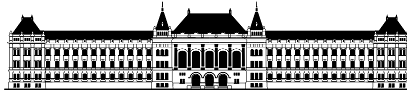



**Budapesti Műszaki és Gazdaságtudományi Egyetem**
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

# Modellellenőrzés és tesztelés: egy kombinált megközelítés szoftverek verifikálására

*Készítette*

Dobos-Kovács Mihály

*Konzulens*

Vörös András
Hajdu Ákos

2018

# Tartalomjegyzék

Tartalomjegyzék ........ 2
Összefoglaló ........ 4
Abstract ........ 5
1. Bevezetés ........ 6
2. Háttérismeretek ........ 8
2.1. Matematikai háttér ........ 8
2.1.1. Matematikai logika ........ 8
2.1.2. Lineáris programozás ........ 9
2.2. Programok formális reprezentációja ........ 10
2.2.1. Control Flow Automata ........ 10
2.2.2. CFA állapottere ........ 11
2.2.3. Modellellenőrzés ........ 12
2.3. Absztrakció-alapú modellellenőrzés ........ 14
2.3.1. CEGAR algoritmus ........ 14
2.3.2. Absztrakció számítása ........ 15
2.3.3. Finomítás ........ 18
2.4. Tesztelés ........ 19
2.4.1. Specifikáció alapú tesztelés ........ 19
2.4.2. Struktúra alapú tesztelés ........ 19
3. Tesztgenerálás absztrakció-alapú modellellenőrzés támogatásával ........ 23
3.1. Áttekintés ........ 23
3.2. CEGAR algoritmus megállási feltétellel ........ 25
3.3. Tesztek generálása ........ 26
3.3.1. Útvonal leképzése lineáris programozási problémára ........ 27
3.3.2. Bemeneti változók határérték analízise ........ 29
3.3.3. Robusztusság tesztelés ........ 31
3.3.4. Számábrázolásból fakadó programhibák tesztelése ........ 32
4. Kiértékelés ........ 35
4.1. Implementáció ........ 35

4.1.1. Forráskód átalakítása ........ 35
4.1.2. Formális verifikáció ........ 36
4.1.3. Tesztek generálása ........ 37
4.1.4. Tesztek futtatása ........ 37
4.2. Esettanulmány ........ 38
4.3. Szoftver verifikációs benchmarkok ........ 41
4.4. Konklúzió ........ 42
5. Összefoglalás ........ 43
5.1. Jövőbeli munka ........ 43
Köszönetnyilvánítás ........ 44
Irodalomjegyzék ........ 45

# Összefoglaló

Különböző szoftveres rendszerek életünk szerves részét képezik. A biztonságkritikus rendszerekben is egyre nagyobb teret kapnak a szoftver alapú megoldások, amelyek gyakran a korábbi, hardver alapú megoldások helyett kerülnek alkalmazásra. Biztonságkritikus környezetben a helyesség biztosítása, illetve az előforduló logikai és programozási hibák megtalálása fontos és nehéz feladat. A gyakorlatban többféle megközelítést is használnak a szoftver hibák felderítésére, továbbá a hibamentesség bizonyítására. A megközelítések egy része leginkább hibák megtalálására használható, míg más megközelítések a helyesség bizonyítására is alkalmasak.

Egy gyakorlatban használt módszer a formális verifikáció, ami a szoftver matematikai modelljét vizsgálja, és matematikailag bizonyítja a specifikációnak megfelelő működést. A formális verifikáció egy számításigényes feladat, hiszen megvizsgálja a szoftver összes lehetséges állapotát, viszont még a legegyszerűbb programoknak is hatalmas lehet az állapottere, vagy sok esetben akár végtelen is. Az elmúlt évtizedben számos megközelítés született, amik hatékonyabbá tették a verifikációs algoritmusokat futási idő és memória felhasználás szempontjából. Ezen előrelépések, és a tudomány gyors fejlődése ellenére sem teljesen megoldott feladat azonban a formális verifikáció: mind az elméleti korlátok, azaz a számításelméleti komplexitásból fakadó nagy számításigény, mind pedig a gyakorlati rendszerek összetettsége, bonyolultsága a sikeres verifikáció ellen dolgoznak.

Egy másik, merőben más megközelítés a tesztelés, ami hatékonyan képes hibákat találni a meglévő rendszerekben. Számításigényét tekintve sokkal szerényebb a formális verifikációnál, és az iparban elterjedten használt, viszont önmagában a helyes működés igazolására nem elegendő.

A munkám célja, hogy ezt a két különböző megközelítést kombináljam, ötvözve a két módszer előnyeit. Bemutatok egy új algoritmust, ami egy absztrakció alapú modellellenőrzési technikát használ a szoftver viselkedésének vizsgálatára. Ha ezek a módszerek képesek bizonyítani a szoftver helyességét, a verifikáció sikeres. Ellenben, ha a valós életben sokkal gyakrabban előforduló eset áll fent, azaz, hogy a formális módszerek sem hiba létét, sem annak hiányát nem voltak képesek igazolni, tesztelési technikák kerülnek alkalmazásra. Ehhez az algoritmus leállítja a formális verifikációt egy bizonyos határ után (ami lehet idő vagy memória korlát), és a formális verifikáció alapján nyert információkat felhasználva teszteket generál. A tesztek generálásához az algoritmus felhasználja az állapottér bejárása során gyűjtött információkat, és az állapottér csak azon részét teszteli, melynek helyességét a formális verifikáció nem tudta igazolni. Ezen kívül külön fókuszálok olyan számábrázolási hibák megtalálására, amelyeket a formális verifikáció önmagában nem tud mindig megtalálni.

Az algoritmusomat az irodalomban fellelhető referencia modellekhez fogom mérni. Remélem, hogy ez az újszerű kombinációja a formális verifikációnak és tesztelésnek segíteni fog a biztonságkritikus szoftverek minőségének javításában.

# Abstract

Software systems are surrounding our life. Even critical infrastructures and safety-critical systems are becoming more and more software driven, where ensuring correctness is an important task. Various approaches exist to find software bugs: formal verification examines the mathematical model of the software and proves the absence of bugs. Formal verification is a computationally expensive task as it explores the possible states of the software, but the state space of even simple software systems can be huge or even infinite. Over the past decade numerous approaches were designed, that were able to reduce the algorithmic complexity of such methods, and the technological development helped to lower the required computation time even further, however such software still exists, that still cannot be formally verified. On the other hand, software testing is an efficient technique to find bugs. It is computationally cheaper than formal verification, and it is widely used by the industry. However testing alone cannot prove correctness.

The goal of my work is to combine formal verification with testing to exploit the advantages of both approaches. I introduce a new algorithm that uses an abstraction-based model checking techniques from the literature to explore the behavior of the software. If the formal verification algorithm can prove the correctness of the system, then the system is verified. However, as it usually happens to real-life problems, formal verification rarely can succeed. In such case, testing needs to be utilized. My algorithm stops the verification after a certain threshold is reached (time or memory limit), and applies test generation. My novel algorithm exploits the result of the formal verification and uses the information gathered during state space traversal: the test generation focuses only on those parts of the state space, which were not fully explored during the model checking. I also introduce improvements to find programming errors that cannot always be found by formal verification.

I will investigate my algorithm on benchmark models from the literature. I hope that the novel combination of formal verification and testing will improve the quality of safety-critical software systems

# 1. Bevezetés

Különböző szoftveres rendszerek életünk szerves részét képezik. Mára már a legegyszerűbb, legkézenfekvőbb otthoni háztartási rendszerekben is megjelentek a különböző szoftver alapú komponensek. Ezzel analóg, hogy az ipar világában is egyre szélesebb körben alkalmaznak szoftveres megoldásokat, még biztonságkritikus környezetekben is. A biztonságkritikus területnek sajátossága, hogy egy-egy esetleges hiba komoly károkat tud okozni, vagy akár emberéletekbe is kerülhet.

Előbbire a tankönyvi példa az European Space Agency (ESA) Ariane 5 nevű rakétájának 1996-os fellövése. A rakéta a fellövését követően nagyjából 40 másodperc múlva letért a kijelölt pályáról, majd felrobbant. A vizsgálatok kimutatták, hogy a hibát az okozta, hogy az egyik komponens a sebességet 64 bites lebegőpontos számként tárolta, míg egy másik komponens 16 bites egész számként várta el. A két formátum között a konverzió nem volt sikeres, ezért a rakéta elvesztette tájékozódási képességét, és megsemmisítette önmagát. Egy évtizednyi, 7 milliárd dollárba kerülő fejlesztés, és 500 millió dollárba kerülő rakomány semmisült meg emiatt az egy hiba miatt.

Sajnos azonban a hibák akár emberéleteket is követelhetnek. Elég csak abba belegondolni, hogy minden atomerőműben előfordulnak szoftveres megoldások, amiknek egy hibája katasztrofális következményekkel járna, de vehetjük a manapság igazán felkapott önvezető autók témáját is. Az többi években többször jelentek meg hírek, hogy önvezető autók szoftverei hibáztak, amik néha emberéleteket is követeltek.

A fenti esetek is bemutatják, hogy a szoftverek ellenőrzése bizonyos körülmények között fontos szerepet kell, hogy betöltsön a fejlesztés folyamatában. Különböző megközelítések léteznek szoftverekben való hibakeresésre. Egyrészről a formális verifikáció, ami a szoftver matematikai modelljét vizsgálja, és matematikailag bizonyítja a specifikáció szerinti működést. A formális verifikáció egy számításigényes feladat, hiszen megvizsgálja a szoftver összes lehetséges állapotát. Viszont még a legegyszerűbb programoknak is hatalmas lehet az állapottere, ha egyenesen nem végtelen. Az elmúlt évtizedben számos megközelítés született, amik csökkenteni tudták ezen feladatok komplexitását, illetve a technológiai fejlődésnek köszönhetően a számításokhoz szükséges idő is csökkent, viszont mindezek ellenére vannak olyan szoftverek, amiknek hibamentes működését még mindig nem tudjuk formális verifikációval igazolni. Egy másik merőben más megközelítés a tesztelés, ami hatékonyan képes hibákat találni a meglévő rendszerekben. Számításigényét tekintve sokkal szerényebb a formális verifikációnál, és az iparban elterjedten használt, viszont önmagában a helyes működés igazolására nem használható.

Mint látható egyik megközelítés sem hibátlan, nem lehet feltétlen alkalmazni kizárólagosan a helyesség bizonyítására. A munkám célja, hogy ezt a két különböző megközelítést kombináljam, ötvözve a két módszer előnyeit, és így javulást érjek el a szoftverellenőrzések hatékonyságában.

A dolgozatban bemutatok egy új algoritmust, ami absztrakció alapú modellellenőrzési technikát használ a szoftver viselkedésének vizsgálatára. Ha ez a módszer képes bizonyítani a szoftver helyességét, vagy ellenpéldát talál és a szoftver hibás voltát, a verifikáció sikeres. Ellenben, ha a valós életben sokkal gyakrabban előforduló eset áll fent, név szerint, hogy a formális módszerek sem hiba létét, sem annak hiányát nem képesek igazolni, tesztelési technikák lesznek alkalmazva. Ehhez az algoritmus leállítja a formális verifikációt egy bizonyos határ után (ami lehet idő vagy memória korlát), és teszteket generál. A tesztek generálásához az algoritmus felhasználja az állapottér bejárása során gyűjtött információkat, és az állapottér csak azon részét teszteli, melynek helyességét a formális verifikáció nem tudta igazolni. Ezen kívül külön fókuszálok számábrázolási hibák megtalálására, amelyeket a formális verifikáció önmagában nem tud megtalálni.

A dolgozatomban a 2. fejezetben bemutatom a dolgozatomban felhasznált háttérismereteket, kitérve a felhasznált formális verifikációs technikára és a tesztelésre. A 3. fejezetben bemutatom a munkámat, ami a formális módszer módosítását, illetve a tesztek generálásának módját takarja. Végül pedig a 4. fejezetben bemutatom a megközelítés alapján elkészült implementációt, és annak gyakorlati eredményeit, korlátait.

# 2. Háttérismeretek

Ebben a fejezetben bemutatom a munkám megértéséhez szükséges háttérismereteket. Kitérek megoldó algoritmusokra, a formális verifikációra és a tesztelésre is.

## 2.1. Matematikai háttér

A dolgozatomban számos formalizmust és algoritmust alkalmazok. A programokat egy gráfként fogom reprezentálni, amiben az utasítások logikai formulákként jelennek meg. Az ellenőrzést egy már létező algoritmus alapjaira fogom építeni, ami absztrakciót használ. Továbbá a tesztgenerálás során logikai megoldó algoritmusokat és a szélsőérték keresésre a lineáris programozást fogom felhasználni.

### 2.1.1. Matematikai logika

A matematikai logikának számos válfaja létezik. Ezek közül a dolgozatban az első rendű logika (First Order Logic) [1] lesz felhasználva. Az elsőrendű logikának ugyan nagy a kifejezőereje, de általános esetben egy ilyen formula kielégíthetősége algoritmikusan eldönthetetlen probléma. A gyakorlatban viszont vannak úgynevezett elméletek (pl. egész aritmetika, tömbök, bitvektorok stb. [2]), amelyek interpretációt adnak a logika szimbólumainak (azaz megszorítják, hogy milyen szimbólumokat használhatunk), és így már gyakran eldönthetővé válik a probléma.

Egy SMT (Satisfiable Modulo Theory) probléma [3] döntési problémát definiál, melyben adott egy elsőrendű logikai kifejezés, és adottak a felhasznált elméletek, és egy megoldó eldönti, hogy létezik-e a kifejezésben lévő változóknak egy olyan behelyettesítése, hogy a logikai kifejezés igazra értékelődjön ki.

Egy *értékadás* egy rendezett pár, aminek első eleme egy a kifejezésben előforduló változó, második eleme pedig a változó értékkészletének egy eleme.

Egy kifejezés *modellje* értékadások olyan halmaza, amiben nincs két azonos változóra vonatkozó értékadás, nincs olyan változó a kifejezésben, ami ne szerepelne egy értékadásban, továbbá ha a kifejezésben lévő összes változót lecseréljük a neki megfelelő értékadásbeli értékre, akkor a kifejezés igazra értékelődik ki.

Azt mondjuk, hogy a kifejezés *kielégíthető*, ha létezik modellje.

Azt mondjuk, hogy a kifejezés *kielégíthetetlen*, ha nem létezik modellje, de létezik egy bizonyítás erre a tényre.

Az SMT problémák megoldására számos szoftver, úgynevezett *SMT megoldó* (SMT solver) [4] létezik. Az SMT megoldók különböző megközelítéseket alkalmaznak, jellemzően más elméletekben (lineáris aritmetika, nemlineáris aritmetika, tömbök, lebegőpontos számok, bitvektorok stb.) erősek és más problémákra hatékonyak. A dolgozat folyamán ezek a szoftverek feketedobozként vannak használva. Különböző problémákat kódolok el SMT problémaként, majd ezeket beadva egy megoldónak felhasználom a kielégíthetőség tényét, illetve gyakran a modelleket is.

**2.1 példa**: *Egy elsőrendű logikai kifejezés például a következő:* $(x < 5 \land x \geq 3 \land y > 7)$*, ahol* $x, y \in \mathbb{Z}$.

*Ebben e kifejezésben egy értékadás az* $(x = 4)$. *Egy modellje a kifejezésnek a* $\{(x = 4), (y = 8)\}$*, hiszen az értékadásokban szereplő változókat behelyettesítve a kifejezésbe igaz értéket kapunk* $(4 < 5 \land 4 \geq 3 \land 8 > 7) = \top$. *Mivel a kifejezésnek létezik modellje, ezért ez egyben kielégíthető is. Megjegyzés, hogy több modellje is létezik, hiszen hasonlóan belátható, hogy* $\{(x = 3), (y = 8)\}$ *is modellje.*

*Példa egy kielégíthetetlen kifejezésre az* $(x > 4 \land x < 5)$*, melynél látható, hogy nincs olyan* x *egész szám, amire ez teljesülne. Ugyanakkor, ha megengedjük a racionális számokat is, már kielégíthetővé válik* $(x = 4.5)$.

### 2.1.2. Lineáris programozás

Egy *lineáris programozási problémánál* [5] adott a kényszerek halmaza, és adott egy célfüggvény.

A kényszerek halmaza *kényszereket* tartalmaz, ahol egy kényszer formája adott: $(a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq L)$. Itt $a_i$ konstans együtthatókat, $x_i$ változókat, $L$ pedig egy adott konstanst jelöl. Azonos változók megjelenhetnek több kényszerben is, míg lehetnek olyan változók, amik csak egy kényszerben jelennek meg.

