MŰEGYETEM 1782

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 8.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Jakab Richárd Benjámin*, 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 8.

Jakab Richárd Benjámin
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

1. fejezet

Bevezetés

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 részcsoportja, 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. Ilyen követelmények lehetnek például a következők [4]:

- 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
- ...

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 fentebb vázolt követelmények közül a biztonságosság követelmény teljesülését ellenőrzi. Ezt a programot a BME VIK Méréstechnika és Információs Rendszerek Tanszék[1] Hibatűrő Rendszerek Kutatócsoportja[2] 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

[1] `https://www.mit.bme.hu/`
[2] `https://www.mit.bme.hu/research/ftsrg`
[3] `https://github.com/FTSRG/theta`

foglalkozva a k-indukció alapú szoftververifikációval, aztán a szakirodalom által bemutatott algoritmust leimplementáltam a *Theta* keretrendszerben, majd végezetül széleskörű tesztelés alá vetettem.

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

- A második fejezetben a dolgozatomhoz szükséges háttérismereteket mutatom be.
- 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 ..., majd a k-indukció nevű matematikai módszert [11] ismertetem a (2.3) alfejezetben, majd végül formalizálom a problémát a (2.4) alfejezetben.

2.1. Általános modellellenőrzés

2.2. Szoftververifikáció

2.3. K-indukció

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

$$P(0) \wedge \forall n(P(n) \Rightarrow P(n+1)) \Rightarrow \forall n P(n). \tag{2.1}$$

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:

$$P(0) \wedge P(1) \wedge \forall n((P(n) \wedge P(n+1)) \Rightarrow P(n+2)) \Rightarrow \forall n P(n). \tag{2.2}$$

Ezt az elvet általánosíthatjuk k lépésre, $k \geq 1$, melyet a irodalom [11] k-indukciónak nevez, formálisan:[1]

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

2.4. A probléma formalizálása

Ahhoz, hogy a problémát precízebben megfogalmazhassuk, szükség van jelölések és fogalmak bevezetésére [9]. 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

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

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:

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

ahol $s_i \in S$ és az $s_{[0..n]}$ rövidítés az $(s_0, s_1, \ldots, s_n)$ állapotsorozatot jelöli. Az útvonal n hosszúságú, ha n darab tranzakcióból áll. A nulla hosszúságú útvonal egy darab állapotot tartalmaz és nem értelmezzük rajta a tranzakció műveletét. Azt a megállapítást, hogy egy Q tulajdonság igaz egy útvonal összes állapotára, úgy fogjuk írni, hogy $\forall .Q(s_{[0..n]})$.

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

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

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:

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

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:

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

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 összetehetőek úgy, hogy azon a probléma szemléletesebb és szimmetrikusabb legyen:

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

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 $\mathit{CFA} = (V, H, I_0, E)$ négyes [1], ahol

- $V = \{v_1, v_2, \ldots\}$ 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 $v_i := \mathit{kif}$ 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 $[\mathit{cond}]$ formában írható le, ahol *cond* egy bináris (*Boolean*) kifejezés (feltétel). Ha egy él rendelkezik $[\mathit{cond}]$ feltétellel, akkor abban az esetben csakis akkor sülhet 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.

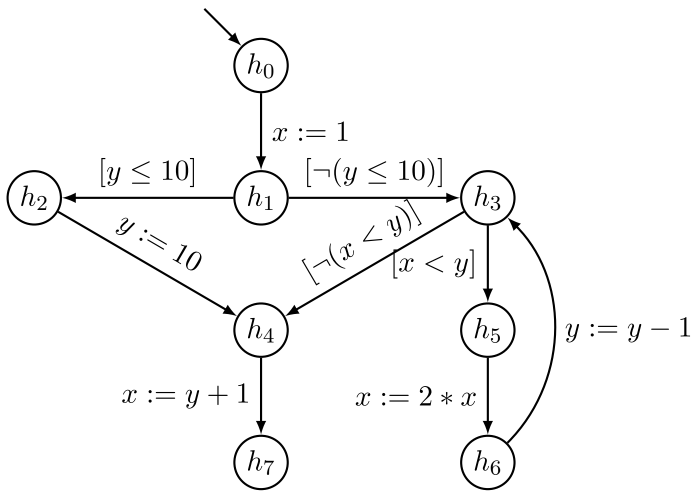
Ha szeretnék egy utasítás nélküli élet húzni két hely között, mely minden körülmények között elsül, azt egy [*igaz*] feltétellel tehetjük meg. TODO.

