FELADATKIÍRÁS

A feladatkiírást a tanszéki adminisztrációban lehet átvenni, és a leadott munkába eredeti, tanszéki pecséttel ellátott és a tanszékvezető által aláírt lapot kell belefűzni (ezen oldal *helyett*, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell beleszerkeszteni ezt a feladatkiírást.

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

Állapottérkép verifikációs MagicDraw plugin továbbfejlesztése

Diplomaterv

Készítette
Gáti László Dávid

Konzulens
Tóthné Farkas Rebeka

2020. december 10.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2020. december 10.

Gáti László Dávid
hallgató

Kivonat

TODO:

Abstract

TODO:

1. fejezet

Bevezetés

A IT technológiák térnyerésével egyre több és komplexebb rendszer készül, melyeknek sokszor valós időben kell működni. Mivel ilyen rendszerek jellemzően valamilyen biztonság kritikus környezetben működnek, elengedhetetlené válik ezek gondos megtervezése és átfogó vizsgálata különösen a helyes működés tekintetében.

A tervezés és ellenőrzés költséges, időigényes folyamat, ezért szükség van olyan eszközökre amelyek megkönnyítik vagy akár teljesen automatizálnak egyes folyamatokat. A tervezés során általában valamilyen modellvezért technikát alkalmaznak, melynek középpontjában a modellek állnak. A tervezés során elkészített tervek nagyon sok értékes információt tartalmaznak, melyeket újra fel tudunk használni és származtatni ezekből kódot, dokumentációt, vagy akár más modelleket, ezáltal időt és erőforrásokat megtakarítva. Ráadásul mivel ezeket automatikusan gépek végzik, minimalizálódnak az emberi hibák például a programkódban, ahhoz képest mintha ezeket kézzel végeznénk el.

Terveinket már érdemes a tervezés korai fázisaiban ellenőrizni, hiszen az itt vétett hibák akár kritikusak lehetnek a későbbiekben. Az ellenőrzésekhez szintén fel tudjuk használni a modelljeinket és szimulálni tudjuk a rendszert, vagy képesek vagyunk magát a modellt is vizsgálni formális módszerek segítségével.

A MagicDraw egy mára de-facto ipari standarddá vált szoftver, és rendszer architektúra modellező eszköz ami fejlett grafikus interfészt nyújt a felhasználók számára. Modelleket elsősorban egy általános célú modellezési nyelvvel UML-el lehet készíteni, azonban UML profilok segítségével akár saját szakterület specifikus nyelvek használatára is lehetőségünk nyílik. Ilyen formában a MagicDraw lehetővé teszi modellek létrahozását SysML nyelven is amihez a profilt maga biztosítja. A dolgozat a továbbiakban SysML modellekkel foglalkozik.

Ugyan a MagicDraw számos fontos és hasznos funkcióval rendelkezik, még mindig megvan az igény újabbakra főleg Verifikáció/Validáció tekintetében. A MagicDrawToGamma nevű MagicDrawhoz készült plugin SysML állapottérképek formális verifikálásához nyújt megoldást, melyhez a Gamma Statechart Composition Frameworköt és az UPPAAL nevű eszközöket használja fel.

Az eszköz ugyan *Proof of Concept* jelleggel már képes a verifikációt elvégezni, azonban, hogy akár szélesebb körben is használható eszközzé válhasson még sok tekintetben fejlesztésre szorul. Jelen dolgozat célja bemutatni azokat a fejlesztéseket amiket a mesterképzés során végeztem az eszközön és visszatekintve kiértékelni azokat a mérnöki megoldásokat melyeket a fejlesztés során hoztam.

A dolgozat felépítése a következő: a második fejezetben ismertetem azokat az ismereteket amelyek a dolgozat során felvetülő problémák illetve az ezekre adott megoldások megértéséhez szükségesek. A harmadik fejezetben ismertetem a projekt céljait és az ezekhez vezető utat, alkalmazott megoldásokat, illetve bemutatom a beépülő modul működését.

A negyedik fejezetben pedig értékelem az alkalmazott megoldások hatékonyságát és az elkészített plugint.

2. fejezet

Háttérismeretek

Ebben a fejezetben bemutatom azokat az ismereteket, technológiákat amelyek segítenek megérteni azokat a megoldásokat melyeket a dolgozat elkészítése során alkalmaztam.

2.1. Kapocsolódó munkák

A fejlesztés megkezdése előtt utána néztem, hogy milyen eszközök állak rendelkezésre, amelyek hasonló problémákat oldanak meg. Specifikusan a MagicDrawhoz nem találtam olyat, ami még most is releváns lenne (korábbi munkáim során találkoztam hasonló célú eszközzel, de úgy tűnik annak a fejlesztése nem folytatódott). Általánosabban az állapottérkép és formális verifikációval kapcsolatban a legszorosabban kapcsolódó munka a Gamma Statechart Compoition Framework amelyet későbbi fejezetekben mutatok be. A megoldások amiket alkalmaztam főleg ennek az eszköznek a segítségével történtek.

