt hívunk meg. Mindkét esetben elkészítettem egy függvényt, amit meghívunk és önmagában nem teszteset. Egy teszteset csak meghívja a függvényt, egy a függvény visszatérési értékétől függően más értéket vissza. A harmadik feltételesen hívja meg az első tesztesetet. A rekurzív függvényeknél két típus van: az egyiknek korlátozott a hívási mélysége, a másiknak nem.

A kivételeknél (**B6**) külön kell tesztelni azt, hogy ellenőrzött vagy futási kivétel érkezik, mivel ezt a Java nyelv is megkülönbözteti. Van minden esetben kivételt dobó, feltételesen kivételt dobó és rekurzió során kivételt dobó függvény. Emellett teszteltem a kivétel újradobását, valamint a saját kivételosztályt is. Fontos, hogy Java-ban a kivételek valójában objektumok, és mindenképpen kötődnek az osztálykönyvtárhoz.

Szempontonként rendre 63, 21, 15, 4, 10 és 10, összesen 123 tesztesetet készítettem. Azért ilyen sok a tesztesetek száma, mert minden típusra és operátorra (**B1**) külön teszteset van, azonban ezek általában csak egysorosak.

### 4.2.2. `_2_structures`: Struktúrák

Java-ban a struktúra nem nyelvi elem, viszont a struktúrák tesztje fontos az adattagok miatt. Így egy struktúrát olyan objektumként valósítottam meg, amitől nem lehet örökölni, csak publikus adattagja van, valamint rendelkezik alapértelmezett és másoló konstruktorral. Két struktúrát készítettem: az egyik egy koordinátát valósít meg, a másik pedig egy szakaszt, ami két koordinátából áll. A szempontrendszernek megfelelően (**S1**–**S4**) valósítottam meg a metódusokat, lefedve a visszatérési értéket, feltételes és ciklikus szerkezeteket.




**4.1. ábra.** *A használt struktúrák modelljét ábrázoló osztálydiagram*

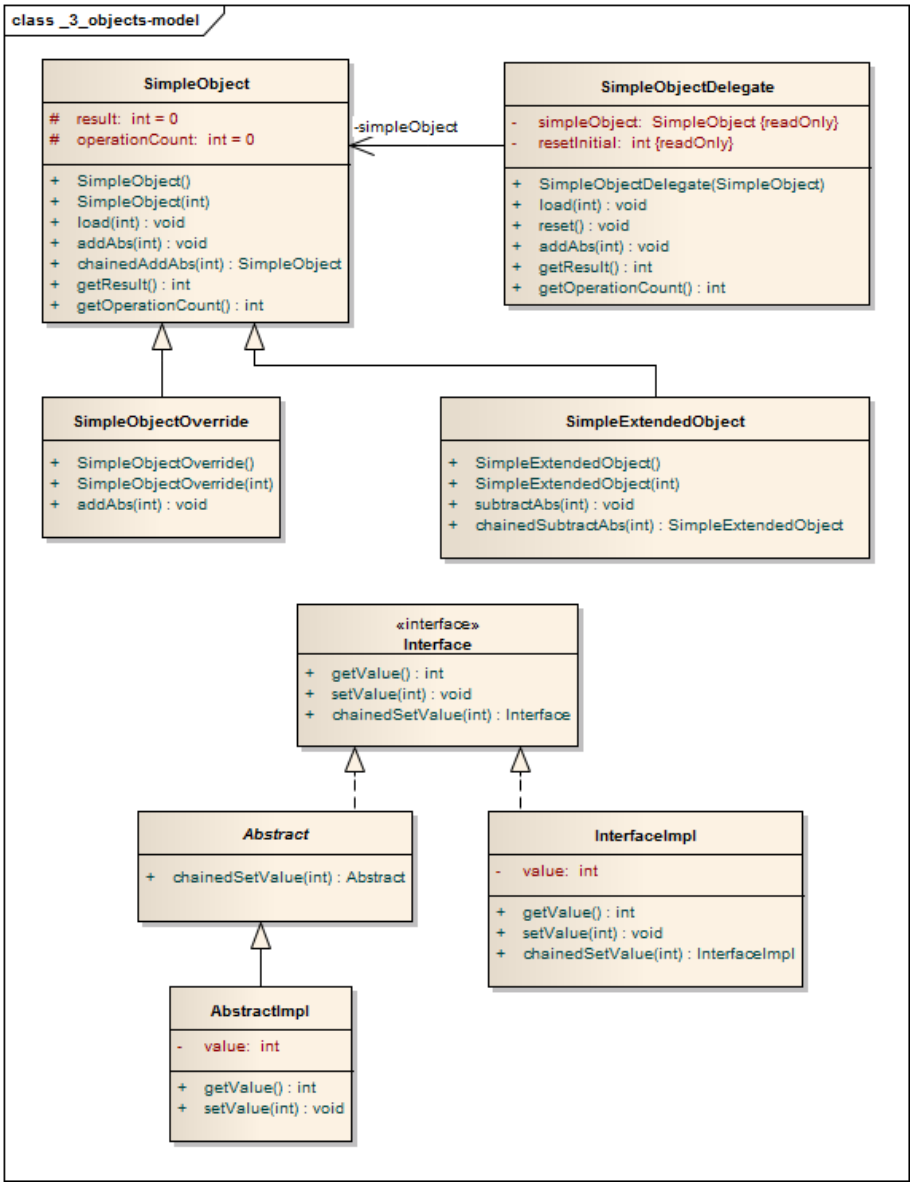
A minimális elvet követve a négy szempontot 5 darab tesztesettel valósítottam meg.

### 4.2.3. `_3_objects`: Objektumok

Az objektumok teszteléséhez is kellenek függvények, amelyek műveletet végeznek rajtuk, hiszen az e nélküli tesztelésre a korábban ismertetett Java-s eszközök közül csak a CodePro AnalytiX és Palus képes. Emellett létre kell hozni az objektumokat, melyek két diszjunkt halmazba sorolhatók. Az egyik halmazban van egy objektum (**O1**), illetve egy ezt használó (**O2**), egy ebből öröklő (**O3**) és egy ebből túlterheléssel öröklő objektum (**O4**). A másik halmazban egy interfész, egy ettől öröklő absztrakt osztály, valamint ezek konkrét implementációi (**O3**).

Több tesztesetet is készítettem egy szemponthoz. Szükség van olyan tesztesetre, ami csak metódust hív, olyanra, ami objektummal tér vissza, olyanra, amiben van feltétel és ciklus. Emellett egy másik síkon is meg kell különböztetni a függvényeket: kell olyan változat, ami az objektumot várja, amin a műveleteket kell végezni, valamint olyanra is szükség van, ami létrehozza az objektumot. Ez azért fontos, mert nem minden eszköz tud objektumot létrehozni.

