



SZAKDOLGOZAT FELADAT

Jakab Richárd Benjámin
Mérnök-informatikus hallgató részére

K-indukciós algoritmus fejlesztése a Theta verifikációs keretrendszerben

Szoftveralapú kritikus rendszerekben a szoftverek és a rendszertervek helyességének vizsgálata fontos feladat. Erre használható a formális verifikáció, amely a matematika eszköztárát használja a helyesség ellenőrzésére. Formális verifikációs keretrendszert egyetemünkön is fejlesztenek, ez a Theta verifikációs keretrendszer. Ebben többféle absztrakciós és egyéb keresési algoritmus szolgál a rendszerek helyességének ellenőrzésére. A k-indukción alapuló verifikáció módszere is egyfajta verifikációs algoritmus, amely kombinálja a hátrafelé keresést az absztrakcióval.

A hallgató feladata megismerkedni a k-indukció alapú formális verifikáció módszerével és egy prototípus algoritmus implementálása a Theta verifikációs keretrendszerbe.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a k-indukciós algoritmus működését az irodalom alapján.
- Mutassa be a Theta verifikációs keretrendszert.
- Vizsgálja meg, hogy a Theta verifikációs keretrendszerbe milyen módon lehetne integrálni a k-indukció módszerét.
- Tervezzon meg a Theta keretrendszer kiegészítését az új algoritmussal.
- Készítse el a megtervezett rendszer prototípusát.
- Igazolja a megközelítés alkalmazhatóságát egy esettanulmány segítségével.

Tanszéki konzulens: Dr. Vörös András, adjunktus

Külső konzulens:

Budapest, 2020.10.11.

.....
Dr. Dabóczi Tamás
tanszékvezető
egyetemi tanár, DSc



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

K-indukciós algoritmus fejlesztése a Theta verifikációs keretrendszerben

SZAKDOLGOZAT

Készítette
Jakab Richárd Benjámin

Konzulens
Dr. Vörös András

2020. december 10.

Tartalomjegyzék

| | |
|---|-----------|
| Kivonat | i |
| Abstract | ii |
| 1. Bevezetés | 1 |
| 1.1. Feladatleírás | 1 |
| 1.2. Dolgozat felépítése | 1 |
| 2. Háttérismeretek | 3 |
| 2.1. Szoftververifikáció | 3 |
| 2.2. K-indukció | 4 |
| 2.3. A probléma formalizálása | 4 |
| 3. K-indukciós algoritmus szoftverellenőrzésre | 6 |
| 3.1. Control Flow Automata | 6 |
| 3.1.1. Assert | 8 |
| 3.1.2. Állapottér | 9 |
| 3.2. Az algoritmus formalizálása | 10 |
| 3.2.1. Elérhetőség vizsgálata | 11 |
| 3.2.2. Algoritmus | 11 |
| 4. Implementáció | 13 |
| 4.1. Theta keretrendszer | 13 |
| 4.2. A program implementálása | 14 |
| 4.2.1. Bemenet | 14 |
| 4.2.2. Architektúra | 14 |
| 4.2.3. Működés | 19 |
| 4.2.4. Kimenet | 23 |
| 5. Kiértékelés | 24 |
| 5.1. Tesztelés | 24 |
| 5.2. Eredmények | 25 |
| 6. Összefoglaló | 31 |
| Köszönetnyilvánítás | 32 |
| Irodalomjegyzék | 33 |
| Függelék | 35 |
| F.1. A parancssori interfész felülete | 35 |

HALLGATÓI NYILATKOZAT

Alulírott *Jakab Richárd Benjámín*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2020. december 10.

Jakab Richárd Benjámín
hallgató

Kivonat

Az elmúlt években növekvő tendenciát mutat a biztonságkritikus rendszerek száma a világban, ahol a nem megengedett viselkedés katasztrofális eredménnyel is járhat. Ezért ezeknek a rendszereknek tesztelése kritikus fontosságú. Viszont méretükből és komplexitásukból adódóan általában lehetetlen a tesztelésük, ezért azokat modellezni vagyunk kénytelenek. Egy ilyen modellezés a Control Flow Automaton mely gráf formájában reprezentálja a programkódokat, így magát a tesztelni kívánt programot is.

Az én feladatom egy program készítése volt, mely kihasználja a Theta verifikációs keretrendszer nyújtotta lehetőségeket, és amely a k-indukció nevezetű algoritmust használja CFA modellek helyességeinek a bebizonyítására. A programom a modelleken a biztonság követelményét ellenőrzi, vagyis azt nézi, hogy a modell hibaállapota nem elérhető a kezdőállapotból. Miképp a CFA modell egy az egyben megfeleltethető az eredeti programkódjával, így ha az előbbire belátjuk a hibamentes működést, úgy belátjuk azt az utóbbira is.

Abstract

Same in English...

1. fejezet

Bevezetés

1.1. Feladateleírás

A körülöttünk lévő világban számos helyen találunk olyan informatikai rendszereket, melyeknél a meghibásodás (hibás működés) következménye elfogadhatatlan. Hagyományosan ilyen területek az egészségügyi alkalmazások, légi közlekedés, atomenergia ipar, fegyverrendszerek stb., vagy például a szoftverrendszerek egy részcsoportha, így az autonóm járművezetés. Ezeket a rendszereket biztonságkritikus rendszereknek nevezzük, és létfontosságú a specifikációnak megfelelő működésük ellenőrzése.

Elmondható, hogy a legtöbb biztonságkritikus rendszer rendelkezik komplex szoftverrendszerrel, melyek ugyanúgy biztonságkritikusak önmagukban is. Ezek ellenőrzésével a szoftververifikáció foglalkozik, mely azt vizsgálja, hogy egy szoftverrendszer megfelel-e a feléje támasztott követelményeknek.

Ennek ellenőrzésére különböző verifikációs technikák szolgálnak. Ezek egyike a modellellenőrzés, mely során a rendszer egy matematikai modelljét vizsgálva lehet azon különböző formalizált követelmények teljesülését ellenőrizni.

A munkám célja egy program leimplementálása mely a követelmények közül a biztonság követelmény teljesülését ellenőrzi. Ezt a programot a BME VIK Méréstechnika és Információs Rendszerek Tanszék¹ Hibatűrő Rendszerek Kutatócsoportja² által fejlesztett *Theta*³ verifikációs keretrendszerben fejlesztettem, majd azt széleskörűen teszteltem.

A munkámat három részre tagolhatjuk, melyet a szakdolgozatom felépítése is követ: először elmerültem a szoftververifikáció és modellezés tématerületében, kiemelten foglalkozva a *k*-indukció alapú szoftververifikációval, aztán a szakirodalom által bemutatott algoritmust leimplementáltam a *Theta* keretrendszerben, majd végezetül széleskörű tesztelés alá vettem.

1.2. Dolgozat felépítése

A dolgozat az alábbi részletesebb tartalmi felosztásban tárgyalja a fentebb felvázolt folyamatot:

- A második fejezetben a dolgozatomhoz szükséges háttérismereteket mutatom be.

¹<https://www.mit.bme.hu/>

²<https://www.mit.bme.hu/research/ftsrg>

³<https://github.com/FTSRG/theta>

- A harmadik fejezetben ismertetem a Control Flow Automata koncepcióját és az algoritmusomat.
- A negyedik fejezetben bemutatom röviden a Theta keretrendszert illetve az implementált programomat.
- Az ötödik fejezetben bemutatom és értékelem az algoritmusom teszteredményeit.

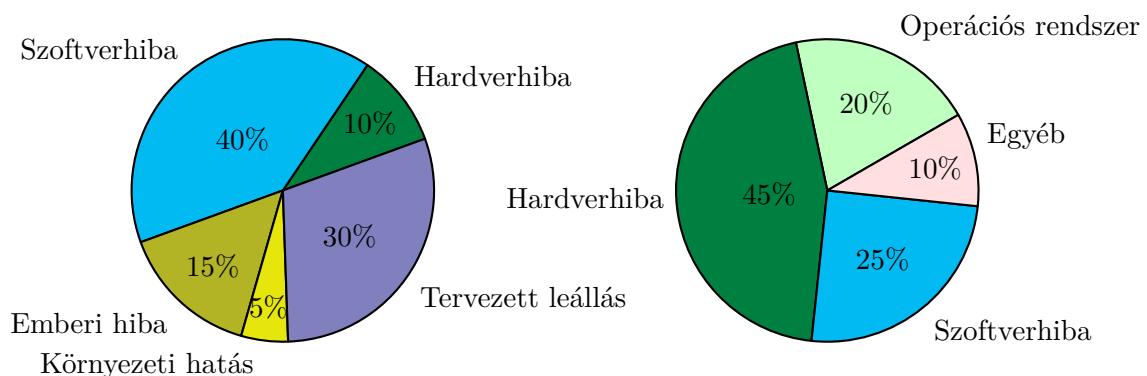
2. fejezet

Háttérismeretek

Ebben a fejezetben a dolgozat további részeinek megértéséhez szükséges elméleti előismereteket mutatom be. Először kitérek a szoftververifikációra a (2.1) alfejezetben, majd a k -indukció nevű matematikai módszert [14] ismertetem a (2.2) alfejezetben, majd végül formalizálom a problémát a (2.3) alfejezetben.

2.1. Szoftververifikáció

Elemzések azt mutatják [6], hogy kliens-szerver illetve beágyazott rendszerek esetén a hiba oka nagy eséllyel visszavezethető szoftverhibára. Ezt mutatja a (2.1) ábra, melyről látszik, hogy nagy igény van a minél pontosabb szoftververifikációs megoldásokra.



2.1. ábra. A leállások illetve hibák okainak százalékos eloszlása. Baloldalt a kliens-szervereken, jobb oldalt a beágyazott rendszereken mért adatok.

A verifikáció az ISO-9000 szabvány [9] szerint nem más, mint egy megerősítés olyan objektív bizonyítékok megvizsgálása által, amelyek a specifikált követelmények beteljesülésére irányulnak. Ebből adódóan a tesztelés, a tesztek futtatása alapvető igény, hogy a szoftvert verifikálni mondhassuk.

A verifikáció egyik legfőbb funkciója, hogy követni tudjuk, hogy a tervezett végtermékből mennyit implementáltunk, illetve mennyit teszteltünk már le [1]. Ezt gyakran használják a szoftverfejlesztésben közismert V-modell alakú fejlesztési folyamatban. A verifikáció során a helyes működést követelmények formájában tudjuk megfogalmazni. Pár példa: [7]:

- Rendelkezésre állás (*availability*) – Helyes szolgáltatás valószínűsége
- Megbízhatóság (*reliability*) – Folyamatos helyes szolgáltatás valószínűsége

- Biztonság (*safety*) – Elfogadhatatlan kockázattól való mentesség
- Integritás (*integrity*) – Hibás változás, változtatás elkerülésének lehetősége
- Karbantarthatóság (*maintainability*) – Javítás és fejlesztés lehetősége

Modellellenőrzés során a tesztelni kívánt rendszert modellezzük, és azon hajtjuk végre a fentebb említett követelmények ellenőrzését. A dolgozatomban Control Flow Automata modelleket tesztelek biztonságosság tulajdonságra. A modellek eredetileg C illetve PLC programozási nyelveken írt programok voltak.

A fejezet további részében az általam használt algoritmus elméleti oldalát ismertetem.

2.2. K-indukció

Tekintsük az alább látható teljes indukció tételét a természetes számok halmaza fölött (kiegészítve 0-val):

$$P(0) \wedge \forall n(P(n) \Rightarrow P(n+1)) \Rightarrow \forall nP(n). \quad (2.1)$$

Lényege, hogy megnézzük az első lépésre teljesül-e a feltétel (az angol szakirodalomban ez a *base-case*). Ha igen, akkor megnézzük ennek ismeretében azt, hogy az $n+1$. lépés következik-e az n . lépésből (indukciós lépés – *induction case*). Ha sikerül ezt belátnunk, akkor készen vagyunk, bebizonyítottuk az összes lépésre a feltételt.

Ezt tovább gondolva megtehetjük azt, hogy az első két lépésre nézzük meg, hogy teljesítik-e a feltételt:

$$P(0) \wedge P(1) \wedge \forall n((P(n) \wedge P(n+1)) \Rightarrow P(n+2)) \Rightarrow \forall nP(n). \quad (2.2)$$

Ezt az elvet általánosíthatjuk k lépésre, $k \in \mathbb{N} : k \geq 1$, melyet a irodalom [14] k -indukciónak nevez, formálisan:¹

$$\left(\bigwedge_{i=0}^{k-1} P(i) \right) \wedge \forall n \left(\left(\bigwedge_{i=0}^{k-1} P(n+i) \right) \Rightarrow P(n+k) \right) \Rightarrow \forall nP(n). \quad (2.3)$$

2.3. A probléma formalizálása

Ahhoz, hogy a problémát precízebben megfogalmazhassuk, szükség van jelölések és fogalmak bevezetésére [12]. Adott egy tranzakciós relációkból felépülő gráf, melyben $T(x, y)$ -al jelöljük azt, ha létezik egy, az $x \in S$ állapotból az $y \in S$ állapotba mutató tranzakciós reláció, ahol S az állapotok halmazát jelöli T pedig a tranzakciós relációt. Így már tudjuk definiálni az útvonal fogalmát, mely állapotok sorozatát jelenti T -n keresztül:

$$utvonal(s_{[0..n]}) \doteq \bigwedge_{0 \leq i < n} T(s_i, s_{i+1}), \quad (2.4)$$

ahol $s_i \in S$ és az $s_{[0..n]}$ rövidítés az (s_0, s_1, \dots, s_n) állapotsorozatot jelöli. Az útvonal n hosszúságú, ha n darab tranzakcióból áll. A nulla hosszúságú útvonal egy darab állapotot

¹A k -indukció helyességének a bizonyítására a dolgozatomban nem térek ki.

tartalmaz és nem értelmezzük rajta a tranzakció műveletét.

