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 4.

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 4.

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.

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 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 keretrendszerbe beillesztettem, 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

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

tesztelés alá vetettem és így finomítottam az algoritmusomat.

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

- A második fejezetben a szoftververifikációt mutatom be általános megközelítésben
- A harmadik fejezetben kifejtem az algoritmusom alapját adó K-indukció módszer elméleti hátterét
- A negyedik fejezetben bemutatom az algoritmusom elkészítésének folyamatait és technikai felépítését
- Az ötödik fejezetben bemutatom az algoritmusom teszteredményeit és összehasonlítom más, verifikáló algoritmusokkal

2. fejezet

Háttérismeretek

Ebben a fejezetben a dolgozat további részeinek megértéséhez szükséges elméleti előismereteket mutatom be. Először a k-indukció nevű matematikai módszert [6] ismertetem (Alfejezet 2.1.), majd formalizálom a problémát (Alfejezet 2.2.), végül pszeudokód szinten bemutatom az ezen a matematikai módszeren alapuló algoritmus működését és annak helyességét (Alfejezet 2.3.).

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 tudatában 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 [6] k-indukciónak nevez, formálisan:[1]

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

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

2.4. A probléma formalizálása

Ahhoz, hogy a problémát precízebben megfogalmazhassuk, szükség van jelölések és fogalmak bevezetésére [5]. 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. 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 a $s_{[0..n]}$ rövidítés az $(s_0, s_1, \ldots, s_n)$ állapotsorozatot jelöli. Az *utvonal* n hosszúságú, ha n darab tranzakcióból áll. A nulla hosszúságú *utvonal* 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, $utvonal_i(s_0, s_i)$ alatt azt jelöljük, hogy *létezik* egy útvonal s_0-ból s_i-be, mely i darab T-ből áll.

Legyen T egy tranzakciós reláció S állapothalmazán. 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 relációt ismétlődően alkalmazva 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 technikai oldalának megértéséhez. Először ismertetem a Control Flow Automata ábrázolás részleteit (Alfejezet 3.1)), aztán az előző fejezetben bemutatott jelölésrendszerrel formalizálom az algoritmust (Alfejezet 3.2).

3.1. Control Flow Automata

A programokat sokféleképpen ábrázolhatjuk [3]. Legismertebb a programkód, melyet az ember könnyen, gyorsan tud olvasni illetve írni, 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 [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 kezdőállapota a gráfnak, a program belépőpontját mutatja,
- $E \subseteq H \times Op \times H$ az irányított élek halmaza, melyek helyeket kötnek össze és a változókra vonatkozó utasításokkal vannak felcímkézve

Utasítások. Háromféle utasítással foglalkozunk:

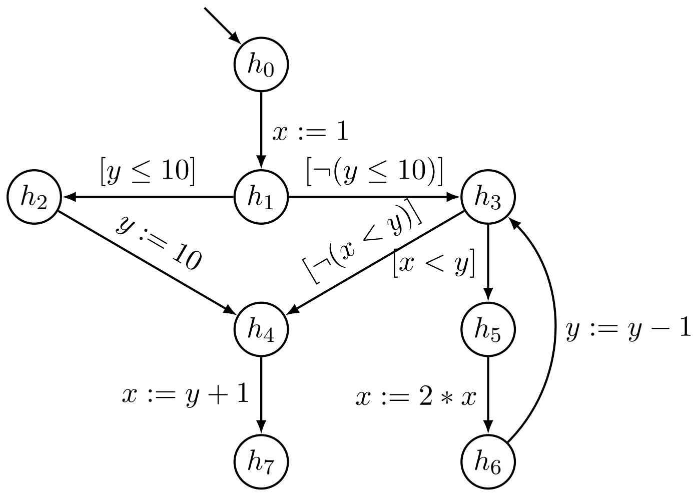
- A *hozzárendelés* utasítás a $v_i := kif$ összefüggéssel írható le. Azt jelöli, hogy a baloldali v_i változóhoz hozzárendeljük a jobb oldali kifejezést. Fontos, hogy a *kif* kifejezésnek is ugyanolyan doménnel kell rendelkeznie, mint a $v_i \in V$ változónak.
- A *feltevés* operátor a $[cond]$ formában írható le, ahol $cond$ egy bináris ($Boolean$) kifejezés (feltétel). Ha egy él rendelkezik $[cond]$ feltétellel, akkor abban az esetben csakis akkor 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.