Szempontonként rendre 10, 2, 12 és 2, összesen 26 tesztesetet készítettem.




**4.2. ábra.** *A használt objektumok modelljét ábrázoló osztálydiagram*

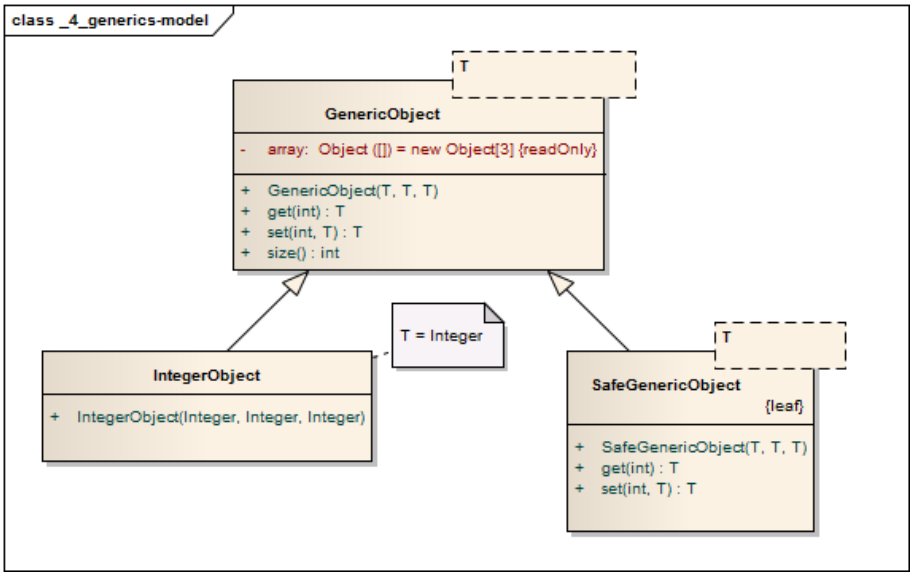
### 4.2.4. `_4_generics`: Generikus adatszerkezetek

Először a generikus függvényeket teszteltem (**G1**), majd az osztályokat (**G2**). Ehhez kellett egy generikus ősosztályt készíteni, valamint kettő ettől öröklőt: az egyik leköti a generikus paramétert, a másik nem. Itt külön érdekesség, hogy az eszközök hogyan birkóznak meg vele: ugyanis futási időben a generikus paraméter ismeretlen, mivel a Java bájtkódba már típuskonverziók kerülnek be.

Szempontonként rendre 3 és 6, összesen 9 tesztesetet készítettem.

### 4.2.5. `_5_api`: Osztálykönyvtár

A `Java` nyelv API-ja elég tág, rettentően sok energia kell a teljes lefedéshez. Így a legfontosabb elemeit teszteltem, melyek a következők: csomagoló osztályok, karakterláncok, aritmetikai függvények, kollekciók, öröklés API osztálytól. Emellett pár egyéb dolgot is




**4.3. ábra.** *A használt generikus objektumok modelljét ábrázoló osztálydiagram*

teszteltem: dátumkezelés, UUID létrehozás[4], valamint reguláris kifejezések.

Összesen 32 tesztesetet készítettem.

**4.3. lista.** *Néhány karakterlánc tesztesetek (a tesztesetek egyre bonyolultabbak)*

```
public final class Strings {
    // [...]
    public static int equality1(final String s) {
        if (s.equals("test")) {
            return 1;
        } else if (s.equals("TeStInG")) {
            return 2;
        } else {
            return 0;
        }
    }

    public static int equality2(final String s) {
        if (s.equals("test")) {
            return 1;
        } else if (s.equalsIgnoreCase("TeStInG")) {
            return 2;
        } else {
            return 0;
        }
    }

    public static boolean regionEquality(final String s) {
        if (s.regionMatches(false, 10, "part", 0, 4)) {
            return true;
        } else {
            return false;
        }
    }
    // [...]
}
```

[4]Ez azért érdekes, mert bizonyos bemenetekre sikeres a művelet, viszont hibás bemenetre kivételt dob.

### 4.2.6. `_6_other`: Egyebek

Ennél a szempontnál teszteltem a névtelen osztályt, enumerációt és a változó számú argumentumokat. A külsőségekhez készítettem egy saját könyvtárat (minimum és maximum műveletek) és ezt használtam fel. A környezettel való kölcsönhatást nem teszteltem, ugyanis ez a probléma nehézsége miatt egy különálló feladat.

Összesen 6 tesztesetet készítettem.

## 4.3. Összegzés

A csomagokat úgy készítettem el, hogy az egy csomagban lévő osztályok más csomag osztályaitól egyáltalán nem függnek (ez alól természetesen kivétel a külső könyvtár). A szempontrendszerhez összesen 201 tesztesetet készült, a részletes megoszlás a 2.1. táblázatban látható.

**4.1. táblázat.** *Tesztesetek számának megoszlása*

| Modul | Teszteset | Összesen |
|---:|---:|---:|
| **Alapműv. és alaptíp. (B)** | 63+21+15+4+10+10 | 123 |
| **Struktúrák (S)** | 5 | 5 |
| **Objektumok (O)** | 10+2+12+2 | 26 |
| **Genericitás (G)** | 3+6 | 9 |
| **Osztálykönyvtár** | 32 | 32 |
| **Egyéb** | 6 | 6 |
| | | **201** |

# 5. fejezet

# A futtatókörnyezet és keretrendszer

Az elkészült Java nyelvű tesztkészlet segítségével 4 darab tesztgeneráló eszközt vizsgáltam meg. Hogy a kapott eredmények minél megbízhatóbbak és reprodukálhatóbbak legyenek, elkészítettem egy automatikus tesztfuttató keretrendszert, ami az adott eszközök bemenetéhez illeszti a tesztkészletet, és megfelelő paraméterezéssel elindítja a tesztbemenetek generálását.

## 5.1. A tesztelendő eszközök meghatározása

A következő – korábban már ismertetett – eszközöket tesztelem: CATG, CodePro AnalytiX, jPET és Symbolic PathFinder. Először ismertetem az eszközök sajátosságait, ezután pedig ismertetem az elkészült futtatókörnyezetet és keretrendszert. Az automatizálás megvalósításakor irányelv volt, hogy független legyen a szempontrendszertől és az implementációtól, így egy olyan keretrendszer készült el, amivel később új tesztesetek is automatikusan futtathatók az eszközökön.