A *célfüggvény*, a kénszerekben előforduló változók függvénye, $f(x_1, x_2, \ldots, x_m) = b_1x_1 + \cdots + b_mx_m$ formában, ahol $b_i$ konstans együtthatókat, $x_i$ pedig változókat jelöl.

Egy *értékadás* egy rendezett pár, aminek első eleme egy a kényszerekben előforduló változó, második eleme pedig a változó értékkészletének egy eleme.

Egy kifejezés *modellje* értékadások olyan halmaza, amiben nincs két azonos változóra vonatkozó értékadás, nincs olyan változó a kényszerekben, ami ne szerepelne egy értékadásban, továbbá ha a kényszerekben lévő összes változót lecseréljük a neki megfelelő értékadásbeli értékre, akkor az összes kényszer igazra értékelődik ki.

Egy lineáris programozási probléma *megoldása* az a probléma azon modellje, ami mellett a célfüggvény maximális. A célfüggvény minimalizálása is lehet cél, ám ez megfelel a maximalizálás feladatának, ha a célfüggvényt $(-1)$-gyel megszorozzuk.

Az SMT problémákhoz hasonlóan, lineáris programozási problémákhoz is léteznek megoldók [6], amiket a dolgozat folyamán fel fogok használni feketedobozként.

**2.2 példa:** *Vegyük a* $\{(x + y \leq 5), (3x + 2y \leq 20)\}$ *kényszerhalmazt, ahol* $x, y \in \mathbb{N}$*, a célfüggvény pedig legyen* $2x + y$. *Ennek egy megoldása az* $\{(x = 5), (y = 0)\}$ *modell, aminél a célfüggvény értéke* $10$*, ami belátható, hogy maximális.*

A lineáris programozás és az SMT problémák között sok párhuzam húzható, ugyanakkor vannak jelentős különbségek. Egy SMT problémában egyszerre többféle elmélet lehet alkalmazva, illetve a formulák tetszőleges logikai operátorokkal, kvantorokkal kombinálhatók. Ezzel szemben lineáris programozásban csak valós vagy egész számokkal lehet dolgozni, és a kényszereknek mind teljesülnie kell. Továbbá utóbbiban

további megkötések vannak az eredmény modell tulajdonságaira, azon kívül, hogy illeszkedjen a kényszerekre.

## 2.2. Programok formális reprezentációja

Ebben a fejezetben bemutatom az irodalomban használt formális modellt, amelyen a formális verifikációs algoritmusok és a teszt generáló megközelítésem is működik.

### 2.2.1. Control Flow Automata

Számítógépes programok sokféle formában jelenhetnek meg, többek között például lehet tekinteni a forráskódot. Ezt a fejlesztő írja, ami érthető és olvasható, illetve lehet tekinteni az ebből létrejövő binárist, ami már a fejlesztő számára olvashatatlan, viszont a számítógép számára érthető, és végrehajtható. Ahhoz, hogy a későbbiekben formális verifikációt tudjunk alkalmazni a szoftverre, meg kell teremteni egy olyan formáját a szoftvernek, ami a matematika eszköztára számára könnyen érthető, feldolgozható.

Az egyik ilyen formának a neve a *control flow automata* (továbbiakban CFA) [7]. Egy CFA egy $(V, L, l_0, E)$ négyes, melynek elemei:

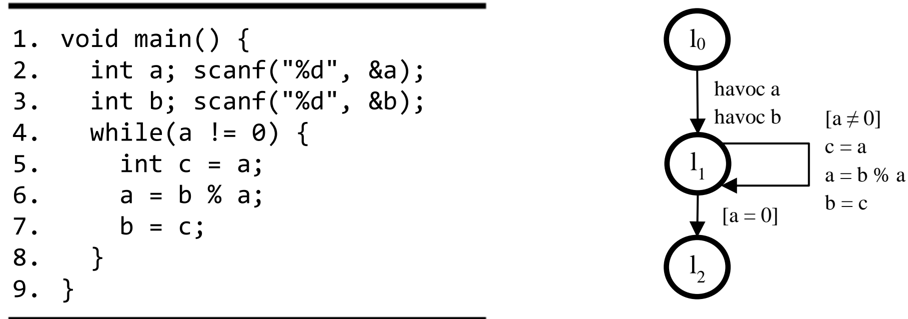
- $V = \{v_0, v_1, \dots\}$ a programban megjelenő *változók* halmaza. Minden $v_i \in V$ változónak van egy $D_{V_i}$értékkészlete.
- $L = \{l_0, l_1, \dots\}$ a program *vezérlési helyeinek* halmaza. Tulajdonképpen a program számláló különböző értékeit modellezi.
- $l_0 \in L$ a *kezdő helyzet*, azaz az a vezérlési hely, amit a program indulásakor felvesz.
- $E \subseteq L \times Ops \times L$ az *átmenetek* halmaza. Egy átmenet egy irányított él két vezérlési hely között, ami fel van címkézve egy művelettel. Művelet két féle lehet:
    - Lehet *értékadás* ($x := x + 1$). Értékadás hatására a baloldalon álló változó értéke módosul a jobb oldalon szereplő kifejezés alapján. Speciális értékadás a *nem determinisztikus értékadás* ($havoc\ x$), amikor a változó értéke bármi lehet, amit a változó értékkészlete megenged. Ez utóbbival modellezhető többek között a felhasználótól érkező nemdeterminisztikus bemenet.
    - Lehet *őrfeltétel* ($[x > 0]$). Egy őrfeltétel esetén pedig csak akkor hajtható végre az átmenet a két vezérlési hely között, ha az őrfeltételben szereplő kifejezés igazra értékelődik ki a változók aktuális értéke mellett.

A dolgozat során egy CFA ábrázolásánál a vezérlési helyeket körökkel, az átmeneteket a megfelelő körök közötti nyilakkal fogom jelölni, a műveleteket a megfelelő nyilak mellé írom. A kezdő helyzet az a kör, amibe egyetlen nyíl sem vezet be.

Mivel egy program reprezentálható egy CFA formában, a továbbiakban, hacsak külön említésre nem kerül, a kettő fogalom egymással felcserélhető.

**2.3 példa:** *A 2.1 ábrán látható egy példa C program, és az ahhoz tartozó CFA. Az ábrán látszik, hogy a strukturált programozás elemei (szekvencia, elágazás, ciklus)*

*leképezhetők CFA-ra. Továbbá látszik, hogy a felhasználótól érkező nemdeterminisztikus bemenetet a havoc utasítással lehet jelölni.*




**2.1 ábra**: Egy C-ben írt mintaprogram, és az annak megfelelő CFA.

### 2.2.2. CFA állapottere

Egy program matematikai reprezentációjához szorosan hozzákapcsolódik az állapottér fogalma. Egy program *állapottere*, a program lehetséges állapotainak a halmaza. A program aktuális állapotát két dolog jellemzi:

- Az aktuális vezérlési hely ($l_i \in L$)
- A program változóinak aktuális értéke $(d_1, d_2, \dots, d_n)$, ahol $d_i \in D_{v_i} \land v_i = d_i$

Tehát formálisan a program egy *(konkrét) állapota* egy n-es $(l_i, d_0, \dots, d_n)$, ahol $l_i \in L$ a vezérlési hely, és $n = |V|$, míg $d_i \in D_{v_i}$ pedig a $v_i$ változó által felvett érték.

Egy CFA esetén egyértelműen ki volt jelölve a kezdő vezérlési hely, azonban ez az állapottér esetében nincs így. Egyrészt azért, mert egy CFA esetében, amennyiben egy változó kezdő értékéről nem nyilatkozik (vagy még nem lett létrehozva), vagy nemdeterminisztikus értékadás történt, a változók értéke az értékkészlet megszorításain belül bármi lehet. Ennek következtében az állapottérben kezdő állapotnak minősül minden olyan állapot, aminél az aktuális vezérlési hely $l_0$.

A program *átmenetei* a következőképpen képződnek le az állapottérre. Adott az aktuális állapot $(l_i, d_0, \dots, d_n)$, és egy a vezérlési helynek megfelelő átmenet $(l_i, op, l_i') \in E$. Ekkor az $op$ típusától függően:

- Ha $op$ értékadás $v_k \coloneqq expr$ formában: Ekkor a következő állapot $(l_i', d_0', \dots d_n')$, akkor $d_i' = d_i$, kivéve $d_k$, aminek az értéke $expr$ kiértékelve $d_0, \dots d_n$ változókkal. Másképpen az aktuális állapot értékeit behelyettesítjük a kifejezésbe, és ennek az értéke lesz a megadott változó új értéke. A többi változó értéke változatlan.
- Ha $op$ nemdeterminisztikus értékadás $v_k$ változónak ($havoc\ v_k$): Ekkor több következő állapot van. Ezek $(l_i', d_0', \dots, d_n')$, ahol $d_i' = d_i$, kivéve $d_k$, amire igaz, hogy $d_k \in D_{v_k}$. Másképpen a változó értékkészletének összes lehetséges értékére létrehozunk egy következő állapotot.

- Ha $op$ őrfeltétel $[cond]$ formában: Ekkor a következő állapot $(l_i', d_0, \ldots, d_n)$, feltéve, hogy $cond$ igazra értékelődik ki $d_0, \ldots, d_n$ alapján. Ha hamisra, akkor nincs következő állapot.

**2.4 példa:** *Legyen az aktuális állapot* $(\mathrm{l}_0, 3, 4)$, *ahol az n-es utolsó két eleme két változó (sorban* $\mathrm{x}, \mathrm{y}$*) jelenlegi értékének felel meg, és legyen egy átmenet* $(\mathrm{l}_0, \mathrm{op}, \mathrm{l}_3)$. *Ekkor a következő állapot* op *típusától függően:*

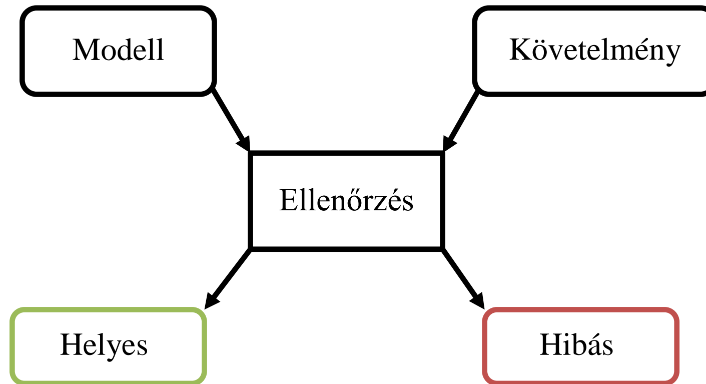
- *Ha* op *értékadás* $\mathrm{x} \coloneqq 2$ *formában: a következő állapot* $(\mathrm{l}_3, 2, 4)$.
- *Ha* op *nem determinisztikus értékadás* $havoc\ x$ *formában: a következő lehetséges állapotok* $\{(\mathrm{l}_3, -\infty, 4), \ldots, (\mathrm{l}_3, 0, 4), (\mathrm{l}_3, 1, 4), \ldots, (\mathrm{l}_3, \infty, 4)\}$, *ha* x *értékkészlete az egész számok halmaza.*
- *Ha* op *őrfeltétel* $[\mathrm{y} = 4]$ *formában: a következő állapot* $(\mathrm{l}_3, 3, 4)$
- *Ha* op *őrfeltétel* $[\mathrm{y} = 5]$ *formában: nincs következő állapot*

Összefoglalva egy CFA a program struktúráját és lehetséges műveleteit kódolja el. Ezzel szemben az állapottérben szerepel az összes olyan állapot, amiben a program a futása során lehet, azaz megjelennek a változók lehetséges értékei, viszont a program struktúrája nehezen olvashatóvá válhat. Az állapotér méretét drasztikusan befolyásolják a nemdeterminisztikus értékadások, és így egy egyszerű program állapottere is lehet hatalmas. Vegyük példának az Euklideszi-algoritmust. Ennek a bemenete két pozitív egész szám, és az algoritmus megadja a legnagyobb közös osztót. Csak ahhoz, hogy a változók értékének bekérését modellezzük, szükség van $2^{32}\ 2^{32} = 2^{64} \approx 10^{19}$ állapotra az állapottérben, feltéve, hogy az egész számokat 32 biten tároljuk. Ez az állapottérrobbanás tipikus esete, amit kezelni kell.

### 2.2.3. Modellellenőrzés

A formális verifikáció számos módszer gyűjtőneve, amiket különféle rendszerek helyességének matematikai bizonyítására lehet használni. Ezek közül az egyik a *modellellenőrzés* [8] [9].

A modellek ellenőrzésének a feladata, hogy ha adott egy *modell*, illetve egy *követelmény*, akkor megvizsgálja, illetve bebizonyítsa, hogy a modell teljesíti-e az adott követelményt. Azt mondhatjuk, hogy egy modell *helyes*, ha matematikai bizonyítás létezik arra, hogy a megadott követelmény teljesül. Ezzel analóg módon, azt mondhatjuk, hogy egy modell *hibás*, ha matematikai bizonyítás létezik arra, hogy előfordulhat, hogy a megadott követelmény nem teljesül. Itt valójában ez a bizonyítás lehet egyetlen ellenpélda, ami megsérti a követelményt.

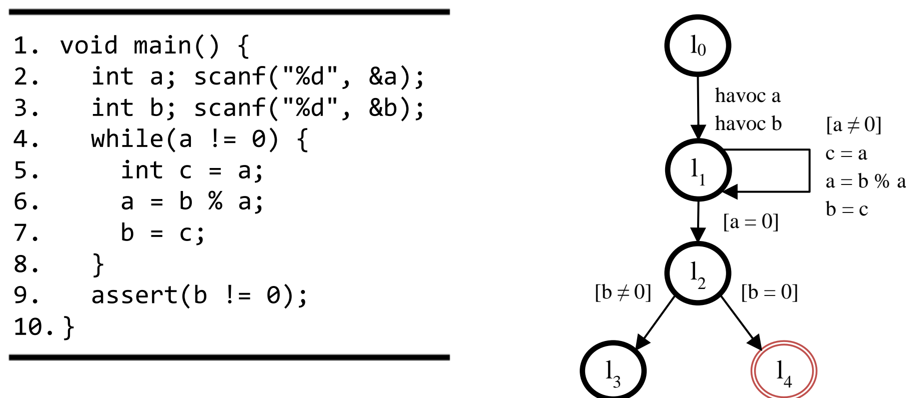



**2.2 ábra:** A modellellenőrzés sematikus ábrája

Mint látható, a fenti megfogalmazás meglehetősen tág teret enged azt tekintve, hogy milyen technikákat lehet modellellenőrzésnek tekinteni. Nem határozza meg sem a modell, sem a követelmények formáját, sem az ellenőrző algoritmusokat. A jelenlegi munkában a következő formában alkalmazzuk a technikát szoftverekre:

- A *modell* legyen a szoftvernek megfeleltethető CFA.
- A *követelmény* legyen az, hogy nem érhető el egyetlen hibás hely sem. A *hibás hely* egy CFA-ban megjelölt vezérlési hely, amit ha a program elér, hiba történt. Ezek a hibás helyek többek között automatikusan képezhetők a programban elhelyezett assertion-ökből.
- Az *ellenőrzés* pedig egy olyan módszer, ami el tudja dönteni, hogy lehet-e úgy értéket adni a program változóinak, hogy annak hatására a program belépjen az egyik hibás helyre. Ha lehet így értéket adni a változóknak, akkor a program *hibás* (és erre a bizonyíték az értékadás), ha nem lehet így értéket adni, akkor a program *helyes*.

A dolgozatban megjelenő ábrákon a hibás helyeket dupla szegélyű körrel fogom jelölni.



```
1. void main() {
2.   int a; scanf("%d", &a);
3.   int b; scanf("%d", &b);
4.   while(a != 0) {
5.     int c = a;
6.     a = b % a;
7.     b = c;
8.   }
9.   assert(b != 0);
10.}
```



**2.3 ábra:** Egy minta C program és az annak megfelelő CFA. A programban az assert makróval fogalmaztuk meg a követelményeket.