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 $h_0 \to h_1 \to h_2 \to h_4$. 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 egyiken 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.*

```c
int x = 1;
if (y <= 10) {
  y = 10;
} else {
  while (x < y) {
    x = 2 * x;
    y = y - 1;
  }
}
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) különváló 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.

```
stmt1;
stmt2;
...
```

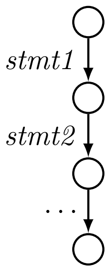
(a) Szekvencia a programban.

```
if (cond) {
 stmt1;
 ...
} else {
 stmt2;
 ...
}
```

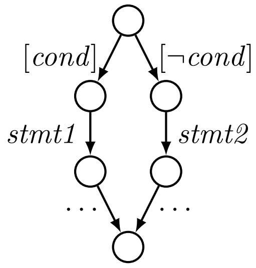
(b) Feltételes elágazás a programban.

```
while (cond) {
 stmt;
 ...
}
```

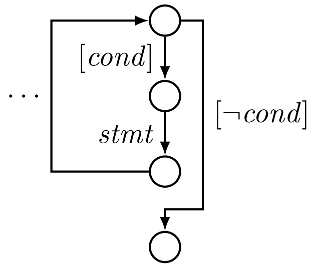
(c) Feltételes ciklus a programban.



(d) Szekvencia a CFA-ban.



(e) Feltételes elágazás a CFA-ban.



(f) Feltételes ciklus a CFA-ban.

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 állapotnál 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 élet, majd a többi hibahelyet visszaminősítjük egyszerű vezérlési helynek. Az egyetlen megmaradt hibahely pedig egy, az Algoritmuselméletből[1] jól ismert nyelőhely 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.

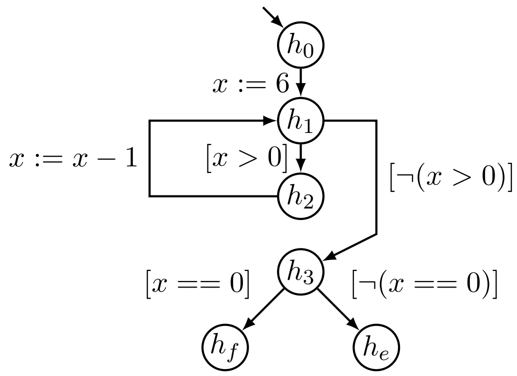
Innentől kezdve az lesz a vizsgálatunk célja, hogy megállapítsuk, elérhető-e a hibahely az adott CFA-ban. Ezért a (CFA, h_e) párost verifikációs feladatnak nevezzük – a program *helyes*, ha a hibahely nem elérhető, különben pedig *hibás*.

[1] `http://www.cs.bme.hu/algel/`

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.*