Definiáljuk emellett a ciklusmentes útvonalat is: olyan útvonal, melyben minden állapot maximum csak egyszer szerepelhet:

$$cmUtvonal(s_{[0..n]}) \doteq utvonal(s_{[0..n]}) \wedge \bigwedge_{0 \leq i < j \leq n} s_i \neq s_j. \quad (2.5)$$

A továbbiakban lesz olyan, mikor egy útvonal alatt nem csak azt értjük, hogy az tranzakciók sorozata, hanem annak létezését is jelöli. Így, az $utvonal_i(s_0, s_i)$ alatt azt jelöljük, hogy *létezik* egy útvonal s_0 -ból s_i -be, mely i darab T -ből áll.

Feltételezzük, hogy T a teljes állapottérre értelmezve van, tehát minden állapotnak (a kezdőállapotokat leszámítva) van egy szülőállapota T -n keresztül. Jelöljük I -vel a kezdőállapotokat, és azt vizsgáljuk, hogy az állapotok teljesítik-e a P tulajdonságot.

A problémát informálisan a következőképp foglalhatjuk össze: beszeretnénk azt látni, hogy ha egy kezdőállapotból elindulunk, akkor a tranzakciós relációkon keresztül csak olyan állapotba fogunk eljutni, mely kielégíti P -t. Formálisan a következőt akarjuk belátni:

$$\forall i : \forall s_0 \dots s_i : (I(s_0) \wedge utvonal(s_{[0..i]}) \rightarrow P(s_i)), \quad (2.6)$$

ahol $i \geq 0$. Később látni fogjuk, hogy az algoritmus felhasználja ennek a megfordítottját is: a „rossz” állapotokból (hibaállapotokból) elindulunk visszafelé, és azt vizsgáljuk, hogy elérjük-e valamelyik kezdőállapotot:

$$\forall i : \forall s_0 \dots s_i : (\neg I(s_0) \leftarrow utvonal(s_{[0..i]}) \wedge \neg P(s_i)), \quad (2.7)$$

ahol $\neg I(s_0)$ azt jelenti, hogy s_0 nem kezdőállapot (nem teljesíti a „kezdőállapot tulajdonságot”), illetve $\neg P(s_i)$ azt, hogy s_i nem elégíti ki a P tulajdonságot. A két egyenlet ekvivalens és összehetők úgy, hogy azon a probléma szemléletesebb és szimmetrikusabb legyen:

$$\forall i : \forall s_0 \dots s_i : \neg(I(s_0) \wedge utvonal(s_{[0..i]}) \wedge \neg P(s_i)). \quad (2.8)$$

Azaz szavakkal elmondva – azt akarjuk megmutatni, hogy *nem létezik* olyan útvonal, mely kezdőállapotból indul és egy *nem- P* állapotba jut.

3. fejezet

K-indukciós algoritmus szoftverellenőrzésre

Ebben a fejezetben bemutatom azokat a technológiákat, melyek szükségesek a programom algoritmikus részének a megértéséhez. Először kitérek a Control Flow Automata modellezés részleteire (Alfejezet 3.1), aztán az előző fejezetben bemutatott jelölésrendszerrel formalizálom és ismertetem az algoritmust (Alfejezet 3.2).

3.1. Control Flow Automata

A programokat sokféleképpen ábrázolhatjuk [5]. Legismertebb a programkód, melyet az ember könnyen, gyorsan tud írni olvasni, szemben a bájtkóddal, melyet a számítógép tud jóval hatékonyabban kezelni. A szoftveres modellellenőrzés elvégzéséhez a programkódot matematikailag pontos, formális ábrázolásban kell megadni, melyet a számítógép is jól tud használni. Egy széleskörűen ismert és használt ábrázolásmód a *Control Flow Automaton* (CFA), mely egy gráf alapú ábrázolást biztosít a programokhoz.

Szintaxis. A CFA formálisan egy $CFA = (V, H, I_0, E)$ négyes [2], ahol

- $V = \{v_1, v_2, \dots\}$ a változók halmaza. Mindegyik $v_i \in V$ változó rendelkezik egy D_{v_i} doménnel, mely megszabja, hogy v_i milyen értékeket vehet fel,
- H a helyek halmaza,
- $I_0 \in H$ a kezdőhelye a gráfnak, a program belépőpontját jelöli,
- $E \subseteq H \times U \times H$ az irányított élek halmaza, melyek helyeket kötnek össze és a változókra vonatkozó utasításokkal vannak felcímkézve.

Utasítások. Háromféle utasítást különböztettem meg a dolgozatomban:

- A *hozzárendelés* utasítás a $v_i := kif$ összefüggéssel írható le. Azt jelöli, hogy a baloldali $v_i \in V$ változóhoz hozzárendeljük a jobb oldali kifejezést. Fontos, hogy a *kif* kifejezésnek is ugyanolyan doménnel kell rendelkeznie, mint a v_i változónak.
- A *feltevés* operátor a $[cond]$ formában írható le, ahol *cond* egy bináris (*Boolean*) kifejezés (feltétel). Ha egy él rendelkezik $[cond]$ feltétellel, akkor abban az esetben csakis akkor süllhet el (kerülünk át az egyik helyről a másikra), ha a feltétel teljesül. A feltétel egyik változóra sem hat ki, azok értékein nem változtat.

- A *havoc* operátor a *havoc* v_i formában írható le, ahol $v_i \in V$ egy változó. A *havoc* hozzárendel a v_i változóhoz egy nem-determinisztikus értéket, a többi változót érintetlenül hagyja. Például arra lehet használni, mikor szimulálni szeretnénk a felhasználói bemenetet.

Ha szeretnénk egy olyan élet húzni két hely között, mely minden körülmények között elsül, akkor azt könnyen megtehetjük úgy, hogy nem adunk neki utasítást, vagy úgy is ha egy `[true]` feltételt adunk neki.

Grafikai megjelenítés. A helyeket körök, az éleket nyilak jelölik. Az egyes élek felett illetve mellett láthatóak az utasítások, amely jelen esetben hozzárendelés vagy feltevés. A kezdőállapotot egy bejövő nyíllal jelöljük. [5].

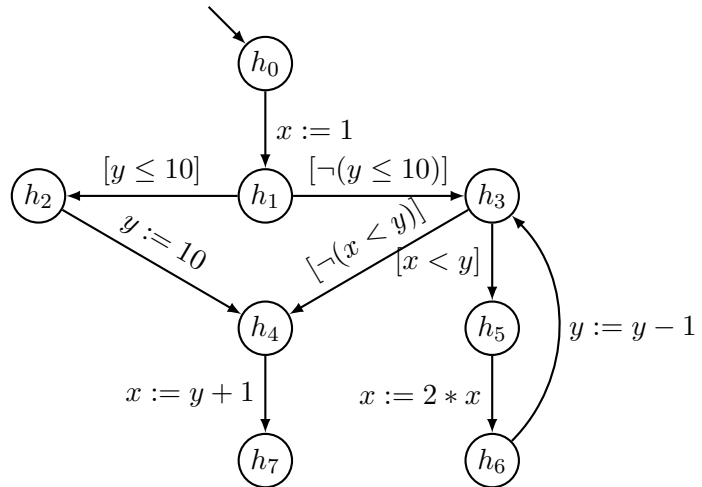
Példa 1. Egy C nyelvű program és egy hozzátartozó CFA látható a (3.1) ábrán. A kezdőhely a h_0 , a termináló hely a h_7 , mely lehet végső- (final location) illetve hibahely (error location). Egy útvonal a kezdőhelytől a h_4 helyre leírható úgy, hogy $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_4$. A h_1 helyen egy elágazást figyelhetünk meg, ahol ha a $[y \leq 10]$ feltétel teljesül, akkor úgy a program a h_1 helyről továbbmegy a h_2 helyre, míg ha nem teljesül, akkor a h_3 helyre kerül a vezérlés. Az elágazásokban a kimenő élekre a feltételek úgy vannak megfogalmazva, hogy míg az egyikben az eredeti feltétel, addig a másikon annak a negáltja figyelhető meg. Ez azért van így, hogy szemléltesse az ábra, hogy ezt algoritmusok fogják feldolgozni, melyeknek könnyebb az egymást kizáró feltételek vizsgálata ebben a formátumban.

```

1  int x = 1;
2  if (y <= 10) {
3      y = 10;
4  } else {
5      while (x < y) {
6          x = 2 * x;
7          y = y - 1;
8      }
9  }
10 x = y + 1;

```

(a) Egyszerű C program.



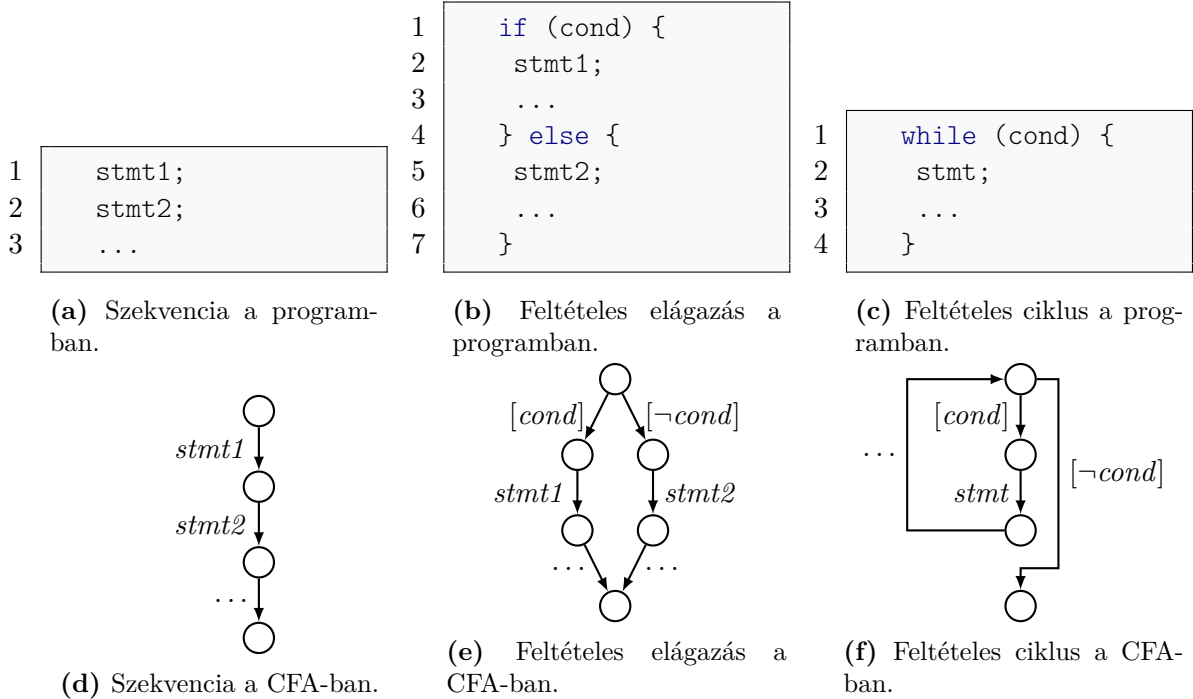
(b) A program CFA ábrázolása.

3.1. ábra. Egy C program és a hozzátartozó Control Flow Automaton (CFA).

Programábrázolás. A (3.2) ábra megmutatja, hogy az alap elemei a strukturált programozásnak miként képezhetőek le CFA alakba [5].

- *Szekvenciális* állításokat (3.2a és 3.2d ábra) úttal reprezentáljuk, mely helyek és élek közt alternál.
- *Feltételes elágazásokat* (pl. *ha-akkor* állítások, 3.2b és 3.2e ábra) különváltó utakkal tudjuk reprezentálni őrfeltételekkel.

- *Feltételes ciklusokat* (3.2c és 3.2f ábra) a CFA-ban körökkel tudunk ábrázolni. Egy vezérlési hely felel a ciklusfejért, amelyből két kimenő él fut. Az egyik bemegy a ciklusba, a másik pedig kilép abból. A ciklusban további szekvenciák, elágazások vagy akár újabb ciklusok is lehetnek, azonban az út mindig visszatér a ciklusfejhez.



3.2. ábra. A strukturált programozás elemei (szekvencia, feltételes elágazás, feltételes ciklus) és a megvalósításuk CFA modelleken.

3.1.1. Assert

A verifikáció célja általában az, hogy a programban valamilyen tulajdonság teljesülését megcáfolja vagy bizonyítsa. Ehhez precízen meg kell fogalmaznunk, hogy pontosan milyen tulajdonságot szeretnénk ellenőrizni. Ezt megtehetjük az *assert*-tel, mely azt ellenőrzi, hogy bizonyos változókon értelmezett feltétel teljesül-e.

A CFA-ban az *assert* egy speciális döntésként értelmezhető. Ha a feltétel igaz, a program a következő helynél folytatódik, ha pedig nem, akkor egy különálló $h_e \in H$ hibahelyre jutunk. Ha több ilyen *assert* szerepel a programban, akkor a CFA hibahelyei összevonhatók: létrehozunk egy új hibahelyet, az összes többi hibahelyből vezetünk ide utasítás nélküli éleket, majd a többi hibahelyet visszaminősítjük egyszerű vezérlési helynek. Az egyetlen megmaradt hibahely pedig egy, az Algoritmuselméletből¹ jól ismert nyelvhely lesz.