**2.5 példa:** *A 2.3 ábrán látható egy egyszerű C program egy assertion-nel, és annak a leképzése CFA-ra. Egy assertion egy olyan predikátum, aminek a kiértékelésekor mindig igaznak kell lennie. Másként megfogalmazva, ha a predikátum nem igaz, a program hibás. Ezt a C nyelvben az assert makróval lehet megfogalmazni, ami egy sérült assertion esetén megszakítja a program futását. A CFA-ban egy assertion a látható módon képezhető le. Adott egy elágazás ($l_2$) és két vezérlési hely ($l_3, l_4$). Az egyikbe akkor lép a vezérlés, ha a predikátumból képzett őrfeltétel teljesül. A másikba, ami egy hibaállapot, akkor lép a rendszer, ha a predikátumból képzett őrfeltétel hamis. Mivel ez hibaállapot, itt véget ér a CFA, míg a másik ágon általános esetben még folytatódhat.*

## 2.3. Absztrakció-alapú modellellenőrzés

A helyesség igazolásának egyik módja, hogy az állapottér összes állapotát bejárva (explicit modellellenőrzés) vizsgáljuk a hibás vezérlési hely elérhetőséget. Ennek viszont van egy sajnálatos hátulütője, mégpedig, hogy egyszerű programnak is lehet hatalmas állapottere. Ezt az állapottérrobbanást többek között absztrakció alkalmazásával lehet mérsékelni.

### 2.3.1. CEGAR algoritmus

Az absztrakciót használó modellellenőrző algoritmusok egyik kategóriája az ellenpélda vezérelt absztrakció-finomítás algoritmusa (counterexample-guided abstraction refinement, továbbiakban CEGAR) [10] [7] [11], amit a dolgozatban a programok ellenőrzésére felhasználtam.

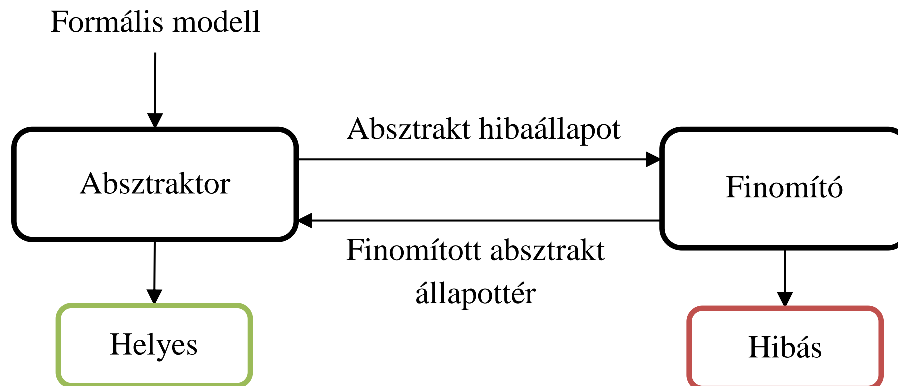
Az algoritmus az *absztrakt állapottérben* dolgozik, ami absztrakt állapotok és átmenetek halmaza. Egy *absztrakt állapot* több (akár végtelen) konkrét állapotot foglalhat magába. Egy konkrét állapot egyszerre csak egy absztrakt állapotban jelenhet meg, és minden konkrét állapotnak meg kell jelennie legalább egy absztrakt állapotban. Egy *absztrakt hibaállapot* egy olyan absztrakt állapot, amiben van olyan konkrét állapot, aminek vezérlési helye hibahely. Általában, bár nem kizárólag, egy absztrakt állapotban olyan konkrét állapotok vannak, amiknek a vezérlési helye megegyezik.

Az algoritmus magját az ún. CEGAR-hurok (**2.4** ábra) adja. A hurok első felében vizsgálni kell, hogy elérhető-e bármely absztrakt hibaállapot. Mivel egy absztrakt hibaállapot a létező hibaállapotok felülbecslése, ezért ha nem érhető el egyik sem, biztosan nem érhető el egy hibaállapot sem, tehát a program helyes.

Viszont ha elérhető, meg kell vizsgálni, hogy valójában az absztrakt hibaállapoton belül, a konkrét hibaállapot elérhető-e. Ez utóbbi a hurok második felének a feladata. Amennyiben elérhető, úgy a program hibás. Ha nem, úgy az absztrakció nem elég precíz, hiszen elérhető olyan absztrakt hibaállapot, amiben egyetlen konkrét hibaállapot sem érhető el. Ezt követően finomítani kell az absztrakciót, hogy az absztrakt hibaállapot ne tartalmazza ezt az elérhetetlen hibaállapotot.

Ezt követően a hurok ismétli önmagát, mindaddig, amíg a helyességet be nem látja, vagy hibát nem talál. Legszélsőségesebb esetben az állapottér addig finomodik, amíg minden

absztrakt állapot csupán egy konkrét állapotot tartalmaz, azaz visszakapjuk az állapotteret. Ebben pedig a helyesség egyértelműen eldönthető.




**2.4 ábra:** CEGAR algoritmus

Az algoritmust technikailag két részben szokás megvalósítani, a huroknak megfelelően. Ez a két rész az absztraktor, és a finomító.

### 2.3.2. Absztrakció számítása

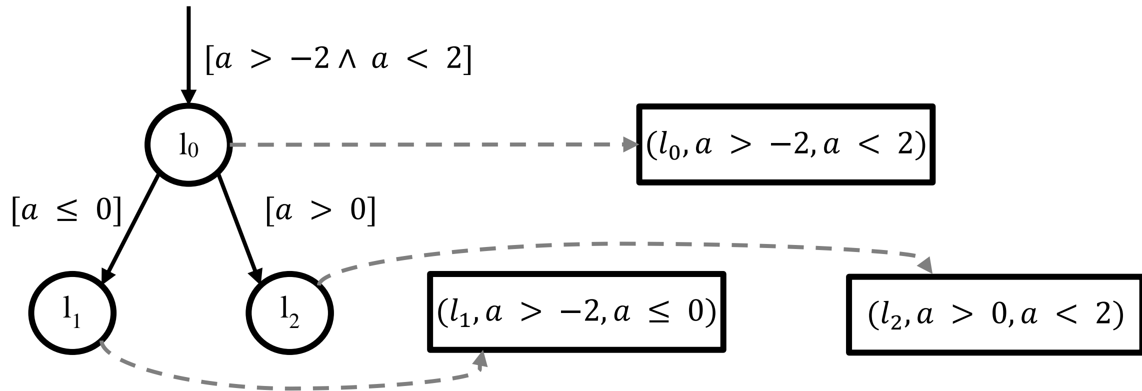
A CEGAR algoritmus formális modellje többek között lehet program is, amit a továbbiakban feltételezünk is. Program esetén az absztrakció számítására több, különböző módszer létezik, ezek közül az egyik a *predikátum absztrakció* [12], amit a dolgozatomban felhasználtam. A predikátum absztrakció úgy próbálja meg csökkenteni az állapottér méretét, hogy a változók konkrét értékei helyett, predikátumok halmazának teljesülését igyekszik követni. Predikátum absztrakció során az absztrakt állapotok olyan konkrét állapotokat tartalmaznak, amiknek a vezérlési helye megegyezik, és ugyanazon kifejezések (predikátumok) teljesülnek rájuk.

Egy *predikátum* egy logikai kifejezés a program változói felett, ami a változók közötti kapcsolatokat rögzít. (Például $(x = 0), (y + 2 < x)$ stb.) A követett predikátumok halmaza (másnéven *pontosság*) $P = \{p_0, \dots, p_n\}$.

Egy absztrakt állapot egy $(l_i, \widehat{p_0}, \dots, \widehat{p_k})$ n-es, ahol $l_i \in L$, a vezérlési hely, $\widehat{p_i} = p_i$ ha a predikátum ponált formában szerepel az adott állapotban, $\widehat{p_i} = \neg p_i$, ha a predikátum negált formában szerepel az adott állapotban, és $\widehat{p_i} = true$, ha nem szerepel a predikátum az adott állapotban. Egy absztrakt állapot magába foglalja az összes olyan állapottérbeli állapotot ahol a vezérlési helyek megegyeznek, és az absztrakt állapot predikátumai igazra értékelődnek ki az állapot változók értékei alapján. Ezt másként úgy is lehet nevezni, hogy az $l_i$ állapot fel van címkézve $p_i, \dots p_j$ predikátumokkal.

**2.6 példa:** *Legyen két predikátum* $\mathrm{p_1 = (x = 0), p_2 = (y + 2 < x)}$. *Ekkor* $\mathrm{p_1}$ *jelenti, hogy* $\mathrm{(x = 0)}$, $\neg\mathrm{p_1}$ *pedig, hogy* $\mathrm{(x \neq 0)}$. *Hasonlóan* $\mathrm{p_2}$ *jelenti, hogy* $\mathrm{(y + 2 < x)}$, $\neg\mathrm{p_2}$ *pedig, hogy* $\mathrm{(y + x \geq x)}$.

A dolgozatomban az absztrakt állapotokat téglalapokként jelölöm, bennük megjelenítve a vezérlési hely, és az aktuális predikátumok.




**2.5 ábra:** Egy CFA részlet, és a vezérlési helyekhez tartozó absztrakt állapotok

**2.7 példa:** *A 2.5 ábra egy CFA egy részletét tartalmazza, és mindhárom vezérlési helyhez, egy hozzá tartozó absztrakt állapotot. Tételezzük fel, hogy a program egyetlen változója* a. *Az* $\mathrm{l}_0$*-hoz tartozó állapot három konkrét állapotot tartalmaz:* $(\mathrm{l}_0, -1), (\mathrm{l}_0, 0), (\mathrm{l}_0, 1)$. *Az* $\mathrm{l}_1$*-hez tartozó absztrakt állapot két konkrét állapotot tartalmaz:* $(\mathrm{l}_1, -1), (\mathrm{l}_1, 0)$, *míg az* $\mathrm{l}_2$*-hőz tartozó állapot egy konkrét állapotot tartalmaz:* $(\mathrm{l}_2, 1)$.

Az absztrakt állapottér felépítésének szabályai némileg eltérnek az állapotér felépítésétől. A program kezdetén még egyik változó sincsen létrehozva, így nincs olyan predikátum, ami jellemezhetné. Ezért egyetlen kezdő állapot van predikátumok nélkül, $(l_0)$, hiszen ez a CFA belépési pontja.

Ha az aktuális állapot $(l_i, \widehat{p_0}, \dots, \widehat{p_k})$, és adott egy a vezérlési helynek megfelelő $(l_i, op, l_i')$ átmenet, akkor az $op$ típusától függően:

- Ha $op$ értékadás $v_k \coloneqq expr$ formában: Ekkor a következő állapot vezérlési helye $l_i{}'$, a predikátumai pedig azok a $P$-beli predikátumok, amiket az aktuális állapot predikátumai, és az értékadás implikálnak.
- Ha $op$ nemdeterminisztikus értékadás $v_k$ változónak: Ekkor a követező állapot vezérlési helye $l_i{}'$, a predikátumai pedig azok a $P$ beli predikátumok, amiket az aktuális állapot predikátumai és az értékadás implikálnak. Értelemszerűen, a következő állapotban nem szerepelhet predikátum $v_k$ értékére, mivel még semmi nem ismert róla.
- Ha $op$ őrfeltétel $[cond]$ formában: Ekkor meg kell vizsgálni, hogy az aktuális állapot predikátumai ellentmondásban vannak-e $cond$ feltétellel. Ellentmondás van, ha nem lehet úgy megválasztani a változók értékét ebben a feltételrendszerben, hogy minden feltétel igaz legyen. Ha van ellentmondás, nincs következő állapot. Ha nincs, akkor a következő állapot vezérlési helye $l_i{}'$, a predikátumai pedig azok a $P$-beli predikátumok, amiket az aktuális állapot predikátumai, és az őrfeltétel implikálnak.

A gyakorlatban az implikáció számítására, ellentmondások megállapítására és az állapothalmazok hatékony kezelésére predikátumok segítségével SMT megoldókat lehet használni [13].

**2.8 példa:** *Legyen az aktuális állapot* $(\mathrm{l}_0, \mathrm{x} > 0, \mathrm{y} < 4)$, *a vezérlési helynek megfelelő művelet pedig* $(\mathrm{l}_0, \mathrm{op}, \mathrm{l}_1)$. *Ekkor a következő állapot* op *típusától függően:*

- *Ha* op *értékadás* $\mathrm{x} \coloneqq \mathrm{x} + 1$ *formában: Ekkor a következő állapot* $(\mathrm{l}_1, \mathrm{x} > 1, \mathrm{y} < 4)$, *feltéve, hogy* $(x > 1) \in P$, *hiszen* $(\mathrm{x} > 0) \wedge (\mathrm{x} = \mathrm{x} + 1) \Rightarrow (\mathrm{x} > 1)$
- *Ha* op *nemdeterminisztikus értékadás* x*-re: Ekkor a következő állapot:* $(\mathrm{l}_1, \mathrm{y} < 4)$.
- *Ha* op *őrfeltétel* $[\mathrm{x} > 3]$ *formában: Ekkor a következő állapot* $(\mathrm{l}_1, \mathrm{x} > 3, \mathrm{y} < 4)$, *feltéve, hogy* $(x > 3) \in \mathrm{P}$, *hiszen* $(\mathrm{x} > 0) \wedge (\mathrm{x} > 3) \Rightarrow (\mathrm{x} > 3)$.
- *Ha* op *őrfeltétel* $[\mathrm{x} < 0]$ *formában: Ekkor nincs következő állapot, hiszen* $(\mathrm{x} > 0)$ *és* $(\mathrm{x} < 0)$ *egyszerre nem lehet igaz, ellentmondás van.*

*A fenti implikációk mind egyben elsőrendű logikai kifejezések, így egy megoldónak átadhatók.*

Az absztrakció-alapú modellellenőrzés során a dolgozatban predikátum absztrakciót fogok alkalmazni. Ekkor az absztrakció számításához adott a pontosság a követett predikátumok halmazaként, és az absztrakt állapottér egy részhalmaza.

A program ellenőrzésének kezdetén, csupán egy absztrakt állapot van az állapottérben, ahonnan elindul az absztrakt állapottér felépítése. Ez a CFA kezdő állapotának megfelelő absztrakt állapot, amin nincs egyetlen predikátum sem.

Adott egy *kifejtés (expand)* művelet, ami megadja egy $s_i$ absztrakt állapotra az összes rá következő absztrakt állapotot. A művelet szisztematikusan veszi az összes, $s_i$ vezérlési helyére illeszkedő átmenetet, és az állapot valamint az előbb bemutatott átmenet alapján meghatározza a következő állapotok halmazát.

Legyen az *absztrakt elérési fa (abstract reachability tree)* egy olyan fa, aminek a gyökere a CFA kezdő állapotának megfelelő absztrakt állapot, és minden csomópontjára igaz, hogy ha vannak gyerekei, akkor azok azon absztrakt állapotok, amik a csomópontra meghívott kifejtés művelet eredményez.

Emellett adott egy *fedési reláció (covering relation)*. Tegyük fel, hogy van egy levél $L = (l_l, p_i, \dots, p_j)$ az eddig felépített absztrakt elérési fában, és egy olyan absztrakt állapot $S = (l_p, p_k, \dots, p_l)$ a fa gyökerébe vezető úton, amire igaz, hogy $l_l = l_p$, és $(p_i, \dots, p_j) \Rightarrow (p_k, \dots, p_l)$, ahol $\Rightarrow$ az implikáció művelete, akkor azt mondhatjuk, hogy $S$ fedi $L$ állapotot, vagy $L$ fedve van $S$ által. Ennek szemléletes jelentése, hogyha ugyanahhoz a vezérlési helyhez két absztrakt állapot is tartozik, akkor, ha a később kifejtett szigorúbb, nem érhető el belőle olyan állapot, ami ne lenne elérhető a korábban kifejtettből. Ezáltal a fedett állapotokra nem szükséges alkalmazni a kifejtést.

Az absztrakt elérési fát a fenti két művelet segítségével lehet építeni. A fa minden csomópontja lehet befejezett, vagy nem befejezett. *Befejezett* egy csomópont, ha már ki lett fejtve, vagy le van fedve, egyéb esetben *nem befejezett*. A fát addig lehet építeni, amíg

egy absztrakt hibaállapot nem lesz a kifejtés eredménye, vagy amíg minden csomópont befejezett nem lesz. Előbbi esetben el kell dönteni, hogy a hibaállapot a nem megfelelő pontosság eredménye, vagy esetleg egy konkrétan elérhető hibaállapotot takar, de ez már a finomító feladata.

```
1.  Abstractor(ARG,PREC):
2.    R := { ARG absztrakt állapotai }, W := { ARG nem befejezett állapotai }
3.    AMÍG W nem üres ÉS nincs absztrakt hibaállapot ARG-ban ADDIG
4.      s := W egyik eleme
5.      W := W \ {s}
6.      HA ∃r ∈ R amire r fedi s-t AKKOR
7.        fedd le s-t r-rel
8.      HA s nincs fedve AKKOR
9.        e := s kifejtése
10.       R := R ∪ e
11.       W := W ∪ e
12.   HA absztrakt hibaállapot találása miatt állt le a ciklus AKKOR
13.     ← ABSZTRAKT HIBAÁLLAPOT
14.   KÜLÖNBEN
15.     ← HELYES
```

**2.1 pszeudokód:** Az absztrakció építése. Az $ARG$ az eddig felépített absztrakt elérési fa, $PREC$ a jelenlegi pontosság.