```
1   int x = 6;
2   while (x > 0) {
3       x--;
4   }
5   assert(x == 0);
```

(a) C program *assert* paranccsal.



(b) A program CFA reprezentációja.

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

Állapottér. TODO.

3.2. Az algoritmus formalizálása

A dolgozatomban arra a kérdésre keresem a választ, hogy a (2.4)-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

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

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 kívánatos 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 $I(s_0) \wedge \mathit{utvonal}(s_{[0..i]})$ feltétel önmagában nem fog gyorsítást eredményezni: vagy végig megy az állapottéren amilyen hosszan csak lehetséges (ezt szeretnénk lerövidíteni), tekintve, hogy minden állapotnak van egy szülőállapota a T tranzakciós reláción keresztül, vagy végtelen ciklusba kerül.

Ennél jobb stratégia, ha akkor állunk meg, mikor $I(s_0) \wedge \mathit{cmUtvonal}(s_{[0..i]})$ ellentmondásos lesz. Ezt használva addig folytatjuk a keresést, míg az összes, ciklusmentes útvonalat be nem jártuk. Legrosszabb esetben ekkor is végigmegy a program a teljes állapottéren, viszont ha az állapottérben ciklikusság figyelhető meg, akkor azt a stratégia maximálisan kihasználja: nem fog végtelen ciklusba kerülni, illetve átlagosan rövidebb (de bizonyosan nem hosszabb) útvonalakat fog bejárni, mint az $I(s_0) \wedge utvonal(s_{[0..i]})$.

Ehhez hasonlóan tehetjük azt is, hogy addig ellenőrzünk, amíg a $cmUtvonal(s_{[0..i]}) \wedge \neg P(s_i)$ nem lesz ellentmondásos: egy, a P tulajdonságot sértő állapotból (hibaállapotból) kiindulva addig megyünk ciklusmentes útvonalakon 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 visszafele 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 előrefelé, addig az utóbbiból visszafelé. Kijelenthető, hogy a modell biztonságos, ha az előrefelé 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*).

A (2.3)-as fejezetben 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 [3], illetve mellé az is, hogy ezzel az indukciós lépést bizonyítottuk [9], 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 csak 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 állapottérben 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* $Sat(utvonal(s_0, s_1, s_2))$ *kifejezés megegyezik a* $Sat(T_0 \wedge T_1 \wedge T_2)$ *kifejezéssel a (2.4) egyenlet miatt.*

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

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

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 eset*nek: 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(...)* visszatérési értéke „igaz” lesz, mely a negált hatására „hamis”ra fordul és folytatjuk tovább az ellenőrzést.

- A második feltétel megfelel a fentebb említett *második eset*nek: elindulunk egy $s_e \in S : \neg P(s_e)$ hibaállapotból visszafele, amíg van olyan elérhető állapot, melyben még nem jártunk.

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 bejárási mélységet.

Algorithm 1: Checking if system is P-safe

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

Állítás. Tegyük fel, hogy az 1-es algoritmus „True” válasszal tért vissza. Ekkor a modell P-biztonságos.

Bizonyítás: ...

4. fejezet