### 5.1.1. Az eszközök sajátosságai

#### CATG

Az eszköz csak `main` függvény alapján képes tesztelni, így azt minden tesztesethez el kell készíteni. Ezeket az általam elkészített keretrendszer automatikusan generálja. A CATG-t minden tesztesetre külön kell lefuttatni, ehhez egy parancssori szkriptet kell előállítani és azt futtatni. A tapasztalataim alapján érdemes az eszköz könyvtárában futtatni a teszteseteket, ugyanis külső könyvtár használata jóval bonyolultabb.

Külön meg kell említeni, hogy a CATG `main` függvényében minden paramétert szimbolikussá kell tenni. Ez csak a CATG osztályaival lehetséges, amelyek viszont csak az egész típusokat és a karakterláncot támogatják – tehát itt rögtön bele is ütköztünk az eszköz egy korlátjába.

### CodePro AnalytiX

A CodePro AnalytiX nem egy szimbolikus végrehajtás elvén működő eszköz, hanem egy forráskód alapú. Az eszközök kiválasztásakor úgy gondoltam, hogy jó összehasonlítási alapot ad, mik a szimbolikus végrehajtást alkalmazó eszközök előnyei és határai.

Az eszköz egy Eclipse plug-in, csak így lehet használni. A tesztgenerálás során először ún. factory osztályokat kell létrehozni, majd utána teszteseteket generálni. Az eszköz tesztesetgeneráló részét nem lehet sem parancssorból, sem `ant`-tal futtatni. Viszont egyszerre több (akár az összes) osztályhoz generálhatunk factory osztályokat és teszteseteket (azonban ilyenkor pár alcsomagot véletlenszerűen kihagy generáláskor).

### jPET

Az eszköz fejlesztői egyaránt elérhetővé tették a forráskódot, valamint a futtatható binárist és az Eclipse plugint. A forráskódból való fordítás nem triviális, az útmutató alapján nem működött, mert néhány függőségnek már újabb verziója van, a fejlesztők által használt verziókat pedig nem sikerült megtalálnom (valószínűleg ezt nem publikálták). Emellett az eszközt csak egy adott könyvtárban ( `/Systems/pet`) lehet fordítani. Ezen okoknál fogva a tesztek futtatásakor a binárist használtam.

Az eszköz futtatásának a legegyszerűbb módja, ha az Eclipse plugin-t használjuk, ugyanis a grafikus felületen az összes opciót beállíthatjuk, a Console nézetben pedig láthatjuk a futás közvetlen kimenetét. Emellett a jPET View segítségével megnézhetjük a generált tesztbemeneteket. Viszont sok teszteset kötegelt futtatásához érdemes parancssorból futtatni az eszközt. Ennek az egyik oka az, hogy a plugin segítségével egyszerre csak egy osztály teszteseteit futtathatjuk, valamint a plugin csak az egyszintű csomaghierarchiákat támogatja, az elkészült implementáció viszont több szintből áll. Ha parancssorból futtatjuk az eszközt, a generált kimenet alapján a plugin ekkor is képes lesz megjeleníteni a tesztbemeneteket (a csomaghierarchia probléma csak futtatáskor jelentkezik, ugyanis a plugin ilyenkor már rosszul paraméterezi a parancsot, ami így kivételt dob).

### Symbolic PathFinder

Az SPF a CATG-hez hasonlóan csak `main` függvényeket képes tesztelni, emellett szükség van minden tesztesetnél egy `.jpf` konfigurációs fájlra. Ebben a fájlban kell megadni a szimbolikusan végrehajtandó metódusokat valamint az SPF beállításait is. Az elérhető konfigurációs paramétereket a fejlesztők publikálták [17], azonban nem volt mind triviális a számomra, így párat ki kellett kísérleteznem.

Az JPF-t és az SPF-t Mercurial repository-ból lehet letölteni. Az eszköz számára létre kellett hozni a `/.jpf/site.properties` fájlt, valamint meg kellett adni a `JAVA_HOME` környezeti változót. Az eszközhöz tartozik egy Eclipse plugin, ami a kiválasztott `.jpf` konfigurációs fájl alapján futtatja az eszközt, így az utóellenőrzések gyorsan végrehajthatók.

## 5.2. Tesztfuttató környezet meghatározása és összeállítása

A CATG és a jPET eszközök csak Linux alatt működnek, a fejlesztők szerint a CATG működik Cygwin segítségével (felkészítették erre az eszközt), viszont a jPET fejlesztői hivatalosan csak Linux alatt működik. A tesztfuttatás során olyan operációs rendszert kell használni, melyen lehetőleg az összes eszköz működik. A választás az Ubuntu 12.04 operációs rendszer 64 bites architektúrára szánt változatára esett. Ennek több oka is van. Az eszközök fejlesztői amikor adtak ajánlást a környezetre, az minden esetben 64 bites Ubuntu volt. Emellett egy gép elég az összes eszköz teszteléséhez. A 12.04-es Ubuntu pedig LTS és várhatóan 2017 áprilisáig lesz hozzá terméktámogatás. A környezet egy virtuális gépen készítettem el, melynek paraméterei a következők:

- *Virtualizációs szoftver:* VMware Player 6
- *CPU:* 2 mag
- *Memória:* 6 GB RAM
- *Háttértár:* 10 GB (rendszer), 2 GB (swap), 10 GB (környezet)

Emellett szükség volt még az Eclipse fejlesztőeszközre, ebből négy példányt használtam (mindegyik 4.3-as verziójú):

- `dev`: plugin-t nem tartalmaz, a fejlesztéshez (tesztesetek és keretrendszer egyaránt) és a CATG ellenőrzéséhez használtam
- `codepro`: csak a CodePro AnalytiX plugin-t tartalmazza, mivel ez jelentősen átszabja a felületet és átszervezi az alapfunkciókat.
- `jpet`: csak a jPET plugint tartalmazza, ezzel futtatam a teszteseteket a jPET eszközön.
- `spf`: a Mercurial és JPF plugin-okat tartalmazza, valamint a JPF és SPF forráskódja ezen belül került letöltésre. Az SPF tesztelését is ezen belül végeztem.

A fejlesztés során négy projekt készült:

- `GeneralLibrary`: a szempontrendszer implementációja.
- `GeneralLibrary-External`: a szempontrendszer által tesztelendő külső könyvtár.
- `Jabba-Common`: az automatizáláshoz szükséges annotációkat tartalmazza.
- `Jabba-Generator`: az automatikus generálásért és futtatásért felelős keretrendszer.