Az építés algoritmusa alapvetően egy várólistát kezel. A várólistába kezdetben belekerülnek az absztrakt elérési fa nem befejezett állapotai. A várólistából több különböző stratégia mentén lehet kiszedni az elemeket (szélességi, mélységi bejárás stb.)

### 2.3.3. Finomítás

A finomítás művelete során adott egy absztrakt hibaállapot, illetve az aktuális pontosság.

Elsőként meg kell vizsgálni, hogy az absztrakt hibaállapot konkrét hibaállapotot takar-e. Ennek egyik módja, hogy a kezdő állapotból a hibaállapotig vezető úton lévő értékadásokat, őrfeltételeket, illetve predikátumokat megfogalmazzuk egy SMT problémaként. Ekkor a probléma kielégíthetősége azt mutatja, hogy létezik olyan változó behelyettesítés, amivel a konkrét hibaállapot elérhető. Ebben az esetben a program hibás, amire bizonyíték a behelyettesítés.

Amennyiben viszont a probléma kielégíthetetlen, több teendő is van. Egyrészt a pontosságot kell finomítani, azaz újabb predikátumokat felvenni. Ennek kapcsán szintén több stratégia létezik, az egyik például, hogy a kielégíthetetlenség bizonyítékát használja fel új predikátumok gyártására [14]. Másrészt pedig az elérhetetlen állapotokat el kell távolítani az állapottérből, másnéven *visszavágni* az állapotteret.

Miután a finomítás véget ért, folytatódhat az absztrakció építése.

A dolgozatomban a finomítást csupán feketedobozként használom, ezért részletesebben nem fejtem ki a tulajdonságait.

## 2.4. Tesztelés

Egy szoftver tesztelésének a célja, hogy megtalálja a szoftverben megbúvó esetleges hibákat, bizonyosságot nyújtson a szoftver minőségét illetően. A formális verifikációs módszerekkel ellentétben széleskörűen elterjedt a gyakorlatban, módszerei kiforrottabbak.

Egy *teszteset* a bementi értékek és végrehajtási előfeltételek, valamint a várt eredmények, és végrehajtási utófeltételek halmaza. A *tesztkészlet* az egyazon programon értelmezett tesztesetek halmaza [15].

Egy teszteset eredménye lehet:

- Sikeres: a program futása konzisztens a várt eredményekkel és végrehajtási utófeltételekkel.
- Sikertelen: a program futása nem konzisztens a várt eredményekkel és végrehajtási utófeltételekkel.
- Nem meggyőző: olyan eredmény, amiről nem eldönthető, hogy sikeres vagy sikertelen-e (például meg van adva a sikeres, és sikertelen kimenetek halmaza, de egy produkált kimenet nem tartozik egyikbe sem.
- Hiba: a teszt futása során hiba történt, emiatt nem eldönthető a sikeresség.

### 2.4.1. Specifikáció alapú tesztelés

A *specifikáció alapú tesztelés* a rendszert fekete dobozként kezeli. Ismert információnak csak a specifikációban rendelkezésre álló adatokat veszi, ami lehet például a bemeneti változók értékkészlete, és feltételek, amiket a kimeneti változóknak teljesíteniük kell.

Egy gyakran használt specifikáció alapú módszer *a határérték analízis* [15]. Ez pont a bemeneti változók értékkészletét veszi alapul. Egy változónak értéke lehet minimális, maximális értéke, illetve minden más esetbe nominális. A tesztelési technika filozófiája, hogy a hibák nagyobb valószínűséggel fordulnak elő a változók szélsőértékénél, mert a szélső esetek kezelésére gyakran kell egyéni megoldást nyújtani, amit a fejlesztő elfelejtett megtenni.

A határérték analízis során két teszteset generálható úgy, hogy egy változóhoz először a maximális, aztán a minimális értékét rendeljük, az összes többihez pedig a nominálisat. A határértékek keresése ezt követően ismételhető az esetleges ekvivalencia partíciók mentén.

### 2.4.2. Struktúra alapú tesztelés

A *struktúra alapú tesztelés* a rendszert fehér dobozként teszteli. Ismert információnak tekinti a program felépítését és vezérlési folyamát. Az erre alapuló technikák célja, hogy a generált tesztkészlet tesztjeit futtatva, a vezérlési folyam helyeinek és átmeneteinek minél nagyobb százaléka kerüljön végrehajtásra.

Egy gyakran használt struktúra alapú módszer a *szimbolikus végrehajtás* [16] [17]. Szimbolikus végrehajtás során, az egyes változók konkrét értékei helyett, a *szimbolikus*

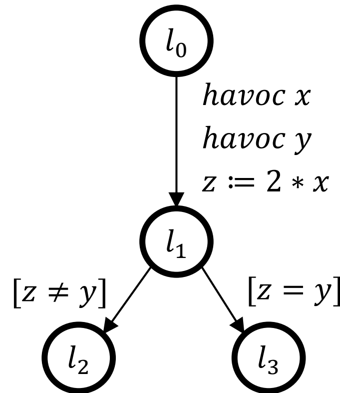
*értékek* lesznek követve, míg az egyes kifejezések helyett a szimbolikus változók felett értelmezett szimbolikus kifejezések.

Egy-egy teszteset a program állapotterében egy konkrét utat jár be. A szimbolikus végrehajtás feladata, hogy olyan teszteseteket állítson elő, amik az összes lehetséges útvonalat bejárják a programon belül. A lehetséges útvonalakat elsősorban a programban lévő elágazások határozzák meg. A szimbolikus végrehajtás sajnos szenved az útvonal-robbanás (path explosion) problémájától, hiszen kis programoknak is nagyon sok, vagy akár végtelen különböző lefutási útvonala lehet.

A szimbolikus végrehajtás karbantart egy *szimbolikus állapotot* ($\delta$), ami minden változóra megadja a jelenlegi szimbolikus értékét, valamint egy *szimbolikus útvonal kényszert* ($PC$), ami egy elsőrendű logikai kifejezés, a szimbolikus változók felett. Az ellenőrzés kezdetén, $\delta = \{\}$ és $PC = \top$. A program bejárása során, a minden művelet hatására módosítani kell a két karbantartott struktúrát. A művelet típusától függően:

- Ha a művelet nemdeterminisztikus értékadás $havoc\, x$ formában, akkor a szimbolikus állapotba fel kell venni egy új, még nem használt szimbolikus értéket. Tehát $\delta(x) = x_i$, ahol $x_i$ nem fordult még elő korábban.
- Ha a művelet értékadás $x \coloneqq expr$ formában, akkor a szimbolikus állapotot frissíteni kell. Tehát $\delta(x) = expr'$, ahol $expr'$ az $expr$ kifejezésből lett képezve, csupán le lett benne cserélve minden változó, a neki megfelelő $\delta$-beli szimbolikus értékre.
- Ha a művelet őrfeltétel $[cond]$ formában, akkor az útvonal kényszer frissítése szükséges. Legyen az új útvonal kényszer $PC' = PC \land cond'$, ahol $cond'$ a $cond$ kifejezésből lett képezve, csupán le lett benne cserélve minden változó, a neki megfelelő $\delta$-beli szimbolikus értékre. Egy őrfeltétel esetén gyakori minta, hogy a program többfelé ágazik (if-else minta). Valójában ezeknél az elágazásoknál keletkeznek lehetséges útvonalak.

A szimbolikus végrehajtás végén, minden lehetséges útvonal mentén meg lesz oldva az útvonal kényszer egy SMT megoldó segítségével, ami megad egy-egy konkrét értéket, amit a bemeneti változóknak fel kell vennie, hogy a program az adott útvonalon haladjon végig.




**2.6 ábra:** Egy példa CFA egy elágazással

**2.9 példa:** *Vegyük a 2.6 ábrán látható CFA-t. Kezdetben* $\delta = \{\}, TC = \top$*, a végrehajtás* $l_0$*-n áll. Lépjen a végrehajtás* $l_1$*-be, a három műveletet végrehajtva. A* $havoc\ x$ *művelet hatására, egy új szimbolikus érték jön létre* $x$ *változóhoz, legyen ez* $x_0$*. Az* $y$ *változóhoz hasonlóan jöjjön létre* $y_0$ *szimbolikus érték. A* $z$ *változóhoz legyen rendelve a* $2 * x$ *kifejezés, amiben minden változó legyen lecserélve a neki megfelelő szimbolikus értékre, azaz* $2 * x_0$*. Összefoglalva* $l_1$*-ben* $\delta(x) = x_0, \delta(y) = y_0, \delta(z) = 2 * x_0, TC = \top$*. Ezután menjen a végrehajtás* $l_2$*-be. Ekkor hozzá kell venni az útvonal kényszerhez az őrfeltételt, de a benne lévő változókat le kell cserélni a nekik megfelelő szimbolikus értékekre. Azaz* $TC = \top \wedge (2 * x_0 \neq y_0)$*. Mivel itt a lehetséges útvonal véget ér, meg kell oldani az útvonal kényszert. A kényszernek jelenleg egy modellje az* $\{(x_0 = 1), (y_0 = 1)\}$*, azaz mind* $x$*, mind* $y$ *változónak 1-et adva értékül, a program ezt az utat fogja bejárni. Hasonlóan be kell járni* $l_3$*-at is. Ekkor a végső útvonal kényszer* $TC = (2 * x_0 = y_0)$ *lesz, aminek egy modellje* $\{(x_0 = 1), (y_0 = 2)\}$*, azaz* $x$ *változónak 1-et,* $y$ *változónak 2-t adva értékül, ez az útvonal lesz bejárva.*

A szimbolikus végrehajtás sajnos sok esetben elakadhat. Erre lehet példa egy nemlineáris értékadás, de elakadhat akár függvények hívása miatt is. Ezt a problémát oldja meg a *dinamikus szimbolikus végrehajtás*, vagy más néven *concolic tesztelés* (*conc*rete + symb*olic*) [18] [19]. A megközelítés párhuzamosan végez szimbolikus és konkrét végrehajtást. Ehhez karbantart egy konkrét állapotot, amiben minden változónak konkrét értéke van, egy szimbolikus állapotot, amiben minden változó szimbolikus értéke van, és egy szimbolikus útvonal kényszert.

Kezdetben minden változó kap egy véletlenszerű kezdő értéket. Ezt követően a szimbolikus végrehajtás menete szerint működik az algoritmus, annyi kiegészítéssel, hogy a konkrét állapotot is frissíti. Amikor viszont elakad a szimbolikus végrehajtás egy művelet miatt, akkor az adott művelet a konkrét értékekkel kerül végrehajtásra, ezáltal folytatható az ellenőrzés. Egy lehetséges útvonal végén a megoldó segítségével elő lehet állítani további bemeneti kezdő értékeket, amikkel újra lehet kezdeni az ellenőrzést. Ez a ciklus ismételhető szisztematikusan, vagy egy adott heurisztika szerint a megfelelő fedés eléréséig.

Számos szoftver létezik, amelyek a forráskód ismeretében generálnak teszteket. Ezek eltérhetnek az alkalmazott technikákban, heurisztikákban, a támogatott programok körében. Vannak eszközök, amik a .NET környezetben működnek [20] [21], illetve vannak, amik natív környezetben használatos forráskódok ellenőrzésére fejlesztettek ki [22] [23], a teljesség igénye nélkül.

Dolgozatomban kombinálom a tesztelés kétfajta megközelítését. A bejárt állapottérben elsősorban szimbolikus végrehajtás szerű bejárást használok, továbbá alkalmazom a határérték analízis elveit is.

# 3. Tesztgenerálás absztrakció-alapú modellellenőrzés támogatásával

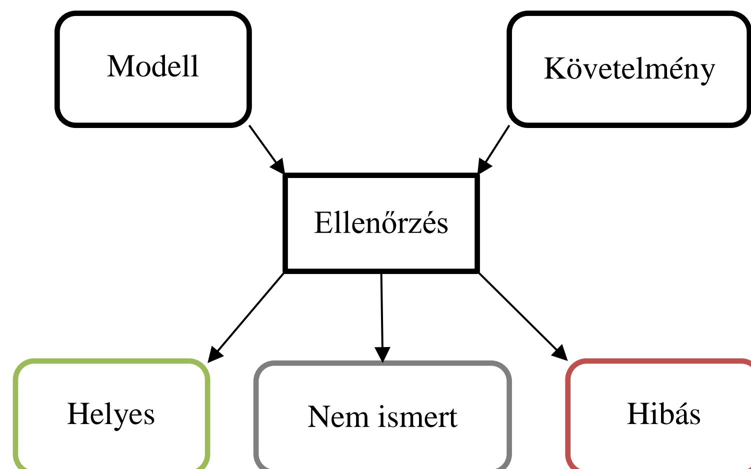
Ebben a fejezetben bemutatom az általam fejlesztett új módszert, amely kombinálja a modellellenőrzést és a tesztgenerálást. Áttekintem a megközelítés lépéseit és az általam fejlesztett algoritmusokat, amelyek a modellellenőrzés során nyert formális állapottér reprezentációt felhasználva képesek teszteket generálni.

## 3.1. Áttekintés

A modellellenőrzés, mint korábban be lett mutatva, egy olyan módszernek tűnhet, ami bármilyen programról el tudja dönteni, hogy helyes, vagy hibás-e. Bár ez matematikailag igaz, sajnos a gyakorlati alkalmazás miatt más szempontokból is meg kell vizsgálni.

A legfőbb szempont, hogy milyen gyorsan képes az algoritmus a helyesség eldöntésére. Sajnos még absztrakciót alkalmazva is, legrosszabb esetben be kell járni az egész állapotteret, ami hatalmas állapottérrobbanással jár. Ha van 8 darab 32 bites egész változónk, az állapottér mérete körülbelül megegyezik az univerzumban található atomok számával! Mivel nagyon gyakran ennél sokkal több változóval dolgoznak a programok, nem várható el az ellenőrzőtől, hogy minden esetben belátható időn belül termináljon.

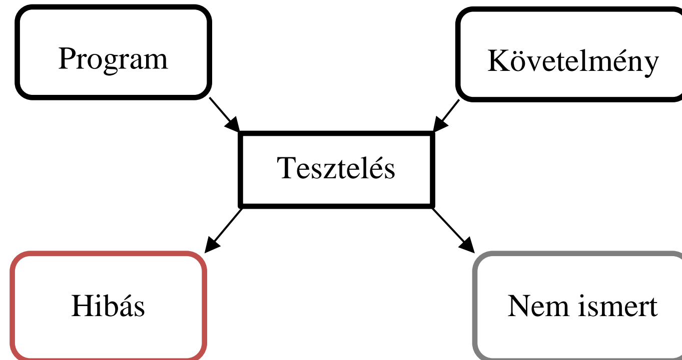
Ennek következtében át kell alakítani a modellellenőrzőről alkotott képünket. Az eddigi két lehetséges kimenet mellett megjelenik egy harmadik is, ami a megadott korláton belül nem eldönthető. Ennek köszönhetően, az algoritmus innentől mindig belátható időn belül ad egy eredményt, még ha azt is, hogy nem tudja eldönteni.