Implementáció

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[3] 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 program melyet fejlesztettem.
- **Sat-megoldó interfész** – Ennek segítségével történik a verifikáció. A Theta a Z3 Sat-megoldót használja jelenleg.
- **Eszközök** – Parancssori alkalmazások melyek futtatható `jar` fájlba fordíthatóak le. Jellemzően csak beolvassák az inputot és meghívják az alsóbb szinten lévő algoritmusokat. TODO: tud az enyém parancssorból futni?

[1] `https://github.com/FTSRG/theta`

[2] 2020 decemberének elején.

[3] Az angol irodalom a *location* kifejezést használja. Én a dolgozatomban a magyar megfelelőjét használom. TODO: nincs ez előbb definiálva már?

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 elöljáróban azt érdemes tudni, hogy bizonyos CFA modellekre az algoritmus lassan terminál. Ezért, ha a felhasználó igényeinek megfelel, bemenetnek megadhat egy maximális időkorlátot vagy egy maximális mélységet is, esetleg mindkettőt, melyek felső korlátot fognak megszabni 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`**

A **`KInduction`** a főosztályom, ő teszi kontextusba és adja meg az ellenőrzés ívét. Benne található a `check(...)` függvény, melyet a tesztelő osztály `CfaTest` 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 Szoftvertechnikából[4] ismert Singleton tervezési mintát, illetve az Initialization-on-demand holder [7] tervezési mintát is. Minden, az útvonal alakításához, feldolgozásához szükséges műveletet ebbe az osztályba szervezem ki függvények formájában.

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

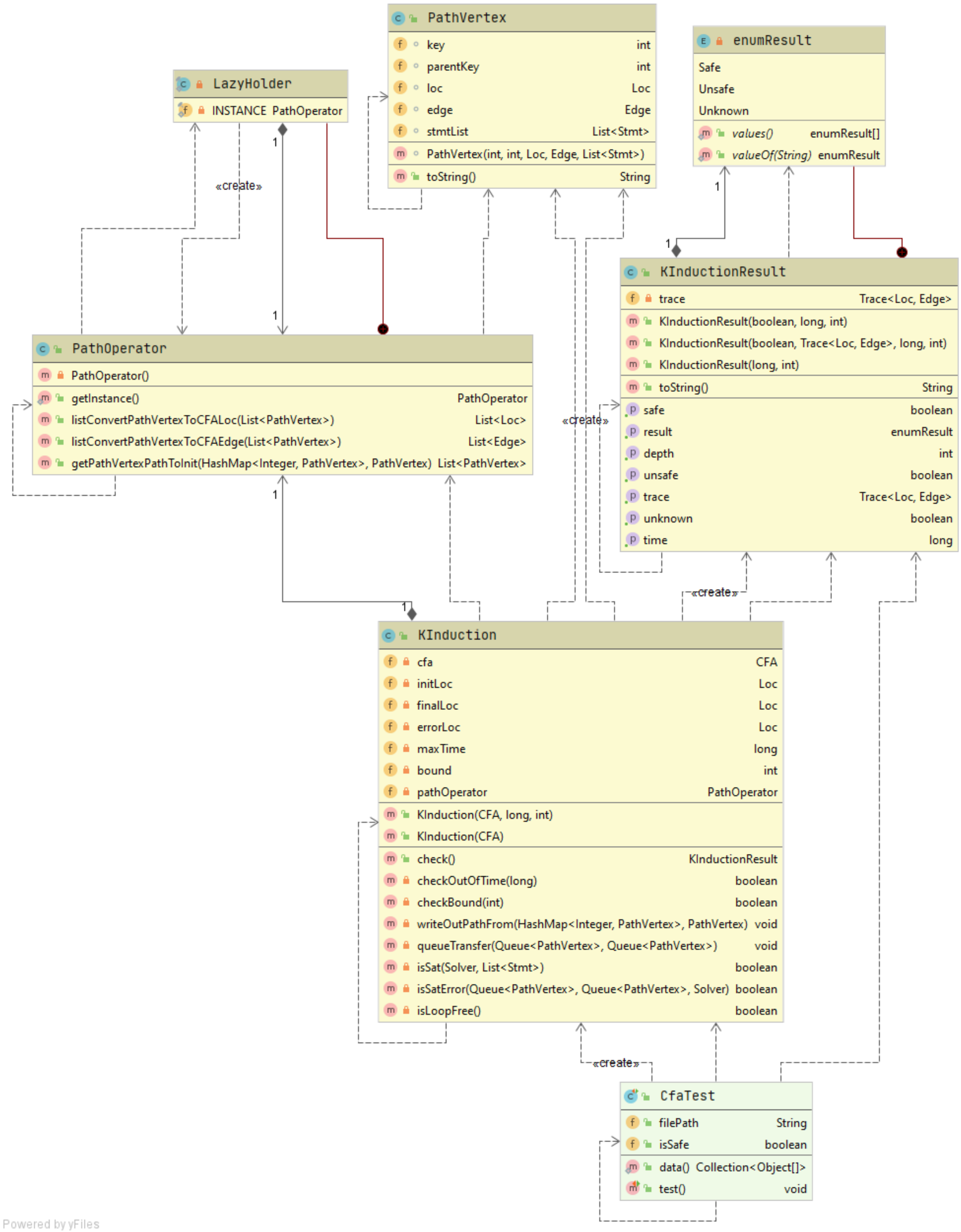
- `key` – int típusú egyedi kulcs (azonosító), hogy minden PathState egyedi legyen.
- `parentKey` – int típusú egyedi azonosító ahhoz a szülő PathState állapothoz, mely az útvonalban eggyel megelőzi őt (tehát a szülő PathState az az állapot, melyből a bejárás során eljutottunk ehhez a PathState állapothoz).

[4]`https://www.aut.bme.hu/Course/VIAUAB00`

- 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 bejárási 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 **KInductionCommandLine** osztály a parancssorból történő tesztelésért illetve verifikációért felel. Lehet csak egy CFA-t tesztelni, de arra is van lehetőség, hogy egy .csv fájlba kigyűjtött teszteket verifikálja. Az utóbbinál a program a futás után létrehoz egy .csv fájlt, amibe beleírja a futtatott program adatait illetve a futás eredményét is.



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.
    - **Bemenet**: CFA modell
    - **Kimenet**: -
  - KInduction(CFA cfa, long maxTime, int bound) – Az osztály három paraméterű konstruktora.
    - **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.
  - checkBound(int depth) – Ellenőrzi, hogy ha van megadott mélységkorlát akkor azt nem-e léptük már át.
    - **Bemenet**: Az aktuális bejárásra váró mélység.
    - **Kimenet**: Boolean.
  - queueTransfer(Queue<PathState> copyFrom, Queue<PathState> pasteTo) – A korlátos szélességi kereséshez két sort használok, az egyik a még bejárandó 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 sor.
    - **Kimenet**: A pasteTo sor tartalmazni fogja a copyFrom sor tartalmát (mást nem, mert előtte kiürítjük), a copyFrom sor pedig üres lesz.