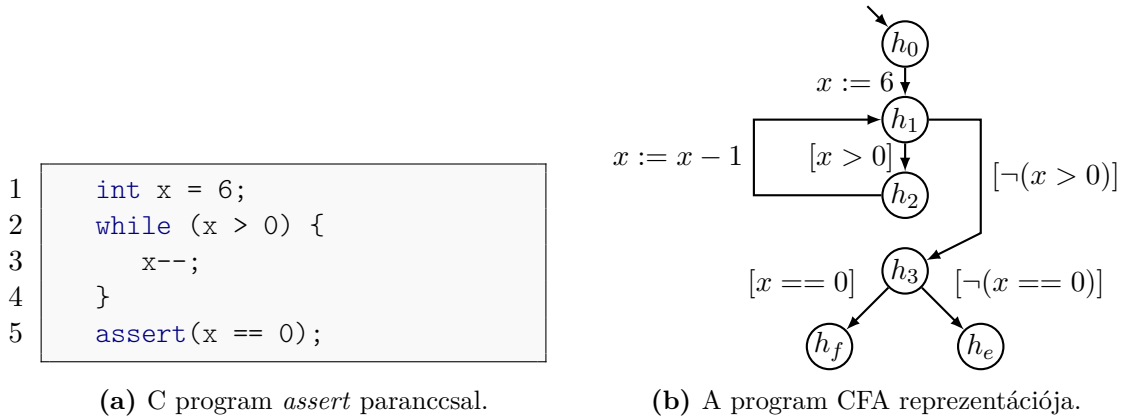
Vegyük észre, hogy a CFA megfelel egy, az előző fejezetben említett tranzakciós modellnek: vannak benne állapotok (helyek), melyek közt relációk húzódnak (élek). A kezdőhely az az állapot mely kielégíti a kezdőállapot tulajdonságot, illetve a hibahely az az állapot, mely nem teljesíti az *assert*-ben megfogalmazott feltételt a bejárás egy olyan pontján, ahol annak teljesítenie kéne.

Innentől kezdve az lesz a vizsgálatunk célja, hogy megállapítsuk, elérhető-e a hibahely az adott CFA-ban. Ezért a (CFA, h_e) párost verifikációs feladatnak nevezzük – a program

¹<http://www.cs.bme.hu/algel/>

helyes, ha a hibahely nem elérhető, így tehát a modell *biztonságos*, különben pedig *hibás* tehát a modell *nem biztonságos*.

Példa 2. Figyeljük meg a 3.3a ábrán az `assert` parancsot az ötödik programsorban. A programkódhoz tartozó CFA a 3.3b ábrán található, ahol a h_3 vezérlési helynél látható elágazás felel meg a program `assert` parancsának. Ha a feltétel teljesül, akkor a h_f végső vezérlési helyre kerülünk és vége az ellenőrzésnek egy „helyes” kimenettel, míg ha nem teljesül a feltétel, akkor a h_e hibahelyre jutunk és a verifikációs feladat egy „hibás” eredménnyel zárul, ekkor a program implementációján változtatni kell.



3.3. ábra. C program `assert` paranccsal és a hozzá tartozó CFA modell. A program `assert` parancsát a CFA modell h_3 helye jelöli, mely hibás működés esetén a h_e hibahelyre viszi a vezérlést.

3.1.2. Állapottér

A helyek ($h_0, h_1, \dots, h_n \in H$) a program futásának egyes állomásait jelölik, amik között a változók értékei változnak. Így nem tudjuk, hogy egy adott helyen például az egyes x, y változók értéke mennyi, mert azt a hely nem tárolja. Ezért bevezetjük az **állapot** fogalmát, mely két elemből fog állni: (1) az aktuális $h \in H$ helyből illetve (2) a $v \in V$ változók aktuális hozzárendelt értékeiből. Így, az összes lehetséges állapot A megkapható úgy, hogy: $A = L \times D_{v_1} \times \dots \times D_{v_n}$. Az aktuális állapot a következő listából áll: $(h, d_1, \dots, d_n) \in A$, ahol $h \in H$ az aktuális hely, $d_i \in D_{v_i}$ pedig a v_i változóhoz hozzárendelt értéke a D_{v_i} doménből.

Az algoritmusom csupán a helyek alkotta gráfot járja be és vizsgálja annak csúcspontjainak az elérhetőségét, az állapotokat (így a változók alakulását) nem veszi figyelembe. Ez azért van, mert már kevés hely és változó esetén is állapottér-robbanásról beszélhetünk, ugyanis az A állapothalmaz mértékét a következőképp kaphatjuk meg: $|A| = |H| \times |D_{v_1}| \times \dots \times |D_{v_n}|$. Ami, ha nézünk egy kis példát, a következőképp alakul: vegyünk egy kis CFA-t 20 darab helyel és 5 darab `int` változóval. Az `int` 4 byte-on van tárolva, tehát a doménjének nagysága: $D_{int} = 2^{32}$. Így az állapottér lehetséges mérete: $|A| = 20 \times 2^{32} \times 2^{32} \times 2^{32} \times 2^{32} \times 2^{32} = 20 \times 2^{160}$. Az igaz, hogy a program nagy valószínűséggel ennek csak a töredékét fogja a futása közben érinteni, viszont (1) ez nem ad garanciát arra, hogy valóban nem fogja bejárni illetve (2) ez még egy nagyon egyszerű CFA volt, egy korszerű biztonságkritikus rendszer nagyságrendekkel nagyobb.

Emiatt a programom ciklusmentes útvonalat nem tud vizsgálni, ugyanis ahhoz, hogy megállapítsuk a ciklusmentességet, az egyes helyekhez a változók értékeit is hozzá kell rendelni tehát az állapotokat kéne nézni. Felmerülhet, hogy miért nem elég csupán azt

nézni, hogy hely ismétlődik-e az útvonalban. A helyek ismétlődhetnek az útvonalban, elég csak megnézni az előző (3.3) ábrát: a helyes működés útvonalát a következőképp írhatjuk le: $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_1 \rightarrow h_2 \rightarrow h_1 \rightarrow h_2 \rightarrow h_1 \rightarrow h_2 \rightarrow h_1 \rightarrow h_2 \rightarrow h_1 \rightarrow h_2 \rightarrow h_1 \rightarrow h_3 \rightarrow h_f$. Azért ismétlődhetnek, mert az helyes működésnek mondható, ha ugyanarra a helyre visszatérünk de más változó értékekkel. Azt viszont jó lenne kiszűrni, ha ugyanarra a helyre értünk vissza ugyanazokkal a változó hozzárendelésekkel, tehát ugyanarra az állapotr, mert ez esetben a modell bejárása közben végig egy végtelen ciklust is vizsgálni fog, amely a helyes működést nem befolyásolja, csak feleslegesen allokál CPU és memória-erőforrásokat.

3.2. Az algoritmus formalizálása

A dolgozatomban arra a kérdésre keresem a választ, hogy a (2.3)-es alfejezetben elmondottak segítségével hogy tudnánk belátni, hogy a modell a P tulajdonságra nézve biztonságos?

Ezt például úgy tehetjük, hogy megnézzük tetszőleges nemnegatív i egész esetén teljesül-e a

$$\forall s_0 \dots s_i : \neg(I(s_0) \wedge \text{utvonal}(s_{[0..i]}) \wedge \neg P(s_i)), \quad (3.1)$$

feltétel. Ha megsérül valamelyik állapotban, akkor ezzel a módszerrel meg fogjuk találni és az oda vezető útvonal ellenpélda lesz a modell P -biztonságosságára. Ez egy kíváncsi eredmény: az algoritmusnak két féle kimenetele kell, hogy legyen: vagy az, hogy a modell P -tulajdonságra nézve biztonságos (minden állapot teljesíti), vagy az, hogy a modell nem P -biztonságos, ekkor egy ellenpéldát kell adnia, mely bizonyítja, hogy a kezdőállapotból elindulva, azon végighaladva valóban egy hibaállapotba kerülünk.

Ha a rendszer P -biztonságos, akkor (3.1) minden i -re igaz lesz, hiszen nem fogunk tudni találni olyan i értéket, melyre ne teljesülne. Felvetődhet a kérdés, hogy mikortól lehet azt mondani, hogy i további növelése céltalan, mert már teljes bizonyossággal kijelenthetjük, hogy a modell P -biztonságos? Az $I(s_0) \wedge \text{utvonal}(s_{[0..i]})$ feltétel önmagában nem fog gyorsítást eredményezni: végig megy az állapottéren amilyen hosszan csak lehetséges (ezt szeretnénk lerövidíteni), tekintve, hogy minden állapotnak van egy szülőállapota a T tranzakciós reláción keresztül.

Ennél jobb stratégia, ha addig ellenőrizzük, amíg az $\text{utvonal}(s_{[e+i..e]}) \wedge \neg P(s_e)$ nem lesz ellentmondásos: egy, a P tulajdonságot sértő állapotból (hibaállapotból) kiindulva addig megyünk visszafelé, míg be nem járjuk a teljes állapotteret (ez esetben kijelenthetjük, hogy a rendszer nem- P -biztonságos), ellenben ha nem járunk be minden állapotot, akkor a rendszer P -biztonságos. Ez azzal magyarázható, hogy ha a kezdőállapot nem elérhető a hibaállapotból (mert visszafelé haladva útközben elakadunk), akkor kijelenthetjük, hogy a modell biztonságos.

A k -indukció alapú szoftververifikáció az előbb elmondottakra épül. A módszer lényege, hogy elindulunk mind a kezdőállapotból, mind a hibaállapotból: míg az előbbiből előre felé, addig az utóbbiból visszafelé. Kijelenthető, hogy a modell biztonságos, ha az előre felé haladó keresés bejárta a teljes állapotteret (azaz minden állapotot bejártunk már – *első eset*)², illetve abban az esetben is, ha a hátrafelé haladó keresés megakad (*második eset*).

²Természetesen ha közben hibaállapotba jutna, akkor a teljes modellellenőrzés megakadna, s így nem tudná bejárni a teljes állapotteret.

A (2.2)-es alfejezetben bemutatott, és így a módszer nevét adó *k-indukció* abból adódik, hogy ha a modellt bejárjuk k mélységig, és arra jutunk a fentebb említett metodika alapján, hogy a modell biztonságos, akkor kijelenthető, hogy $k + 1$ mélységre is biztonságos lesz [4], illetve mellé az is, hogy ezzel az indukciós lépést bizonyítottuk [12], tehát kijelenthetjük, hogy a modell valóban biztonságos. Ha találunk ellenpéldát, akkor azzal a modell nem-biztonságosságát láttuk be.

3.2.1. Elérhetőség vizsgálata

Definíció. Egy $s_i \in S$ állapot elérhető az $s_j \in S$ állapotból, ha létezik olyan útvonal, melynek első állapota s_j , az utolsó állapota s_i , és beadva az út tranzakciós reláció listáját bejárási sorrendben egy Sat-megoldóba az „igaz” értékkel tér vissza.

Eddig a tranzakciós relációról úgy volt szó, mint egy, az állapotok közti éleket leíró halmaz. Ezt most kibővítjük – a relációknak lehetnek állításai, melyek vagy leírnak egy utasítást ($x := x + 1$), vagy egy feltételt fogalmaznak meg ($x > 0$) vagy nem-determinisztikus értékadást képviselnek (*havoc*). Ahogy bejárjuk az utat az állapot-térben a relációkon keresztül, abban a sorrendben egymás mellé fűzzük konjunkcióval a tranzakciókat, melyről a Sat-megoldó eldönti, hogy kielégíthető-e vagy sem.

Példa 3. Ha például az útvonalunk az $s_0 \xrightarrow{T_1(s_0, s_1)} s_1 \xrightarrow{T_2(s_1, s_2)} s_2 \xrightarrow{T_3(s_2, s_3)} s_3$, és azt szeretnénk megtudni, hogy s_3 elérhető-e, akkor azt a $Sat(T_0 \wedge T_1 \wedge T_2)$ kifejezéssel meghívott Sat-megoldó fogja nekünk eldönteni, mely bináris típusú válasszal tér vissza: „igaz” választ ad ha a kifejezés kielégíthető, különben „nem” választ. Vegyük észre, hogy a $Sat(\text{utvonal}(s_0, s_1, s_2, s_3))$ kifejezés megegyezik a $Sat(T_0 \wedge T_1 \wedge T_2)$ kifejezéssel a (2.4) egyenlet miatt.

Feltételezzük, hogy a Sat-megoldó a tranzakciós reláció szekvencián kívül képes kezelni az egyes tulajdonságok teljesülését is. Így például a $Sat(I(s_0) \wedge \neg P(s_5))$ akkor lesz igaz, ha s_0 kezdőállapot és s_5 nem elégíti ki a P tulajdonságot.

3.2.2. Algoritmus

Tekintsük az (1) algoritmust, mely a kezdőállapotból indulva inkrementálisan járja be az állapotteret. Az első *if* a 3. sorban két feltételt ellenőriz, melyek egy diszjunkcióval vannak összekapcsolva:

- Az első feltétel megfelel a fentebb említett *első esetnek*: azt nézi, hogy az aktuális bejárási mélységben van-e olyan állapot, melyet nem látogattunk meg még a kezdőállapotból indulva és elérhető. Ha van, akkor akkor a $Sat(\dots)$ visszatérési értéke „igaz” lesz, mely a negált hatására „hamis”-ra fordul és folytatjuk tovább az ellenőrzést.
- A második feltétel megfelel a fentebb említett *második esetnek*. Azt vizsgálja, hogy ha az $s_e \in S$ hibaállapotból indulva eljutunk egy $s_{e+i} \in S$ állapotba, akkor az utóbbiból a hibaállapot felé visszaindulva kialakuló útvonal kielégíthető-e. Ha igen, akkor folytatjuk tovább az ellenőrzést.

A második *if* a 6. sorban azt vizsgálja, hogy az állapot, melyben éppen vagyunk (s_i) az:

- hibaállapot-e, illetve,
- a kezdőállapotból elindulva elérhető-e.

Ha elérhető és hibaállapot, a modellellenőrzés véget ért mert egy ellenpéldát találtunk, mellyel vissza is tér az algoritmus, ha nem, akkor az ellenőrzés folytatódik tovább és növeljük eggyel a bejárési mélységet.

Algorithm 1: Checking if system is P -safe

```

1  $i = 0$ 
2 while  $True$  do
3   if  $\neg Sat(I(s_0) \wedge utvonal(s_{[0..i]})) \vee \neg Sat(utvonal(s_{[e+i..e]}) \wedge \neg P(s_e))$  then
4      $\text{return } True$ 
5   end
6   if  $Sat(I(s_0) \wedge utvonal(s_{[0..i]}) \wedge \neg P(s_i))$  then
7      $\text{return } s_{[0..i]}$ 
8   end
9    $i = i + 1$ 
10 end

```

4. fejezet

Implementáció

Ebben a fejezetben bemutatom a programom technikai oldalát. Az első (4.1) alfejezetben bemutatom a keretrendszert, amiben a programomat implementáltam. A második (4.2) alfejezetben ismertetem részletesen a programomat és annak felépítését.