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. [3].

```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).

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.*

Programábrázolás. A 3.2 ábra megmutatja, hogy az alap elemei a strukturált programozásnak miként képezhetőek le CFA alakba [3].

- *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 történhetnek, azonban az út mindig visszatér a ciklusfejhez.

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

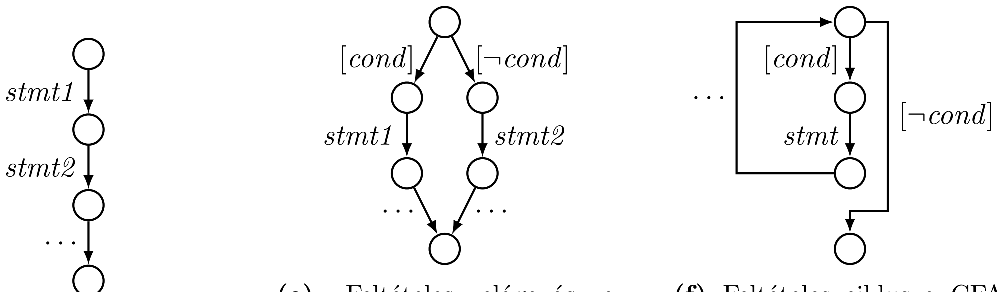
(a) Szekvencia a programban.

```
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.

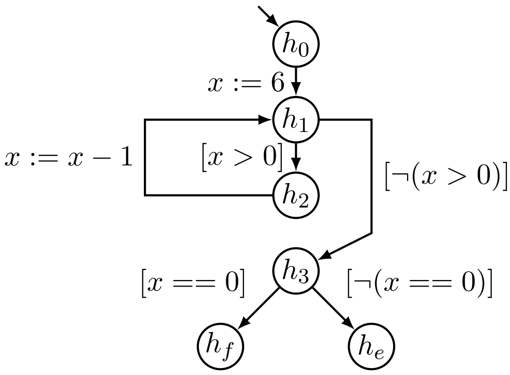
A CFA-ban az *assert* egy speciális döntésként értelmezhető. Ha a feltétel igaz, a program a következő állapotnál folytatódik, ha pedig nem, akkor egy különálló $h_e \in H$ hibahelyre jutunk. Ha több ilyen *assert* szerepel a programban, akkor a CFA hibahelyei összevonhatók: létrehozunk egy új hibahelyet, az összes többi hibahelyből vezetünk ide egy operátor nélküli élet (amely úgy is felfogható, hogy az él feltétele $[igaz]$), majd a többi hibahelyet visszaminősítjük egyszerű vezérlési helynek. Az egyetlen megmaradt hibahely pedig egy, az Algoritmuselméletből[1] jól ismert nyelőhely lesz. Innentől kezdve az lesz a vizsgálatunk célja, hogy megállapítsuk, elérhető-e hibahely az adott CFA-ban. Ezért a (CFA, l_e) párost verifikációs feladatnak nevezzük – a program *helyes*, ha a hibahely nem elérhető, ha elérhető, akkor pedig *hibás*.

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

```c
int x = 6;
while (x > 0) {
    x--;
}
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.

3.2. Az algoritmus formalizálása

A 2.4-es alfejezetben elmondottak segítségével hogy tudnánk belátni, hogy a modell a P tulajdonságra nézve biztonságos?

Például úgy, ha megnézzük, hogy 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 elvárható 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 az adott 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? A $I(s_0) \wedge \mathit{utvonal}(s_{[0..i]})$ 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 $I(s_0) \wedge \mathit{utvonal}(s_{[0..i]})$.

Ehhez hasonlóan tehetjük azt is, hogy addig ellenőrzünk, amíg a $\mathit{cmUtvonal}(s_{[0..i]}) \ \wedge \ \neg P(s_i)$ 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 belátjuk a fentebb említett metodika alapján, hogy a modell biztonságos, akkor kijelenthető, hogy $k+1$ mélységre is biztonságos lesz [2], illetve mellé az is, hogy ezzel az indukciós lépést bizonyítottuk [5].

3.2.1. Elérhetőség vizsgálata

Definíció. Egy $s_i \in S$ állapot elérhető az $s_j \in S$ állapotból, ha létezik olyan *utvonal*, 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 ($x > 0$). 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 elérhető, 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.*

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.

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

- 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 ¬Sat(I(s_0) ∧ cmUtvonal(s_[0..i])) ∨ ¬Sat(cmUtvonal(s_[e..e+i]) ∧ ¬P(s_e)) then
4          return True
5      end
6      if Sat(I(s_0) ∧ utvonal(s_[0..i]) ∧ ¬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 *Symbolic Transition Systems*
- **Háttéranalízis** –
- **SMT megoldó** –
- **Eszközök** – Parancssori alkalmazások melyek futtatható `jar` fájlba fordíthatóak le. Jellemzően csak beolvassák az inputot és meghívják az alsóbb szinten lévő algoritmusokat.

[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.

5. fejezet

Kiértékelés

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] 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.

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

[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] 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.

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