  - isSat(Solver solver, List<Stmt> stmtList) – Megnézi a solver megoldó 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 említetthez 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 `KIduction` 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 `KIduction` osztály akkor inicializál ezzel egy objektumot, mikor a futás eredménye *helyes* volt.
    * **Bemenet**: A program futási eredménye (hiba, ha nem `true` az érték), 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 `KIduction` 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), 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**: -

- `KInductionCommandLine` osztály
  - main(final String[] args) – Létrehoz egy `KInductionCommandLine` objektumot aminek átadja a parancssori argumentumokat és aminek meghívja utána a `run()` függvényét.
    * **Bemenet**: - Parancssori argumentumok.
    * **Bemenet**: -
  - KInductionCommandLine(final String[] args) – A kapott parancssori argumentumokat eltárolja.
    * **Bemenet**: - Parancssori argumentumok.
    * **Bemenet**: -
  - run() – A parancssorból beolvasott paramétereket feldolgozza.
    * **Bemenet**: -
    * **Bemenet**: -

- `CfaTest` osztály
  - test() – @Test annotációval ellátott függvény, mely a JUnit tesztelésért felelős.
    * **Bemenet**: -
    * **Bemenet**: -

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 bejárandó 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 (*BackWard*), 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 a `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. Utóbbit utána 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 modellről 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.

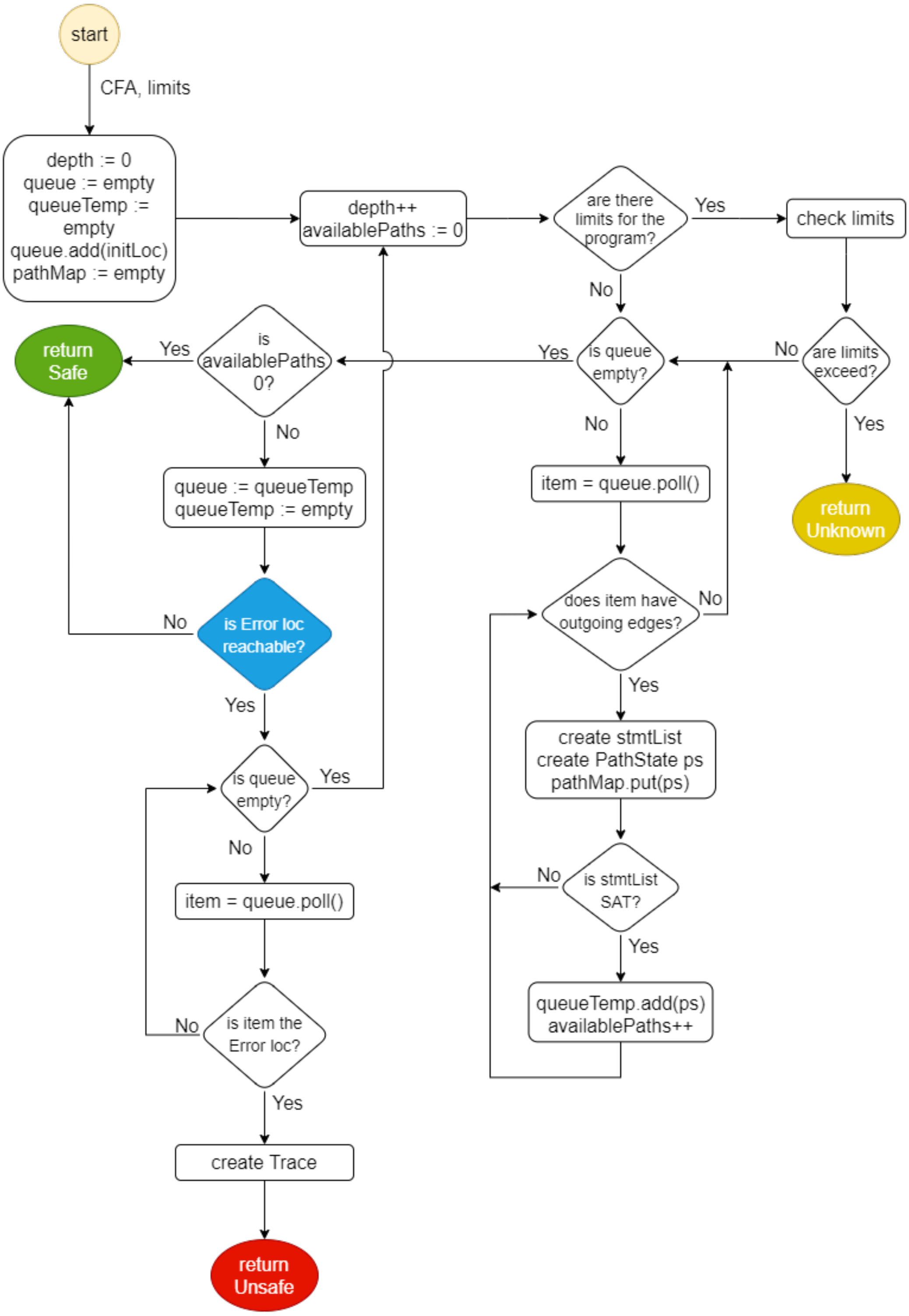
   (c) Ha a hibahely nem érhető el, akkor létrehoz egy KInductionResult objektumot és befejezi a program a futását *helyes* válasszal.

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.

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 [10], 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-ról[1] származik ahol eredetileg C kódok voltak amik aztán át lettek CFA-ba alakítva [8]. Ezek különböző csoportba sorolhatóak [6]:

- `locks` kicsi (94-234 LoC[2]) 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) feladatok gyors teszteléshez.

Továbbá volt alkalmam tesztelni olyan CFA modelleket, melyek eredetileg ipari PLC szoftverkódok voltak a CERN-nél. [2] 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 [6]:

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

- `--model` – A CFA teszt elérési útvonala, ha csak egy darab tesztre szeretnénk lefuttatni. Nem kötelező.

[1] `https://sv-comp.sosy-lab.org/2018/`
[2] Source lines of code - Hány sorból áll a CFA modellt leíró szöveges fájl.

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
Total	*479*				

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

- `--input` – A CFA teszteket, 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 bejárási mélységét tudjuk vele korlátozni. Nem kötelező, alapértelmezetten -1.
- `--output` – A kimenetel fájl neve (input paraméter esetén). Nem kötelező, alapértelmezetten „output.csv".

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