4.1. Theta keretrendszer

A *Theta*¹ egy nyílt forráskódú, általános célú, moduláris és konfigurálható modellellenőrző keretrendszer, melyet absztrakciós finomításon alapuló algoritmusok tervezésének és értékelésének támogatására hoztak létre a különböző formalizmusok elérhetőségi elemzéséhez.

A keretrendszer a már évek óta tartó fejlesztéseknek köszönhetően számos eszközt tud nyújtani modellellenőrzéshez:²

- `theta-cfa-cli` – Control Flow Automata hibahelyeinek az elérhetőségét vizsgálja CEGAR alapú algoritmusokkal.
- `theta-sts-cli` – *Symbolic Transition Systems* biztonsági tulajdonságainak verifikációját végzi CEGAR alapú algoritmusokkal.
- `theta-xta-cli` – Uppaal időzített automaták verifikációját lehet vele elvégezni.
- `theta-xsts-cli` – *eXtended Symbolic Transition Systems* biztonsági tulajdonságainak verifikációját végzi CEGAR alapú algoritmusokkal.

A Theta architektúrája négy rétegre osztható. Nevezetesen:

- **Formalizmusok** – A Theta legalapvetőbb elemei, melyek való-életbeli problémákat modelleznek le (pl. szoftvereket, hardvereket, protokollokat). A formalizmusok általában alacsony szintű, matematikai ábrázolások melyek elsőrendű logikai kifejezéseken és gráfszerű struktúrákon alapulnak. Ilyen például a *Control Flow Automata*.
- **Háttéranalízis** – Itt történik a formalizmus feldolgozása és ellenőrzése. Ide sorolható a programom belseje.
- **Sat-megoldó interfész** – Ennek segítségével történik a verifikáció. A Theta a Z3 Sat-megoldót használja jelenleg.

¹<https://github.com/FTSRG/theta>

²2020 decemberének elején.

- **Eszközök** – Parancssori alkalmazások melyek futtatható jar fájlba fordíthatóak le. Jellemzően csak beolvassák az inputot és meghívják az alsóbb szinten lévő algoritmusokat. Az én programom is rendelkezik *command line interface (CLI)*-vel azaz parancssori interfésszel.

4.2. A program implementálása

4.2.1. Bemenet

A program bemenete egy CFA, mely számos hellyel rendelkezik, melyek közül kiemelkedik a kezdő-, a hiba- illetve a végső hely. Az utóbbit a k-indukciós algoritmus nem veszi figyelembe, mert az a teljes teret bejárja, viszont más algoritmusok működéséhez szükségesek lehetnek. Az ezután következő (5). fejezetben részletesen kitérek a programom tesztelésére, de előjáróban azt érdemes tudni, hogy bizonyos CFA modellekre az algoritmus lassan fut le. Ezért, ha a felhasználó igényeinek megfelelő, bemenetnek megadhat egy maximális időkorlátot vagy egy maximális mélységet is a bejárando állapotternek, esetleg mindkettőt, melyek felső korlátot fognak szabni a programnak.

4.2.2. Architektúra

Ebben az alfejezetben részletesen kitérek a programom felépítésére. A program állapotdiagramja a (4.1) ábrán látható.

Osztályok. A következő osztályokat definiáltam:

- **KInduction**
- **KInductionResult**
- **PathOperator**
- **PathVertex**
- **CfaTest**
- **KInductionCLI**

A **KInduction** a főosztályom, ő teszi kontextusba és adja meg az ellenőrzés ívét. Benne található a `check(...)` függvény, melyet a tesztelő **CfaTest** osztály hív és amely végzi az ellenőrzést.

A **KInductionResult** osztályú objektummal tér vissza a `check(...)` függvény és így a programom. Magába foglal minden olyan információt, mely az eredményhez kötődik: az ellenőrzés tényleges eredményét, ha nem volt biztonságos a modell akkor egy ellenpéldát, illetve hány másodpercig futott és hogy milyen mélységig jutott a program.

A **PathOperator** egy osztály mely megvalósítja a Szoftvertechnikából³ ismert Singleton tervezési mintát, illetve az Initialization-on-demand holder [10] tervezési mintát is. Minden, az útvonal átalakításához, feldolgozásához szükséges műveletet ebbe az osztályba szervezem ki függvények formájában.

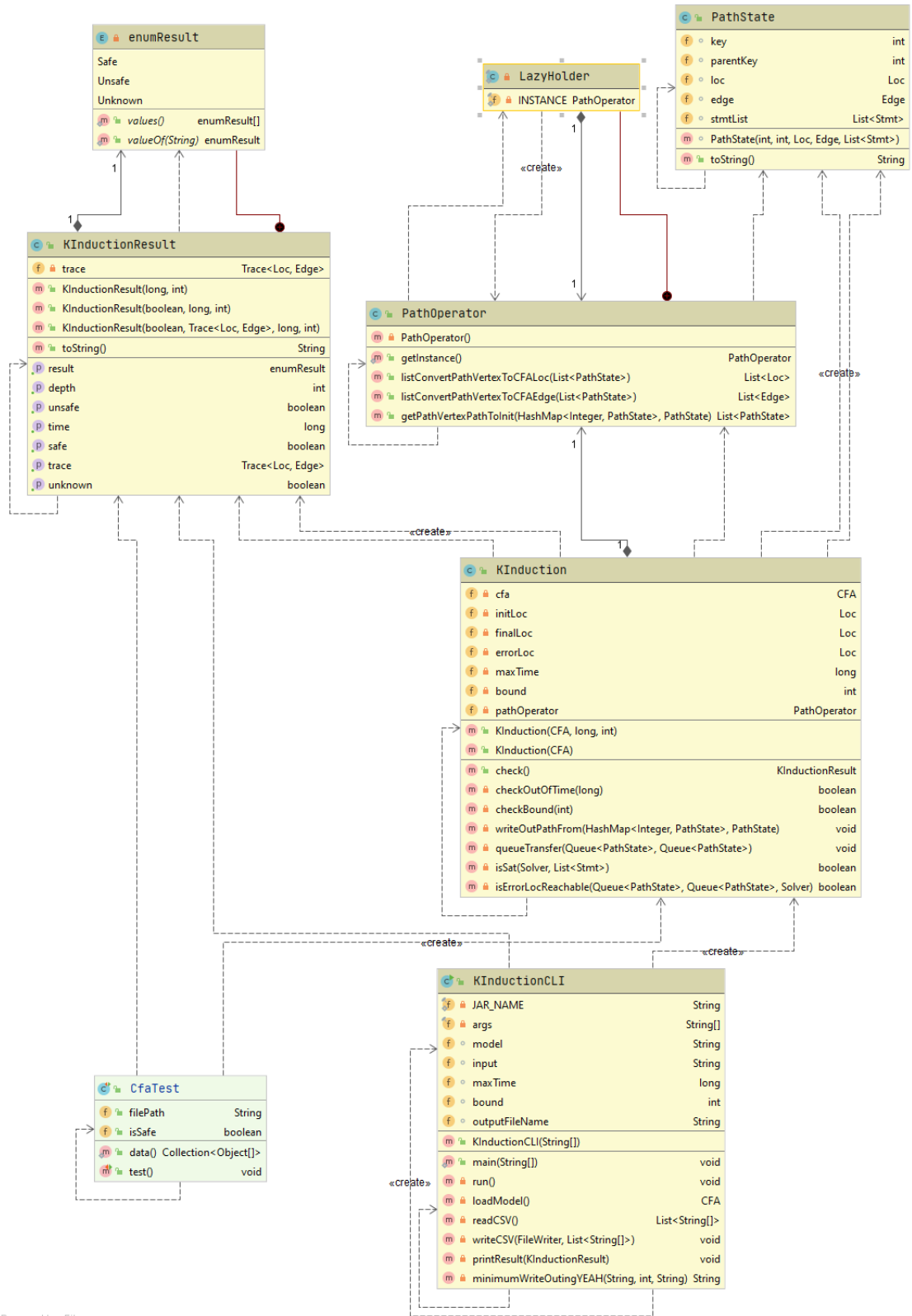
³<https://www.aut.bme.hu/Course/VIAUAB00>

A **PathState** osztály az útvonal bejáráshoz kell, az útvonalaim ezekből az állapotokból épülnek fel. A PathState a következő ötöt tárolja public változóiban:

- `key` – `int` típusú egyedi kulcs (azonosító), hogy minden PathState egyedi legyen.
- `parentKey` – `int` típusú egyedi azonosító ahhoz a szülő PathState állapothoz, mely az útvonalban eggyel megelőzi őt (tehát a szülő PathState az az állapot, melyből a bejárás során eljutottunk ehhez a PathState állapothoz).
- `loc` – `CFA.Loc` típusú változó, melyben a PathState egy helyet tárol.
- `edge` – `CFA.Edge` típusú változó, melyben a PathState azt az élet tartalmazza amelyen keresztül a szülő PathState állapotból eljutottunk ebbe a PathState állapotba.
- `stmtList` – `List<Stmt>` típusú változó, melyben a PathState azon stmt állítások listáját tartalmazza bejárési sorrendben, melyek azokon az éleken voltak amiken a bejárás során az útvonal végigment, a kezdőhelytől (vagy visszafele keresésnél a hibahelytől) egészen eddig a PathState állapotig.

A **CfaTest** osztály a tesztelésért felelős a JUnit egységteszt-keretrendszer segítségével.

A **KInductionCLI** osztály a parancssorból történő tesztelésért (Command Line Interface) illetve verifikációért felel. Lehet csak egy CFA-t tesztelni, de arra is van lehetőség, hogy egy .csv fájlba kigyűjtött tesztek verifikálják. Az utóbbinál a program a futás után létrehoz egy .csv fájlt, amibe beleírja a futtatott program adatait illetve a futás eredményét is.



Powered by yFiles

4.1. ábra. A programom UML állapotdiagramja.*

* Automatikusan generálva az IntelliJ IDEA fejlesztőkörnyezettel

Függvények. Miután van egy széles, de nem túl mély rálátásunk a programra, most elmerülnék benne és bemutatnám részletesebben, függvényekre bontva az osztályokat.

- **KInduction** osztály

- **KInduction(CFA cfa)** – Az osztály egy paraméterű konstruktora. Akkor használjuk, mikor nem akarunk a programnak korlátozó követelményeket beadni.
 - * **Bemenet:** CFA modell.
 - * **Kimenet:** -
- **KInduction(CFA cfa, long maxTime, int bound)** – Az osztály három paraméterű konstruktora, a CFA mellett a két korlátozó követelmény megadására van lehetőség. Alapértelmezetten mindegyik -1, tehát nincsen.
 - * **Bemenet:** CFA modell, maximális megengedett idő és maximális bejárható mélység.
 - * **Kimenet:** -
- **check()** – Az osztály modellellenőrzést végző függvénye.
 - * **Bemenet:** -
 - * **Kimenet:** A verifikáció eredménye, az ellenpélda, az eltelt idő és az elért mélység összecsomagolva egy KInductionResult típusú objektumba.
- **checkOutOfTime(long timeInSeconds)** – Ellenőrzi, hogy ha van megadott időkorlát akkor azt nem-e léptük már át.
 - * **Bemenet:** A program indulása óta eltelt idő másodpercben.
 - * **Kimenet:** Boolean – igen vagy nem.
- **checkBound(int depth)** – Ellenőrzi, hogy ha van megadott mélységkorlát akkor azt nem-e léptük már át.
 - * **Bemenet:** Az aktuális bejárásra váró mélység.
 - * **Kimenet:** Boolean – igen vagy nem.
- **queueTransfer(Queue<PathState> copyFrom, Queue<PathState> pasteTo)** – A korlátos szélességi kereséshez két vermet használok, az egyik a még bejárando helyeket tartalmazza, a másik az ebben a mélységben már bejárt helyeket tárolja. A mélység bejárása után az utóbbit (copyFrom) beleteszem a másikba (pasteTo), így haladva egyre beljebb a térben.
 - * **Bemenet:** Két, PathState állapotokat tartalmazó verem.
 - * **Kimenet:** A pasteTo verem tartalmazni fogja a copyFrom verem tartalmát (mást nem, mert előtte kiürítjük), a copyFrom verem pedig üres lesz.
- **isSat(Solver solver, List<Stmt> stmtList)** – Megnézi a solver segítségével, hogy az utasításlistát tartalmazó stmtList kielégíthető-e.
 - * **Bemenet:** Egy Z3 megoldó és egy utasításlista.
 - * **Kimenet:** Boolean: ha kielégíthető, akkor igaz, különben hamis.
- **isErrorLocReachable(Queue<PathState> queueBW, Queue<PathState> queue2BW, Solver solver)** – A hibahelyről hátrafelé indulva járja be a teret két, az előbb említettthez hasonló sorral. Minden meghíváskor egy mélységet halad, a solver megoldót az elérhető helyek ellenőrzéséhez használja.
 - * **Bemenet:** Két, PathState állapotokat tartalmazó sor és egy Z3 megoldó.
 - * **Kimenet:** Boolean: ha belátja, hogy a hibahely nem elérhető, akkor igaz, különben hamis.

