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

Állapottérképek modellellenőrzése

Szakdolgozat

Készítette
Román Dávid

Konzulens
Tóth Tamás
Vörös András

2018. június 27.

Tartalomjegyzék

1. fejezet

Bevezetés

A technológia fejlődésével életünk legtöbb területén egyre jobban hagyatkozunk szoftverek használatára. Általában a feladat elvégzését gyorsabban, megbízhatóbban végzik el, mintha emberi munkaerőt alkalmaznánk. Így nem meglepő, hogy olyan helyeken is alkalmazzuk, ahol a biztonságos, hibamentes üzemeltetés alapfeltétel. Az olyan rendszereket ahol a hibás működés súlyos anyagi, vagy akár ember életekben mérhető károkat okoz, biztonságkritikus rendszereknek nevezzük. Ilyen rendszerek vannak például az autóknál, vasutaknál, repülőknél, reaktoroknál, vagy egészségügyi rendszereknél. Ezeken a területeken is egyre több feladatot programok végeznek el. Mivel a helyes viselkedés létfontosságú, fontos, hogy ezeknek a szoftvereknek tudjuk bizonyítani a hibamentességét.

Viszont ezek a rendszerek általában nagyon komplexek, ezért nehéz rajtuk bizonyításokat végezni. Erre az egyik megoldás, hogy a rendszernek egy egyszerűsítésén dolgozunk, ahol csak a számunkra fontos jellemzőit tartjuk meg. Ez a modellezés. Számos modellezési technológiák vannak, én a félév során az állapottérképes modellezéssel foglalkoztam. Ezt a technikát gyakran alkalmazzák iparban is, sok eszköz elérhető ezek szerkesztésére, vizsgálatára. Hasznos tehát, hogy minél megbízhatóbb ugyanakkor elég gyors ellenőrzéseket tudjunk végezni ezeken a modelleken.

Félév során én az állapotgép modellezéssel, és az ezeken a modelleken végzett ellenőrzéssel foglalkoztam. A második fejezetben megmutatom, hogy milyen meglévő projekteket, használtam, illetve mi módon próbáltam ezeket tovább fejleszteni.

Harmadik fejezetben az általam is tovább fejlesztett java állapotgép implementációt mutatom be, kiemelve, hogy mi a saját munkám.

Negyedik fejezetben bemutatom a Gamma állapotgépek transzformációját Theta állapotgéppé.

Ötödik fejezet az állapotgép konfigurációt mutatja meg.

Hatodik fejezetben leírom azt a modellellenőrzőt, amit a félév során sikerült implementálnom.

Hetedik fejezet egy rövid verifikáció az elvégzett munkámhoz.

2. fejezet

Arhitektúra

Ebben a fejezetben megmutatom milyen már működő keretrendszereket használtam, illetve próbáltam hozzájárulni fejlesztésükhöz a féléves munkám során

2.1. Gamma keretrendszer

A félév során megpróbáltam már meglévő keretrendszerekbe beledolgozni. Ilyen keretrendszer a Gamma[1], ami a tanszéken fejlesztett rendszer és összetett állapotgépek analízisét teszi lehetővé számos eszköz segítségével.

Az Uppaal model checker[2] például lehetővé teszi a Gamma állapotgép modellellenőrzését. Viszont ez az eszköz nem tudja kezelni az összetett állapotgépeket, ezért át kell alakítani a Gamma állapotgépet, hogy ne legyenek már benne összetett állapotok, ekkor viszont információt veszítünk[3] és az analízis jelentősen lassabb lehet. Van igény tehát olyan modellellenőrzésre a Gamma keretrendszeren belül, ami kihasználja az összetettséget.

A Gamma állapotgépek EMF[4] és XTEXT[5] technológiát használnak, ami nehezebbé teszi a modell architektúra tetszőleges kialakítását, ezért a félév során egy új ezektől a technológiáktól nem függő állapotgép Pojo-t csináltam amit, majd a következő fejezetben fogok bemutatni.

Még egy hasznos funkciója a Gammának, hogy lehet vele Yakindu[6]-ban szerkesztett állapotgépeket Gamma állapotgéppé alakítani.

2.2. Theta keretrendszer

A Theta[7] szintén a tanszéken fejlesztett eszköz, (ami egyéb dolgok mellett) interfészt nyújt több SMT Solverhez, megvalósít különböző absztrakciós módszereket. Ezek a funkciók nagyon hasznosak lehetnek az állapotgépek modellellenőrzéséhez.

Nem meglepő tehát, hogy a már említett Pojo-t ebbe a keretrendszerbe szeretnénk beépíteni. Ezt a Pojo-t innentől tehát Theta állapotgép-nek fogom nevezni.