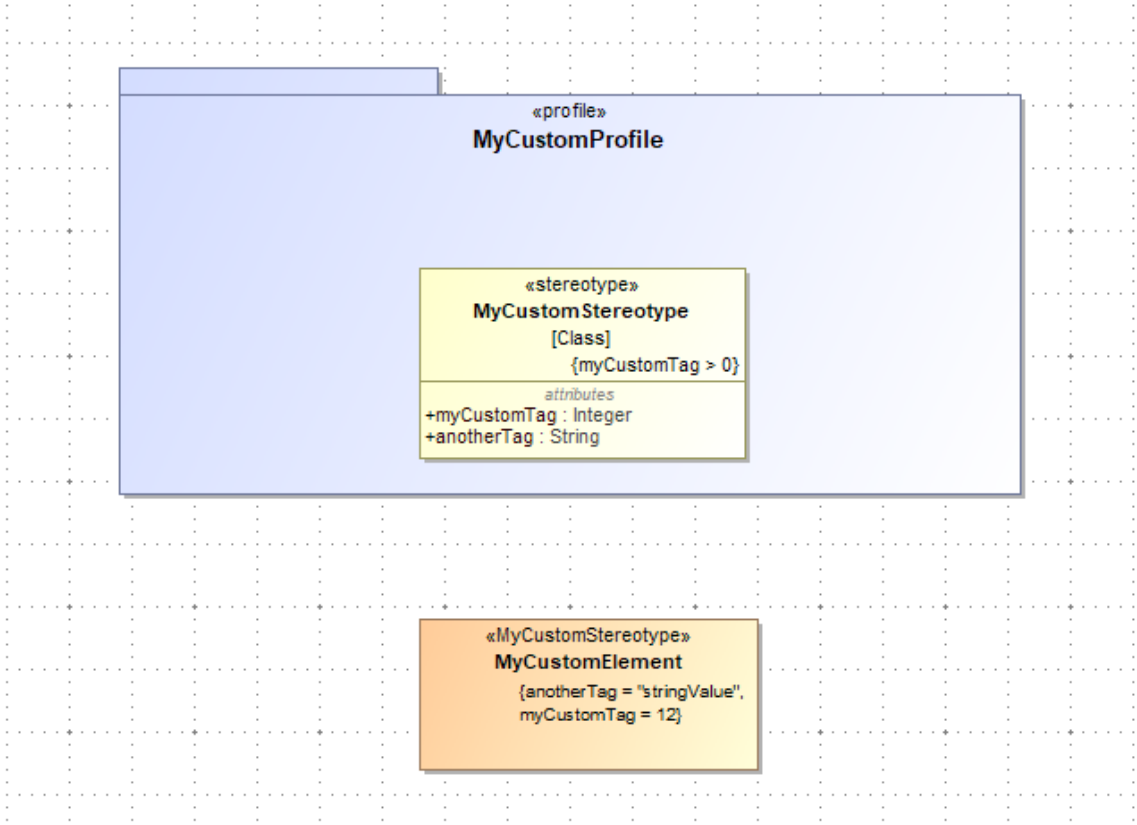
2.2. SysML

A System Modeling Language (SysML) egy általános célú architektúra modellező nyelv, melyet elsősorban rendszermérnökök használnak. A nyelvvel különböző komplex rendszereket tudunk leírni többféle megközelítésből mint magas szintű funkcionális modellek mind pedig az alacsonyabb akár fizikai modellekig.

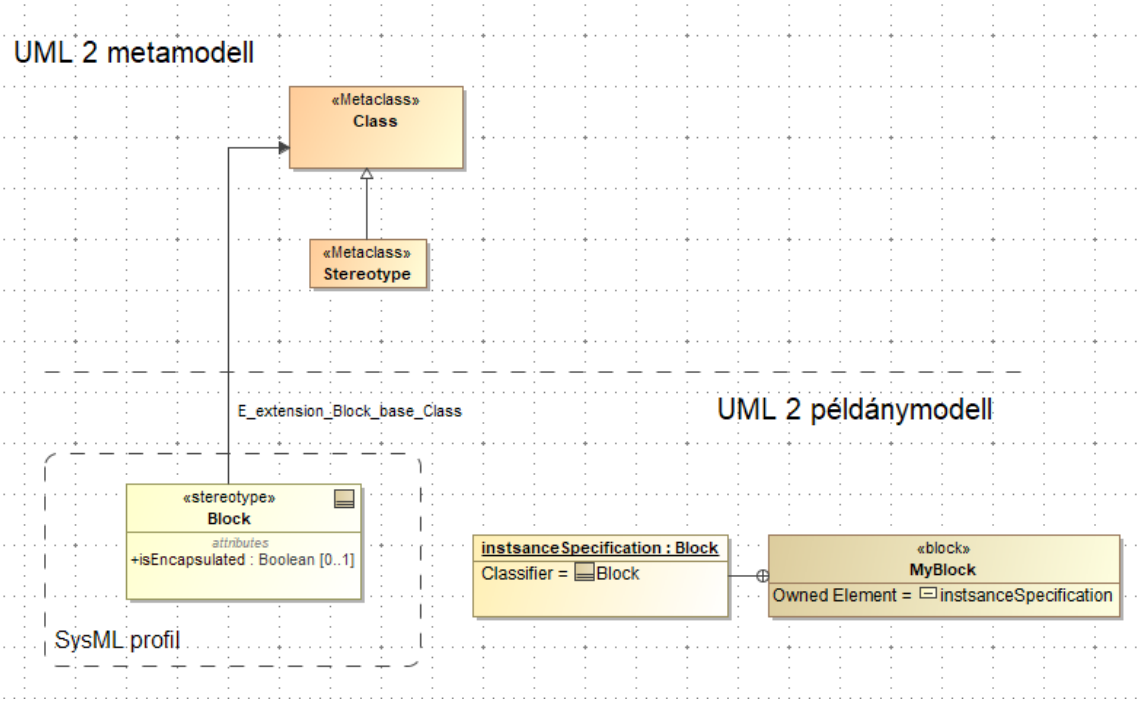
A SysML-t mint ahogyan az UML-t is az Object Management Group (OMG) fejleszti, sőt az UML 2-nek egy dialógusa a nyelv, úgynevezett UML profil segítségével van definiálva. Az UML profil a nyelv specializálásának egy módja sztereotípiák (Stereotype), megkötések (Constraint) és címkézett értékek (Tagged Value) segítségével (2.1. ábra).

Egy nyelv lehetséges elemeit és kapcsolatait leíró struktúrát metamodellnek hívjuk, a tényleges elemekből és ezek kapcsolatából álló modellt pedig példánymodellnek. SysML esetében az UML profil az amit metamodellnek tudunk tekinteni, igaz a metaszintek eléggé összemosódnak és nem elegendő csak a profilt ismerni, hanem az UML metamodellt is ismerni kell. Például egy SysML Block valójában egy sztereotipizált UML Class. Ez azt jelenti, hogy van egy UML példány specifikáció (Instance Specification) aminek az osztálya (Classifier) egy Block sztereotípia és ez hozzá van rendelve az UML-s osztály példányunkhoz a modellünkben. Az UML-s osztály példány a sztereotípia és a példány specifikáció tehát hagyományos értelemben azonos metaszinten helyezkednek el (2.2. ábra).

SysML segítségével sokféle modellt lehet készíteni a teljesség igénye nélkül: Követelmény modellek, viselkedés modellek, struktúra modellek, allokáció modellek, mindezek közül azonban kettőre koncentrálok a dolgozat elkészítése közben, mégpedig a viselkedés modellek egy részére az állapotgépekre és a funkcionális architektúrára amely funkcionális architektúra egy de-komponált állapotgépet ír le.