## 5.3. Tesztelési folyamat automatizálása

Az automatizáláshoz a következő munkafázisokat kellett elvégezni:

1. Tesztkészlet kiegészítése metaadatokkal
2. Generálandó bemenet meghatározása az összes tesztelendő eszköz számára
3. Kimenet gyűjtésének meghatározása az összes tesztelendő eszköz számára
4. Futtatás módjának meghatározása az összes tesztelendő eszköz számára
5. A bemenetgeneráló és futtató elkészítése

A munkafázisok elvégzése után a teszteseteket már tudtam automatikusan futtatni. Az automatizáló eszközt `Java` nyelven valósítottam meg, ugyanis így lehetőségem nyílt a *reflection* használatára. A cél az volt, hogy ha változnak a tesztesetek, az automatizáló eszközt ne kelljen módosítani.

A futtató esetében célszerű minden tesztesetet külön folyamatban futtatni, így ha egy teszteset miatt elszáll egy eszköz, attól még egy másik esetet letesztelhetünk. Emellett javasolt egy futási időkorlát meghatározása minden tesztesetre, hogy leállítsuk az esetlegesen túl sokáig futó teszteket.

### 5.3.1. Tesztkészlet kiegészítése metaadatokkal

Az automatizálás úgy a legegyszerűbb, ha egy könnyen kiterjeszthető program minden vizsgált eszköz számára előállítja a megfelelő bemenetet. Ezt célszerű Java nyelven írni, így használhatjuk a *reflection*-t is. A teszteseteket annotációkkal láttam el, jelölve, hogy melyeket kell futtatni. Ezek az annotációk a következők:

- `@SymbTest`: az adott statikus metódus szimbolikusan tesztelendő.
- `@SymbTestAll`: az osztály összes statikus metódusa szimbolikusan tesztelendő[1].
- `@SymbTestSkip`: az adott statikus metódust nem kell szimbolikusan tesztelni (akkor van értelme, ha az osztályra a `@SymbTestAll` definiálva van).

Sajnos a Java nem ad nyelvi lehetőséget arra, hogy az annotációkra komplex megkötéseket tegyünk, így ezeket az elemző program ellenőrzi. Ezek a megkötések a következők:

- `@SymbTest` csak olyan metóduson lehet, ami `public` láthatóságú és statikus, valamint az osztály nem rendelkezik `@SymbTestAll` annotációval.
- `@SymbTestSkip` csak olyan metóduson lehet, ami `public` láthatóságú és statikus, valamint az osztály rendelkezik `@SymbTestAll` annotációval.

[1] Belső osztályokra nem öröklődik, hanem külön kell megadni.

### 5.3.2. Generálandó bemenet, kimenet és futtatás módjának meghatározása az összes tesztelendő eszköz számára

Ebben a részben meghatározom a bemenetek pontos formátumát minden eszközre. Ezt egy egyszerű teszteseten mutatom be, ami az alábbi kódrészletben látható. Olyan példát választottam, amit minden eszköz képes futtatni.

**5.1. lista.** *Egy egyszerű teszteset forráskódja*

```
package generallibrary._1_basic.B2_conditionals;

import jabba.annotations.SymbTestAll;

@SymbTestAll
public final class B2a_IfElse {
    private B2a_IfElse() {
        throw new UnsupportedOperationException("Static class");
    }

    // ...

    public static int int1(final int x) {
        if (x == 12345) {
            return 2;
        }
        if (x > 0) {
            return 1;
        } else if (x < 0) {
            return -1;
        } else {
            return 0;
        }
    }

    // ...
}
```

#### CATG

A futtatáshoz az alábbi kiegészítő osztályra van szükség:

**5.2. lista.** *main() függvény a CATG számára*

```
package generallibrary._1_basic.B2_conditionals;

import generallibrary._1_basic.B2_conditionals.B2a_IfElse;
import catg.CATG;

public final class B2a_IfElse_int1 {

    public static void main(final String[] args) throws Exception {
        int a1 = CATG.readInt(0);

        System.out.println("B2a_IfElse.int1");
        System.out.println("  int a1 = " + a1);
        System.out.println("  result: " + B2a_IfElse.int1(a1));
    }

}
```

Azért kellenek a kiírások, mert az eszköz nem írja ki az eredményeket. Az eszközt futtatni a következő paranccsal lehet:

```
./concolic 100 generallibrary._1_basic.B2_conditionals.B2a_IfElse_int1
```

Az első paraméter azt határozza meg, hogy maximum hányszor indíthat keresést az eszköz, a második pedig megadja, hogy mely tesztesetet kell futtatni. A kimenet ehhez hasonló lesz:

**5.3. lista.** *A futás során előállt kimenet*

```
Now testing generallibrary._1_basic.B2_conditionals.B2a_IfElse_int1
[Input 1]
B2a_IfElse.int1
  int a1 = 0
  result: 0
[Input 2]
B2a_IfElse.int1
  int a1 = -1
  result: -1
[Input 3]
B2a_IfElse.int1
  int a1 = 2147483647
  result: 1
[Input 4]
B2a_IfElse.int1
  int a1 = 12345
  result: 2

================================================================================
Exit value: 1
```

Amennyiben nem futtatja annyiszor az eszköz a tesztesetet, ahányszor megengedtük neki, hanem kevesebbszer futtatja, az azt jelenti, hogy az összes futási utat lefedte.

### CodePro AnalytiX

Az eszköz bemenete csak a teszteset forráskódja, kimenete pedig JUnit tesztek. Az eszközt az Eclipse grafikus felületén kell futtatni.

### jPET

Az eszköz bemenete csak a teszteset bájtkódja. Az eszköz képességeit korlátozza az Eclipse plugin, így érdemes parancssorból futtatni:

```
./pet 'generallibrary/_1_basic/B2_conditionals/B2a_IfElse.int1(I)I' -cp bin
-c bck 10 -td num -d -100000 100000 -l ff -v 2 -w -tr statements -cc yes
-xml '[...]/pet-testcases/generallibrary/[...]/B2a_IfElse.int1(I)[TC].xml'
```

A paraméterek részletezése:

- `-cp bin`: Java classpath.
- `-c bck 10`: maximimális blokkménység 10 (lehetőség van a mélység korlátozására is).
- `-td num -d -100000 100000`: konkrét értékek kiszámítása valamint egész típus alkalmazandó értékkészlete.

- `-l ff`: címkézési (változókiválasztási) stratégia.
- `-v 2`: legnagyobb *verbosity* szint.
- `-w`: az előállított CLP program mentése (a `/tmp/pet` könyvtárba menti).
- `-tr statements`: követés (tracing) módja, lehet még blokk alapján is.
- `-cc yes`: kódfedettségi eredmények megjelenítése.
- `-xml [file]` kimeneti XML fájl elérési útja.