[1] A Gamma hivatalos oldala: `https://inf.mit.bme.hu/node/6028`

[2] Az Uppaal weboldala: `http://www.uppaal.org/`

[3] Magyarázat Az összetett állapotgépeknél

[4] Eclipse Modelling Framework `https://www.eclipse.org/modeling/emf/`

[5] `https://www.eclipse.org/Xtext/`

[6] Egy nagyon elterjedt állapotgép szerkesztő eszköz, weboldala: `https://www.itemis.com/en/yakindu/state-machine/`

[7] hivatalos weboldala: `https://inf.mit.bme.hu/en/theta`

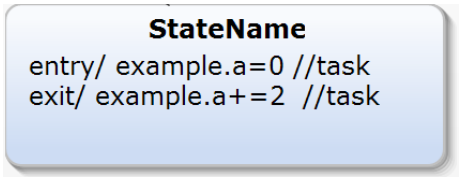
3. fejezet

A Theta állapotgép bemutatása

Ebben a fajezetben bemutatom a Theta állapotgép reprezentációt. Mivel egy már korábban mások által elkezdett projektet fejeztem be külön kitérek arra is, hogy mely részek a saját fejlesztésem részei.

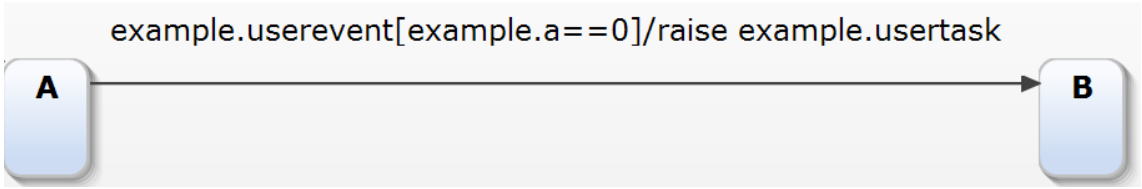
3.1. Az állapotgép

Az állapotgép egy széles körben használt modellezési módszer. A modellezni kívánt rendszerünket több egymástól jól elkülöníthető állapotokra bontjuk, ez lesz a rendszer állapota. Elvégezhetünk különféle feladatokat, akkor ha belépünk, vagy akkor ha kilépünk az adott állapotból.



3.1. ábra. Egy állapot Yakinduban.

A rendszer természetesen működés közben változik, másik állapotba kerül. Az állapotváltást a tranzíciók segítségével írjuk le. Ezeket mindig valamilyen esemény váltja ki, lehet egy felhasználói esemény, vagy egy időzített esemény. Őrfeltételt is adhatunk `[]`-en belül ilyenkor csak akkor lépünk át a másik állapotba, ha ez a feltétel teljesül. A `/` után pedig újabb parancsokat hajthatunk végre



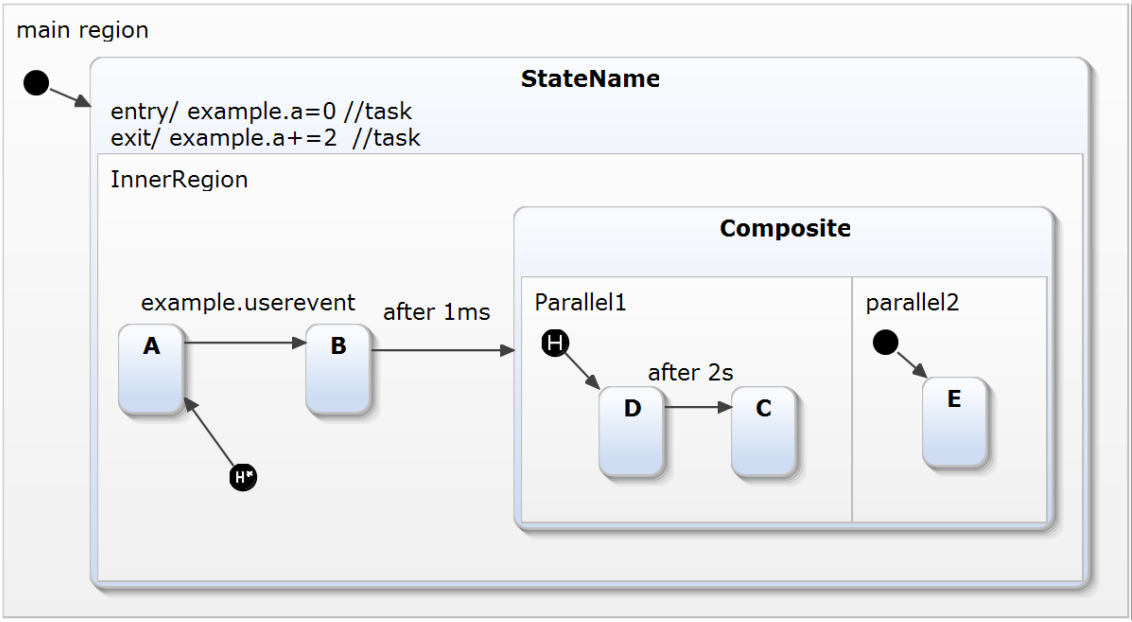
3.2. ábra. Egy tranzíció Yakinduban.