2.1. ábra. UML nyelv specializálása sztereotípiákkal



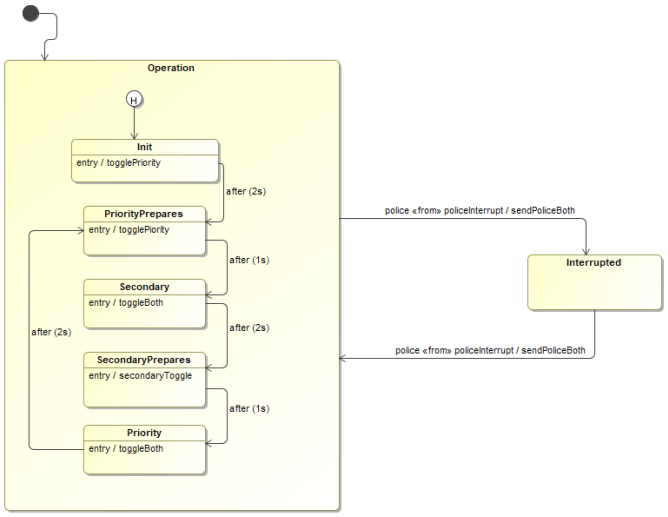
2.2. ábra. Meta szintek és Block sztereotípia alkalmazása

2.2.1. Állapotgépek, állapottérképek

Állapotgépek (State Machine) alatt olyan modelleket értünk amik a rendszert véges számú jól elkülönülő állapotát írják le, illetve az ezek közötti átmeneteket. Az állapotgépeket SysML-ben állapottérkép diagramokon (Statechart Diagram) tudunk definiálni. Szintaktikáját tekintve oválisok amik állapotokat jelentenek illetve az ezeket összekötő nyilak, melyek az állapotok közötti lehetséges váltások. A nyilak általában címkézettek. A címke három részből állhat: esemény (trigger), őrfeltétel (guard), akció (action).

```
esemény [örfeltétel] / akció
```

Állapotváltás egy esemény bekövetkezésekor lehetséges. Amennyiben van őrfeltétel annak igaznak kell lennie. Az állapotváltás bekövetkezését szokás tüzelésnek is nevezni. Ha egy állapotátmenet tüzel és van akció az átmenethez rendelve ez végrehajtódik. Akció nem csak állapotátmenetek, hanem állapotok be és kilépésénél is végrehajthatók, illetve létezik olyan akció is ami folyamatosan fut amíg a rendszer adott állapotban van.



2.3. ábra. Állapottérkép SysML-ben

Az állapotoknak van egy típusa melyet pszeudo állapotoknak nevezünk. Ezek nem tényleges állapotai a rendszernek olyan értelemben, hogy a rendszer soha nem tartózkodhat ezekben mindig azonnal át kell lépniük belőle egy tényleges állapotba, viszont fontos szemantikai jelentéssel bírnak. A kezdő állapot mely fekete körként jelenik meg: ●, egy régió belépési pontjai. Az állapotok mindig régiókban helyezkednek el, ezek tulajdonképpen állapotoknak egy olyan halmaza, amely halmazon belül mindig csak egy állapot lehet aktív, régiókból viszont lehet több is párhuzamosan. Így egy rendszernek lehet egyszerre több aktív állapota is régiónként viszont csak egy (a fork-joinra a dolgozat nem tér ki részletesen).

Pszeudo állapotok a *History Statek* is. Ezek segítségével meg lehet jegyezni, hogy egy összetett állapot mely belső állapota volt aktív, így az állapotba visszalépve nem a belső régió kezdő állapota hanem az utolsó aktív állapot lesz újra aktív. *History State*eknek két változata van *Deep* és *Shallow*. Előbbi egymásba ágyazott esetén minden régióba az utolsó aktív állapotba lép át, míg utóbbi csak a saját régiójának utolsó aktív állapotába. Jelölésük kör, benne "H"-val vagy "H*": Ⓗ. Létezik még választó pszeudo állapot (*Choice state*) ez szintaktikailag egy rombuszként jelenik meg ◇. Ezzel lehet elágazásokat megvalósítani oly módon, hogy a kimenő állapot átmeneteken lévő őrfeltételek igazsága jelöli ki azt az élt

ami tüzelhet (több igazra értékelt átmenet esetén nem determinisztikus módon az egyik tüzelhet). Választó állapotok esetén lehet egy őrfeltételben az *else* kulcsszót használni, ilyenkor, ha nem volt más átmeneten igazra értékelődő őrfeltétel akkor ezen az átmeneten keresztül folytatódik a futtatás. A választó állapotból kimenő éleken nem lehet *trigger*.

Az előbbieken kívül létezik még fork-join amivel párhuzamos futtatást lehet modellezni, csatlakozás (*junction*), amivel élek vonhatók össze. Végső állapot (*Final State*) ami a futtatás végét jeleni.

2.2.2. A rendszer architektúrája

Egy rendszer felépítését többféle szempont szerint is lehet modellezni. Ez egyik ilyen, hogy a rendszerünk milyen fizikai részekből áll és ezek között mi a kapcsolat, de fel lehet funkciók szerint is bontani. Utóbbit a rendszer funkcionális architektúrájának nevezzük.

Egy rendszer funkcionális architektúráján olyan modelleket értünk amelyek ennek a logikai felépítését írják le azaz, hogy milyen funkciói vannak a rendszernek, ezek hogyan kapcsolódnak egymáshoz. Jelen esetben ezeket a modelleket a rendszer állapotgépének a dekomponálására használjuk. Tehát van egy összetett rendszerünk melyeket funkcionális egységekre tudunk bontani úgy, hogy minden funkcionális egység állapot alapú viselkedések egy összessége. A komponensek képesek egymással kommunikálni és egymás viselkedését ezáltal befolyásolni.

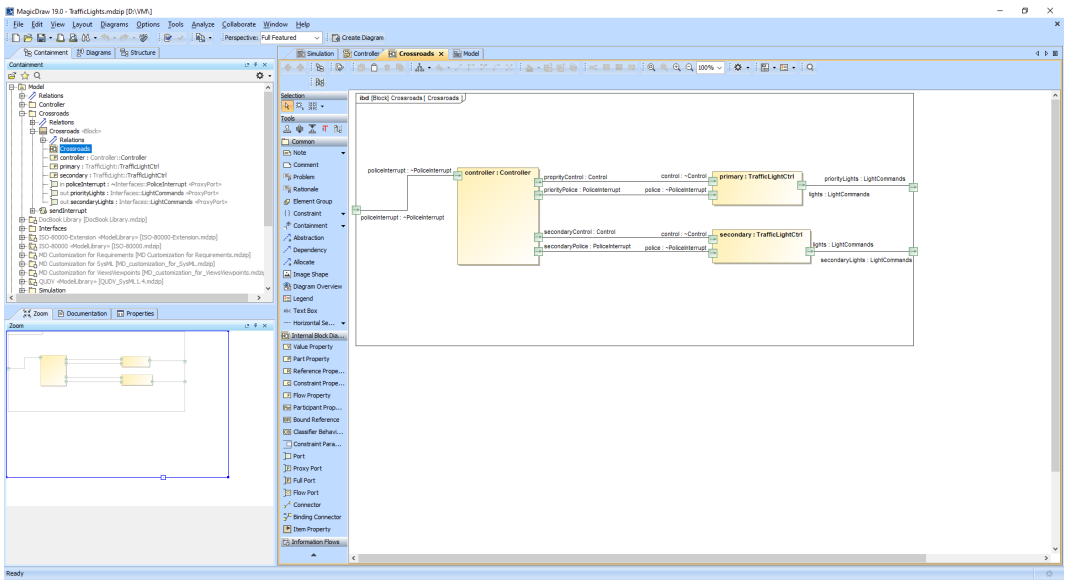
A rendszert két szinten tudjuk modellezni: komponens definíciók és ezek kompozíciója, illetve ezeknek a kapcsolatai és belső szerkezete szerint. Előbbiek blokkok formájában jelennek meg és SysMLben Blokk definíciós (Block Definition) vagy BDD diagramokon jelennek meg A tartalmazásokat pedig *Composition* élekkel tudjuk modellezni. Azt, hogy a blokkon belüli részek (*Part*) hogyan kapcsolódnak egymáshoz Internal Block Diagramok (IBD) tartalmazzák. A részek közötti kapcsolatot konnektorok írják le. Ezek jellemzően portokon keresztül kötik össze a részeket. A portok tekinthetők egy adott blokk interfészének is.