```
if (!input.equals("") && !model.equals("")) {
  Assert.fail("Only one input source is allowed .");
}
if (input.equals("") && model.equals("")) {
  Assert.fail("One input source is required .");
}
```

5.2. Eredmények

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

```
java -jar theta-k-induction.jar --input input_svcomp_xx.csv --output
  output_svcomp_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 tesztkategóriát 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 kapott eredményfájlokat a Pandas[3] Python programkönyvtárral dolgoztam fel.

[3] https://pandas.pydata.org/

6. fejezet

Összefoglaló

Köszönetnyilvánítás

Irodalomjegyzék

[1] 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.

[2] 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.

[3] 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.

[4] Dr. Majzik István: Rendszertervezés és -integráció. URL: `https://www.mit.bme.hu/system/files/oktatas/targyak/10019/VIMIMA11_RTI_08_Biztonsagi_alapfogalmak_1.pdf`, 2018. 12.

[5] Dr. Ákos Hajdu: Formal software verification. URL: `https://ftsrg.mit.bme.hu/software-verification-notes/software-verification.pdf`, 2020. 11.

[6] 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`.

[7] 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`.

[8] 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.

[9] 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.

[10] Tamás Tóth. URL `https://github.com/ftsrg-ca/ca/tree/master/program-verification-1/src/test/resources`.

[11] Thomas Wahl: The k-induction principle. URL `http://www.comlab.ox.ac.uk/people/Thomas.Wahl/Publications/k-induction.pdf`.