Előfordulhat az is, hogy több állapotot valamilyen közös tulajdonságuk szerint szeretnénk csoportosítani. Ilyenkor jók az összetett állapotok. Ezt az információt veszítjük tehát el, ha ”kilapítjuk”[1] az állapotgépet.

[1]Több állapot felvételével helyettesítjük az összetett állapotokat

Itt kell bevezetni a régió fogalmát. Minden állapot egy régióban van, és minden régióban egyszerre csak egy állapot írhatja le a rendszerünket. Az összetett állapotok tehát tartalmaznak legalább egy régiót.

A régión belül van egy úgynevezett pszeudó állapot, ami kijelöli, hogy egy régióba lépés esetén melyik állapotba lépjünk. Három fajtája van a history, deephistory és a sima kezdő állapot. A sima mindig ugyanarra az állapotra mutat, a history megjegyzi, hogy kilépéskor melyik volt aktív és oda léptet vissza. A deephistory még ennél is többet jegyez meg, ha a kilépéskori aktív állapot, egy összetett állapot, akkor az azon belüli aktív állapotokat is megjegyzi.



3.3. ábra. Egy összetett állapotgép Yakinduban.

3.2. A Theta állapotgép

Az állapotgép reprezentáció Java-ban készült el. Az egyes osztályok megfeleltethetők az előző fejezetben leírt állapotgép elemekkel. Mindegyik osztálynak vannak a tárolt elemeihez getter/setter, a tárolóikhoz pedig add függvényei. Illetve néhány segédfüggvény, ami segít majd az analízis közben. A projekt már korábbi munkák folytatásaként jött létre. Az én hozzájárulásom a pszeudó állapotok implementálása, az időzített események kezelése és néhány segéd függvény, illetve apróbb javítások.

```
interface Sc {}
```

Az állapotgépet reprezentáló interfész. Ez tárolja a gyökér régiót. Van neve, tárolja az összes tranzíciót az állapotgépben. Tárolja a változó deklarációkat egy a Thetában értelmezett osztályban (VarDecl).

Az egyetlen megvalósító osztály a `MutableSc` minden tárolója hashSet-ben van megvalósítva

```
interface Region {}
```

Van neve, tárolja az állapotait (State), ezek lekérdezhetőek és pontosan egy pszeudó állapota (PseudoState) van. Lehet szülő állapota; az állapot ami közvetlen őt tartalmazza. Továbbá van egy Boolean attribútuma, ami azt jelzi, hogy RootRegion-e a régió azaz

nincs neki szülőállapota. Ellenőrizhető, hogy szabályos-e a régió: Minden tartalmazás igaz fordított irányban (vagyis az elem szülő eleme a régió)

Az egyetlen megvalósító osztály a `MutableRegion` egy hashSet-ben tárolja az állapotait

```
interface State {}
```

Van neve és tárolja a régióit (Region) ez lehet egy üres lista. Kötelezően van szülő régiója. Van két akciója (Action), egy a kimeneti, egy a bemeneti akcióknak. Van két tranzíció tárolója, tárolja ugyanis a beérkező tranzíciókat és a kimenőket is.

Az egyetlen megvalósító osztály a `MutableState` minden tárolója hashSet-ben van megvalósítva

```
interface Transition {}
```

A tranzíciót reprezentáló interfész. Tárolja a forrás állapotát (honnan indul ki) és a cél állapotát (hova érkezünk). Továbbá van egy kiváltó (trigger) eseménye (Event). Lehet őrfeltétele, ami a Thetában is használt Expr<Type> osztály. Van egy akciója (Action)

Az egyetlen megvalósító osztály a `MutableTransition`

```
interface PseudoState {}
```

Van neve, tárolja azt az egy állapotot ami a régiójába lépéskor az aktuális állapot lesz. Kötelezően kell legyen egy szülő régiója (Region).

Három megvalósító osztály a `MutableInitState` a sima kezdő a `MutableHistorytate` a history a `MutableDeepHistoryState` pedig a deephistory pszeudó állapothoz

```
interface Action {}
```

Interfész az akcióknak. Visitor mintát követi. Négy különböző osztály van.

Az `AssignmentAction` egy egyszerű változó értékadást reprezentál, van tehát egy változója (VarDecl) és egy kifejezése (Expr).

A `SignalAction` egy esemény (Event) jelzést reprezentál. Ennek megfelelően tárol egy eseményt (csak egyet)

Az `EmptyAction` azért van, hogy ne null-okat tároljunk ha egy tranzíciónak, vagy állapotnak nincs akciója

A `SequenceAction` Több akciót (Action) tárol

```
interface Event {}
```

Az eseményeket reprezentáló interfész alapvetően, csak egy neve van.

Két megvalósító osztály van az `EventImpl` a user eseményekhez, a `TimeEventImpl` pedig az időzített eseményeknek a 3.3-as ábrán is látható after 2s is egy ilyen esemény. Tárolja a nevén kívül azt, hogy mennyi az időzítés (milliszekundumban).