A dolgozat elkészítésénél kétféle blokkot különböztetek meg. Azokat melyek állapotgépeket definiálnak és nem bonthatók részekre ezeket állapot blokkoknak nevezem. Illetve olyan blokkokat melyen részekre bonthatók, de nem definiálhatnak saját állapotgépet, ezeket pedig kompozit blokkoknak.

2.3. MagicDraw

A MagicDraw egy modellező eszköz melyet a NoMagic fejlesztett elsősorban UML modellek készítésére, bár talán inkább a SysML modellezés vált meghatározóvá. Ahhoz, hogy SyML-ben is tudjunk modelleket készíteni egy beépülő modulra van szükségünk (kivéve ha Cameo System Modellert használunk ami a SysML pluginnal integrálva szállít). A MagicDraw az iparban egy egyre inkább elterjedt eszköz. Szigorúan követi az UML és SysML szabványt. Az modellezőeszköz gazdag funkcionalitással rendelkezik mint fejlett felhasználói interfész (2.4. ábra), validációs motor - ami képes akár futásidőben a felhasználók által készített szabályok futtatására is, sablon alapú kódgenerátor és még sok egyéb. Ezen felül a funkcionalitás beépülő modulok segítségével bővíthető.

MagicDrawhoz lehetőségünk van saját plugin fejlesztésére is, amik lehetővé teszik számunkra, új funkcionalitás integrálását az eszközben illetve modellek manipulálását is. A feladatot is egy ilyen plugin fejlesztésével oldottam meg.



2.4. ábra. MagicDraw felhasználói felülete

2.4. Modelltranszformációk

Az ipari standardok mellett sok speciális modellezési nyelv is létezik, amelyek megannyi céllal és az ezeket használó eszközökkel jöttek létre. Ezek között vannak magas absztrakciójú általánosabb nyelvek és alacsony szintűek is amik egy része olyan formalizmusokra épül amik felett bizonyos problémákra matematikai eszközökkel tudunk megoldást keresni.

A modellvezéreltség egyik alapötlete, hogy különböző modellekből származtatni tudunk más modelleket feltéve, hogy elegendő információ áll rendelkezésünkre a konverzió elvégzéséhez. Ezt a fajta származtatást például modell transzformációk segítségével tudjuk végrehajtani. A modelltranszformációk használata lehetővé teszi, hogy ne csak azokat a technológiákat használjuk modellünk feldolgozására melyek speciálisan az adott modellezési nyelvhez készültek hanem a modelleket megpróbáljuk átalakítani - lehetőleg a szemantikai tartalom megőrzésével és automatizáltan - egy olyan modellezési nyelvre amelyhez már létezik az általunk használni kívánt funkcionalitást támogató technológia.

Ezen felül dokumentumokat is tudunk származtatni a modellekből a kódgenerálás tulajdonképpen ennek egy speciális esete. Az sem ritka, hogy a származtatások több lépésen és modellen keresztül történnek. A kód generálásnál maradva a generált kód is egy modell amit általában egy fordítónak valamilyen futtatható állománnyá kell alakítani.

2.5. Felhasznált technológiák

Ebben az alfejezetben röviden kitérek azokra a tényleges technológiákra, amelyekre a feladat elvégzése során használtam, mint például a VIATRA, a kiindulási alapot szolgáltató MagicDraw plugin és a Gamma.

2.5.1. VIATRA[1]

A VIATRA egy keretrendszer mely lehetővé teszi eseményvezért modell transzformációk fejlesztését. Ehhez egy inkrementális modell lekérdezéseket leíró nyelvre a VIATRA Query

[1] https://www.eclipse.org/viatra/

Language-re (2.5. ábra) támaszkodik. A létrehozott transzformációkat egy reaktív Query Engine futtatja és tartja karban.

```
pattern StateMachines(stateMachine: StateMachine, name: java String){
    StateMachine.name(stateMachine, name);
}

pattern ParametersInStateMachine(stateMachine: StateMachine, parameter: Parameter){
    StateMachine.ownedParameter(stateMachine, parameter);
}

pattern RegionsInRegion(container: Region, region: Region){
    Region.subvertex(container, vertex);
    State.region(vertex, region);
}
```

2.5. ábra. Példa: VIATRA Query Language

A VIATRA használata jelentősen megkönnyíti a modellekkel való munkát, bár én elsősorban a modell lekérdezés részére támaszkodtam a dolgozat elkészítése alatt. MagicDrawban egy pluginon keresztül van lehetőség VIATRA használatára melyet V4MD-nek hívnak. A plugin egy Query Engine-t biztosít számunkra per projekt. Ezen keresztül tudunk lekérdezéseket és transzformációkat regisztrálni. A V4MD leveszi a vállunkról a terhet az engine életciklus menedzsmentjét illetően. A VIATRA hátránya lehet azonban, hogy nagyon nagy modelleket komplex lekérdezések esetén a memóriaigénye elég jelentősre duzzadhat a háttérben, cserébe viszont az inkrementális működésnek köszönhetően a lekérdezések már csak a VIATRA által karbantartott táblázatokból való kiolvasás ezért költségül elhanyagolható. Éppen ezért olyan esetekben érdemes használni, amelyek kellően gyakran futnak és gyorsan van szükség az eredmény meghatározására mint például a futás idejű validáció.

2.6. MagicDraw állapottérkép verifikációs plugin

Ebben az alfejezetben bemutatom azt a SysML állapottrékép verifikációs eszközt, mely korábbi egyetemi munkám eredményeként képes volt MagicDraw modellek ellenőrzésére. A funkció megvalósításához a *Gamma Statechart Composition framework* nevű keretrendszert használtam fel.