Az eszköz kimenete a következőhöz hasonló:

**5.4. lista.** *A jPET által generált példakimenet*

```
<?xml version="1.0"?>
<pet>
  <test_case>
    <method>generallibrary/_1_basic/B2_conditionals/B2a_IfElse.int1(I)I</method>
    <args_in>
      <data>0</data>
    </args_in>
    <heap_in></heap_in>
        <heap_out></heap_out>
    <return>0</return>
    <exception_flag>ok</exception_flag>
    <!-- ... trace information ... -->
  </test_case>
  <test_case>
    <method>generallibrary/_1_basic/B2_conditionals/B2a_IfElse.int1(I)I</method>
    <args_in>
      <data>-100000</data>
    </args_in>
    <heap_in></heap_in>
        <heap_out></heap_out>
    <return>-1</return>
    <exception_flag>ok</exception_flag>
    <!-- ... trace information ... -->
  </test_case>
  <test_case>
    <method>generallibrary/_1_basic/B2_conditionals/B2a_IfElse.int1(I)I</method>
    <args_in>
      <data>1</data>
    </args_in>
    <heap_in></heap_in>
        <heap_out></heap_out>
    <return>1</return>
    <exception_flag>ok</exception_flag>
    <!-- ... trace information ... -->
  </test_case>
  <test_case>
    <method>generallibrary/_1_basic/B2_conditionals/B2a_IfElse.int1(I)I</method>
    <args_in>
      <data>12345</data>
    </args_in>
    <heap_in></heap_in>
        <heap_out></heap_out>
    <return>2</return>
```

```
    <exception_flag>ok</exception_flag>
    <!-- ... trace information ... -->
  </test_case>
</pet>
```

Emellett még a parancs futásának a kimenetét is mentjük egy szövegfájlba, hogy az később visszanézhető legyen. Ebben kapunk pontos visszajelzést a futásról, az esetlegesen fellépő hibákról valamint a kódlefedettségről.

### Symbolic PathFinder

A CATG-hez hasonlóan itt is el kell készíteni egy `main` függvényt, azonban itt csak a tesztesetet kell meghívni. Emellett minden tesztesethez szükség van egy `.jpf` konfigurációs fájlra. A kimenetet külön könyvtárba, tesztesetenként külön fájlba kell gyűjteni. A teszteseteket a konfigurációs fájlok segítségével lehet egyesével futtatni, azonban érdemes egy olyan tesztfuttató létrehozása, ami kezeli az időtúllépést és több tesztesetet is képes egymás után lefuttatni.

**5.5. lista.** *SPF teszteset*

```
package generallibrary._1_basic.B2_conditionals;

import generallibrary._1_basic.B2_conditionals.B2a_IfElse;

public final class B2a_IfElse_int1 {

    public static void main(final String[] args) throws Exception {
        B2a_IfElse.int1(0);
    }

}
```

**5.6. lista.** *SPF számára elkészített konfigurációs fájl*

```
target=generallibrary._1_basic.B2_conditionals.B2a_IfElse_int1
symbolic.method=generallibrary._1_basic.B2_conditionals.B2a_IfElse.int1(sym)
classpath=bin/,libs/GeneralLibrary-External.jar,libs/Jabba-Common.jar
listener=gov.nasa.jpf.symbc.SymbolicListener
symbolic.debug=on
```

Az SPF kimenete szöveg, viszont minden lényeges információt tartalmaz:

**5.7. lista.** *SPF által előállított kimenet*

```
Running Symbolic PathFinder ...
[...]

====================================================== system under test
generallibrary._1_basic.B2_conditionals.B2a_IfElse_int1.main()

====================================================== search started: 23/10/13 18:03
[numeric PC: constraint # = 1
x_1_SYMINT != CONST_12345 -> true

### PCs: total:1 sat:1 unsat:0

[...]
```

```
======================================================== Method Summaries
Inputs: x_1_SYMINT

int1(12345)  --> Return Value: 2
int1(1)  --> Return Value: 1
int1(-1000000)  --> Return Value: -1
int1(0)  --> Return Value: 0
Inputs: x_1_SYMINT

int1(12345)  --> Return Value: 2
int1(1)  --> Return Value: 1
int1(-1000000)  --> Return Value: -1
int1(0)  --> Return Value: 0

======================================================== Method Summaries (HTML)
[...]

============================================================ results
no errors detected

============================================================ statistics
elapsed time:       00:00:00
states:             new=7, visited=0, backtracked=7, end=4
search:             maxDepth=4, constraints hit=0
[...]
```

## 5.4. Tesztesetek futtatása

A tesztesetek kötegelt futtatásakor külön oda kell figyelni arra, hogy mindig külön folyamatot indítsunk. A folyamatoknak 5 másodperc futási időkorlátot határoztam meg, mivel ennél több ideig csak kevés teszteset futott, azok nagy része pedig végtelen ciklusba torkollott. Minden teszt kimenetét és hibakimenetét menti a futtató, így a teszt futásának részletes eredménye később visszanézhető. Az öt másodpercen belül nem végző teszteket pedig kézzel újrafuttattam és részletesen megnéztem, hogy mi lehetett a probléma.

# 6. fejezet

# Eredmények és értékelés

## 6.1. Eredmények

Az előző fejezetben bemutatott keretrendszer segítségével lefuttattam a teszteseteket a kiválasztott négy eszközre. Mivel az eszközök kimenete teljesen különböző ezért az eredmények összegyűjtését manuálisan végeztem el. Minden egyes teszteset futási eredményét kézzel ellenőriztem. Ezt az is indokolta, hogy ha az alapértelmezett paraméterezés nem felelt meg egy eszköznek, akkor finomhangoltam a beállításokat és újrafuttattam a tesztesetet (pl. tesztesethez való SMT megoldó kiválasztása).

A következő oldalon található táblázat foglalja össze a részletes eredményeket. Az oszlopelnevezések a következőket jelentik:

**N/A** Az eszköz által nem futtatható tesztesetek száma.

**EX** Azon tesztesetek száma, melyek futtatása kivétel miatt leállt[1].

**T/M** Azon tesztesetek száma, melyek futtatása során vagy időtúllépés volt, vagy elfogyott a folyamat rendelkezésére álló memória[2].

**NC** Azon tesztesetek száma, melyek futtatása sikeres volt, de nem értek el teljes utasítási lefedettséget[3].

**C** Azon tesztesetek száma, melyek futtatása sikeres volt, és elérték a maximálisan elérhető utasítási lefedettséget[4].

[1] Azt nem soroltam ide, amikor az eszköz detektálta a kivétel érkezését.

[2] Az időtúllépéses eseteket újrafuttattam 30 másodperc-es maximális várakozási idővel.