- **KInductionResult** osztály

- **KInductionResult(long time, int depth)** – Az osztály két paraméteres konstruktora. Helyes használat esetén a KInduction osztály akkor inicializál ezzel egy objektumot, mikor a futás eredménye *ismeretlen* volt.
 - * **Bemenet:** A program futási ideje illetve a mélység ameddig jutott.
 - * **Kimenet:** -
- **KInductionResult(boolean isSafe, long time, int depth)** – Az osztály három paraméteres konstruktora. Helyes használat esetén a KInduction osztály akkor inicializál ezzel egy objektumot, mikor a futás eredménye *helyes* volt.
 - * **Bemenet:** A program futási eredménye (hiba, ha nem true az érték, ezt assert-tel ellenőrzi), a program futási ideje illetve a mélység ameddig jutott.
 - * **Kimenet:** -
- **KInductionResult(boolean isUnsafe, Trace<CFA.Loc, CFA.Edge> trace, long time, int depth)** – Az osztály négy paraméteres konstruktora. Helyes használat esetén a KInduction osztály akkor inicializál ezzel egy objektumot, mikor a futás eredménye *nem helyes* volt. A Trace egy Theta beépített osztály, amelyben én az ellenpéldát tárolom.
 - * **Bemenet:** A program futási eredménye (hiba, ha nem true az érték, ezt assert-tel ellenőrzi), egy Trace típusú ellenpélda, a program futási ideje illetve a mélység ameddig jutott.
 - * **Kimenet:** -

Az osztály minden változója privát, ezért mindhez létezik get függvény. Ezeket külön nem sorolom fel.

- **PathOperator** Singleton osztály

- **getInstance()**
 - * **Bemenet:** -
 - * **Kimenet:** Az egyetlen static final PathOperator példány.
- **listConvertPathVertexToCFALoc(List<PathState> path)**
 - * **Bemenet:** PathState állapotokat tartalmazó lista.
 - * **Kimenet:** A PathState állapot helyei listában, ugyanabban a sorrendben.
- **listConvertPathVertexToCFAEdge(List<PathState> path)**
 - * **Bemenet:** PathState állapotokat tartalmazó lista.
 - * **Kimenet:** A PathState állapot élei listában, ugyanabban a sorrendben. A path lista utolsó PathState állapotának élét nem adja hozzá a listához, mert az a kezdőállapot éle lenne, ami pedig nincs (null).
- **getPathVertexPathToInit(HashMap<Integer, PathState> pathMap, PathState item)**
 - * **Bemenet:** PathState állapotokat és az egyedi kulcsukat tartalmazó HashMap és egy PathState állapot.
 - * **Kimenet:** Egy PathState lista (útvonal) az item PathState állapotból indulva, mely a HashMap kiinduló eleméig tart (ami vagy a kezdőhely vagy a hibahely).

- **PathState** osztály

- `PathState(int key, int parentKey, CFA.Loc loc, CFA.Edge edge, List<Stmt> stmtList)` – az osztály öt elemű konstruktora.
 - * **Bemenet:** A `PathState` egyedi kulcsa, melynek segítségével a `HashMap`-ben lehetséges keresni, a megelőző állapot kulcsa, az állapothoz rendelt hely, az él amin keresztül a helyhez értünk és egy `Stmt` lista, mely tárolja a kiindulási helytől a `PathState` helyéig vezető út állításait.
 - * **Kimenet:** -
- **KInductionCLI** osztály
 - `main(final String[] args)` – Létrehoz egy `KInductionCLI` objektumot aminek átadja a parancssori argumentumokat és aminek meghívja utána a `run()` függvényét.
 - * **Bemenet:** - Parancssori argumentumok.
 - * **Kimenet:** -
 - `KInductionCLI(final String[] args)` – A kapott parancssori argumentumokat eltárolja.
 - * **Bemenet:** - Parancssori argumentumok.
 - * **Kimenet:** -
 - `run()` – A parancssorból beolvasott paramétereket kezeli (meghívja a `KInduction` osztály `check(...)` függvényét), illetve az interfész kinézetéért felel még.
 - * **Bemenet:** -
 - * **Kimenet:** -
- **CfaTest** osztály
 - `test()` – `@Test` annotációval ellátott függvény, mely a JUnit tesztelésért felelős.
 - * **Bemenet:** -
 - * **Kimenet:** -

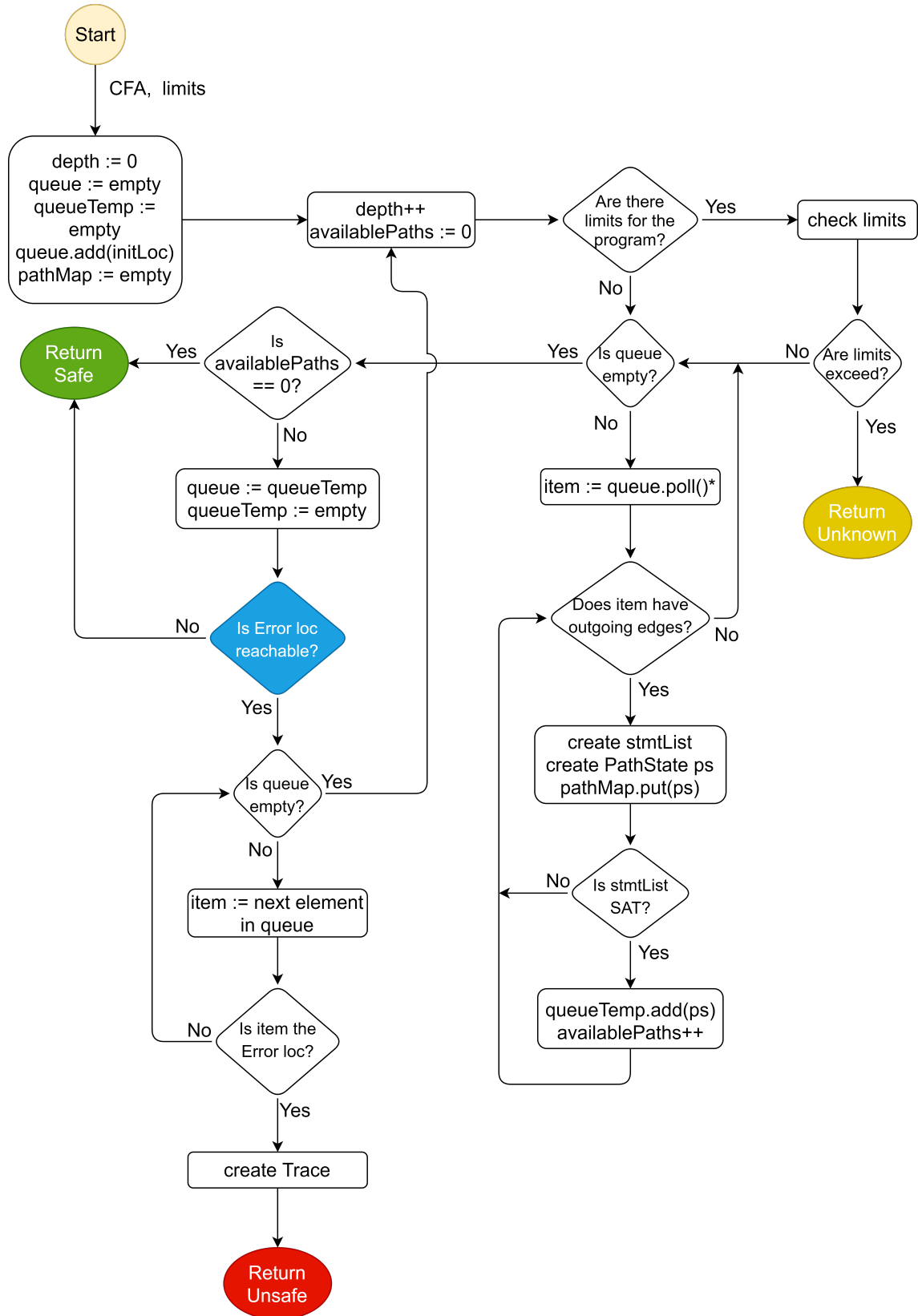
4.2.3. Működés

Ebben az alfejezetben részletesen kifejtem a programom működésének folyamatát, melyet vázlatosan összefoglal a (4.2) diagram.

1. lépés: A program bemenetként kap egy CFA modellt és opcionálisan korlátozó feltételeket.
2. lépés: A következő változókat inicializálja:
 - (a) `depth` – az aktuális, bejárásra váró mélységet tárolja, kezdetben nulla,
 - (b) `queue` – tárolja az aktuális bejárando mélység állapotait, kezdetben csak a kezdőállapotot tárolja,
 - (c) `queueTemp` – az aktuális mélység bejárása közben talált elérhető állapotokat tárolja,
 - (d) `availablePaths` – a bejárás utáni `queueTemp` méretét tárolja, a programkód jobb olvashatóságának érdekében van külön is vezetve,
 - (e) `queueBW` – ugyanaz, mint a `queue`, csak a hátrafelé haladó kereséshez (*Back-Ward*), kezdetben csak a hibaállapotot tárolja,

- (f) `queueTempBW` – ugyanaz, mint a `queueTemp`, csak a hátrafelé haladó kereséshez (*BackWard*),
 - (g) `pathMap` – egy `HashMap` mely `PathState` állapotokat tárol s azok egyedi kulcsait használja kulcsnak, kezdetben csak a kezdőállapotot tárolja, illetve
 - (h) a program futási idejének a mérését segítő változókat.
3. lépés: Elindít egy végtelen ciklust.
 4. lépés: Növeli a `depth` változó értékét és beállítja az `availablePaths` értékét nullára.
 5. lépés: Ellenőrzi, hogy vannak -e korlátozások, és ha igen, teljesülnek-e:
 - (a) Ha teljesülnek, létrehoz egy `KInductionResult` objektumot és befejezi a program a futását *ismeretlen* válasszal,
 - (b) Ha nem teljesül egyik sem (vagy nincsenek), akkor megyünk tovább a következő lépésre.
 6. lépés: Végig iterál a `queue` vermen, és kiveszi belőle az éppen utolsó elemet (`item : PathState`).
 - (a) Végig iterál az `item` állapot `loc` változójának a kimenő élein (`edge : CFA.Edge`)
 - i. Eltárolja a `loc : CFA.Loc` helyet, ahova jutott az élen keresztül
 - ii. Az `item stmtList` listájának a végére beszúrja az `edge` `stmt` utasításait.
 - iii. Létrehoz egy új állapotot `nextPS : PathState` néven, melynek megad egy egyedi kulcsot, a szülő kulcsa az `item` egyedi kulcsa lesz, a helye a `loc` és az éle pedig az `edge`.
 - iv. `pathMap` `HashMap`-hez hozzáadja a `nextPS` állapotot.
 - v. Ellenőrzi solver segítségével, hogy az `stmtList` kielégíthető-e.
 - A. Ha igen, `queueTemp` veremhez hozzáadja a `nextPS` változót, és megnöveli eggyel az `availablePaths` értékét.
 7. lépés: Beleteszi az üres `queue` verembe a `queueTemp` verem tartalmát. Utóbbi után kiüríti.
 8. Ellenőrzi, hogy az `availablePaths` értéke nulla -e:
 - (a) Ha nulla, létrehoz egy `KInductionResult` objektumot és befejezi a program a futását *helyes* válasszal.
 9. Ellenőrzi, hogy a hibahely elérhető-e hátra felől:
 - (a) Ugyanaz, mint a 6. lépés, csak `queue` helyett `queueBW` veremmel, `queueTemp` helyett `queueTempBW` veremmel, illetve egy lokális `availablePaths` változóval, leszámítva a (a) \rightarrow iv lépést: `pathMap` `HashMap`-hez itt nem adjuk hozzá a `nextPV` állapotot, mert az csak az ellenpélda meghatározásához kell, amit meg úgy értelmeztünk, hogy csak a kezdőhelyből indulhat ki.
 - (b) Ha `availablePaths` nem nulla, azaz van elérhető hely, akkor a modelltől nem tudtunk meg új információt, csak továbbra is azt látjuk, hogy talán elérhető a hibahely. Ha viszont `availablePaths` nulla, azaz a hibahelytől bejárva a gráfot arra jutunk egy bizonyos szint után, hogy nincs több elérhető hely, azzal akkor beláttuk, hogy a hibahely nem érhető el.

- (c) Ha a hibahely nem érhető el, akkor létrehoz egy KInductionResult objektumot és befejezi a program a futását *helyes* válasszal.
10. Ellenőrzi, hogy azok a helyek között, melyeket a most bejárt mélység után kaptunk (tehát amiket queue tárol), ott van-e a hibahely:
- (a) Végig iterál a queue vermen, az aktuális elem az item : PathState.
 - i. Ha az item loc helye a hibahely és elérhető (az item stmtList listáját a solver ki tudja elégíteni), akkor létrehoz egy KInductionResult objektumot és befejezi a program a futását *nem helyes* válasszal.
11. Ha ideáig eljutott a program, akkor a 4) lépésre ugrik, ezzel újra kezdve még egy szint bejárását.



4.2. ábra. A programom folyamatábra diagramja. A kék színnel jelölt elágazás magába foglal még egy ekkora folyamatábrát, csak az hátulról ellenőriz a hibahelytől. A *-al jelölt poll() függvény kiválasztja a verem következő elemét és abból törli is.

4.2.4. Kimenet

A program kimenetele egy `KInductionResult` osztályú objektum, melynek a következő változói vannak:

- `enumResult` mely egy `enum` típusú változó és tárolja a program futásának a kimenetelét, ami az egyik a következőkből:
 - `Safe` (*Biztonságos*)
 - `Unsafe` (*Nem biztonságos*)
 - `Unknown` (*Ismeretlen*)
- `trace` mely egy `Trace` típusú változó és ami tárolja az ellenpéldát, ha van, különben `null`.
- `time` mely `long` típusú és tárolja, hogy a program mennyi másodpercig futott.
- `depth` mely `int` típusú és azt mondja meg, hogy milyen mélységben fejeződött be a program futása.

A *biztonságosról* és a *nem biztonságosról* az előző fejezetekben sok szó esett. Az *ismeretlen* válasszal a programom akkor tér vissza, ha kifutott az időből illetve ha elérte a maximális mélységet (a két feltétel között diszjunkció van), és addigra nem sikerült belátnia sem a modell helyességét, sem annak ellentettjét.

5. fejezet

Kiértékelés

Ebben a fejezetben a programom tesztelését mutatom be. Az első alfejezetben (5.1) a tesztelés részleteiről írok, a második alfejezetben (5.2) az elért eredményeket összesítem és értékelem ki.