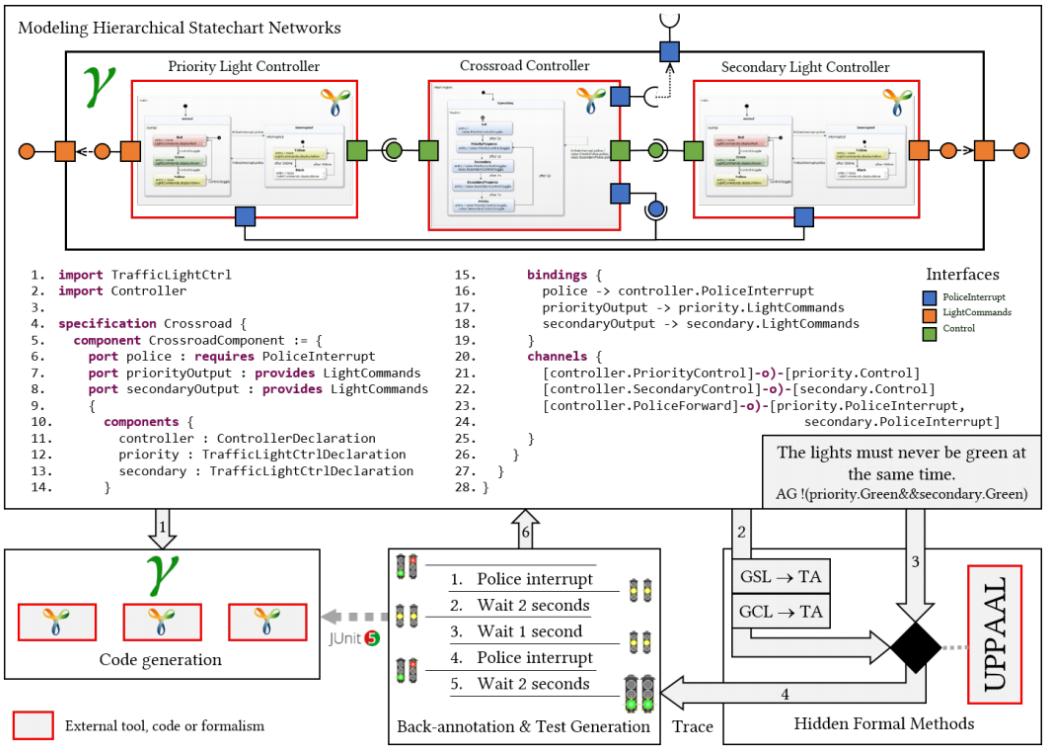
2.6.1. Gamma Statechart Composition Framework

A Gamma Statechart Composition Framework (továbbiakban Gamma) egy állapottérkép modellező keretrendszer Eclipse felett ami integrálja a YAKINDU-t, de ezen felül saját nyelveket definiál nem csak állapottérkép leíráshoz, hanem ezek kompozíciójának modellezéséhez is. Gammával a felhasználónak lehetősége van modellek formális verifikációját elvégezni, az eredményből pedig teszt kódot generálni, futtatható kód mellett.

A verifikáció eredményeként előálló időzítéseket és lépéseket visszavezeti a modellbe. Ezt Back-annotationnek nevezzük. Ezekhez szintén definiált egy nyelv, mely lehetővé teszi ezek dokumentálását.

A dolgozat elkészítése során nagy mértékben támaszkodtam a Gamma nyújtotta lehetőségekre. Ezt a Gamma mint egy köztes nyelv használatával érem el úgy, hogy modell transzformációk segítségével megvalósítok SysML-Gamma leképzést. A Gamma a formális verifikáció elvégzéséhez az UPPAAL nevű eszközt használja. Ez időzített automatákon tudja elvégezni a formális verifikációt (2.6. ábra). Ezeket a Gamma szintén modell

transzformációk segítségével állítja elő, összekapcsolva a mérnöki modelleket matematikai modellekkel.



2.6. ábra. A Gamma Statechart Composition Framework[2]

A Gamma nagy erőssége, hogy állapotgép kompozíciókat is lehet modellezni benne. A komponenseket végrehajtás szempontjából háromféle szemantika van megkülönböztetve.

Szinkron komponensek melyek egy lépésen belül fogadnak eseményeket ezekre lépnek és elküldik az eseményeket, ezek viszont a következő iterációban fognak fogadásra kerülni a hozzájuk kapcsolódó komponensekben.

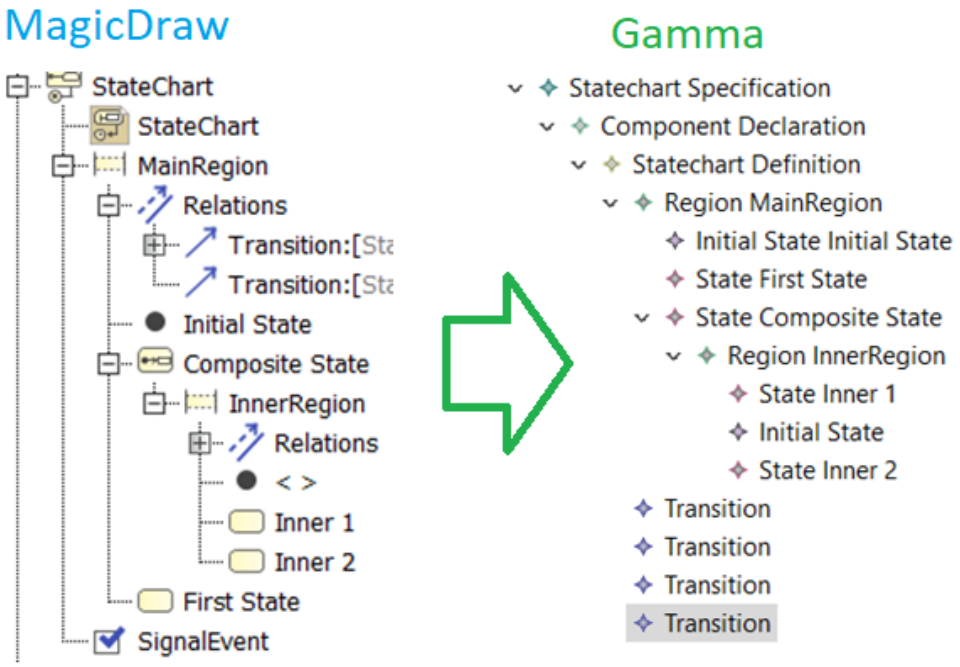
Kaszkád komponensek (Cascade Component) ezek hasonlóan működnek, mint a szinkron komponensek, viszont ebben az esetben a kiküldött üzenetek ugyan abban az iterációban kerülnek fogadásra. Ebből kifolyólag az üzenet áramlásnak aciklikusnak kell lennie, hiszen a feldolgozást végtelen ciklusba kerülne.

Aszinkron komponensek Az események feldolgozása aszinkron módon, üzenet sorok kiolvasásával történik. Állapottérkép definíciók alapból szinkron szemantikával bírnak, így ahhoz hogy asszinkron szemantikával ruházzuk fel őket be kell csomagolni őket egy *Wrapperbe*. Ez definiálja számukra az üzenet sort is.