3.3. Sorosítás

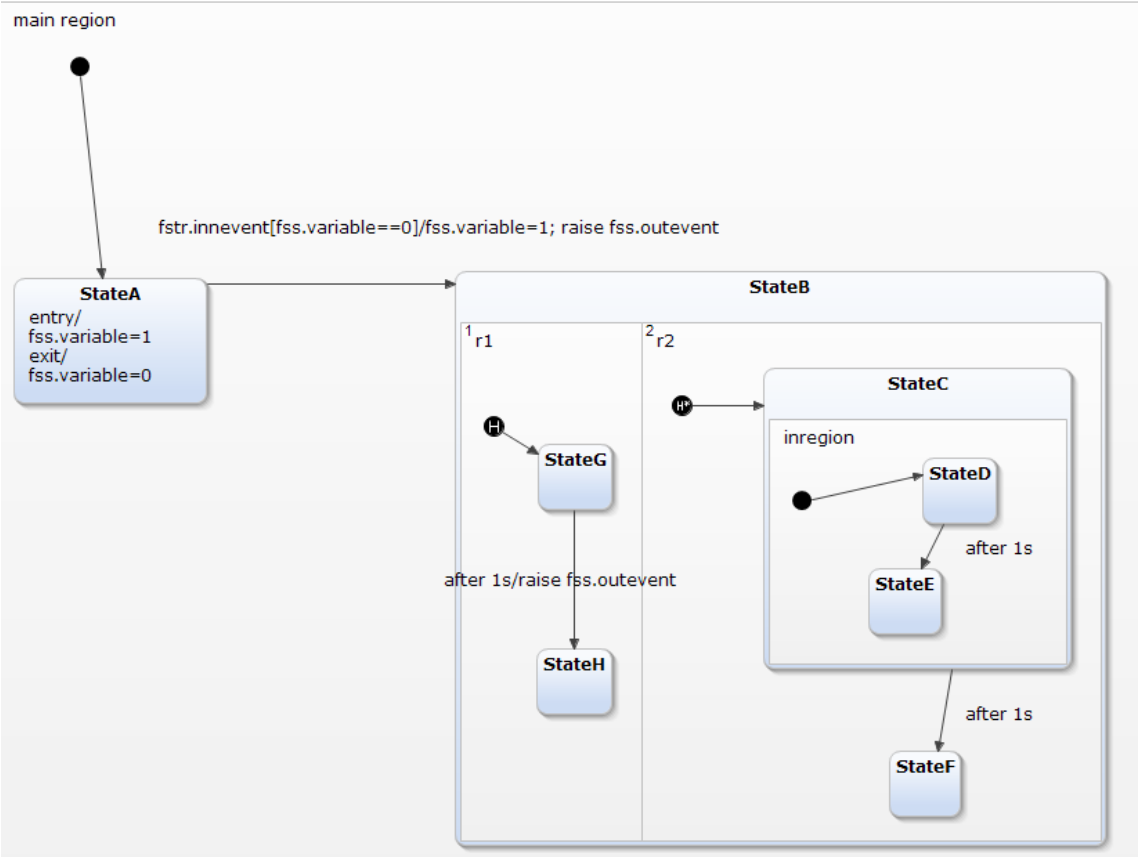
Amikor egy adott modellen akarunk végezni analízist, valószínűleg nem csak egyszer tesszük ezt meg, hanem többször is. Nem lenne túl hatékony, ha mindig újra kéne építeni az állapotgépet, erre van a sorosítás, vagy szerializáció, ami lehetővé teszi az állapotgépek gyors beolvasását, illetve akár néhány módosítás utáni kiírását.

A séma Lisp-szerű szintaxist követ. A fent említett elemek sorosítva a következőképp néznek ki:

```
(statechart nameOfStateChart
  (var varName varType)
  (var var2 type)
  (event Esemény)
  (region regionName
    (state stateName
```

```
    (entry assign varName value)
    (exit assign var2 value))
  (state compositeState
    (region innerRegion
      (state inState)
      (state stateA))
      (init inState))
  (history stateName))
(transition stateName compositeState
  (trigger Esemény)
  (guard varName>2)
  (effect
    (sequence
        (assign varName value)
        (signal Esemény))))
```

Azt érdemes megemlíteni, hogy egy-egy elemet előbb deklarálunk, például (var varName varType) és utána már elég a nevével hivatkozni rá. Például (assign varName value)



3.4. ábra. példa állapotgép a szerializációhoz

```
<terminated> ParserTest [JUnit] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe (2018. jún. 26. 14:22:48)
(statechart name
  (var variable Int)
  (normal fss.0)
  (normal fstr.0)
  (timeout StateDTimeout4 1000)
  (timeout StateCTimeout5 1000)
  (timeout StateGTimeout6 1000)
  (region main_region
    (state StateA
      (entry (assign variable 1))
      (exit (assign variable 0)))
    (state StateB
      (region r1
        (state StateG)
        (state StateH)
        (history StateG))
      (region r2
        (state StateC
          (region inregion
            (state StateD)
            (state StateE)
            (init StateD)))
        (state StateF)
        (deephistory StateC)))
    (init StateA))
  (transition StateD StateE
    (trigger StateDTimeout4)
    (guard true))
  (transition StateA StateB
    (trigger fstr.0)
    (guard (= variable 0))
    (effect (sequence (assign variable 1) (signal fss.0))))
  (transition StateG StateH
    (trigger StateGTimeout6)
    (guard true)
    (effect (signal fss.0)))
  (transition StateC StateF
    (trigger StateCTimeout5)
    (guard true)))
```