**3.1 ábra**: A modellellenőrzés gyakorlati modellje

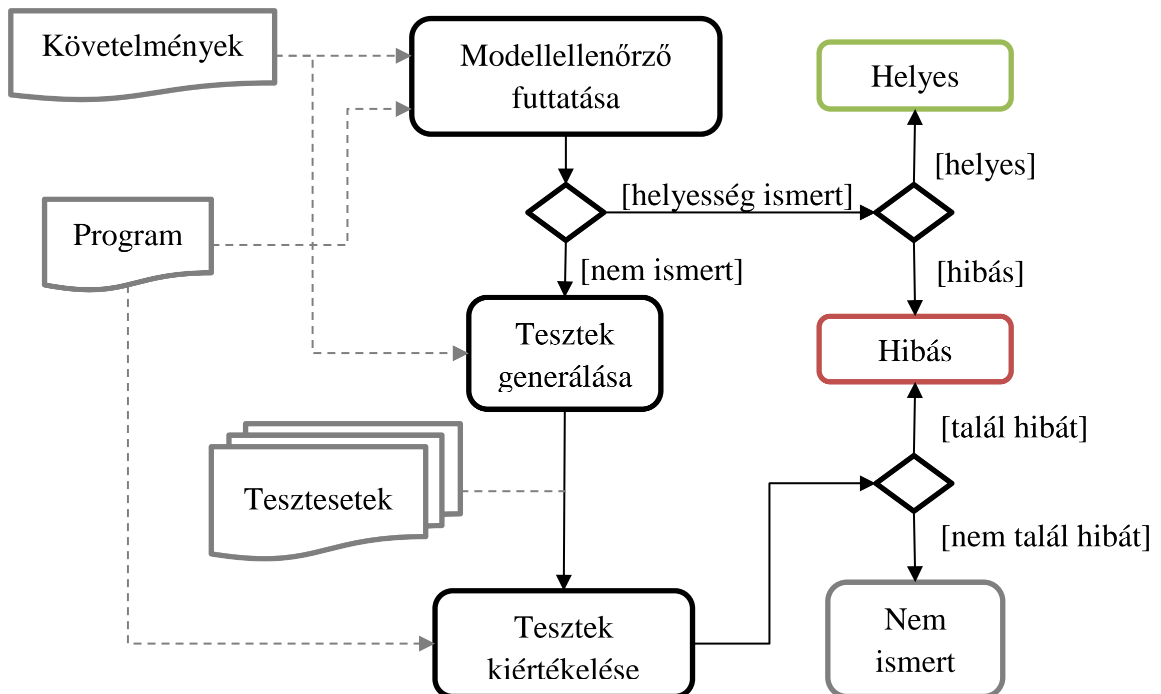
A kérdés az, hogy mit lehet tenni akkor, ha a modellellenőrző nem tud dönteni a helyességről. A dolgozatomban egy olyan új módszert javaslok, amely ilyenkor teszteket generál a modellellenőrző által előállított információkat felhasználva, melyek célja hogy felfedje a szoftverben lévő hibákat.

A tesztelés a jelenlegi megközelítésben kétféle végeredményt produkálhat a program helyessége szempontjából. Egyrészt lehet hibás, ha a tesztesetek között van olyan, ami sikertelen, illetve lehet nem ismert, ha az összes teszteset sikeresen lefutott, hiszen ettől sajnos nem mondhatjuk biztosan, hogy nincs is hiba.




**3.2 ábra**: A tesztelés sematikus modellje

A tesztelést alkalmazva a bemutatott módszerrel szembeni elvárás, hogy csökkenthető azon programok száma, amikre a helyesség nem eldönthető, illetve a tesztelés segítségével a lehetséges hibákat szeretnénk kiszűrni. A megközelítés első lépése, hogy a modellellenőrző igazolni tudja, hogy a program helyes, vagy hibás-e. Ha nem tudja eldönteni, akkor a tesztek generálása és futtatása után a nem eldönthető programok halmaza ketté bontható azokra, amikről a teszteset igazolja, hogy hiba van, és azokra, amikre továbbra sem lehet eldönteni a helyességet.

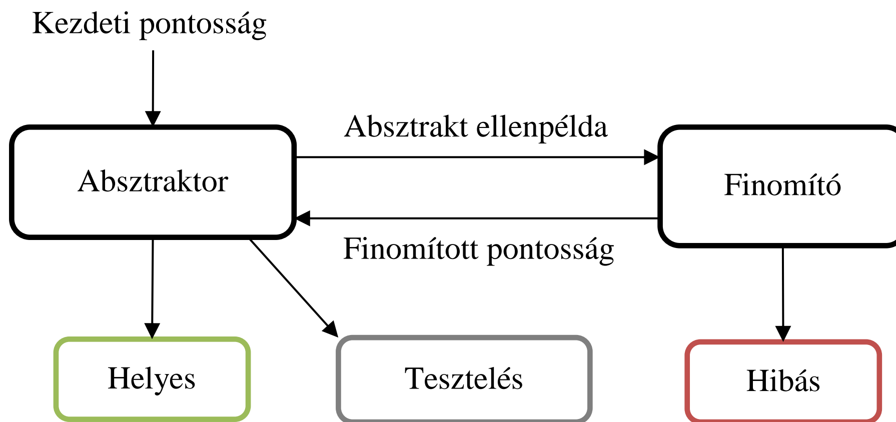



**3.3 ábra**: Modellellenőrzés és tesztelés kombinálása

## 3.2. CEGAR algoritmus megállási feltétellel

Mivel az ellenőrzőtől nem várható el a belátható időn belül való terminálás, lehetőséget kell biztosítani a modellellenőrző algoritmus leállítására. A leállításnak általánosan akkor kell bekövetkeznie, ha elfogytak az ellenőrző rendelkezésére álló erőforrások, esetleg már nem éri meg azt tovább futtatni. Leállási feltétel lehet:

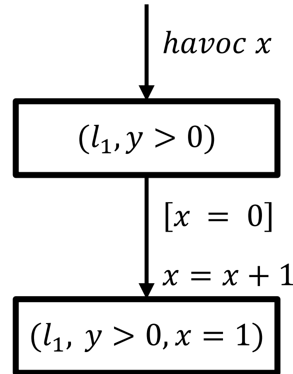
- Memória korlát elérése. Mivel az ellenőrzőket véges kapacitással rendelkező számítógépeken futtatják, számolni kell azzal, hogy az algoritmus feléli az erőforrásokat, és így nem válható a továbbhaladás.
- Időkorlát. Az algoritmus futására véges idő áll rendelkezésre.
- Költségkorlát. Az algoritmus által használt erőforrások után költségek kerülnek felszámításra, és véges a keret.
- Bármilyen egyéb, a helyzet által indokolt feltétel, ami akár függhet az állapottértől.




**3.4 ábra**: A programok vizsgálatára módosított predikátum absztrakciót használó CEGAR algoritmus

A CEGAR algoritmusnak alapvetően két kilépési pontja van. Egy, amikor a helyességet igazolta, egy pedig, amikor a hiba jelenlétét. Ahhoz, hogy a **3.1** ábrának megfeleljen az algoritmus egy harmadik kilépési pontot is létesíteni kell, ami a leállási feltétel teljesülése esetén lép érvényre.

Ezt a kilépési pontot érdemes a hurok első felébe, az absztrakció építésébe közbeiktatni. Ennek az oka, hogy az absztrakció építése egy hosszú művelet, értve ezalatt, hogy ciklikusan ismétel műveleteket, amíg bizonyos feltételek nem teljesülnek. Ha ennek a ciklusnak a kilépési feltételébe felvételre kerül a leállási feltétel, akkor az építés rugalmas feltételek mellett állítható le.




**3.5 ábra:** Egy absztrakt elérési fa részlete

Az algoritmus leállítása után az addig felépített absztrakt elérési fából információk nyerhetők ki. Információt hordoz magában a jelenlegi pontosság, az egyes csomópontokon érvényes predikátumok, illetve az átmeneteken lévő műveletek. A **3.5** ábrán látható absztrakt elérési fa részletén például csomópontokon lévő predikátum az $(y > 0)$ két esetben is, az $(x = 1)$, átmeneten lévő művelet az $[x = 0]$ és az $x = x + 1$.

```
1.  Abstractor(ARG, PREC, TERM):
2.     R := { ARG absztrakt állapotai }, W := { ARG nem befejezett állapotai }
3.     AMÍG W nem üres ÉS nincs kifejtett absztrakt hibaállapot ÉS
       nem TERM(ARG) ADDIG
4.        s := W egyik eleme
5.        W := W \ {s}
6.        HA ∃r ∈ R amire r fedi s-t AKKOR
7.           fedd le s-t r-rel
8.        HA s nincs fedve AKKOR
9.           e := s kifejtése
10.          R := R ∪ e
11.          W := W ∪ e
12.    HA TERM(ARG) miatt állt le a ciklus AKKOR
13.       ← NEM ISMERT
14.    HA absztrakt hibaállapot miatt állt le a ciklus AKKOR
15.       ← ABSZTRAKT HIBAÁLLAPOT
16.    KÜLÖNBEN
17.       ← HELYES
```

**3.1 pszeudokód:** Az absztrakció építésének módosított működése. Az $ARG$ az eddig felépített absztrakt elérési fa, $PREC$ a jelenlegi pontosság, $TERM$ a leállási feltétel.

## 3.3. Tesztek generálása

Amennyiben a helyesség nem dönthető el, mert a leállási feltétel megállítja a modellellenőrző futását, tesztek generálására van szükség. Ehhez felhasználom az absztrakt állapottérben fellelhető információkat.

A modellellenőrzés leállítását követően az absztrakt elérési fa csomópontjai két részre oszthatók. Vannak a befejezett, és a nem befejezett csomópontok.

A befejezett csomópontok azok az csomópontok, amik vagy ki lettek már fejtve, vagy le lettek fedve. Ezekkel a csomópontokkal a későbbiekben nem kell már foglalkozni, a következő okokból. Ha le lett már fedve, akkor az összes, ebből az állapotból elérhető hibaállapot elérhető abból az állapotból, ami fedi, ez pedig szükségszerűen ki van fejtve. Ha pedig ki lett már fejtve, akkor az összes ebből elérhető hibaállapot elérhető lenne a kifejtése eredményeként kapott állapotokból.

A nem befejezett csomópontok azok a csomópontok, amik még nem lettek kifejtve, és nincsenek is fedve. A fentiekből következik, hogy az összes elérhető hibaállapot ezen állapotokon átvezető úton keresztül érhető csak el.

### 3.3.1. Útvonal leképzése lineáris programozási problémára

Amennyiben adott az absztrakt elérési fában egy csomópont, legyen az *útvonal* a gyökérből ebbe a csomópontba vezető állapotok és köztük lévő átmenetek sorozata.

Az útvonallal kapcsolatban *kényszereket* fogalmazhatnak meg az átmeneteken található műveletek, illetve az állapotokon lévő predikátumok. Az alábbiakban bemutatom, hogy a kényszerekből hogyan lehet egy lineáris programozási problémát felírni.

Az útvonalon található változókat cseréljük le indexelt változókra. Egy *indexelt változó* egy olyan változó, aminek csak egyszer lehet értéket adni. Ennek oka, hogy a programban fellelhető változók másként viselkednek, mint a matematikában használatosak. Egy program esetén az $x = x + 1$ jelentése, hogy a következő állapotban $x$ értéke $x + 1$ lesz. Matematikailag felírva $x = x + 1$ egy ellentmondás. Ahhoz, hogy a „következő állapot" fogalmát értelmezni tudjuk matematikai formulákon, indexelésre van szükség ($x_1 = x_0 + 1$).

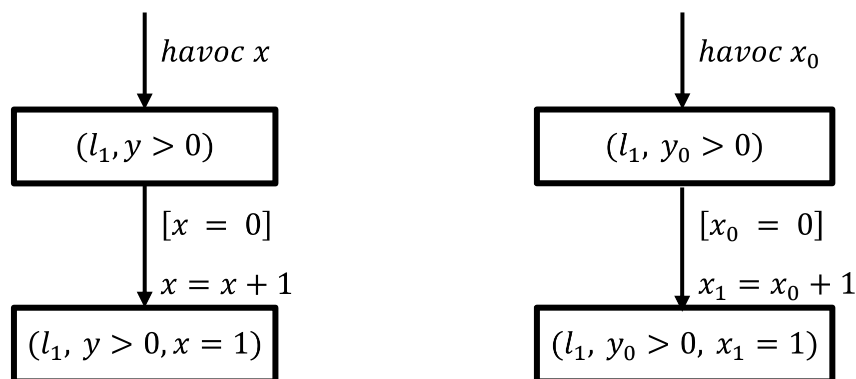
Legyen az indexelés egy olyan $I(n, x)$ függvény, ami megadja, hogy egy adott csomópontban, egy változót milyen indexelt változóra kell lecserélni. Ekkor, ha adott egy $s_i$ állapot $p_i$ predikátummal, $s_j$ állapot, és köztük $op$ művelet, akkor a következő átírási szabályok érvényesek:

- Ha $p_i$-ben adott egy $x$ változó, akkor $p_i$-ben $x$ összes előfordulását le kell cserélni $I(s_i, x)$-re. Ekkor $I(s_j, x) = I(s_i, x)$. Azaz az adott állapot predikátumaiban a változókat megindexeljük, az indexelés pedig nem változik a következő állapotra nézve.
- Ha $op$ egy őrfeltétel, és benne $x$ egy változó, akkor az őrfeltételben $x$ összes előfordulását le kell cserélni $I(s_i, x)$-re. Ekkor $I(s_j, x) = I(s_i, x)$. Azaz az adott őrfeltételben a változókat megindexeljük, az indexelés pedig nem változik a következő állapotra nézve.
- Ha $op$ egy értékadás, akkor külön kell kezelni az egyenlőségjel bal és jobb oldalát:
    - Ha a jobb oldali kifejezésben található egy $x$ változó, akkor $x$ összes előfordulását le kell cserélni $I(s_i, x)$-re a kifejezésben. Azaz az adott kifejezésben a változókat megindexeljük.

- Ha a baloldalon lévő változó $x$, akkor helyette egy olyan új, indexelt változót kell írni, ami korábban még nem szerepelt, azaz $x_i$-t, amire igaz, hogy $\nexists n, x : I(n, x) = x_i$.
- Ekkor $I(s_j, x) = x_i$, vagyis a következő állapotban a változóhoz, az újonnan létrehozott indexelt változó fog tartozni.

- Ha $op$ egy nemdeterminisztikus értékadás $x$ változóra, akkor helyette egy olyan új, indexelt változót kell írni, ami korábban még nem szerepelt, azaz $x_i$-t, amire igaz, hogy $\nexists n, x : I(n, x) = x_i$. Ekkor $I(s_j, x) = x_i$, vagyis a következő állapotban a változóhoz, az újonnan létrehozott indexelt változó fog tartozni.
- Általánosan igaz, hogy ha egy $y$ változónak nincs szerepe az adott műveletben, akkor $I(s_i, y) = I(s_j, y)$, vagyis az indexelése nem változik.

Ekkor látható, hogy minden indexelt változónak valóban csak egyszer lesz érték adva, hiszen minden értékadásnál a baloldalra egy új változót vezetünk be. Továbbá, mivel a fában nem fordulhat elő, hogy egy csomópontnak két különböző szülője is van, $I(n, x)$ értékére sem lehet ellentmondó információkat megfogalmazni.




**3.6 ábra:** Egy absztrakt elérési fa részlete (balra), és annak a vonatkozó átírása indexelt változók használatára (jobbra)

Ehhez nagyon hasonló, elsősorban számítógéparchitektúráknál használt, adatfüggőséget elimináló regiszter átnevezés technikája [24].

Ezután adott egy útvonal, ami mentén csak indexelt változók, és indexelt változókkal megfogalmazott kényszerek vannak. Első körben el kell dönteni, hogy milyen kényszerek fordíthatók le lineáris programozási kényszerekre.

- Ha egy őrfeltétel vagy predikátum aritmetikai egyenlőtlenség, akkor az átrendezhető úgy, hogy a konstans tagok összevonva egyoldalon, míg az indexelt változók a másik oldalon szerepeljenek. Ebből pedig egyértelműen előállítható a lineáris programozás által igényelt kényszerforma. Egyedül azt kell meggondolni, hogy egy esetleges $<$ jel mindig átírató $\leq$ jelre. Ha az összes változó egész szám, ez a konstans igazításával megtehető. Racionális számok esetén azt kell kihasználni, hogy ezeket a számítógép kvantálva tárolja, így ez az átírás a konstans igazításával szintén megtehető.

- Ha egy őrfeltétel vagy predikátum egyenlőségvizsgálat, akkor az átrendezhető úgy, hogy a konstans tagok összevonva egyoldalon, míg az indexelt változók a másik oldalon szerepeljenek. Ekkor a relációs jelet mindkét irányban a két oldal közé írva, két kényszer születik, ami ekvivalens az eredeti feltétellel, hiszen $(a = b) \Leftrightarrow (a \geq b) \wedge (a \leq b)$. Ez utóbbi azonban csak akkor tehető meg, ha a kifejezésben csak numerikus változók szerepelnek. (A logikai változók is leképezhetők, ha az igaz értéknek az 1-et, hamis értéknek a 0-t feleltetjük meg.)
- Egy értékadás, az egyenlőségvizsgálattal analóg módon képezhető le, a konstans tagokat az egyik, a változókat a másik oldalra rendezve, majd két kényszert képezve.
- Egy nemdeterminisztikus értékadás nem képezhető le, hiszen az értékéről nem állapítható meg semmi.