2.6.2. MagicDraw - Gamma transzformáció

A MagicDraw beépülő modul fő célja SysML állapottérképek formális verifikációja. A Gamma képes állapotgépek ellenőrzésére ezek viszont a Gamma saját nyelvén kellenek, hogy legyenek definiálva. Modell transzformációk segítségével azonban lehetőségünk nyílik Gamma modellek származtatására SysML modellekből és ezáltal ellenőrizni őket.

Ez a származtatás vagy modell transzformáció képezi az ötlet alapját (2.7. ábra). A kihívás pedig az két nyelv közötti szemantikai különbségek feloldása illetve az elemek megfelelő egymáshoz rendelésének megtervezése és végrehajtása.



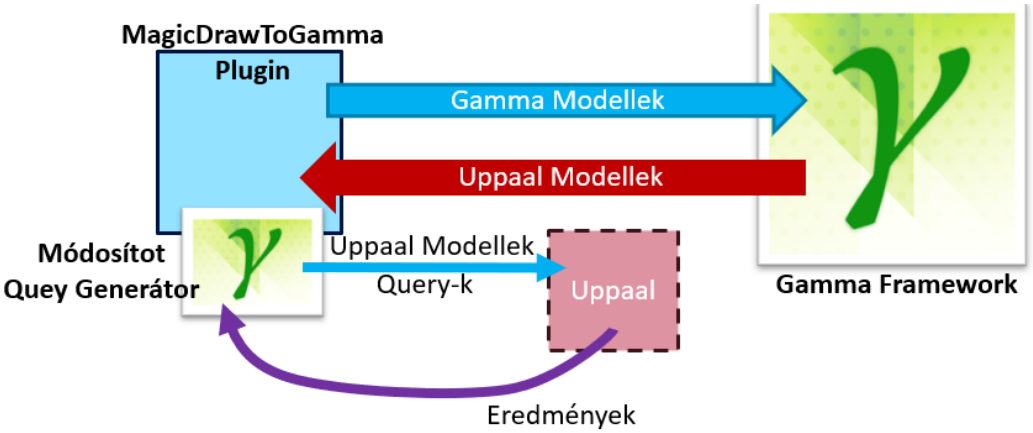
2.7. ábra. Példa: egy modell transzformációra MagicDrawról Gammára

2.6.3. Verifikáció menete

A verifikáció elvégzése a MagicDraw beépülő modul szemszögéből négy lépésből áll:

- MagicDraw modellek Gammává transzformálása
- Gamma modellek UPPAALmodellé transzformálása (ezt a Gamma keretrendszer végzi el)
- tulajdonságok ellenőrzése UPPAAL segítségével
- Eredmény megjelenítése

Ezt a folyamatot 2.8. ábra szemlélteti. A Gamma által elvégzett lépéseket a zöld ”Gammák” jelölik a modellben. Az ábrán a verifikációt a ”Módosított Query Generátor” kezdeményezi, mely szintén a ”Gamma” jelölést kapta. Ennek oka, hogy a beépülő modul ezen verziójában az ellenőrizendő tulajdonságok megadása még nem volt kiforrott ezért egy a Gamma keretrendszerből átemelt megoldás segítségével lehetett ezeket megadni és a verifikációt kezdeményezni.



2.8. ábra. Verifikáció menete a pluginban

2.6.4. Őrfeltételek, akciók

SysML nem definiál akció nyelvet őrfeltételek és akciók végrehajtásához. Míg akciókat tudunk modellezni bármilyen viselkedést leíró modell segítségével, addig az őrfeltételeket jellemzően valamilyen szöveges nyelvtannal szokás megadni.

Ahhoz, hogy a modell ellenőrzést végre lehessen hajtani ezeknek a Gamma számára érthető kifejezéseknek kell lenniük. A pluginnak ez a verziójában ezért minden őrfeltételt és akciót a Gamma által biztosított nyelvtan segítségével kellett megírni. Ez abból a szempontból nem szerencsés, hogy ez nem egy olyan elterjedt nyelv, mint például a javascript amit a MagicDraw felkínál, továbbá nem áll még rendelkezésre olyan interpreter a MagicDraw-ban ami lehetővé tenné ezek futtatását ami elengedhetetlen többek között a modellek szimulációjához.

2.7. Xtext

Az Xtext egy keretrendszer amivel programozási nyelveket és egyéb szakterület specifikus nyelveket lehet készíteni, melyek szöveges formában manifesztálódnak. Ezeket valamilyen már meglévő modellezési nyelv egy konkrét szintaxisaként érdemes készíteni. Ez azt jelenti, hogy egy szövegesen megadott leírás elemeket és ezek kapcsolatait írja egy metamodellnek megfelelően.

Az Xtext nyelvtanok készítéséhez egy erős nyelvtan leíró nyelvet kínál. Az ezzel leírt nyelvtanhoz pedig teljes infrastruktúrát generál mint: parser, linker, típus ellenőrző, fordító. Ezen felül Eclipses környezetben a szerkesztéshez kapunk szintaxis kiemelést, content assistot.

MagicDraw-ban is ki tudjuk használni az Xtextben rejlő lehetőségeket és akár saját kiértékelő motort is tudunk írni, az általunk létrehozott nyelvtanokhoz. Sajnos a MagicDraw beépített szerkesztőjében elesünk olyan hasznos felhasználói funkcióktól mint a content assist vagy a szintaxis kiemelés. Erre megoldást jelenthet az Xtext language szerver támogatásának a kihasználása a dolgozat azonban ennek a vizsgálatára nem tér ki.

Az Xtext használatára a dolgozat elkészítése során két helyen volt szükségem. Az egyik az őrfeltételek *parse*olása, a másik pedig külön kérésre egy olyan export funkció integrálása ami XMI helyett a Gamma saját nyelvtanára sorosítva képes modelleket kimenteni.

3. fejezet

Plugin továbbfejlesztése

Ebben a fejezetben ismertetem a beépülő modulon végzett továbbfejlesztéseket. Kezdetben bemutatom a fejlesztés céljait ezt követően ismertetem, hogy ezeket hogyan valósítottam meg. Bemutatom a feladat elvégzése során készített UML profilt és ennek elemeit illetve azt, hogyan valósítottam meg a kompozíció leképzését ennek segítségével. Ezt követően emutatom milyen megoldást alkalmaztam a modellen ellenőrizendő tulajdonságok megfogalmazására. Kitérek a plugin validációs szabályaira és elmagyarázom miért van szükség rájuk, végül rövid példán bemutatom az elkészült funkciók használatát.

3.1. Fejlesztés céljai

A fejlesztés során hozott alkalmazott megoldások áttekintése előtt érdemes végigvenni, hogy mik is voltak a fejlesztés fő céljai és mi volt ezeknek a motivációja.