3.5. ábra. a fenti állapotgép sorosítva

4. fejezet

Transzformáció

A Gamma keretrendszert is szeretnénk összekötni az előző fejezetben bemutatott Theta állapotgéppel. Ezért írtam egy segéd osztályt, ami képes a beolvasott EMF-es Gamma állapotgépet átalakítani Theta állapotgéppé.

4.1. Gamma állapotgép

Már korábban említettem az Architektúra fejezetben, hogy a Gamma állapotgép egy EMF alapú állapotgép reprezentáció. Így a sémája nem sokban különbözik az előző fejezetben leírt Theta állapotgéptől. Ez megkönnyíti a transzformáció folyamatát. Az erre a célra írt segédosztályokat mutatom be a következő részekben

4.2. GammaSctoThetaSCConverter

Ez az osztály egy statikus `convert()` függvény meghívásával transzformál

```
public static MutableSc Convert(final StatechartDefinition gammaSC, final String name)
```

Látható, hogy bemenetként egy beolvasott EMF gyökér elemet várunk (ez a Gamma állapotgépet reprezentáló StatechartDefinition osztály) a name pedig a Theta állapotgép neve lesz.

Először beolvassuk a változókat, ezekkel létrehozok egy ExpressionConverter-t amit a következő szekcióban mutatok be. Utána beolvassuk a gyökér régiót ami az addRegiontoThetaSC() rekurzív függvény meghívásával történik utána a tranzíciókat adom az állapotgéphez (frissítve a már korábban létrehozott állapotokat)

```
private static void addRegiontoThetaSC(final Region reg, final MutableRegion r)
```

Ez a függvény rekurzív, abból a szempontból, hogy ha egy régión belül van egy másik régió akkor meghívja önmagát. Minden elemhez van egy ehhez hasonló függvény, aminek a bemenete, egy gammás elem és egy annak megfelelő thetás elem, és a transzformátor a gammás elemben lévő belső elemeket létrehozza és berakja a thetás elembe. Ez így megy addig amíg az adott elemekben vannak újjabb elemek.

4.3. ExpressionConverter

A legnagyobb nehézség az akciókban illetve őrfeltételekben használt kifejezések megfelelő átkonvertálása, ehhez használom a ExpressionConverter segéd osztályt. A Thetában van erre egy segítség a DispachTable. Ennek a segítségével könnyen és átláthatóan össze lehet kötni a különböző gammás Expression-öket a thetás Expr<> osztályokkal

5. fejezet

Állapotgép konfiguráció

5.1. Aktív állapot

Az állapotgép a rendszerünk összes lehetséges állapotát mutatja. Analízis közben azonban fontos hogy azt is tudjuk nézni, hogy mik az aktív állapotok, azaz melyek írják le a rendszerünk pillanatnyi helyzetét.

Az aktív állapotoknak van több szabálya.

- Egy régión belül, pontosan egy aktív állapot lehet (közvetlen gyerekekre nézve)
- Egy aktív állapot összes szülő állapota is aktív
- Minden régióban van aktív állapot

5.2. A megvalósító osztály

A StateConfiguration osztályban valósítom meg. Egy ilyen példány tartalmazza az állapotgép reprezentációt (Sc) továbbá egy listát az épp aktuális állapotokról és -egy később igen hasznosnak bizonyuló segítség- tároljuk, hogy az adott konfiguráció mely tranzíciók elsütésével érhető el a kiindulási állapotból, illetve hogy eközben milyen akciók lettek végrehajtva. Oly módon, hogy minden egyes tranzícióhoz ebben a listában két akció tartozik, egy azokhoz az akciókhoz, amik a tranzíció őrfeltétel ellenőrzés előtt hajtódnak végre, egy pedig azokhoz amik utána.

```
public static StateConfiguration create(final Sc sc)
```

Létrehozható egy Theta állapotgépből.

```
public void init()
```

Aktiválja az állapotgépet, azaz mindenhol a kezdő állapot szerint kijelöli az aktív állapotokat

```
  public Collection<Transition> getFireableTransitions()
```

Visszaad egy listát az összes olyan tranzícióról, ami tüzelhető az adott konfigurációban. Vagyis a tranzíció forrásállapota aktív.

```
public StateConfiguration fire(final Transition tr)
```

Visszaad egy új állapotgép konfigurációt, ami a paraméterben megadott tranzíció elsütése után kialakul. Saját magát nem változtatja!