[3] Azokat az eseteket, amelyeket csak túlcsordulással lehet elérni elérhetetlennek tekintettem a kiértékeléskor.

[4] Nem csak egy olyan teszteset volt, amit nem lehet 100%-osan lefedni.

**6.1. táblázat.** *A tesztfuttatás részletes eredménye*

| | | CATG | | | | | CodePro AnalytiX | | | | | jPET | | | | | Symbolic PathFinder | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Össz.** | N/A | EX | T/M | NC | C | N/A | EX | T/M | NC | C | N/A | EX | T/M | NC | C | N/A | EX | T/M | NC | C |
| **B1** | 63 | 0 | 4 | 0 | 0 | 59 | 0 | 0 | 0 | 0 | 63 | 20 | 0 | 0 | 0 | 43 | 0 | 4 | 0 | 0 | 59 |
| **B2** | 21 | 8 | 4 | 0 | 0 | 9 | 2 | 0 | 0 | 10 | 9 | 7 | 0 | 0 | 0 | 14 | 0 | 0 | 0 | 0 | 21 |
| **B3** | 15 | 0 | 0 | 9 | 0 | 6 | 0 | 0 | 0 | 0 | 15 | 0 | 0 | 0 | 0 | 15 | 0 | 0 | 9 | 0 | 6 |
| **B4** | 4 | 4 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 2 | 2 | 0 |
| **B5** | 10 | 0 | 1 | 0 | 1 | 8 | 0 | 0 | 0 | 4 | 6 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 2 | 0 | 8 |
| **B6** | 10 | 0 | 8 | 0 | 2 | 0 | 0 | 0 | 0 | 8 | 2 | 4 | 0 | 0 | 0 | 6 | 0 | 8 | 0 | 0 | 2 |
| **S1–S4** | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 2 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 5 | 0 |
| **O1** | 10 | 5 | 0 | 2 | 0 | 3 | 0 | 0 | 0 | 2 | 8 | 5 | 0 | 0 | 3 | 2 | 0 | 2 | 2 | 3 | 3 |
| **O2** | 2 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| **O3** | 12 | 8 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 2 | 10 | 3 | 0 | 0 | 9 | 0 | 0 | 0 | 2 | 8 | 2 |
| **O4** | 2 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| **G1** | 3 | 3 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 |
| **G2** | 6 | 6 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 1 | 0 | 5 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 6 | 0 |
| **API** | 32 | 13 | 3 | 0 | 15 | 1 | 0 | 0 | 0 | 27 | 5 | 31 | 0 | 0 | 1 | 0 | 6 | 3 | 0 | 18 | 5 |
| **Egyéb** | 6 | 3 | 2 | 0 | 0 | 1 | 0 | 1 | 0 | 5 | 0 | 5 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 4 | 2 |
| **Összesen** | 201 | 57 | 22 | 11 | 18 | 93 | 7 | 8 | 0 | 64 | 122 | 87 | 0 | 0 | 16 | 98 | 9 | 17 | 17 | 42 | 116 |

## 6.2. Értékelés

Az értékelést a korábban felállított szempontrendszer mentén végighaladva végeztem, és az eszközöket a szempontokon belül hasonlítottam össze.

### 6.2.1. Alapműveletek és alaptípusok

#### B1: Alaptípusokkal műveletet végző függvények

A CATG nem tudja szimbolikusan kezelni a lebegőpontos típusokat, így egy olyan metódus se volt futtatható, amelynek legalább egy paramétere lebegőpontos típus. Viszont minden mást kezel, amit a szempont felsorol. Emellett fontos megemlíteni, hogy érvénytelen művelet esetén hibát dob az eszköz, és egyáltalán nem generál bemenetet (pl.: osztás nullával). Ez azért rossz, mert ha ilyen művelet van egy metódusban, azt elég nagy eséllyel nem fogjuk tudni ellenőrizni az eszközzel.

A CodePro AnalytiX minden primitív típust kezel, emellett tudja használni a konstansokat és az összes operátort is. Ez várható volt, hiszen a programot futtatja és forráskód alapján próbál teszteseteket generálni.

A jPET esetén már korán meglátszik, hogy a bájtkódot CLP programra fordítja (ami Prolog nyelvű), majd ezt futtatja, ugyanis ez a fordító nincs felkészítve többek közt a `byte`, `short` és `char` típusokra. A logikai típust olyan `integer` típusként kezeli ami csak 0 vagy 1 értéket vehet fel (ez nem meglepő, hiszen a JVM is így kezeli). Az aritmetikai és relációs operátorokat támogatja, viszont a bitműveleteket nem. A hiba oka az, hogy a CLP program nem találja az adott művelet Prolog szabályát.

Az SPF minden tesztesetet sikeresen futtat.

Megjegyzés: a ternáris operátor valójában egy feltételes szerkezet, de ezt minden eszköz sikeresen kezelte.

#### B2: Alaptípusokkal dolgozó, elágazást tartalmazó függvények

Az egyszerű elágazásokat, ahol a feltétel kielégítése triviális és az argumentum egész, az összes eszköz jól kezelte, a nagyobb eltérések a lineáris és nemlineáris feltételek kielégítésénél jöttek elő.

Meg kell jegyezni, hogy a kényszer kielégítésének két korlátja is lehet: vagy az eszköz nem tudja a bemenetben található kényszert az SMT megoldó számára lefordítani, vagy az SMT megoldó nem tudja megoldani.

A CATG az egyik nemlineáris problémát nem tudta megoldani, mivel az általa használt SMT megoldó (choco) ezt nem támogatta[5].

A CodePro sajnos már nem tudta a lineáris feltételt sem kielégíteni, azonban a generált tesztbemeneteken látszik, hogy a forrásban megtalálható literálokkal próbálkozott, de azokkal már műveletet nem végzett. Valós számok esetén négy tesztesetet generált, de mindegyik bemenete azonos volt (mint paraméter értékenek 1-et választott).

[5]Lehetőség van más megoldó használatára is, pl. a szerzők a yices-t ajánlják nemlineáris kifejezésekre, azonban ehhez implementációt nem találtam.

A jPET a feltételeket egész számokra jól elégítette ki, azonban a lebegőpontos számok esetén a CLP program hibát dobott, ugyanis a literálokat nem tudta kezelni.

AZ SPF egyaránt jól teljesített mind az egész, mind a tört számok esetében. Arra a tesztesetre, aminek csak túlcsordulással van megoldása, ő sem tudott bemenetet generálni[6]. Fontos megemlíteni, hogy a nemlineáris kifejezéseket az alapértelmezett `choco` nem tudja megoldani, így ezekben az esetekben a `coral`-t használtam.