3.1.1. Kompozit állapotgép definíciók támogatása

Egy rendszert állapotait és állapotváltásait le lehetne modellezni egy mindent tartalmazó állapotgéppel, párhuzamos régiókkal és egyéb modellezési megoldásokkal. Ahogy a modell mérete nő a funkcionalitást érdemes feldarabolni és részenként modellezni. Ez nem csak az áttekinthetőséget segíti, de megkönnyíti a modellen való csapatmunkát is a mérnökök számára hiszen a szétbontott részeket külön, egymástól független lehet modellezni, majd ezeket összekapcsolni.

Szerencsére a Gamma erre a problémára is megoldást kínál, támogatja állapotgépek kompozíciójának modellezését és formális verifikációját. A fejlesztés egyik fő célja modell alapú kompozit rendszerek modellezésének és transzformációjának támogatása MagicDraw - SysML modellek esetében.

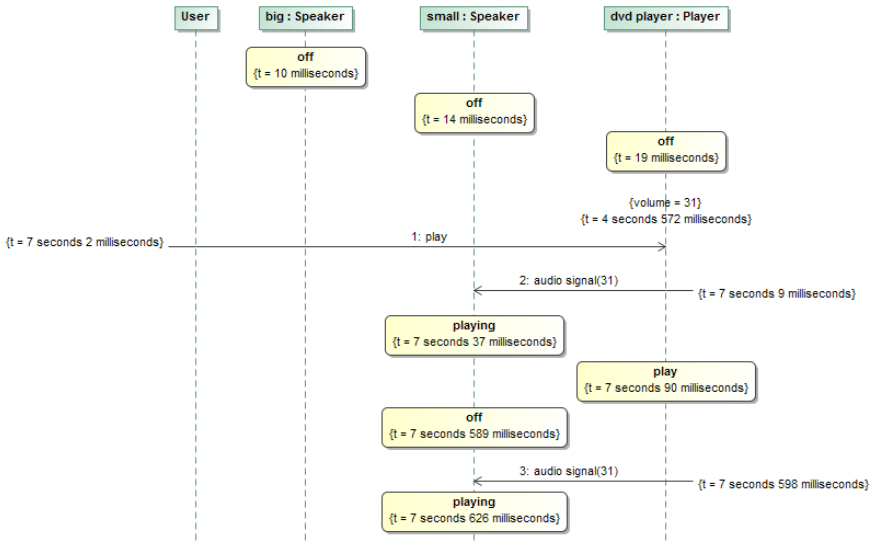
3.1.2. Eredmények megjelenítése

Fontos kérdés, hogy a formális verifikáció eredményét milyen formában kívánjuk megjeleníteni a mérnökök számára. A plugin korábbi verziója csak arra a kérdésre tudott választ adni, hogy teljesülnek-e a modellel szemben támasztott megkötések vagy sem.

Az UPPAAL és a Gamma is képes sokkal részletesebb választ adni. Ezt a Gamma *back-annotation* formájában teszi azaz a verifikációban előállt időzítések és lépéseket visszavezeti az eredeti modellbe. Ehhez a Gamma egy elég jól értelmezhető nyelvtant definiál, azonban a SysML-t jellemzően rendszermérnökök használják akik általában a szöveges leírások helyett a diagramokat preferálják.

A modellbe visszavezetett lépések, *trace*k, a modell egy futtatását írják le. Ezek szinkron esetben egymástól jól elkülönülő lépés sorozatok. Ezt leginkább szekvencia diagramo-

kon lehet ábrázolni. További érv a szekvencia diagramok alkalmazása mellett az a MagicDraw, illetve a Cameo Simulation toolkit. Ez ugyanis lehetőséget biztosít szimulációk rögzítésére szekvencia diagramok formájában (3.1. ábra)) melyekkel a futtatás elméletben reprodukálható. A cél tehát olyan szekvencia diagramok előállítása úgy minthogyha a szimulátort használva találtuk volna meg a hibautakat, így ezekről nem csak egy jól áttekinthető megoldást kapunk szekvenciák formájában, hanem ezek akár szimulálhatóak is lehetnek Cameo Simulation Toolkit segítségével.



3.1. ábra. Rögzített szimuláció szekvencia diagramon[1]

3.1.3. Követelmények definiálása

A formális verifikáció elvégzéséhez a modelleken felül meg kell tudnia a felhasználónak azokat a kérdéseket melyekre választ szeretne kapni a modell ellenőrzése során. Ezek jelen esetben a ”Kerülhet-e a rendszer adott állapotba” illetve ”Adott állapotból el tud-e jutni egy másikba” típusúak lehetnek. Az UPPAAL-ban ezeket temporális logikai kifejezések segítségével tudjuk megtenni UPPAALQueryk formájában ezért ezeket az ellenőrzés során elő kell állítanunk.

A Gammához készült egy úgynevezett *Property Language* és ez ennek az UPPAALra transzformáló funkciója és ezt terveztem felhasználni. Ez szintén temporális logikai kifejezéseket ír le viszont nem az UPPAALbemenetét képező időzített automatákon hanem magukon az állapotgépeken.

3.1.4. Validáció

A fejlesztés során hozott számos döntés megköveteli, hogy a modellekre vonatkozzanak bizonyos jól-formáltsági kényszerek. Ezek egy része a Gammából örökölt. Például a *Property Language* egy komponensen belül egy állapotra a régión keresztül tudunk hivatkozni (*[component]+.region.state*). Ebből következik, hogy a régióknak a MagicDraw modellben nevet kell adni, illetve, hogy ezek egyediek legyenek egy állapottérképen belül.

A munkám egyik célja egy validációs szabálykészlet létrehozása ami segít a felhasználóknak az eszköz helyes használatában.

[1]Forrás: https://docs.nomagic.com/display/CSTD184/Recording+simulation+as+a+Sequence+diagram

3.2. Gamma UML profil

A dolgozat elkészítéséhez pusztán a SysML nyelv nem volt elegendő ugyanis szükséges volt a modellben is eltárolni bizonyos információkat, mint például az ellenőrizendő követelmények, a *back-annotation*, vagy éppen, hogy milyen kompozit szemantikát kívánunk érvényesíteni az adott modellekre. Ezt többféleképpen meg lehet valósítani például speciális nevezéktannal, én viszont egy UML profil készítése mellett döntöttem. Ez lehetővé teszi, hogy az egyes elemeket könnyebb legyen keresni és nagyobb flexibilitást is ad például saját mezőket tudtam definiálni amik akár lehetnek származtatottak is. Az elemeknek továbbá megkötéseket tudok előírni a tartalmazási hierarchiára.