6. fejezet

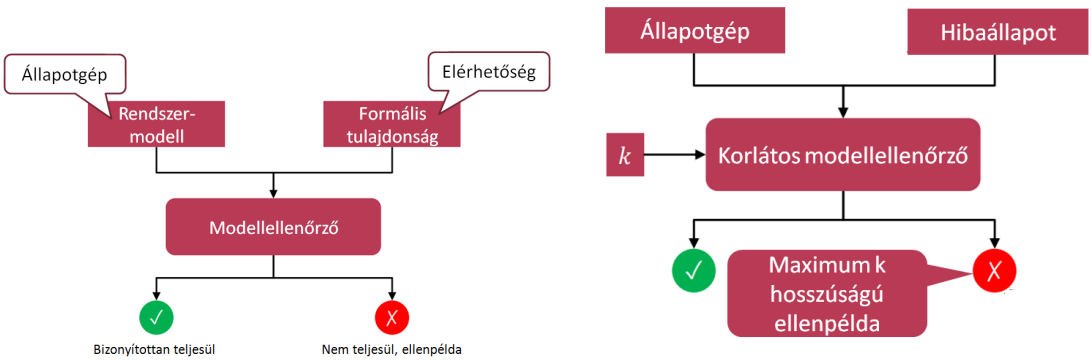
Korlátos modell ellenörző

6.1. Modell ellenőrzés

A bevezetőben már elmondtam miért fontos a modellellenőrzés, most inkább arról beszélek, hogy ez mit jelent az állapotgépek esetén. A 6. 1-es ábrán látható a működés elve. A modellellenőrző tehát egy modell reprezentációt, és egy tulajdonságot kap bemenetként. És megadja, hogy teljesül-e az adott tulajdonság, vagy nem. Ilyenkor még egy példát is ad arra, amikor nem teljesül, amivel könnyen ellenőrizhetjük, hogy igaz volt-e. A másik esetben viszont, ha nem talál ellenpéldát akkor azt bebizonyítani, hogy valóban nincs, csak úgy lehet, ha a modellellenőrző működés közben bizonyítottan megtalál minden ellenpéldát. Ezt sokszor csak bizonyos korlátok mellett lehet biztosítani.

A tulajdonság, általában egy elérhetőségi vizsgálatot jelent[1]. A BMC[2]-nél sincs ez másképp, ilyenkor az ellenpélda egy útvonal az elérni kívánt állapothoz[3]. Mivel azt szeretjük, ha a hibás esetekben van ellenpélda ez az állapot egy hibaállapot, ami a rendszer nem megfelelő működését jelzi.

A BMC a helyes működést, azaz hogy egy állapot biztosan nem érhető el nem tudja bizonyítani, mert végtelen ideig is tarthatna a futás ideje. Ezért inkább megfogalmaz egy korlátot: maximum K lépésből elérhető-e az adott állapot



6.1. ábra. Az általános (balra) és a korlátos modellellenőrző (jobbra) működési elve

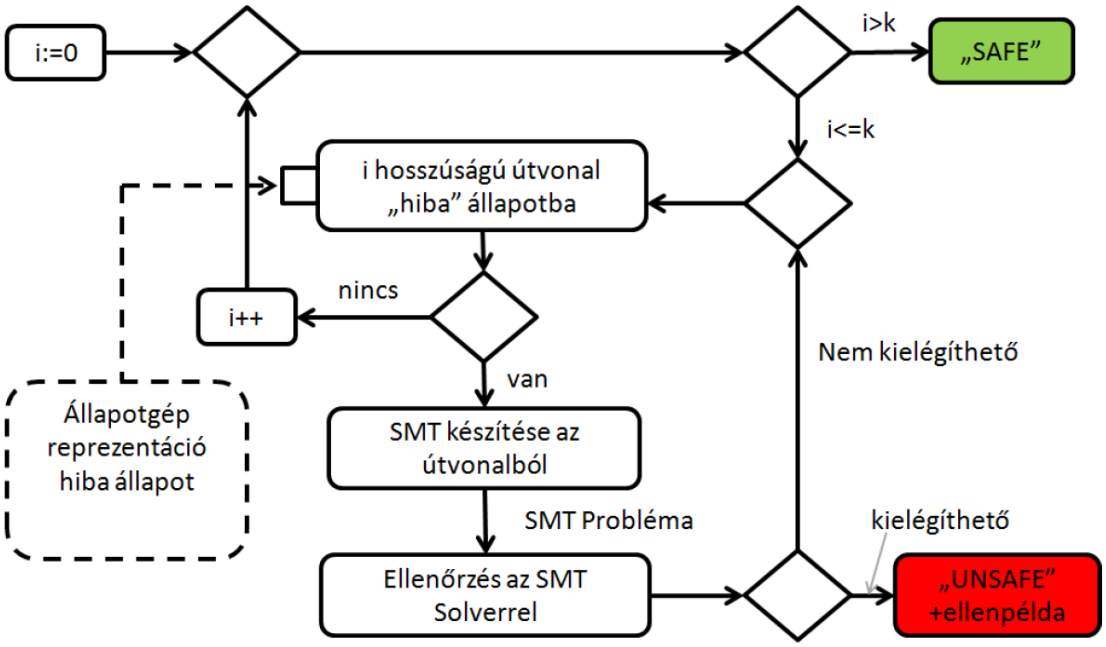
[1] Elérhető-e egy állapot a kezdeti állapotkonfigurációból, úgy, hogy minden tranzíció elsütése érvényes, azaz teljesül az őrfeltétel