### B3: Alaptípusokkal dolgozó, ciklust tartalmazó függvények

A CATG minden futási utat megkeres, nem csak a kódsorokat igyekszik lefedni. Ennek eredménye viszont az, hogy sokszor végtelen ciklusba jut az eszköz, és a futási utak nyilvántartása miatt egy idő után elfogy a memória.

A CodePro következetesen a $0, 1, 2 \ldots$ értékekkel próbálkozik, viszont javára szól, hogy nem keveredik végtelen ciklusba, mert generáláskor egy idő után leállítja a meghívott metódust.

A jPET minden kódsort lefed, amit lehetséges[7], egyedül a tisztán végtelen ciklus esetén nem generál tesztbemenetet és 0%-os fedettséget jelez.

Az SPF a CATG-hez hasonlóan a futási utakat keresi, így ő is több esetben végtelen ciklusba jut még a futási utak felderítésekor, így vagy a futtató állítja le, vagy a memória fogy el.

### B4: Alaptípusok tömbjeivel műveletet végző függvények

A CATG nem tudja kezelni a tömböket.

A CodePro érdekes módon azzal az esettel, amikor ellenőrizzük a paramétereket, és a tesztesetek nem dobhatnak kivételt, nem tud megbirkózni. Itt is a literálokkal próbálkozott, azonban nem sikerült minden kódsort lefednie.

A jPET akkor is jól generál bemenetet, amikor kivétel érkezhet. Ennek ellenére a generált tesztbemenet érvénytelen, pl. olyan tömböt generál, aminek a hossza nagyobb, mint a benne lévő elemek száma. Ez azért van, mert a kényszerek alapján a jPET olyan halmot (heap) állít elő, ami kielégíti azokat.

Az SPF nem talál feltételt a futás során, így ő sem támogatja a tömböket.

### B5: Függvényt hívó függvények, rekurzív függvények

Az eredmények között egy eszköz esetén sem volt különbség, annak tekintetében, hogy a hívott függvény láthatósága privát vagy publikus (tehát még a generált bemenetek is megegyeztek).

A CATG ezeket az eseteket viszonylag jól kezelte, egyedül a rekurzív függvényekkel nem tudott tökéletesen megbirkózni.

A CodePro nem fedte le az összes kódsort, viszont sosem került végtelen ciklusba az eszköz.

[6] A PEX ezt is figyelembe veszi.

[7] A tesztesetekben voltak olyan ágak, melyeket logikailag sosem érhet el a program.

A jPET minden esetben 100%-os fedettséget ért el.

Az SPF a CATG-hez hasonlóan az összes olyan esetre jó tesztbemenetet generált, ami nem kerülhetett végtelen ciklusba.

#### B6: Kivételek

A CATG szinte mindig leállt, ha kivétel érkezett egy metódusból, csak a rekurzív esetekben nem érte el a kivételt.

A CodePro ha kivételt talált, jó tesztesetet generált és rögzítette, hogy milyen típusú kivételt kell kapni. Azonban a korábban említettekhez hasonlóan nem derített fel minden futási utat.

A jPET az ellenőrzött kivételek esetében nem tudott mindig bemenetet generálni, a hibakimenet arra utal, hogy nem tudja példányosítani a kivétel objektumot (mivel a CLP program nem találja a hozzá tartozó szabályt). A nem ellenőrzött kivételeket használó eseteket viszont jól kezelte.

Az SPF ha kivételt talált, azonnal leállt két eset kivételével. Az SPF tesztbemenet-generáláskor csak az általánosan előforduló kivételeket (`NullPointerException`, túlindexelés, stb.) kapja el.

### 6.2.2. Struktúrák

A CATG csak azokat az eseteket tudta kezelni, ahol a bemeneti paraméterek értéke nem struktúra, ezekben az esetekben viszont lefedte a futási utakat. A CodePro csak a generált factory osztályok által generált struktúrákkal végez műveletet, így teljesen önállóan nem tud érdemi tesztbemenetet generálni. A jPET megoldása minden esetben kifogástalan, mindig 100%-os fedettséget ér el. Az SPF nem találja meg a futási utakat.

### 6.2.3. Objektumok és kapcsolataik

A CATG csak azokkal az esetekkel tudott megbirkózni, amelyekben mi hoztuk létre az objektumot, azokat viszont tökéletesen kezelte. Ugyan ez a helyzet a jPET-tel és az SPF-fel is. A CodePro előnye ezzel szemben az, hogy hoz létre factory osztályt az interfészeknek és az absztrakt osztályoknak is. Az osztály metódusai viszont már konkrét objektumokkal térnek vissza. Emellett a CodePro mock csonkokat is tud generálni.

A függőségeket, túlterhelést az összes eszköz jól kezelte. Ez a jPET esetén különösen fontos, hiszen itt a Java bájtkódot az eszköz CLP programra fordítja át.

### 6.2.4. Generikus függvények és adatszerkezetek

A generikus függvényeket és adatszerkezeteket egy eszköz sem támogatta. Amely eszköz tudott futtatni egy tesztesetet, az se produkált teljes fedettséget.

### 6.2.5. A nyelvhez tartozó osztálykönyvtár használata

A karakterláncot csak a CATG és a CodePro AnalytiX támogadja, de ezek az eszközök is csak az egzakt egyenlőség vizsgálatát. A CATG nem instrumentálja a Java osztálykönyvtárának a releváns bájtkódjait. A CodePro AnalytiX csak a triviális bemeneteket állítja elő. A jPET egyáltalán nem tud függőségeket használni (erre csak egy kivétel volt), ugyanis a CLP programhoz nincsenek meg az osztálykönyvtár releváns szabályai. Az SPF viszonylag sok esetet futtat, de csak keveset teljes lefedettséggel. A nem teljes lefedettségű esetek nagy részében csak triviális bemenettel próbálkozik, viszont rögzíti a kivételek érkezését.

### 6.2.6. Egyebek

A CATG csak a névtelen osztály esetét tudta megfelelően futtatni. A külső könyvtárat nem tudta használni, mivel nem a saját fordítóbeállításaival fordítottuk, így ezek az esetek kivételt dobtak.

A CodePro AnalytiX egy esetet sem tudott teljesen lefedni.

A jPET egyedül a változó argumentumokat tudta kezelni. Ez várható volt, hiszen a tömbökkel is boldogult.

Az SPF tökéletesen csak a külső osztálykönyvtár használatát támogatja, a többi tesztesetnél nem generált teljes lefedettséget.