Az útvonal összes predikátumára, őrfeltételére, értékadására alkalmazva a fentieket, adott lesz egy kényszerhalmaz. Ebben lesznek ugyan kötött változók az értékadások következtében, viszont lesznek szabad változók is, mégpedig azok, amiknek nemdeterminisztikusan lett érték adva. Egy célfüggvény megadása után így megadható, hogy a nemdeterminisztikus értékadásoknál milyen értékeket kell valójában adni a változóknak, hogy a célfeltétel teljesüljön.

**3.1 példa:** *Vegyük a 3.6 ábra jobb oldalán lévő útvonal részletet, és írjuk át az ott található kényszereket lineáris programozási problémára. Tételezzük fel, hogy mind* x, *mind* y *egész változók.*

- havoc $x_0$ → *nem kerül átírásra*
- $y_0 > 0 \rightarrow (-y_0 \leq -1)$
- $[x_0 = 0] \rightarrow (x_0 \leq 0), (-x_0 \leq 0)$
- $[x_1 = 1] \rightarrow (x_1 \leq 1), (-x_1 \leq -1)$
- $x_1 = x_0 + 1 \rightarrow (-x_0 + x_1 \leq 1), (x_0 - x_1 \leq -1)$

### 3.3.2. Bemeneti változók határérték analízise

A határérték analízis futtatásához szükség van a bemeneti változók azon értékeire, amelyek a program futtatása során szélsőséges helyzetekhez vezethetnek, tehát a program futása során az ekvivalencia partíciók határán vannak.

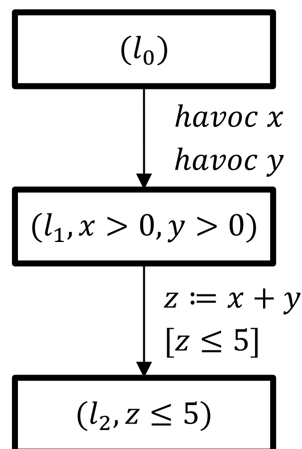
Vegyük a modellellenőrzés utáni nem befejezett csomópontokat az absztrakt elérési fában. Az összes elérhető hibaállapot ekkor ezeken keresztül érhető el. Az adott csomópontokra kényszereket fogalmaz meg az útvonal, jó esetben korlátozva az ebből elérhető állapottér méretét. Ha vesszük a kényszerek mellett a bemeneti változók lehetséges minimális és maximális értékét, akkor ezek az adott változó lehetséges szélsőértékei az innen elérhető állapottérben.

Ezek a szélsőértékek általában szűkebbek lesznek a változó valódi szélsőértékeihez képest. Ennek oka, hogy az ellenőrzés az állapottér egy részét bejárja, és megállapítja, hogy elérhetetlen, így tesztelni sem szükséges.

```
1. InputMinMaxGenerator(ARG):
2.   N := { ARG nem befejezett nódusai }
3.   T := {}
4.   MINDEN n ∈ N ESETÉN:
5.     T := útvonal n-be
6.     K := { T-ből kinyerhető lineáris programozási kényszerek }
7.     H := {T-beli nemdeterminisztikus értékadások indexelt változói }
8.
9.     MINDEN h ∈ H ESETÉN:
10.      MAX := LP_MEGOLD(K,max(h))
11.      HA MAX kielégíthető AKKOR
12.        T := T ∪ { MAX modelljében H elemeihez tartozó értékadások }
13.      MIN := LP_MEGOLD(K,min(h))
14.      HA MIN kielégíthető AKKOR
15.        T := T ∪ MIN { modelljében H elemeihez tartozó értékadások }
16.  ← T
```

**3.2 pszeudokód:** Bemeneti változók határérték analízisén alapuló tesztgenerálás. Az $ARG$ a modellellenőrző által felépített absztrakt elérési fa. Az $LP_MEGOLD(K, F)$ függvény megoldja azt a lineáris programozási problémát, aminek a kényszerei $K$, célfüggvénye $F$.

A tesztkészlet generálása a következőképpen zajlik. Minden nem befejezett csomóponthoz tartozó útvonal más kényszereket fogalmazhat meg a változókkal kapcsolatban. Ezeket a kényszereket le lehet fordítani lineáris programozási kényszerekre. Ezt követően a célfüggvényt kell még meghatározni. A célfüggvény legyen sorra minden bemeneti változó minimalizálása, majd maximalizálása. Ez utóbbihoz venni kell az összes nemdeterminisztikus értékadás következtében létrehozott indexelt változót. Az így kapott lineáris programozási problémák, egyesével megoldhatók egy megoldó segítségével.




**3.7 ábra:** Egy absztrakt elérési fa részlete

Ha egy probléma megoldása kielégíthetetlen, az azt jelenti, hogy az adott állapot valójában elérhetetlen, csak még nem került visszavágásra (mert nem lett még a visszavágáshoz szükséges hibaállapot kifejtve), nem szükséges vele tovább foglalkozni.

Ha viszont kielégíthető, akkor adott lesz egy modell, amiből kinyerhető az összes bemeneti változó kezdőértéke. Ez utóbbihoz venni kell az összes nemdeterminisztikus értékadás következtében létrehozott indexelt változókat, és megkeresni a hozzájuk tartozó értékadásokat a modellben.

**3.2 példa:** *Vegyük a 3.7 ábrán látható absztrakt elérési fa részletet. Ha az ezen található kényszereket lefordítjuk lineáris programozási kényszerekre, akkor a következőt kapjuk:* $\{ (-x_0 \leq 0),\ (-y_0 \leq 0),\ (\mathrm{z}_0 - x_0 - \mathrm{y}_0 \leq 0),\ (x_0 + \mathrm{y}_0 - \mathrm{z}_0 \leq 0), (\mathrm{z}_0 \leq 5)\}$.

*A bemeneti változóknak megfelelő indexelt változók kinyerhetők a nemdeterminisztikus értékadásokból, jelen esetben $x_0$, és $y_0$. Két változó esetén négy célfüggvényt lehet megfogalmazni, rendre*

- $\max(x_0)$*: ekkor a megoldás* $\{(x_0 = 4), (y_0 = 1)\}$.
- $\min(x_0)$*: ekkor egy lehetséges megoldás* $\{(x_0 = 1), (y_0 = 1)\}$.
- $\max(\mathrm{y}_0)$*: ekkor a megoldás* $\{(x_0 = 1), (y_0 = 4)\}$.
- $\min(\mathrm{y}_0)$*: ekkor egy lehetséges megoldás* $\{(x_0 = 1), (y_0 = 1)\}$.

### 3.3.3. Robusztusság tesztelés

A programok írása során egy másik gyakori hibaforrás lehet, hogy különböző feltételeknél (elágazásokban vagy ciklusokban) az egyik numerikus változó szélsőértékére nem megfelelően viselkedik a program. A robusztusság analízis alapja szintén a határérték analízis, a kérdés, hogy egyes változók minimális, maximális értékei, milyen bemeneti értékek mellett érhető el.

```
1.  AssumptionMinMaxGenerator(ARG):
2.    N := { ARG nem befejezett nódusai }
3.    T := {}
4.    MINDEN n ∈ N ESETÉN:
5.      T := útvonal n-be
6.      K := { T-ből kinyerhető lineáris programozási kényszerek }
7.      H := { T-beli nemdeterminisztikus értékadások indexelt változói }
8.      A := { T-beli aritmetikai feltételekben lévő indexelt változók }
9.
10.     MINDEN a ∈ A ESETÉN:
11.       MAX := LP_MEGOLD(K,max(a))
12.       HA MAX kielégíthető AKKOR
13.         T := T ∪ { MAX modelljében H elemeihez tartozó értékadások }
14.       MIN := LP_MEGOLD(K,min(a))
15.       HA MIN kielégíthető AKKOR
16.         T := T ∪ MIN { modelljében H elemeihez tartozó értékadások }
17.   ← T
```

**3.3 pszeudokód:** Aritmetikai feltételek határérték analízisén alapuló tesztgenerálás. Az $ARG$ a modellellenőrző által felépített absztrakt elérési fa. Az $LP_MEGOLD(K, F)$ függvény megoldja azt a lineáris programozási problémát, aminek a kényszerei $K$, célfüggvénye $F$.

Egy nem befejezett csomóponthoz tartozó útvonal kényszereket fogalmaz meg az innen elérhető állapottérrel kapcsolatban. Vizsgálható, hogy ezen kényszerek mellett, mi a változók lehetséges minimum és maximum értéke, illetve mi az ezekhez tartozó bemeneti érték.

A tesztkészlet a következőképpen generálható. Minden nem befejezett csomóponthoz tartozó útvonal más kényszereket fogalmazhat meg a változókkal kapcsolatban. Ezeket a kényszereket le lehet fordítani lineáris programozási kényszerekre. Ezt követően a célfüggvényt kell még meghatározni. A célfüggvény legyen az összes, aritmetikai feltételben előforduló változó először minimalizálása, majd maximalizálása. Természetesen itt is a változókhoz meg kell keresni a feltétel pillanatában megfelelő indexelt változót, és arra megfogalmazni a célfüggvényt.

Az így kapott lineáris programozási problémákat megoldva, ha egy probléma kielégíthetetlen, az azt jelenti, hogy az adott állapot valójában elérhetetlen, csak még nem került visszavágásra. Ha viszont kielégíthető, akkor adott lesz egy modell, amiből kinyerhető az összes bemeneti változó kezdőértéke. Ez utóbbihoz venni kell az összes nemdeterminisztikus értékadás következtében létrehozott indexelt változókat, és megkeresni a hozzájuk tartozó értékadásokat a modellben.

**3.3 példa:** *Vegyük a 3.7 ábrán látható absztrakt elérési fa részletet. Ha az ezen található kényszereket lefordítjuk lineáris programozási kényszerekre, akkor a következőt kapjuk:* $\{\,(-x_0 \leq 0),\ (-y_0 \leq 0),\ (\mathrm{z}_0 - x_0 - \mathrm{y}_0 \leq 0),\ (x_0 + \mathrm{y}_0 - \mathrm{z}_0 \leq 0), (\mathrm{z}_0 \leq 5)\}$.

*A bemeneti változóknak megfelelő indexelt változók kinyerhetők a nemdeterminisztikus értékadásokból, jelen esetben* $x_0$*, és* $y_0$*. Az aritmetikai feltételeknek megfelelő indexelt változók kinyerhetők a műveletekből, jelen esetben ez* $\mathrm{z}_0$*. Egy változó esetén két célfüggvényt lehet megfogalmazni, rendre*

- $\max(z_0)$*: ekkor egy lehetséges megoldás* $\{(x_0 = 4), (y_0 = 1)\}$.
- $\min(\mathrm{z}_0)$*: ekkor egy lehetséges megoldás* $\{(x_0 = 1), (y_0 = 1)\}$.

### 3.3.4. Számábrázolásból fakadó programhibák tesztelése

Az eddigi vizsgálatokban nem foglalkoztunk a változók értékkészletével. Bár a matematikában egy változó értékkészlete lehet végtelen, számítástechnikában a véges bitszámon való tárolás miatt a változók értékkészlete véges. Az ebből eredő probléma, ha egy változónak olyan értéket kéne eltárolnia, amire az értékkészlete alapján nem képes, túlcsordul, aminek számos (általában negatív) következménye lehet. Tehát fontos a számábrázolási korlátokból fakadó szoftver hibák ellenőrzése.

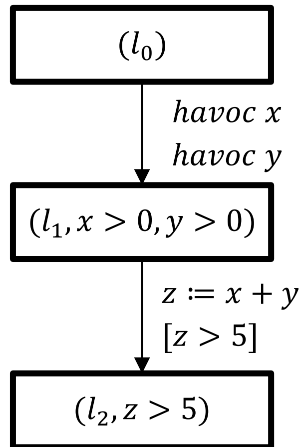
A modellellenőrző algoritmusok jellemzően természetes számokkal dolgoznak, ezért ezeket a hibákat nem tudják észrevenni. Az alábbi megoldás kiegészíti ezen algoritmusok ellenőrzéseit, segít vizsgálni a programok robusztusságát.

```
1.  OverflowGenerator(ARG, MAX_VAL, MIN_VAL):
2.    N := { ARG nódusai }
3.    T := {}
4.    MINDEN n ∈ N ESETÉN:
5.      T := útvonal n-be
6.      H := { T-beli nemdeterminisztikus értékadások indexelt változói }
7.      V := { T-n lévő indexelt változók }
8.
9.      K := { T-ből kinyerhető lineáris programozási kényszerek }
10.     MINDEN h ∈ H ESETÉN:
11.       K := K ∪ { h ≤ MAX_VAL(h), −h ≤ MIN_VAL(h) }
12.
13.     MINDEN v ∈ V ESETÉN:
14.       MAX := LP_MEGOLD(K, max(v))
15.       HA MAX kielégíthető ÉS MAX(v) ≥ MAX_VAL(v) AKKOR
16.         T := T ∪ { MAX modelljében H elemeihez tartozó értékadások }
17.       MIN := LP_MEGOLD(K, min(v))
18.       HA MIN kielégíthető ÉS MIN(v) ≤ MIN_VAL(v) AKKOR
19.         T := T ∪ MIN { modelljében H elemeihez tartozó értékadások }
20.   ← T
```

**3.4 pszeudokód:** Túlcsordulás tesztelésén alapuló tesztgenerálás. Az $ARG$ a modellellenőrző által felépített absztrakt elérési fa. $MIN_VAL(v)$ és $MAX_VAL(v)$ olyan függvények, amik megadják minden változóra annak lehetséges minimum, maximum értékét. Az $LP_MEGOLD(K, F)$ függvény megoldja azt a lineáris programozási problémát, aminek a kényszerei $K$, célfüggvénye $F$.

Ezt a korábbiakhoz hasonlóan lehet elvégezni. Venni kell az absztrakt elérési fa összes csomópontját, és az odavezető útvonalon lévő kényszereket lefordítani lineáris programozási problémára. Ekkor viszont még további kényszerek felvétele szükséges. Meg kell kötni az összes bemeneti változót, hogy az a típusa által megkívánt értelmezési tartományban legyen. A típusok hordozta információk el lettek absztrahálva a CFA formalizmusra való áttérésnél, így e vizsgálathoz ezeket külön kell biztosítani az ellenőrző számára.

A célfüggvényhez venni kell az összes, útvonalon előforduló indexelt változót, és egyszer a minimumukat, egyszer a maximumukat keresni. Ha az így keletkezett lineáris programozási probléma kielégíthetetlen, akkor az azt jelenti, hogy az adott állapot valójában elérhetetlen, csak még nem került visszavágásra. Ha kielégíthető, akkor meg kell vizsgálni a kapott modellben, az aktuálisan minimalizált/maximalizált indexelt változó értékét. Ha ezek közül bármelyik, a megengedett értékkészleten kívül esik, a program futása során túlcsordulás keletkezhet. A túlcsorduláshoz szükséges bemeneti értékek kinyerhetők a modellből.




**3.8 ábra:** Egy absztrakt elérési fa részlete

**3.4 példa** *Vegyük a 3.8 ábrán látható absztrakt elérési fa részletet. Ez szemléletesen a* **3.7** *ábrán látható fa „else ága”. Ha az ezen található kényszereket lefordítjuk lineáris programozási kényszerekre, akkor a következőt kapjuk:* $\{(-x_0 \leq 0),\ (-y_0 \leq 0),\ (z_0 - x_0 - y_0 \leq 0),\ (x_0 + y_0 - z_0 \leq 0), (-z_0 \leq -6)\}$.