5.1. Tesztelés

A programom készítése közben folyamatosan teszteltem azt JUnit tesztek segítségével az Architektúra (4.2.2) alfejezetben bemutatott CfaTest osztállyal. A tesztekhez a CFA modelleket egyrészt az ftsrg kutatócsoport ca github repository-jából nyertem [13], illetve készítettem sajátokat is, de a tesztek túlnyomó többsége a Thetához kapcsolódó, privát GitHub repository-ból való, amelyek különböző frontendekkel lettek generálva. Az utóbbiban többek között 479 darab CFA teszt található, melyekhez tartozik előre ismert eredmény is. A tesztek egyik fele az SV-Comp-ról¹ származik ahol eredetileg C kódok voltak amik aztán át lettek CFA-ba alakítva [11]. Ezek különböző csoportba sorolhatóak [8]:

- Locks kicsi (94-234 LoC²) kizárási feladatokat ír le.
- ECA (event-condition-action) feladathalmaz nagy (591-1669 LoC) eseményvezérelt rendszereket tartalmaz.
- SSH nagy (557-716 LoC) kliens-szerver rendszereket ír le.
- Simple kicsi (14-40 LoC) feladatok gyors teszteléshez.

Továbbá volt alkalmam tesztelni olyan CFA modelleket, melyek eredetileg ipari PLC szoftverek voltak a CERN-nél. [3] Ezek mérete roppant változatos – a pár tucattól a több ezerig is terjedhet. Mindegyik tesztről tudjuk, hogy abban mennyi változó van (azok között mennyi int és mennyi boolean típusú), mennyi hely, mennyi él, az egyes taszkok ciklikus komplexitása, az éleken mennyi hozzárendelés, mennyi őrfeltétel illetve hogy mennyi havoc van. Ezeket vázlatosan bemutatja a (5.1) táblázat [8]:

Míg a program fejlesztése közben azt 28 darab, véletlenszerűen kiválasztott teszttel ellenőriztem, a végén teszteltem mind a 479 tesztre is. A széleskörű teszteléshez KInductionCLI osztályt használtam, mely egy interfészt biztosít a programom parancssori futtatásához, és amelyet a következő paraméterekkel lehet meghívni:

¹<https://sv-comp.sosy-lab.org/2018/>

²Source lines of code - Hány sorból áll a program melyet a CFA modellez.

| Category | Tasks | Vars | Locs | Edges | CC |
|--------------|------------|-------|----------|----------|--------|
| Simple | 10 | 1–2 | 4–12 | 3–13 | 3–9 |
| Locks | 143 | 4–32 | 9–40 | 10–57 | 3–23 |
| SSH | 17 | 64–81 | 187–267 | 262–375 | 87–121 |
| PLC | 129 | 1–596 | 8–4614 | 7–4782 | 4–188 |
| ECA | 180 | 9–30 | 302–1301 | 375–1516 | 73–231 |
| <i>Total</i> | <i>479</i> | | | | |

5.1. táblázat. Az egyes taszkok tulajdonságainak statisztikai jellemzői. A CC rövidítés a Cyclomatic Complexity azaz a ciklikus komplexitást jelöli.

- `--model` – A CFA teszt elérési útvonala, ha csak egy darab tesztre szeretnénk lefuttatni. Nem kötelező.
- `--input` – A CFA tesztek, az elvárt eredményeiket és egyéb, a tesztek tulajdonságait leíró információkat tartalmazó .csv fájl elérési útvonala. Nem kötelező.
- `--time` – A futási időt tudjuk vele korlátozni, másodpercben. Nem kötelező, alapértelmezetten -1.
- `--bound` – Az algoritmus bejárési mélységét tudjuk vele korlátozni. Nem kötelező, alapértelmezetten -1.
- `--output` – A kimenetel fájl neve (input paraméter esetén). Nem kötelező, alapértelmezetten „output.csv”.

Habár se a `--model` sem az `--input` paraméter nem kötelező, a program assert-tel ellenőrzi, hogy legalább egy meg legyen adva. Viszont ugyanúgy hiba, ha kettő bemenet van adva. Ezt az alábbi kóddal ellenőrzöm:

```

1  if (!input.equals("") && !model.equals("")) {
2      Assert.fail("Only one input source is allowed.");
3  }
4  if (input.equals("") && model.equals("")) {
5      Assert.fail("One input source is required.");
6  }

```

5.2. Eredmények

Azon teszteknel, melyekre a programom készítése közben folyamatosan teszteltem azt, az ideális ellenpéldahossz is meg volt adva. A programom kivétel nélkül mindre (11 darab) az optimális megoldást produkálta.

A 479 tesztet az előző oldalon található felsorolás szerint külön .csv fájlalba szerveztem. Ezután a

```

1  java -jar theta-k-induction.jar --input input_xx.csv --output
    output_xx_yy.csv --time yy

```

parancssori utasítást adtam ki, ahol `xx = {locks, eca, ssh, simple, plc}`, illetve `yy =`

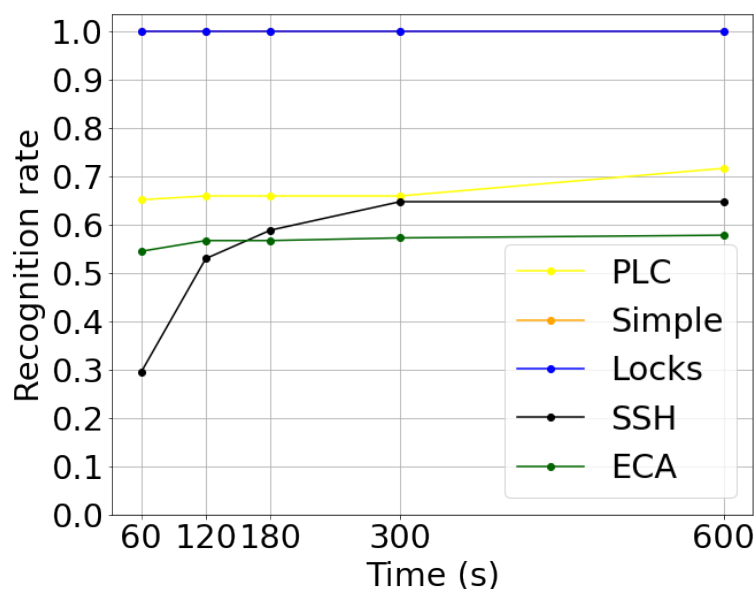
{60, 120, 180, 300, 600}, azaz szavakkal elmondva külön-külön mindegyik teszt kategóriát futtattam 1 perc, 2 perc, 3 perc, 5 perc illetve 10 perc időkorláttal. Azért döntöttem időkorlát használata mellett, mert tapasztalataim szerint egyes tesztek 8, 10 vagy annál több órás futási időt igényelnek, ami a nagy darabszámot is figyelembe véve korlátozásra ad okot.

A parancssori interfészről készítettem képeket, melyek megtekinthetők a Függelékben (F.1). Itt a harmadik ábra (F.1.3) érdekes: a program futása után statisztikát közlök arról, hogy hány teszt lett sikeres, mennyi nem sikerült azért, mert ismeretlen lett és mennyi nem sikerült azért, mert rosszat mondott a programom. Ez az utóbbi minden futtatásnál **0** volt.

A kapott eredményfájlokat a Pandas³ Python programkönyvtárral dolgoztam fel. A következő eredményekre jutottam:

- Azt tapasztaltam, hogy a tesztek 64.23%-a lett biztonságos, 8.88%-a nem biztonságos és 26.89%-a lett ismeretlen.
- Viszont arra lettem figyelmes, és ez az (5.1) ábrán jól is látszik, egyes teszt kategóriákban 100% felismerési arányt sikerült elérni.

A felismerési arány alatt azt értem, hogy az adott kategóriában a tesztek hány százaléka lett nem ismeretlen, tehát a tesztek hány százalékát láttuk be. Szeretném kihangsúlyozni, hogy minden egyes tesztre, ami nem ismeretlen lett, **helyes** választ adott a programom. Néhány nagyobb, bonyolultabb tesztnél engedtem végig futni (a statisztikában nem szerepelnek), és azokra is mind **helyes** eredményt kaptam, így arra jutottam, hogy a korlátozás nem befolyásolja a programom működését, egyszerűen csak időmenedzsmenti okai és következményei vannak.



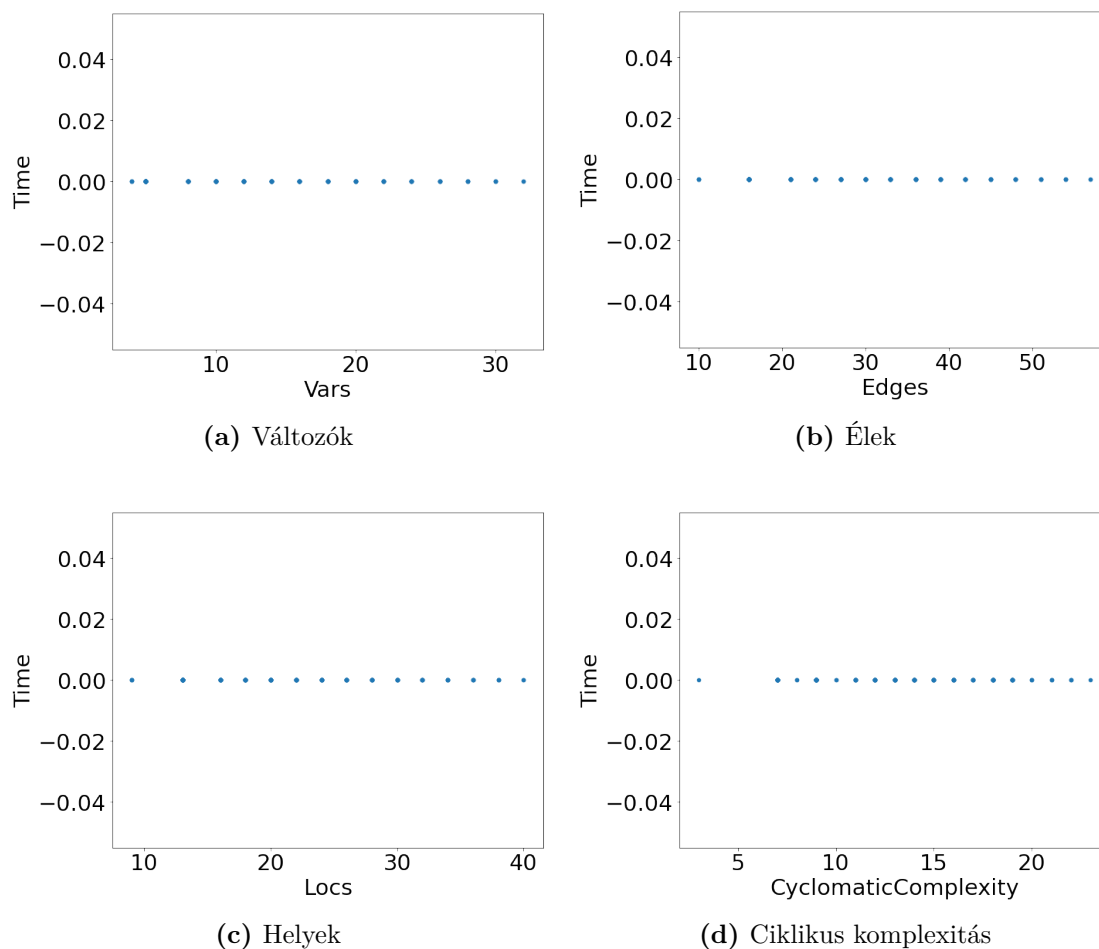
5.1. ábra. A tesztek felismerési arányainak a változása különböző időkorláttal futtatva. A Simple végig 100%-os felismerési arányt produkált, mint a Locks, csak az utóbbi kitakarja az ábrán.

A fejezet maradék részében a futtatási eredményeket fogom ellenőrizni és elemezni –

³<https://pandas.pydata.org/>

arra vagyok kíváncsi, hogy a tesztek egyes tulajdonságai hogyan hatnak ki a futási időre. Mindegyik képen az X tengely a tesztek egy adott tulajdonsága látható, az Y tengelyen pedig a futási idő az adott tulajdonságra vetítve másodpercben. Mindegyik kép a 10 percig futtatott program eredményeit mutatja, azért, hogy a lehető legtöbb információt tudjam megjeleníteni.