# 7. fejezet

# Összefoglalás

## 7.1. Eredmények összegzése

A TDK munka keretében az irodalomkutatás után meghatároztam egy általános szempontrendszert, mellyel összehasonlíthatóak különböző automatikus tesztbemenet-generáló eszközök. A szempontrendszert úgy terveztem meg, hogy az összegyűjtött imperatív nyelvi szerkezeteket, valamint a programszervezési módszereket is lefedje. A szempontrendszer tervezésekor figyelembe vettem a szimbolikus végrehajtás aktuális kihívásait is, így a szempontrendszer alkalmazásával egy eszköz felkészültségét is vizsgálhatjuk.

Korábban ilyen szempontrendszert még nem készítettek, ahogyan részletes értékelés sem született több szimbolikus végrehajtással működő tesztbemenet-generáló eszközről. A szempontrendszer felépítése szisztematikus, így segítségével lépésről lépésre kiértékelhetjük egy eszköz képességeit, sőt, a szempontok alapján több eszközt is össze tudunk hasonlítani.

Bemutattam a szempontrendszer egy leképzését is, ami Java nyelvre történt. Létrehoztam egy keretrendszert, aminek a segítségével a tesztkészlet könnyen futtatható bármely eszközre. A keretrendszer egyszerűen kiterjeszthető, így nem igényel sok erőforrást, hogy a tesztkészletet egy új eszközön futtassuk.

Futtattam a tesztkészletet a CATG, CodePro AnalytiX, jPET és Symbolic PathFinder eszközökön, valamint ezeket részletesen ki is értékeltem. A tesztkészlet futtatásával sikerült azonosítani az eszközök képességei közötti különbségeket, valamint azokat a nyelvi elemeket, amelyekkel egyetlen vizsgált eszköz sem képes megbirkózni. Korábban nem készült ezeknek az eszközöknek a képességeiről részletes visszajelzés.

Úgy gondolom, hogy a munkám hasznos kontribúció és elősegíti a szakterületen folyó kutatásokat.

## 7.2. Távlati tervek

A jövőben érdemes lehet a teszteseteket más Java-s eszközön (pl.: LCT, Palus), vagy akár más platform eszközein is lefuttatni. Így sok eszközről lesz részletes visszajelzésünk.

A jelenlegi keretrendszerben a futási eredmények kiértékelése nem automatikus. Az automatikus kiértékelést a tesztesetek nagy száma is indokolja. Minden eszköz más formá-

tumban állítja elő a kimenetét, így ez nehezíti a feladatot.

Továbbá tervben van a tesztkészlet és a tesztfuttató keretrendszer nyilvánosságra hozatala, így a fejlesztők frissíthetik a tesztelt eszközöket, javíthatják a hibákat, úgy, hogy közvetlen és gyors visszajelzést kapnak az eszköz működéséről.

# Irodalomjegyzék


[1] E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET: a partial evaluation-based test case generation tool for java bytecode. In *Proc. of workshop on Partial evaluation and program manipulation*, PEPM'10, pages 25–28. ACM, 2010. doi:10.1145/1706356.1706363.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978 – 2001, 2013. doi:10.1016/j.jss.2013.02.061.

[3] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. of Int. Conf. on Automated Software Engineering*, ASE'08, pages 443–446, 2008. doi:10.1109/ASE.2008.69.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224. USENIX Association, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, Dec. 2008. doi:10.1145/1455518.1455522.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proc. of the 33rd Int. Conf. on Software Engineering*, ICSE '11, pages 1066–1071. ACM, 2011. doi:10.1145/1985793.1985995.

[7] T. Chen, X. song Zhang, S. ze Guo, H. yuan Li, and Y. Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758 – 1773, 2013. doi:10.1016/j.future.2012.02.006.

[8] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *Proc. of the eighteenth in. symp. on Software testing and analysis*, ISSTA '09, pages 129–140. ACM, 2009. doi:10.1145/1572272.1572288.

[9] P. Godefroid and J. Kinder. Proving memory safety of floating-point computations by combining static and dynamic program analysis. In *Proc. of the 19th*

*int. symp. on Software testing and analysis*, ISSTA '10, pages 1–12. ACM, 2010. doi:10.1145/1831708.1831710.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. ISSN 0362-1340. doi:10.1145/1064978.1065036.

[11] Google. CodePro AnalytiX web page. `https://developers.google.com/java-dev-tools/codepro/doc/`, 2013. Utolsó hozzáférés: 2013.10.25.

[12] K. Kähkönen. LCT web page. `http://www.tcs.hut.fi/Software/lime/`, 2013. Utolsó hozzáférés: 2013.10.25.

[13] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for c programs using cute and austin. *J. Syst. Softw.*, 83(12):2379–2391, Dec. 2010. doi:10.1016/j.jss.2010.07.026.

[14] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Int. Symp. on Code Generation and Optimization*, pages 75–86, 2004. doi:10.1109/CGO.2004.1281665.

[15] Microsoft Research. Pex for fun web page. `http://www.pexforfun.com/`, 2013. Utolsó hozzáférés: 2013.10.25.

[16] MSDN. C# language specification. `http://msdn.microsoft.com/en-us/library/vstudio/ms228593.aspx`, 2013. Utolsó hozzáférés: 2013.10.25.

[17] NASA. Symbolic PathFinder – tool documentation. `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc/doc`, 2013. Utolsó hozzáférés: 2013.10.25.

[18] T. C. R. Network. C++ documentation. `http://www.cplusplus.com/doc/`, 2013. Utolsó hozzáférés: 2013.10.25.

[19] Oracle. Java platform standard edition 7 documentation. `http://docs.oracle.com/javase/7/docs/`, 2013. Utolsó hozzáférés: 2013.10.25.

[20] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *Int. Conf. on Software Engineering*, ICSE'07, pages 75–84, 2007. doi:10.1109/ICSE.2007.37.

[21] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353, 2009. ISSN 1433-2779. doi:10.1007/s10009-009-0118-1.

[22] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013. ISSN 0928-8910. doi:10.1007/s10515-013-0122-2.

[23] X. Qu and B. Robinson. A case study of concolic testing tools and their limitations. In *Int. Symp. on Empirical Software Engineering and Measurement*, ESEM'11, pages 117–126, 2011. doi:10.1109/ESEM.2011.20.

[24] K. Sen. CATG web page. `https://github.com/ksen007/janala2`, 2013. Utolsó hozzáférés: 2013.10.25.

[25] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006. doi:10.1007/11817963_38.

[26] N. Tillmann and J. Halleux. Pex – white box test generation for .NET. In *Tests and Proofs*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008. doi:10.1007/978-3-540-79124-9_10.

[27] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proc. of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 353–363. ACM, 2011. doi:10.1145/2001420.2001463.