*A bemeneti változóknak megfelelő indexelt változók kinyerhetők a nemdeterminisztikus értékadásokból, jelen esetben* $x_0$*, és* $y_0$*. A kényszereket ki kell egészíteni a változók értékkészletével. Ehhez a példa kedvéért tegyük fel, hogy* $x, y, z \in [0; 15]$*, azaz 4 bites előjel nélküli egész változók. A kényszerek halmaza kiegészül a* $\{(-x_0 \leq 0), (x_0 \leq 15), (-y_0 \leq 0), (y_0 \leq 15)\}$ *kényszerekkel. Az előforduló indexelt változók jelen esetben* $x_0$, $y_0$*, és* $z_0$*. Három változó esetén hat célfüggvényt lehet megfogalmazni, rendre*

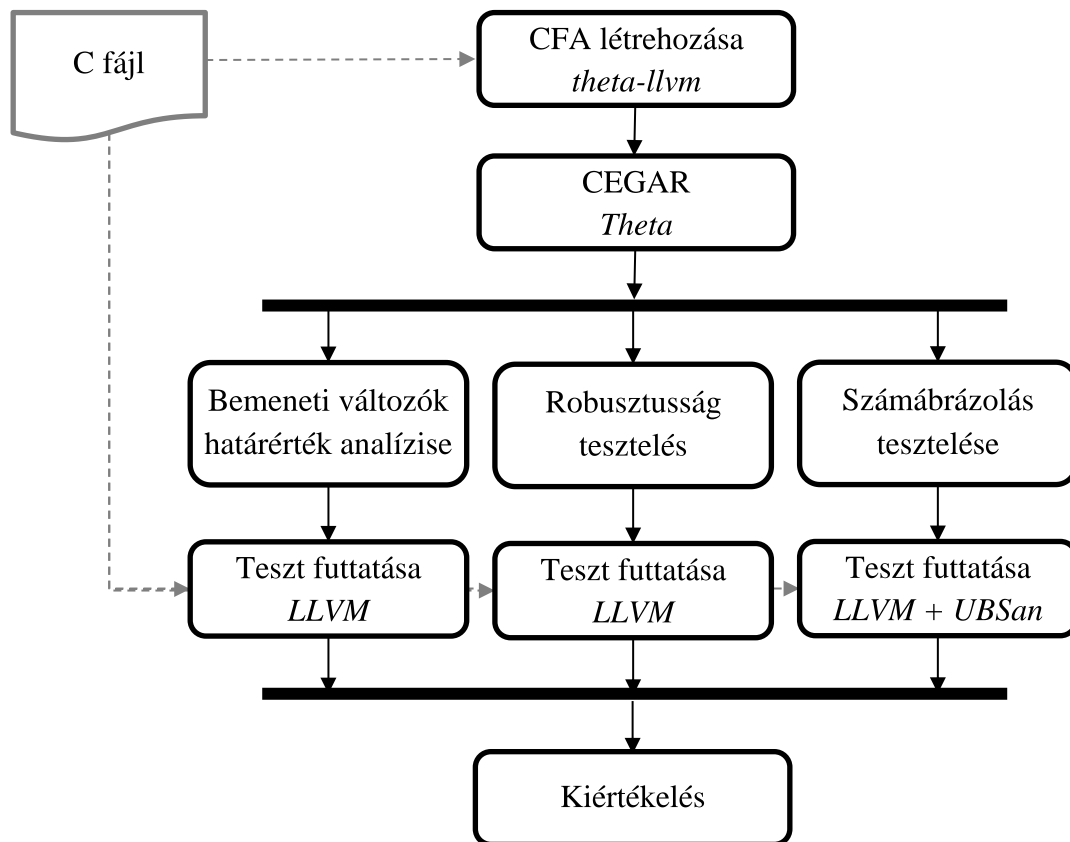
- $\max(x_0)$*: ekkor nincs megoldás, mert* $x_0$ *lehet végtelen.*
- $\min(x_0)$*: ekkor egy lehetséges megoldás* $\{(x_0 = 1), (y_0 = 5)\}$.
- $\max(y_0)$*: ekkor nincs megoldás, mert* $y_0$ *lehet végtelen.*
- $\min(y_0)$*: ekkor egy lehetséges megoldás* $\{(x_0 = 5), (y_0 = 1)\}$.
- $\max(z_0)$*: ekkor egy lehetséges megoldás* $\{(x_0 = 15), (y_0 = 15)\}$*, amire* $z_0$ *értéke 30, tehát túlcsordulás történt.*
- $\min(z_0)$*: ekkor egy lehetséges megoldás* $\{(x_0 = 3), (y_0 = 3)\}$

# 4. Kiértékelés

Ebben a fejezetben bemutatom, hogy a fenti elvek mentén, hogyan készült el az algoritmus egy implementációja, és az azzal végzett mérések eredményét.

## 4.1. Implementáció

A dolgozatomban kifejtett ellenőrzési és tesztgenerálási ötleteket egy szoftver formájában megvalósítottam. A szoftver alapvetően Java nyelvre épül, és több, külső szolgáltatást veszek igénybe.




**4.1 ábra:** Az implementált szoftver felépítése nagyvonalakban

### 4.1.1. Forráskód átalakítása

A korábbi fejezetekben, mind a formális verifikáció, mind a tesztek generálása során erősen támaszkodtunk a program matematikai reprezentációjára, a CFA-ra. Azonban ahhoz, hogy gyakorlatban használható eszközt kapjunk, a programoknak elő kell állítani ezt a reprezentációját.

Az implementáció tervezésekor döntést kellett hozni a támogatandó programok köréről. Az egyszerűségre törekedve a C nyelvű forráskódok feldolgozása mellett döntöttünk, bizonyos, később ismertetett megkötések mellett.

A programok átalakítására a *theta-llvm* [25] szoftver használata mellett döntöttünk. Ez a Low Level Virtual Machine (továbbiakban LLVM) [26] környezetet használja fel a működéséhez.

Az LLVM infrastruktúrát felhasználva először elő kell állítani a C program LLVM bájtkód reprezentációját. A későbbiekben aztán ezt a bájtkódot lehet továbbfordítani más nyelvekre, gondolva itt egy x86-os processzor utasításkészletére, vagy akár webes környezetben használható JavaScript nyelvre.

A korábban említett theta-llvm szoftver is egy fordítót valósít meg, ami az LLVM bájtkódból egy CFA-t állít elő.

A szoftver megkülönböztet néhány függvényt, aminek különleges jelentése van:

- `int __theta_nondet_int()`: Ez a függvény jelöli a nemdeterminisztikus (32 bites egész) értékadást.
- `void __theta_assert(bool):` Ezzel a függvénnyel lehet a követelményeket (assertion) megfogalmazni. Amennyiben a paramétere hamisra értékelődik ki, a szoftver hibával befejezi a futását.
- `void __theta_exit(int):` Egy kilépési pontot biztosít a programból, a megadott értékkel.

Az implementáció használójának felelőssége annak a biztosítása, hogy a fenti függvények megfelelően szerepeljenek a forráskódban.

#### 4.1.1.1. Korlátozások

Mivel célom egy prototípus segítségével a megközelítés kipróbálása, ezért a feladatomhoz elég volt egy olyan fordító használata, amely a C programok csupán egy részhalmazára valósítja meg a fordítást. A fordító nem támogatja a globális változók, illetve a függvények használatát, csak egy megkötéssel: fordítás időben bele kell őket ágyazni a main függvénybe. Ennek következménye, hogy a rekurzív függvényhívásokat alkalmazó szoftverek egyelőre nem fordíthatók le.

Összefoglalva a megkötések:

- Dinamikus memória kezelése nem támogatott
- Mutatók használata erősen korlátozott
- A teljes programnak egy forrásfájlon belül kell elhelyezkednie.
- Csak 32 bites egész és logikai változók szerepelhetnek a programban.
- Rekurzív függvényhívások nem támogatottak.
- Csak a forrás fájlban implementációval rendelkező függvények támogatottak.

### 4.1.2. Formális verifikáció

A formális verifikáció céljából a Theta [27] nevű keretrendszert választottam a rugalmassága miatt. Ebben a formális verifikációt egy CEGAR alapú algoritmus valósítja meg, ami többek között predikátum absztrakciót használ. A Theta mögött, az SMT problémák megoldásához a Microsoft által fejlesztett Z3 bizonyítót [28] használom.

Megállási feltétel gyanánt több lehetőséget fejlesztettem le. Lehetőség van a verifikációt egy megadott ideig, az absztrakt elérési egy megadott csomópontszámáig, vagy utóbbi egy megadott mélységéig futtatni.

### 4.1.3. Tesztek generálása

A tesztek generálásához a Theta [27] által visszaadott állapottér reprezentáció (ami egy absztrakt elérési fa) lett felhasználva, az SMT problémák megoldására pedig továbbra is a Z3 [28] bizonyító. A generálás során felmerülő lineáris programozási problémák megoldására szintén a Z3 bizonyító lett felhasználva.

Mivel egy tesztesetnél ismertek az egyes értékadások, könnyedén lehet ezt felhasználva olyan C fájlt írni, ami a forrással történő összelinkelés esetén le is futtatja a tesztesetet. Ahhoz, hogy ezt megtegyük, egy implementációt kell biztosítani a 4.1.1-es fejezetben ismertetett három függvénynek.

- `int __theta_nondet_int():` Az implementációnak a teszteset által megkívánt sorrendben kell visszaadnia az értékeket. Ez utóbbi legegyszerűbben egy statikus tömbbel valósítható meg, ami sorrendben tartalmazza a visszaadandó értékeket, és mindig annyiadikat adja vissza, ahányadikként meg lett hívva a függvény. Amennyiben többször lesz meghívva a függvény, mint ahány érték visszaadására képes, a teszt eredménye nem meggyőző.
- `void __theta_assert(bool):` Ha a paraméter hamisra értékelődik ki, akkor leállítja a program futását, és jelzi a program hívójának, hogy a teszteset egy hibaállapotot ért el.
- `void __theta_exit(int):` Leállítja a program futását, és jelzi a program hívójának, hogy a teszteset hiba találása nélkül futott le.

A különböző típusú teszteket, különböző komponensek generálják, ezek a teszt generálási stratégiák. Ezek a stratégiák egymástól függetlenül működnek, egymás eredményeit nem befolyásolják.

### 4.1.4. Tesztek futtatása

A különböző típusú teszteket, különböző komponensek futtatják. Erre azért van szükség, mert egy-egy típusú teszt különleges elvárásokat támaszthat a futtatókörnyezet felé, mint az a túlcsordulás esetén meg is történik.

A legenerált tesztek tulajdonképpen az előző fejezet alapján előállított C fájlok. Ezeket ezt követően le lehet fordítani és össze lehet linkelni a program forrásával az LLVM keretrendszert használva. A futtatókörnyezet figyelemmel kíséri az így létrejövő tesztprogram által adott eredményt.

#### 4.1.4.1. Számábrázolásból fakadó programhibák tesztelése

A fenti egyszerű futtatás képes a logikai hibák megtalálására, azonban a túlcsordulás ellenőrzésére külső eszközök is szükségesek. Ennél az LLVM keretrendszer UndefinedBehaviorSanitizer (továbbiakban UBSan) [29] eszközére esett a választás, mert ez volt integrálható legkönnyebben az addigi technológiákkal.

Ez az eszköz röviden összefoglalva belefordít további kódrészeket a programba, amik minden egyes aritmetikai műveletet ellenőriznek, hogy ne történjen benne túlcsordulás. Túlcsordulás esetén a viselkedés konfigurálható, jelenleg leállítja a programot, és jelzi a futtatónak, hogy hibaállapot lett elérve.

## 4.2. Esettanulmány

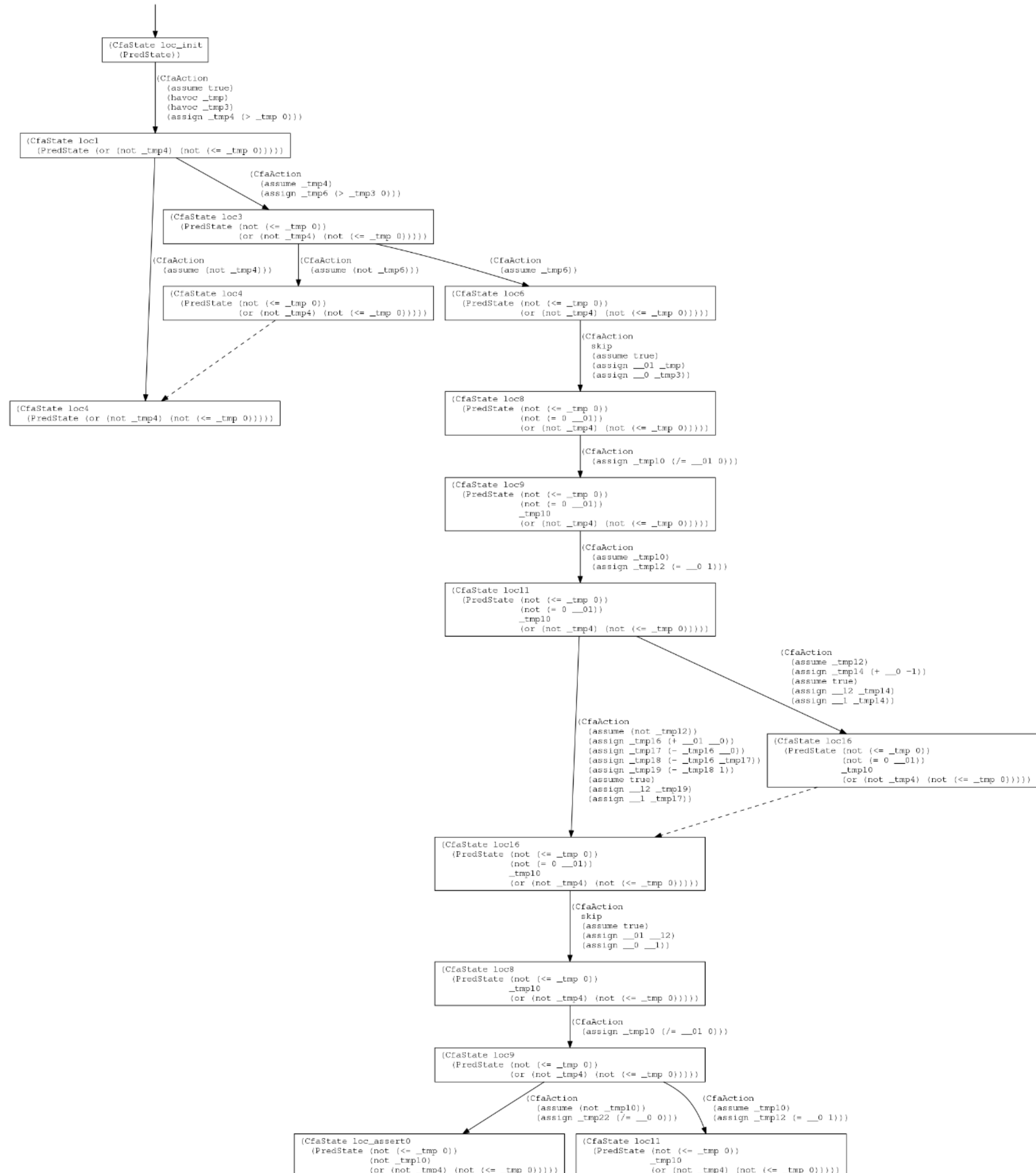
Ebben a fejezetben bemutatok egy példa programot, és az implementált szoftver a programról alkotott kiértékelését.

```
1. #include <stdbool.h>
2. extern int __theta_nondet_int();
3. extern void __theta_assert(bool);
4. extern void __theta_exit(int);
5. #define assert(X) __theta_assert(X)
6. #define exit(X) __theta_exit(X)
7. #define assume(X) if(!(X)) exit(0);
8.
9. int main(void) {
10.       int a = __theta_nondet_int();
11.       int b = __theta_nondet_int();
12.       assume(a > 0 && b > 0);
13.
14.       while(a != 0) {
15.         if(b == 1) {
16.           b--;
17.           a = b;
18.         }
19.         else {
20.           a = a + b;
21.           b = a - b;
22.           a = a - b - 1;
23.         }
24.       }
25.       assert(b != 0);
26.
27.       return 0;
28. }
```

**4.1 kód:** Egy elrontott C nyelvű program

A fenti kódon az alábbiak figyelhetők meg. Két bemeneti változója van ($a$, és $b$). Az algoritmusnak előfeltétele, hogy mindkét bemeneti változója pozitív legyen, ennek hiányában jelenleg leáll a program, hiba nélkül. Ez a helyességen nem változtat, mivel a helyesség feltétele, hogy hibás állapot nem érhető el. A követelmény az, hogy az algoritmus futása után a $b$ változó értéke ne lehessen 0. A ciklust addig ismétli, amíg $a \neq 0$. Ha $b$ értéke 1 lesz, akkor mind $a$, mind $b$ változó értéke 0 lesz, tehát véget ér a ciklus, és megsérült a követelmény. Amíg $b$ értéke nem 1, addig minden iterációban megcseréli a két változó tartalmát, majd $a$-t csökkenti 1-gyel. A cserét azért ezzel az algoritmussal hajtja végre, mert így demonstrálható lesz a túlcsordulás tesztelése is.