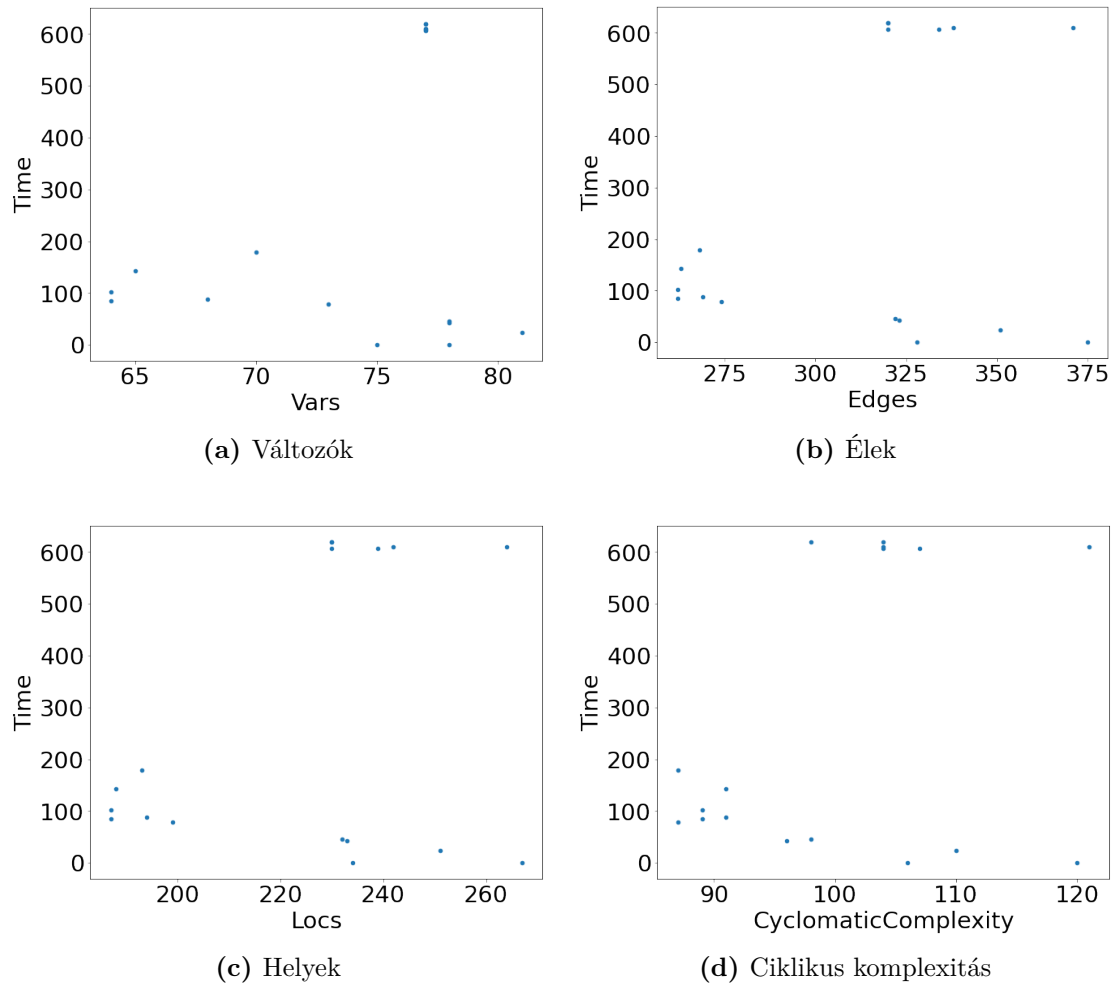
A 100%-os felismerési arány a Simple és a Locks kategóriák esetén nem meglepő, ha ránézünk a (5.2) ábrára. Látható, hogy másodpercek alatt képes volt a programom ellenőrizni a modelleket, így még csak közel sem került az időkorláthoz.



5.2. ábra. Locks teszt kategória. Az időtengely mindegyiken másodpercben van.

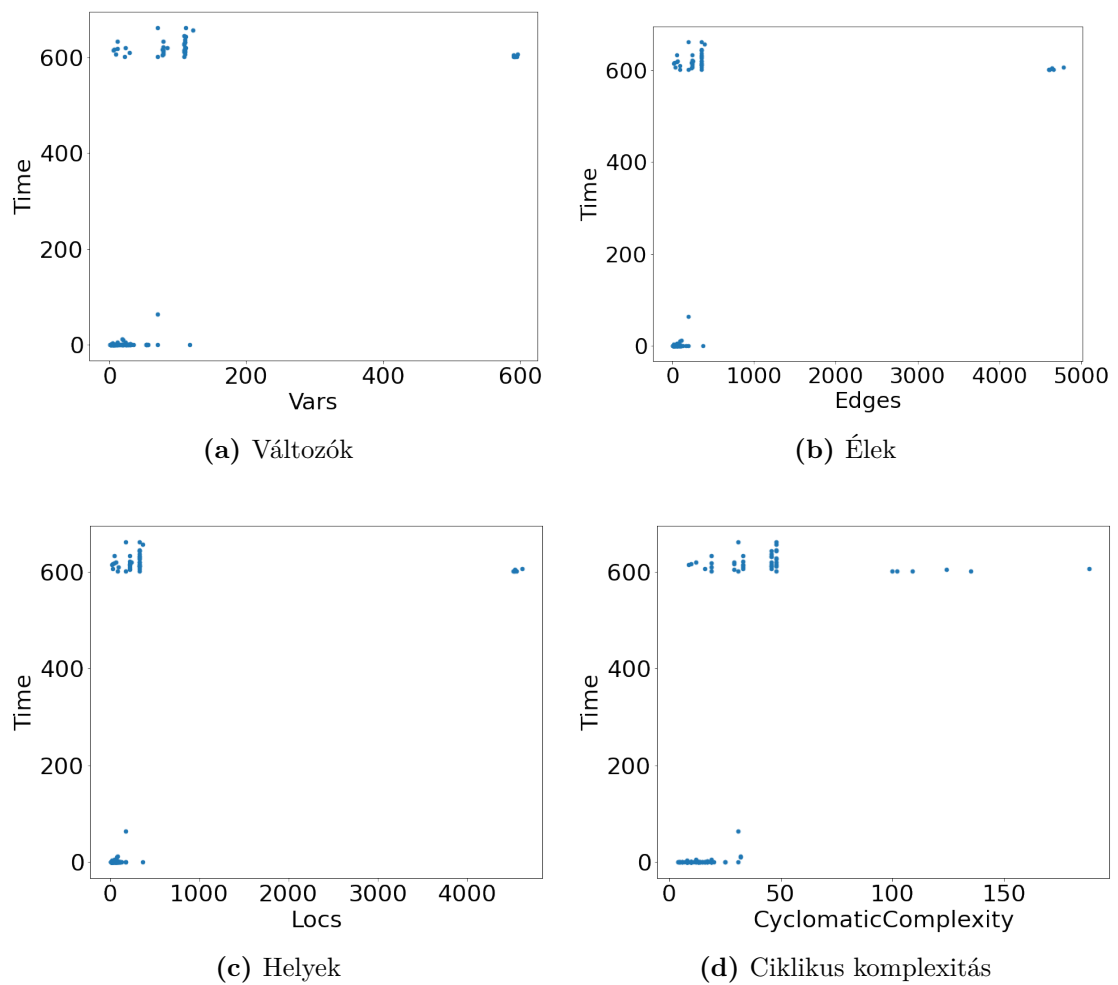
Az (5.1) ábrán látható, hogy az SSH teszt kategóriának az egyperces futási idő kevés, viszont ahogy emeljük a korlátot, úgy nő a sikeresen lefutott tesztek száma. Egészen öt percig, ahol is megtorpanni látszik: nem változik utána a felismerési arány. Öt percig az eredmény 3 biztonságos, 8 nem biztonságos és 6 ismeretlen, és hiába emeltük a duplájára az időkorlátot, a maradék 6 ismeretlenből egyiket se sikerült belátni.

A (5.3) ábrán a következőt láthatjuk: A változók számától önmagában egyedül biztosan nem függ a futási idő, tekintve, hogy volt olyan teszt eset ahol 81 változóval a program 23 másodpercig futott illetve 78 változóval 42 és 45 másodpercekig, míg 77 változóval rendre (hatszor) túllépte a 600 másodperces időkorlátot. Tehát kell lennie legalább még egy tulajdonságnak, amivel együtt határozzák meg a tesztek futási idejét.

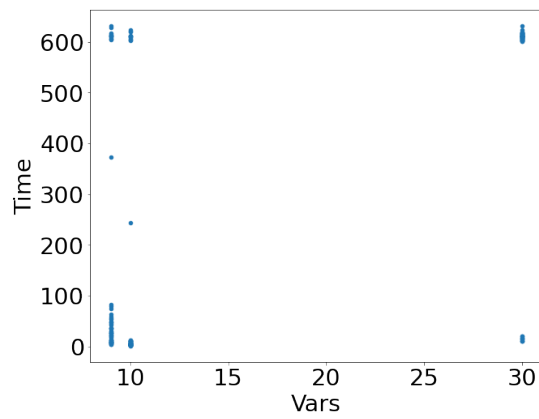


5.3. ábra. SSH tesztkategória. Az időtengely mindegyiken másodpercben van.

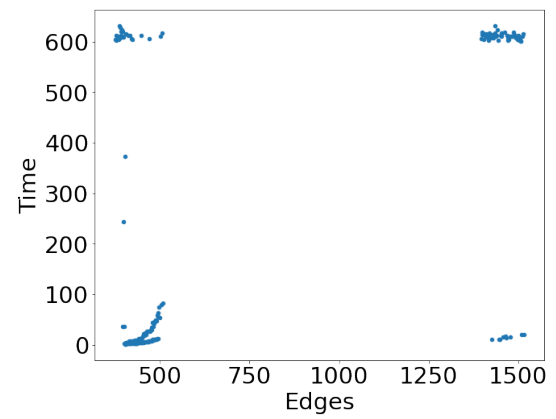
A (5.4) ábrán láthatjuk a PLC tesztkategória eredményeit. Az biztosan kijelenthető, hogy nagy (változó, él, hely illetve komplexitás) értékekre a futásidő nem fog beleférni a 10 perces korlátba. Ami ennél érdekesebb, hogy



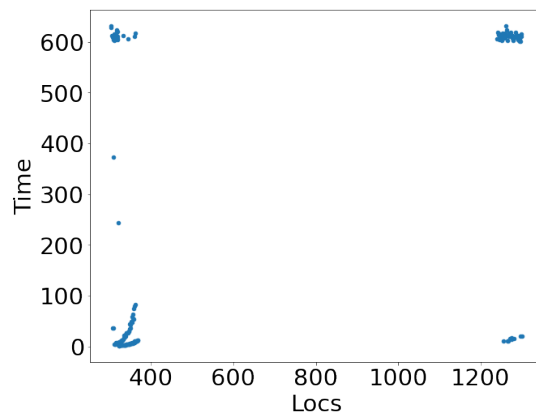
5.4. ábra. PLC tesztkategória. Az időtengely mindegyiken másodpercben van.



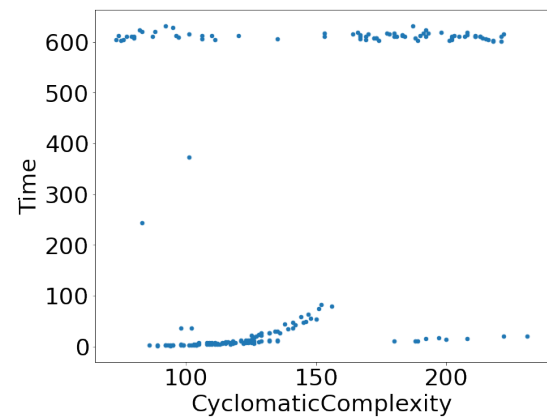
(a) Változók



(b) Élek



(c) Helyek



(d) Ciklikus komplexitás

5.5. ábra. ECA tesztkategória. Az időtengely mindegyiken másodpercben van.

6. fejezet

Összefoglaló

Köszönetnyilvánítás

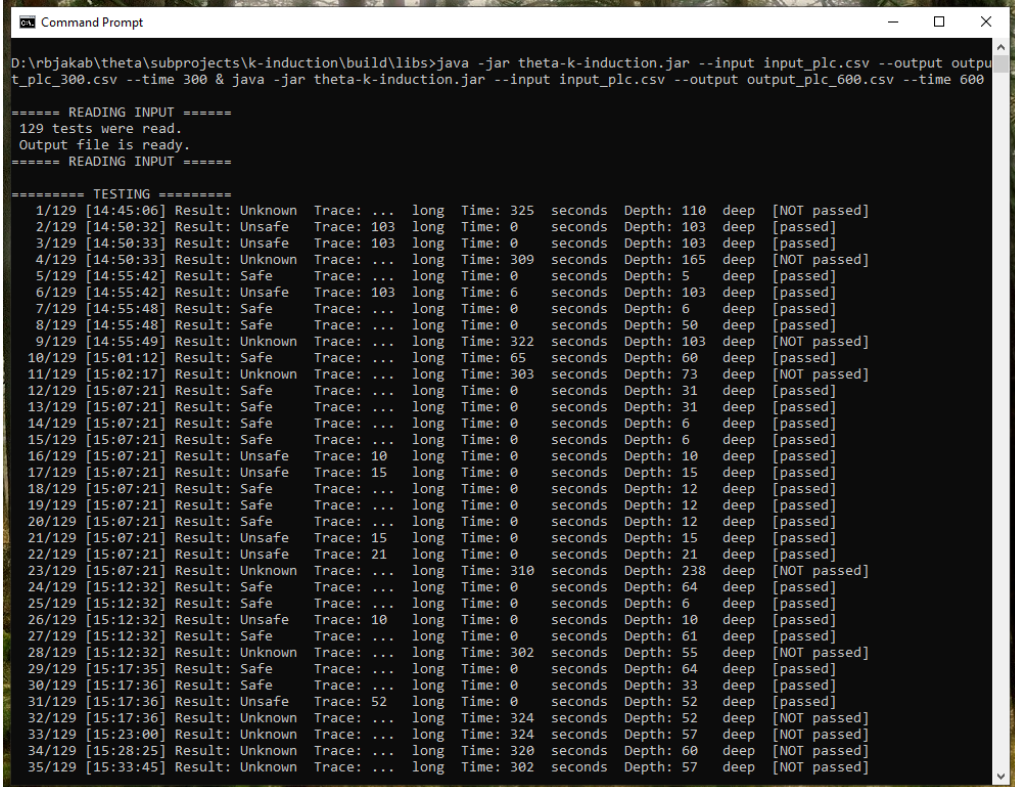
Irodalomjegyzék

- [1] Denke Ákos: Beágyazott szoftver verifikáció virtuális környezetben. Szakdolgozat (Budapest University of Technology and Economics). 2014. 12.
- [2] Dirk Beyer – Stefan Löwe: Explicit-state software model checking based on CEGAR and interpolation. In *Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science sorozat, 7793. köt. 2013, Springer, 146–162. p.
- [3] Dániel Darvas – Enrique Blanco Viñuela – Vince Molnár: PLCverif re-engineered: An open platform for the formal analysis of PLC programs. In *Proceedings of the 17th International Conference on Accelerator and Large Experimental Physics Control Systems* (konferenciaanyag). 2019, JACoW.
- [4] Alastair F. Donaldson – Leopold Haller – Daniel Kroening – Philipp Rümmer: Software verification using k-induction. In Eran Yahav (szerk.): *Static Analysis* (konferenciaanyag). Berlin, Heidelberg, 2011, Springer Berlin Heidelberg, 351–368. p. ISBN 978-3-642-23702-7.
- [5] Dr. Ákos Hajdu: Formal software verification. URL: <https://ftsrg.mit.bme.hu/software-verification-notes/software-verification.pdf>, 2020. 11.
- [6] Dr. Majzik István: Szoftver verifikáció és validáció. URL: https://inf.mit.bme.hu/sites/default/files/materials/category/kategória/oktatás/doktorandusz-targyak/szoftver-verifikáció-és-validáció/11/SZVV_EA01_intro.pdf, 2011. 9.
- [7] Dr. Majzik István: Rendszertervezés és -integráció. URL: https://www.mit.bme.hu/system/files/oktatás/targyak/10019/VIMIMA11_RTI_08_Biztonsagi_alapfogalmak_1.pdf, 2018. 12.
- [8] Ákos Hajdu: *Effective Domain-Specific Formal Verification Techniques*. PhD értekezés (Budapest University of Technology and Economics). 2020. URL <https://repozitorium.omikk.bme.hu/bitstream/handle/10890/13523/ertekezes.pdf>.
- [9] ISO Central Secretary: Quality management systems — fundamentals and vocabulary. ISO/TC 176 9000:2015. Standard, 2015, International Organization for Standardization. URL <https://www.iso.org/obp/ui/#iso:std:iso:9000:ed-4:v1:en>.
- [10] Jeremy Manson – Brian Goetz: Jsr 133 (java memory model) faq, 2004. 2. URL <http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>.
- [11] Gyula Sallai – Ákos Hajdu – Tamás Tóth – Zoltán Micskei: Towards evaluating size reduction techniques for software model checking. In *Proceedings of the Fifth International Workshop on Verification and Program Transformation*. Electronic Proceedings in Theoretical Computer Science sorozat, 253. köt. 2017, Open Publishing Association, 75–91. p.