[2] Bounded Model Checker, a korlátos modell ellenőrző rövid neve

[3] Mely tranzíciókat kell elsütni, milyen sorrendben

6.2. A BMC menete

A korlátos modellellenőrző működése a 6.2-es ábrán látható. A keresés egy szélességi gráf bejárás az állapotkonfigurációkon ahol a k korlát mélységig megyünk. A hiba állapot olyan állapotgép konfigurációt jelent, ahol a hibaállapot aktív.



6.2. ábra. A korlátos modellellenőrző folyamat modellje

6.3. Az útvonal

A `Path` osztályt használom ennek a leírására. Tárolom az összes állapotgép konfigurációt az kiinduló állástól kezdve, és az útvonal során elsütött tranzíciókat.

```
public boolean checkpath()
```

Azt ellenőrzi, hogy az útvonal helyes-e azaz minden egyes konfigurációból valóban jó konfigurációba léptünk és az elsütött tranzició elsüthető volt (a forrásállapota aktív volt)

```
public StateConfiguration getResultConfiguration()
```

Az útvonal végigfutása utáni állapotkonfigurációt adja vissza

6.4. SMT Solver

Az SMT solver egy olyan eszköz, ami több aritmetikai kifejezést kapva, megtudja mondani, hogy ezek teljesülhetnek-e egyszerre. Ha igen akkor ad egy példa értéket az összes benne szereplő változóra, hogy azok kielégítsék mindegyik aritmetikai kifejezést.

Ezek most nekünk azért kellenek, mert az útvonalakon, a tranzíciókon lévő őrfeltételek is egy ilyen aritmetikai kifejezéshalmazt képeznek.

A Theta projektben van egy interfész a Z3 solveréhez, ezt használtam.

Az `SMTBuilder` segéd osztályban csinálok az útvonalak segítségével SMT problémát, amit át tudok adni a solvernek.

```
public SMTBuilder(final Path p) {
```

A segéd osztály egyszerűen létrehozható az útvonal alapján.

```
public Collection<Expr<BoolType>> unfold()
```

Ezzel a függvénnyel hozzuk létre azt a listát, amit közvetlen átadhatunk a solvernek.

Elsőre egyszerűnek tűnik, hiszen a tranzíciók, már ezt a formátumot használják az őrfeltételek tárolásához, de figyelemmel kell kísérni azt is, hogy az akciók során változhatnak a változó értékei -akár többször is-. Ennek a problémának a megoldásához van egy eszköz a Theta projekten belül (`PathUtils`), ami pont ennek a "változó frissítésnek" ad implementációt. Itt válik hasznossá, hogy az állapotgép konfigurációkban tároljuk az összes akciót, ami a kezdeti állapothoz képest történt.

6.5. BoundedChecker

Maga az ellenőrző osztály a `BoundedChecker`.

```
public BoundedChecker(final Sc chart, final int K, final State errorState)
```

Bemenetként egy állapotgépet várunk, amin végezzük az ellenőrzést. Bekérünk egy K korlátot, ami megadja a maximum lépés számot, és persze maga a hiba állapotot is kell.

```
  public SafetyStatus check()
```

Ezzel a függvénnyel lehet futtatni az ellenőrzést, egy `SafetyStatus` al tér vissza amiből kiolvasható, hogy biztonságos-e, ha nem, akkor az ellenpéldát is ki lehet olvasni. Meghívja az `algorithm` függvényt

```
public SafetyStatus algorithm(final Collection<Path> paths, final int deep)
```

Egy rekurzív függvény ami, minden egyes lépésben növeli a mélységet (deep) és, minden útvonalból, (ami nem bukott már meg) létrehozza az összes lehetséges új útvonalakat. Ehhez elég az útvonal utolsó konfigurációján minden elsüthető tranzíciót elsütni.

Így biztosítjuk azt, hogy az összes lehetséges útvonalat megnézzük, aminek a hossza az inicializáláskor megadott maximumnál nem nagyobb. Tehát, ha nem találunk ellenpéldát akkor biztosan állíthatjuk, hogy nincs is.

A bizonyítás azért nem ennyire egyszerű, mert ugyan kipróbálunk minden lehetőséget, előfordulhat, hogy egy jó ellenpéldát nem fogadunk el, ez persze úgy lehet, ha például, rosszul építjük meg az SMT problémát és azt kapjuk, jogtalanul, hogy az útvonal nem teljesíthető.

További információ a verifikációs résznél

7. fejezet

Verifikáció

7.1. A Theta állapotgép tesztelése

Az egyes elemeknek egyenlőre nem készültek Unit tesztjeik.