A fenti kódot először át kell alakítani CFA-ra. Az átalakítás után el kell indítani a modellellenőrzést. A leállási feltétel jelenleg legyen a példa kedvéért, hogy a kifejtett csomópontok száma több mint 14. Ennél a feltételnél az algoritmus már bejárja az állapottér egy részét, de még nem találja meg az utat egyértelműen egy elérhető hibaállapot felé. A modellellenőrzés után az alábbi absztrakt elérési fát kapjuk.




**4.2 ábra:** A modellellenőrzés eredményeként előálló absztrakt elérési fa. A szaggatott nyilakkal a fedési reláció van jelölve.

Az ábrán a következő dolgok figyelhetők meg a struktúrával kapcsolatban. A bal felső sarokban találhatóak a beolvasás, és a bemeneti feltétel ellenőrzése. A ciklus ebben a pillanatban egyszer lett csak kifejtve (ciklus eleje a $loc11$ vezérlési hely). Szintén megfigyelhető, hogy jelenleg a pontosság nem elég precíz, ezért a ciklusban lévő elágazás

két ága közül az egyik fedi a másikat (pedig a két ág teljesen mást csinál). Továbbá látható, hogy két nem befejezett állapot van, mégpedig az ábra legalján lévő két állapot ($loc_assert0, loc11$).

Az ábra struktúrája, változónevei nehezen áttekinthetők. Ezek a CFA-t előállító eszköz sajátosságai. Az eszköz az LLVM bájtkódot alakítja CFA-vá, és mert ez egy assembly-szerű nyelv, egy műveletben nem lehet összetett aritmetikai műveleteket elvégezni, azokat több lépésre kell bontani. Ez a struktúra bonyolultságát okozza, míg az LLVM által használt Single Static Assignment (SSA) elv a sok segédváltozó használatát. A változónevek sajátossága, hogy az eszköz belül szintén a regiszter átnevezéshez hasonló műveleteket végez. A $__01 = a$, míg $__0 = b$ a két fontosabb változó, amit nyomon kell követni.

Az alábbi példaprogramra mindhárom korábban ismertetett megközelítés talál egy tesztesetet. A bemeneti változóknál, a kényszereket végig követve látható, hogy $a$ minimális értéke 1, míg $b$ minimális értéke 2. A lehetséges maximális érték mind két változóra végtelen.

```
1. #include <stdbool.h>
2. #include <stdlib.h>
3.
4. void __theta_exit(int n) {
5.       exit(n);
6. }
7.
8. void __theta_assert(bool b) {
9.       if(!b) exit(3);
10.}
11.
12.int __theta_nondet_int() {
13.      static int callNum = 0;
14.      static int callMax = 2;
15.
16.      static int tests[] = {
17.        1,
18.        2,
19.        0
20.      };
21.
22.      if(callNum == callMax) exit(4);
23.      return tests[callNum++];
24.}
```

**4.2 kód:** Egy tesztesek leképzése C nyelvre

A $b$ minimális értéke érdekes. A kódot nézve lehetne 1, viszont a jelenlegi absztrakcióban valóban 2 a lehetséges legkisebb érték. A végtelen értékek esetén a modell nem képezhető le véges értékekre, így ezek a tesztek eldobásra kerülnek. Az $a = 1$ értéke esetén nincs kizárva a $b = 2$ értéke, és ez fordítva is igaz. A kényszerekre a Z3 által visszaadott modell valójában mind $a$, mind $b$ minimalizálása esetén $a = 1$ és $b = 2$ értékadásokat

tartalmazza, ezekből pedig egy teszteset generálható. Teljesen hasonló gondolatmenet belátható az aritmetikai értékadásokra is, így azokat nem fejtem ki.

A túlcsordulás vizsgálata szintén talál egy hibát. A $_tmp16$ változó értéke megegyezik $a + b$ értékével, ami ha nagyobb, mint az $int$ értelmezési tartománya, túlcsordulás történik. Ez megtehető akkor, ha $b = 2$ értéket kap, míg $a$ a lehetséges maximumot.

Az így talált teszteseteket le kell képezni C fájlokra. Egy példa esetén, ez a **4.2** kódban látható módon tehető meg. A teszteset esetén a 3-as kilépési érték jelenti a hibás eredményt, a 4-es kilépési érték, ha több értéket kéne adni, mint tud a teszt, a nem meggyőző eredményt. A $tests$ tömb tartalmazza a nemdeterminisztikus értékadások értékét, amiből mindig a hívásnak megfelelőt adja vissza.

## 4.3. Szoftver verifikációs benchmarkok

Az implementáció erőforrásigényeinek vizsgálatára a Software and Computational System Lab (SoSy-Lab)[1] szoftveres ellenőrző eszközök kiértékelésére használatos gyűjteménye[2] [30] lett használva. A gyűjtemény tartalmaz C, Java kódokat, de tartalmaz klóz formában elérhető problémákat is. Az implementáció kiértékeléséhez értelemszerűen a C kódok közül lett válogatva.

A gyűjtemény több szempont alapján is kategóriába sorolja a programokat. Az első szempont a program felépítése, a második szempont a vizsgált követelmények típusa. A dolgozatban felvetett vizsgálatoknak az utóbbi szempont szerinti *unreach-call*, *és no-overflow* kategóriák felelnek meg. Előbbi egyes hívások elérhetetlenségét kívánja igazolni, az utóbbi a túlcsordulás hiányát. Sajnálatos módon, az utóbbi kategóriába eső összes program használ olyan nyelvi elemeket, amik megsértik az implementáció C nyelvvel szemben bevezetett korlátozásait, így ezzel kapcsolatos mérést nem lehetett végezni.

A fennmaradó programokból szűrve, 6 példaprogram lett kiválasztva mérésre, mind az *ssh_simplified* kategóriából, mert ezeket nem érintették az implementáció korlátozásai. Ezek közül 2 helyes, 4 hibás volt a követelmény megsértése alapján. A 6 programból mindösszesen 1 alkalommal sikerült bizonyítania a modellellenőrzőnek egy órás futásidő korlátot beállítva a helyességet. Ezzel az egy alkalommal megtalálta a hibát a szoftverben. A további 2 helyes, illetve 3 hibás program esetén tesztek lettek generálva.

A bemeneti változók határérték analízisén alapuló tesztgenerálás minden program esetében nagyjából 130 tesztesetet generált, azonban minden teszt nem meggyőző eredménnyel ért véget (se nem volt sikeres, se nem volt sikertelen). Hasonlóan, az robusztusság tesztelés alapján egyenként generált nagyjából 270 teszteset közül egyik sem volt meggyőző. Ennek okaira a következő a hipotézisek lettek felállítva.

Elsőként, az összes kiválasztott program struktúrája úgy nézett ki, hogy volt egy ciklus, és azon belül számos elágazás. Az elágazásokon belül szereplő programkódokban viszont

[1] https://www.sosy-lab.org/
[2] https://github.com/sosy-lab/sv-benchmarks

voltak nemdeterminisztikus értékadások, amik a ciklus minden iterációjában meg lettek hívva. A hipotézis az, hogy a modellellenőrző nem tudta kellően kifejteni az állapotteret, és a tesztelés futása közben a teszteseteknek nem sikerült terminálnia a ciklust, ezért többször lett meghívva a ciklus belsejében lévő nemdeterminisztikus értékadó függvény, mint ahány értéket tárolt, ezáltal megszakítva a teszt futását (lásd: 4.1.3).

A második hipotézis a változók értelmezéséhez tartozik. A szoftver számos változója közül néhány értékkészlete néhány diszkrét érték felvételét engedte meg. A programokban csupán egyenlőség vizsgálatok szerepeltek a feltételekben, így viszont a minimum, maximum kereséseknél túlságosan megkötött volt e változók értéke, valójában csupán egy érték felvételét engedték meg. Így pedig a tesztelés során a program nem tudott olyan útvonalakat bejárni, amik nem voltak részei a kifejtett állapottérben. Azonban az állapottérben nem volt hiba a modellellenőrző alapján, így a teszteset sem tudott találni.

A harmadik stratégia, a túlcsordulás tesztelése viszont mindegyik programban talált hibát, programonként átlagosan 40-et. A tesztelt programok nem voltak kategorizálva a túlcsordulás tesztelésére, viszont a generált tesztesetek közül mindegyik kimutatott legalább egy túlcsordulást.

## 4.4. Konklúzió

Munkám eredményeképpen előállít egy prototípus implementáció, amely már képes tesztgenerálással támogatni a formális verifikációt. Az esettanulmány megmutatta, hogy a megközelítés hatékonyan alkalmazható szoftverekben a hibák megtalálására. Azonban a benchmark modellek nagyszámú nemdeterminisztikus értékadása a ciklusokban egyelőre még problémát jelent az algoritmusom számára. Emellett sajnos a kisszámú támogatott utasítás is problémát jelent a széles körű alkalmazhatóság szempontjából, ez elsősorban implementációs feladatokat jelent a jövőben. További megfigyelés, hogy az állapotgépeket megvalósító programokat a sok egyenlőségvizsgálat miatt szintén nem tudja hatékonyan kezelni. A jövőben további méréseket fogok végezni az implementáció továbbfejlesztéséhez.

# 5. Összefoglalás

A munkám a szoftverek helyességét ellenőrző technikák vizsgálata, és hatékonyságuk javítása volt. Ennek során egy olyan módszer lett kifejlesztve, ami képes két merőben eltérő megközelítés előnyeinek ötvözésére.

Az általam adott megközelítés az absztrakció alapú modellellenőrzés által előállított információkat felhasználva próbált a programban hibákat kereső teszteket generálni, feltéve, hogy a formális ellenőrzés nem járt sikerrel. A tesztek generálásához a bemeneti változók, illetve aritmetikai feltételekben előforduló változók határérték analízise, valamint az aritmetikai műveletek túlcsordulásának lehetősége lett felhasználva.

A megközelítés újszerűségét az algoritmusok innovatív kombinációja adja, amely során a tradicionális tesztgeneráló megközelítésekhez képest sokkal kevesebb, célzott tesztesetet állítok elő. Ezáltal a megközelítésem középen helyezkedik el a formális módszerek által nyújtott precizitás és a tesztelés által nyújtott hatékonyság között.

Az adott megközelítés alapján elkészült egy implementáció is, ami képes C nyelven írt programok egy részének ellenőrzésére. Az elkészült implementációt szoftverellenőrző rendszerek vizsgálatára használt példákon ellenőriztem, megállapítva annak jelenlegi korlátait.

## 5.1. Jövőbeli munka

A jövőbeli munka fő célja a jelenlegi implementáció korlátainak csökkentése több téren.

- Meg kell vizsgálnom, hogy a jelenlegi CEGAR alapú algoritmus fejleszthető-e, esetleg helyettesíthető-e másik algoritmussal, ami hatékonyabban támogatja a tesztek generálását.
- Meg kell vizsgálnom további tesztgenerálási stratégiák létjogosultságát, amik célzottabban képesek hibák megtalálására, mint a jelenleg alkalmazottak.
- Az implementáció egyes részeit kell tovább fejleszteni, esetleg C-től különböző nyelvek támogatására, illetve a C-t érintő korlátozások csökkentése érdekében.
- Mérésekkel meg kell vizsgálnom, hogy a tradicionális forráskód alapú tesztgenerálási stratégiákhoz viszonyítva milyen az én algoritmusom által adott tesztek számossága/hibafedése.

# Köszönetnyilvánítás

Szeretném megköszönni konzulenseimnek, Vörös Andrásnak, és Hajdu Ákosnak a rendkívül sok támogatást és segítséget, amit a munkám során kaptam tőlük. Továbbá szeretném megköszönni Tóth Tamásnak a munkám során nyújtott segítséget és azt a rengeteg tudást, amit megtanulhattam Tőle a formális verifikációs algoritmusokkal kapcsolatban.

# Irodalomjegyzék


[1] A. R. M. Z. Bradley, The Calculus of Computation, Springer, 2007, pp. 10-32.

[2] C. S. R. S. S. &. T. C. Barrett, „Satisfiability modulo theories,” in *Handbook of Satisfiability*, 2009, pp. 825-885.

[3] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM,* vol. 54, no. 9, pp. 69-77, 2011.

[4] S.-L. Initaitive, „The Satisfiable Modulo Theories Library,” [Online]. Elérhető: http://smtlib.cs.uiowa.edu/solvers.shtml. [Hozzáférés dátuma: 2018.10.24.].

[5] V. Chvátal, Linear Programming, W.H. Freeman, 1983.

[6] B. Meindl és M. Templ, „Analysis of Commercial and Free and Open Source Solvers for the Cell Suppression Problem,” *Transactions on Data Privacy,*vol. 6, no. 2, pp. 147-159, 2013.

[7] D. Beyer és S. Löwe, „Explicit-State Software Model Checking Based on CEGAR and Interpolation,” *Lecture Notes in Computer Science, vol.* 7793, pp. 146-162, 2013.

[8] C. Baier és J.-P. Katoen, *Principles of Model Checking,* MIT Press, 2008.

[9] E. H. T. V. H. B. R. Clarke, *Handbook of Model Checking,* Springer, 2018.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu és H. Veith, „Counterexample-guided Abstraction Refinement for Symbolic Model Checking,” *J. ACM, vol.* 50, no. 5 pp. 752-794, 2003.

[11] Á. Hajdu, T. Tóth, A. Vörös és I. Majzik, „A configurable CEGAR framework with interpolation-based refinements,” *Lecture Notes in Computer Science,* vol. 9688, pp. 158-174, 2016.

[12] S. Graf és H. Saidi, „Construction of abstract state graphs with PVS,” *Lecture Notes in Computer Science, vol.* 1254, pp. 72-83, 1997.

[13] D. Beyer és M. Dangl, „SMT-based Software Model Checking: An Experimental Comparison of Four Algorithms,” *Lecture Notes in Computer Science, vol.* 9971, 2016.

[14] K. L. McMillan, „Applications of Craig interpolants in model checking.,” *Lecture Notes in Computer Science, vol.* 3440, pp. 1-12, 2005.

[15] I. S. T. Q. Board, *Certified Tester Foundation Level Syllabus,* 2018.

[16] J. C. King, „Symbolic Execution and Program Testing,” *Communications of the ACM, vol.* 19, no. 7, pp. 385-394, 1976.

[17] C. Cadar és K. Sen, „Symbolic Execution for Software Testing: Three Decades Later,” *Communications of the ACM, vol.* 56, no. 2, pp. 82-90, 2013.

[18] K. Sen, „Concolic testing,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007.

[19] X. Qu és B. Robinson, „A Case Study of Concolic Testing Tools and their Limitations,” in *International Symposium on Empirical Software Engineering and Measurement*, 2011.

[20] N. Tillmann, J. de Halleux és T. Xie, *Pex for Fun: Engineering an Automated Testing Tool for Serious Games in Computer Science,* 2018.

[21] J. de Halleux és N. Tillmann, „Moles: Tool-Assisted Environment Isolation with Closures,” *Lecture Notes in Computer Science, vol.* 6141, pp. 253-270, 2010.

[22] C. Cadar, D. Dunbar és D. Engler, „KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[23] G. LiIndradeep, G. Sreeranga és P. Rajan, „KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs,” *Lecture Notes in Computer Science, vol.* 6806, pp. 609-615, 2011.

[24] D. Sima, „The design space of register renaming techniques,” *IEEE Micro, vol.* 20, no. 5, pp. 70-83, 2000.

[25] G. Sallai, T. Tóth, Á. Hajdu és A. Vörös, *Development of a Verification Compiler for C Programs,* 2016. Elérhető: http://docs.inf.mit.bme.hu/theta/publications/sallaigyBsc2016.pdf. [Hozzáférés dátuma: 2018.10.24.]

[26] C. Lattner és V. S. Adve, „The LLVM Compiler Framework and Infrastructure Tutorial,” in *LCPC*, 2004.

[27] T. Tóth, Á. Hajdu, A. Vörös, Z. Micskei és I. Majzik, „Theta: a Framework for Abstraction Refinement-Based Model Checking,” in *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, 2017.

[28] L. M. d. Moura és N. Bjørner, „Z3: An Efficient SMT Solver,” *TACAS,* 2008.

[29] T. C. Team, „Clang 8 documentation,” The Clang Team, [Online]. Elérhető: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. [Hozzáférés dátuma: 2018.10.24.].

[30] D. Beyer, „Software Verification with Validation of Results,” in *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2017.