- [12] Mary Sheeran–Satnam Singh–Gunnar Stålmarck: Checking safety properties using induction and a sat-solver. In Warren A. Hunt–Steven D. Johnson (szerk.): *Formal Methods in Computer-Aided Design* (konferenciaanyag). Berlin, Heidelberg, 2000, Springer Berlin Heidelberg, 127–144. p. ISBN 978-3-540-40922-9.
- [13] Tamás Tóth. URL <https://github.com/ftsrg-ca/ca/tree/master/program-verification-1/src/test/resources>.
- [14] Thomas Wahl: The k-induction principle. URL <http://www.comlab.ox.ac.uk/people/Thomas.Wahl/Publications/k-induction.pdf>.

Függelék

F.1. A parancssori interfész felülete



```
D:\rbjakab\theta\subprojects\k-induction\build\libs>java -jar theta-k-induction.jar --input input_plc.csv --output output_plc_300.csv --time 300 & java -jar theta-k-induction.jar --input input_plc.csv --output output_plc_600.csv --time 600

***** READING INPUT *****
129 tests were read.
Output file is ready.
***** READING INPUT *****

***** TESTING *****
1/129 [14:45:06] Result: Unknown Trace: ... long Time: 325 seconds Depth: 110 deep [NOT passed]
2/129 [14:50:32] Result: Unsafe Trace: 103 long Time: 0 seconds Depth: 103 deep [passed]
3/129 [14:50:33] Result: Unsafe Trace: 103 long Time: 0 seconds Depth: 103 deep [passed]
4/129 [14:50:33] Result: Unknown Trace: ... long Time: 309 seconds Depth: 165 deep [NOT passed]
5/129 [14:55:42] Result: Safe Trace: ... long Time: 0 seconds Depth: 5 deep [passed]
6/129 [14:55:42] Result: Unsafe Trace: 103 long Time: 6 seconds Depth: 103 deep [passed]
7/129 [14:55:48] Result: Safe Trace: ... long Time: 0 seconds Depth: 6 deep [passed]
8/129 [14:55:48] Result: Safe Trace: ... long Time: 0 seconds Depth: 50 deep [passed]
9/129 [14:55:49] Result: Unknown Trace: ... long Time: 322 seconds Depth: 103 deep [NOT passed]
10/129 [15:01:12] Result: Safe Trace: ... long Time: 65 seconds Depth: 60 deep [passed]
11/129 [15:02:17] Result: Unknown Trace: ... long Time: 303 seconds Depth: 73 deep [NOT passed]
12/129 [15:07:21] Result: Safe Trace: ... long Time: 0 seconds Depth: 31 deep [passed]
13/129 [15:07:21] Result: Safe Trace: ... long Time: 0 seconds Depth: 31 deep [passed]
14/129 [15:07:21] Result: Safe Trace: ... long Time: 0 seconds Depth: 6 deep [passed]
15/129 [15:07:21] Result: Safe Trace: ... long Time: 0 seconds Depth: 6 deep [passed]
16/129 [15:07:21] Result: Unsafe Trace: 10 long Time: 0 seconds Depth: 10 deep [passed]
17/129 [15:07:21] Result: Unsafe Trace: 15 long Time: 0 seconds Depth: 15 deep [passed]
18/129 [15:07:21] Result: Safe Trace: ... long Time: 0 seconds Depth: 12 deep [passed]
19/129 [15:07:21] Result: Safe Trace: ... long Time: 0 seconds Depth: 12 deep [passed]
20/129 [15:07:21] Result: Safe Trace: ... long Time: 0 seconds Depth: 12 deep [passed]
21/129 [15:07:21] Result: Unsafe Trace: 15 long Time: 0 seconds Depth: 15 deep [passed]
22/129 [15:07:21] Result: Unsafe Trace: 21 long Time: 0 seconds Depth: 21 deep [passed]
23/129 [15:07:21] Result: Unknown Trace: ... long Time: 310 seconds Depth: 238 deep [NOT passed]
24/129 [15:12:32] Result: Safe Trace: ... long Time: 0 seconds Depth: 64 deep [passed]
25/129 [15:12:32] Result: Safe Trace: ... long Time: 0 seconds Depth: 6 deep [passed]
26/129 [15:12:32] Result: Unsafe Trace: 10 long Time: 0 seconds Depth: 10 deep [passed]
27/129 [15:12:32] Result: Safe Trace: ... long Time: 0 seconds Depth: 61 deep [passed]
28/129 [15:12:32] Result: Unknown Trace: ... long Time: 302 seconds Depth: 55 deep [NOT passed]
29/129 [15:17:35] Result: Safe Trace: ... long Time: 0 seconds Depth: 64 deep [passed]
30/129 [15:17:36] Result: Safe Trace: ... long Time: 0 seconds Depth: 33 deep [passed]
31/129 [15:17:36] Result: Unsafe Trace: 52 long Time: 0 seconds Depth: 52 deep [passed]
32/129 [15:17:36] Result: Unknown Trace: ... long Time: 324 seconds Depth: 52 deep [NOT passed]
33/129 [15:23:00] Result: Unknown Trace: ... long Time: 324 seconds Depth: 57 deep [NOT passed]
34/129 [15:28:25] Result: Unknown Trace: ... long Time: 320 seconds Depth: 60 deep [NOT passed]
35/129 [15:33:45] Result: Unknown Trace: ... long Time: 302 seconds Depth: 57 deep [NOT passed]
```

F.1.1. ábra. A parancssori interfész a futás elején.

```
Command Prompt
49/129 [22:15:32] Result: Unknown Trace: ... long Time: 602 seconds Depth: 65 deep [NOT passed]
50/129 [22:25:35] Result: Unknown Trace: ... long Time: 605 seconds Depth: 76 deep [NOT passed]
51/129 [22:35:40] Result: Unknown Trace: ... long Time: 620 seconds Depth: 59 deep [NOT passed]
52/129 [22:46:01] Result: Unknown Trace: ... long Time: 643 seconds Depth: 65 deep [NOT passed]
53/129 [22:56:44] Result: Unknown Trace: ... long Time: 620 seconds Depth: 74 deep [NOT passed]
54/129 [23:07:04] Result: Safe Trace: ... long Time: 0 seconds Depth: 51 deep [passed]
55/129 [23:07:05] Result: Unsafe Trace: 76 long Time: 2 seconds Depth: 76 deep [passed]
56/129 [23:07:07] Result: Safe Trace: ... long Time: 0 seconds Depth: 11 deep [passed]
57/129 [23:07:07] Result: Unknown Trace: ... long Time: 611 seconds Depth: 60 deep [NOT passed]
58/129 [23:17:18] Result: Unknown Trace: ... long Time: 626 seconds Depth: 59 deep [NOT passed]
59/129 [23:27:45] Result: Unknown Trace: ... long Time: 645 seconds Depth: 59 deep [NOT passed]
60/129 [23:38:31] Result: Unknown Trace: ... long Time: 628 seconds Depth: 60 deep [NOT passed]
61/129 [23:49:00] Result: Unknown Trace: ... long Time: 607 seconds Depth: 63 deep [NOT passed]
62/129 [23:59:07] Result: Unknown Trace: ... long Time: 619 seconds Depth: 59 deep [NOT passed]
63/129 [00:09:27] Result: Unknown Trace: ... long Time: 657 seconds Depth: 61 deep [NOT passed]
64/129 [00:20:24] Result: Safe Trace: ... long Time: 0 seconds Depth: 56 deep [passed]
65/129 [00:20:25] Result: Safe Trace: ... long Time: 0 seconds Depth: 56 deep [passed]
66/129 [00:20:25] Result: Safe Trace: ... long Time: 0 seconds Depth: 56 deep [passed]
67/129 [00:20:25] Result: Safe Trace: ... long Time: 0 seconds Depth: 51 deep [passed]
68/129 [00:20:26] Result: Unsafe Trace: 76 long Time: 1 seconds Depth: 76 deep [passed]
69/129 [00:20:28] Result: Safe Trace: ... long Time: 0 seconds Depth: 11 deep [passed]
70/129 [00:20:28] Result: Unsafe Trace: 11 long Time: 0 seconds Depth: 11 deep [passed]
71/129 [00:20:28] Result: Safe Trace: ... long Time: 0 seconds Depth: 8 deep [passed]
72/129 [00:20:28] Result: Unsafe Trace: 15 long Time: 0 seconds Depth: 15 deep [passed]
73/129 [00:20:28] Result: Unknown Trace: ... long Time: 616 seconds Depth: 170 deep [NOT passed]
74/129 [00:30:44] Result: Unsafe Trace: 29 long Time: 0 seconds Depth: 29 deep [passed]
75/129 [00:30:44] Result: Unsafe Trace: 41 long Time: 0 seconds Depth: 41 deep [passed]
76/129 [00:30:45] Result: Unknown Trace: ... long Time: 606 seconds Depth: 133 deep [NOT passed]
77/129 [00:40:51] Result: Unsafe Trace: 23 long Time: 0 seconds Depth: 23 deep [passed]
78/129 [00:40:51] Result: Safe Trace: ... long Time: 0 seconds Depth: 15 deep [passed]
79/129 [00:40:52] Result: Safe Trace: ... long Time: 0 seconds Depth: 15 deep [passed]
80/129 [00:40:52] Result: Safe Trace: ... long Time: 0 seconds Depth: 14 deep [passed]
81/129 [00:40:52] Result: Safe Trace: ... long Time: 0 seconds Depth: 15 deep [passed]
82/129 [00:40:52] Result: Safe Trace: ... long Time: 0 seconds Depth: 16 deep [passed]
83/129 [00:40:52] Result: Safe Trace: ... long Time: 0 seconds Depth: 16 deep [passed]
84/129 [00:40:52] Result: Safe Trace: ... long Time: 0 seconds Depth: 14 deep [passed]
85/129 [00:40:52] Result: Unsafe Trace: 86 long Time: 4 seconds Depth: 86 deep [passed]
86/129 [00:40:57] Result: Safe Trace: ... long Time: 0 seconds Depth: 12 deep [passed]
87/129 [00:40:57] Result: Unsafe Trace: 8 long Time: 0 seconds Depth: 8 deep [passed]
88/129 [00:40:57] Result: Unsafe Trace: 44 long Time: 10 seconds Depth: 44 deep [passed]
89/129 [00:41:07] Result: Unknown Trace: ... long Time: 601 seconds Depth: 1293 deep [NOT passed]
90/129 [00:51:09] Result: Unknown Trace: ... long Time: 601 seconds Depth: 2982 deep [NOT passed]
91/129 [01:01:10] Result: Unknown Trace: ... long Time: 604 seconds Depth: 815 deep [NOT passed]
92/129 [01:11:15] Result: Unknown Trace: ... long Time: 601 seconds Depth: 639 deep [NOT passed]
93/129 [01:21:17] Result: Unknown Trace: ... long Time: 601 seconds Depth: 2903 deep [NOT passed]
```

F.1.2. ábra. A parancssori interfész futás közben.

```
Command Prompt
124/129 [02:12:45] Result: Safe Trace: ... long Time: 0 seconds Depth: 4 deep [passed]
125/129 [02:12:45] Result: Safe Trace: ... long Time: 0 seconds Depth: 305 deep [passed]
126/129 [02:12:45] Result: Safe Trace: ... long Time: 0 seconds Depth: 112 deep [passed]
127/129 [02:12:46] Result: Safe Trace: ... long Time: 0 seconds Depth: 25 deep [passed]
128/129 [02:12:46] Result: Safe Trace: ... long Time: 0 seconds Depth: 61 deep [passed]
129/129 [02:12:46] Result: Unsafe Trace: 52 long Time: 0 seconds Depth: 52 deep [passed]
===== TESTING =====
===== WRITING OUTPUT =====
Successfully completed.
===== WRITING OUTPUT =====
===== STATISTICS =====
The testing ran for 27400 seconds.
Passed: 85/129 tests.
Failed: 44 unknowns.
Failed: 0 not unknowns.
There were 59 safe tests.
There were 26 unsafe tests.
There were 44 unknown tests.
===== STATISTICS =====
```

F.1.3. ábra. A parancssori interfész a futás végén.