A szerializációt több különböző állapotgépre is lefuttattam, kipróbálva minden elemet. Erre egy példa van a 3.4 és a 3.5-ös ábrán. Erre egy teszt osztály a `ParserTest`, ami beolvas egy sorosított állapotgépet, majd kiírja a standard output-ra

7.2. A transzformáció tesztelése

Több állapotgépet is létrehoztam Yakindu-ban, ezeket a Gamma Framework segítségével Gamma állapotgéppé alakítottam. Ezekkel az állapotgép modellekkel futtattam a `ConverterTest` tesztet, ami Theta állapotgéppé transzformálja a kapott Gamma modellt. A 3.4-es ábrán látható állapotgép is így lett Theta állapotgéppé konvertálva és utána sorosítva.

7.3. Az állapotgép konfiguráció tesztelése

A `StateConfigurationTester` osztállyal teszteltem az állapotgép konfigurációt. A 7.1-es ábrán átható az állapotgép, amin futtattam a tesztet. A következő tranzíciókat tüzeljük el:

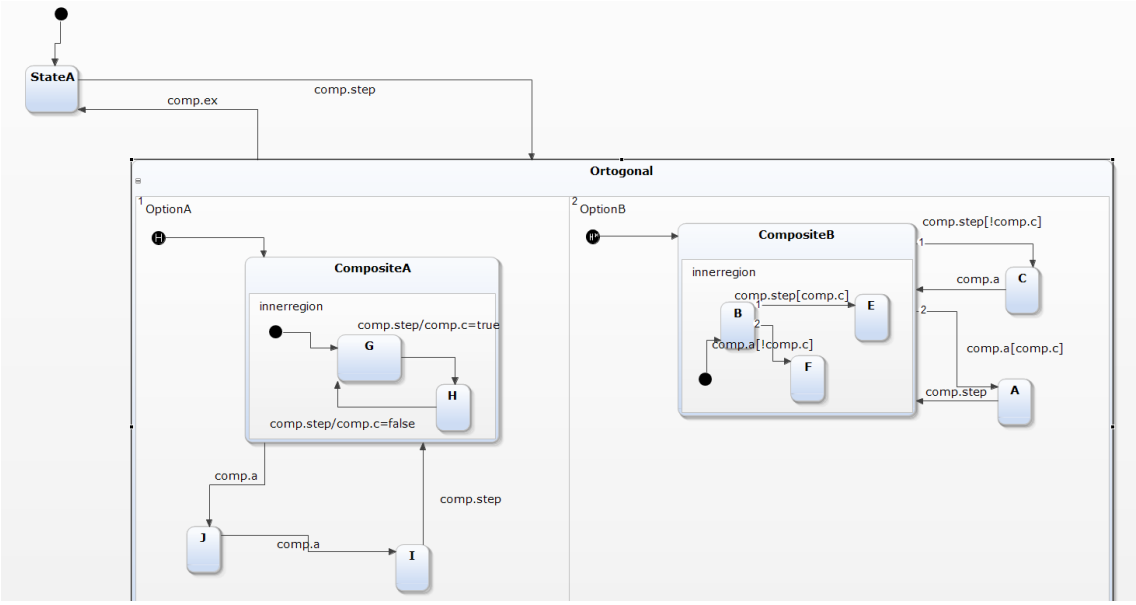
1. StateA –> Ortogonal
2. B –> E
3. G –> H
4. Ortogonal –> StateA
5. StateA –> Ortogonal

Ezek után a következő állapotoknak kell aktívnak lennie:

- CompositeA,
- CompositeB,
- G mert a history megjegyezte, hogy kilépéskor CompositeA volt aktív, viszont az azon belüli állapotot már nem jegyezte meg tehát hiába volt H aktív, most már újra G-az,

- E, mert a deep history azt is megjegyezte, hogy CompositeB-n belül mi volt az aktív állapot
- Ortogonal

A teszt sikeresen lefutott.



7.1. ábra. Az állapotgép, amin a konfigurációs tesztet végeztem

7.4. A BMC tesztelése

A `BoundedCheckerTest` osztállyal lehet beolvasott Theta állapotgépeken tesztelni. Például a 3.4-es állapotgépen keresve az F hiba állapotot. A következő eredményt kapjuk:

```
<terminated> BoundedCheckerTest (1) [JUnit] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe (2018. jún. 26. 19:47:38)
(val (_variable:2 0)
     (_variable:1 1)
     (_variable:3 1))
UNSAFE
Fired Events: fstr.0 StateCTimeout5
```

7.2. ábra. A BoundedCheckerTest futási eredménye

Itt láthatjuk, hogy a solver által vissza adott példában három különböző változó szerepel, pedig csak egy változó van az állapotgépben. A másik kettő az új értékadás után jött létre. Az `UNSAFE` azt jelenti, az F állapot elérhető. Utána felsorolásként ott vannak azok az események, amik kiváltják a szükséges tranzíciókat. Viszont az időzítési feltételeket egyenlőre nem vizsgáljuk, ez a jövőbeni tervek közé tartozik.