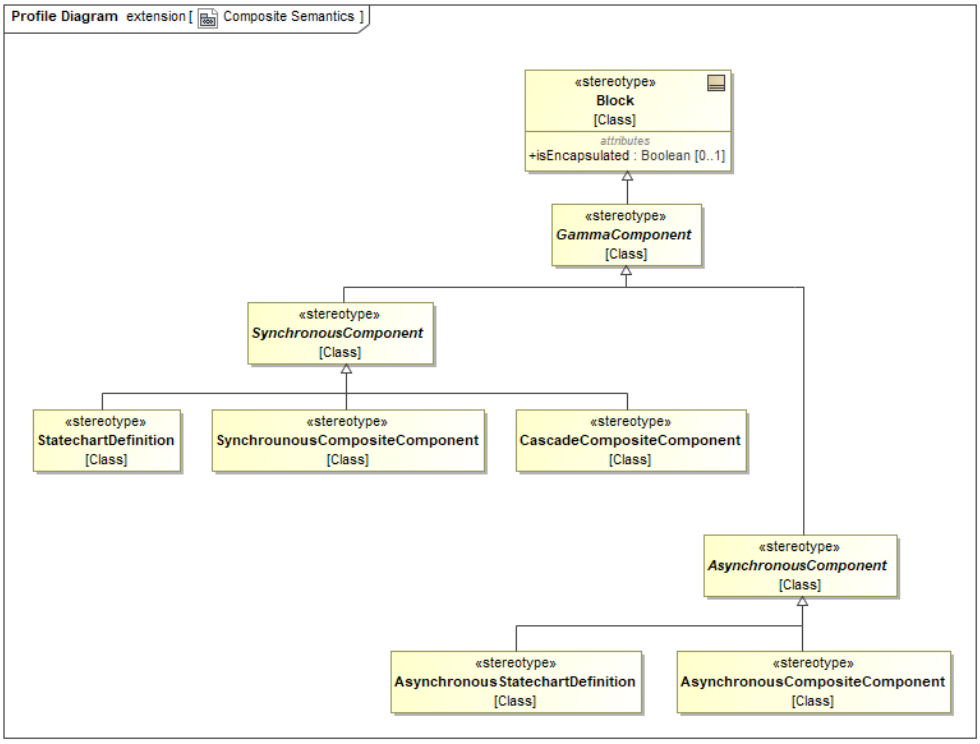
Az UML három részből áll:

- Kompozit szemantika
- Check modell
- Back-annotation modell

Az UML profil érdekessége, hogy tartalmazásokat csak megkötés szintjén *Customization*on keresztül tudunk megadni. UML profil diagramon csak öröklést és *tag*eket van lehetőségünk definiálni.

3.2.1. Kompozit szemantika

A Gamma háromféle komponens végrehajtási szemantikát biztosít a felhasználók számára. Ugyan ezt implicit módon is meg lehet határozni, például a kommunikáció típusából, mégis fontos lehet ezek egyértelmű jelölése. Ezért az UML profil (3.2. ábra) a SysML nyelvet kiegészíti néhány sztereotípiával melyek segítségével explicit jelölhetők, hogy milyen szemantikát szeretne a felhasználó érteni Blokkjain.



3.2. ábra. Szinkron, asszinkron szemantikát támogató UML profil

A sztereotípiák leszármaznak a Block sztereotípiájából, így lényegében lecserélik azt. Ennek előnye hogy szintaktikailag is asszinkron és szinkron blokkok fognak megjelenni a

modelljeinkben. Viszont van egy nagyon nagy hátránya is mégpedig az, hogyha már meglévő modelleken szeretnénk használni az eszközt szükség van azok módosítására, amire nem mindig van lehetőség, például elosztott környezetben. Sokszor azonban az őrfeltételekre, akciókra és interfészekre vonatkozó megkötések már önmagukban megkövetelik a modellek módosítását, hiszen ezek nagy valószínűséggel nem a támogatott módszertant követik.

Egy alternatív megoldás az lehetne, hogy a sztereotípiákat valamilyen él például *Dependency* segítségével rendeljük az egyes komponens definíciókhoz, vagy komponensekhez, *part*okhoz. Ez megoldást nyújthat arra a problémára is, hogyha a modell egyes részei külső könyvtárból jönnek, vagy nincs hozzáférésünk hozzájuk.

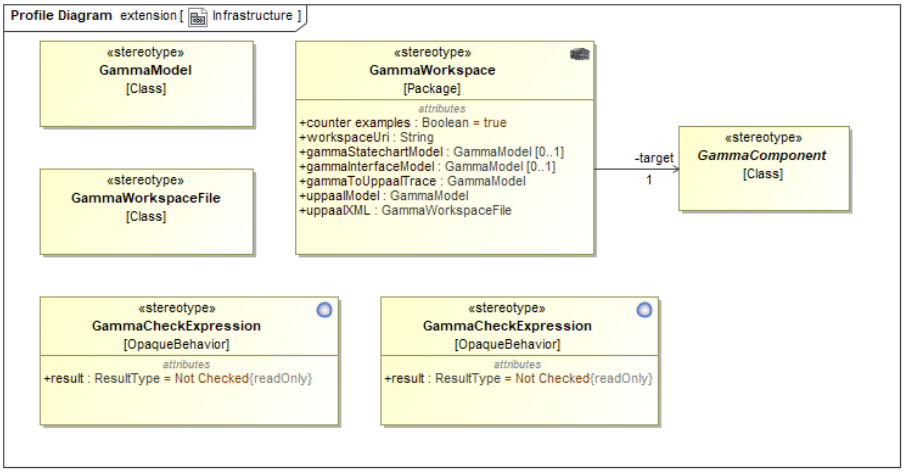
3.2.2. Check modell

Ahhoz, hogy a formális verifikáció végrehajtható legyen ki kell választani, hogy milyen modelleket szeretnénk ellenőrizni és ezeken milyen követelményeket. Az formális verifikációt modell elemeken keresztül lehet paraméterezni.

Az elképzelés szerint a felhasználó *Workspace*eket hoz létre. Ezek hivatkoznak a felhasználó számítógépén egy könyvtárra melyet az infrastruktúra sajátosságából adódó háttértárra kimentendő modelleket tárolására használok. Ezen felül rajta keresztül kell behivatkozni az ellenőrizendő modellt a projektből.

A modell transzformációk során *Trace*k keletkeznek. Ezek alapján lehet visszakeresni, hogy mely a MagicDraw - Gamma transzformáció során milyen leképzések történtek. Ezek valójában UML propertyk egy *Class*on belül melyeknek a neve egy azonosító amivel a Gamma modell egy kisorosított XMI fájlban lévő elemei vannak hivatkozva. A *property*kből egy *Trace* él mutat a megfelelő MagicDraw elemekre. A kisorosított XMI ugyan ezen *Class* modell elemekhez mint *Comment* vannak hozzárendelve, innen lehet őket kiolvasni.

A követelményeket *GammaProperty* segítségével lehet megadni a Gamma által definiált Property nyelvtan segítségével. Az ezek ellenőrzéséből származó Back-annotaion modellek is a *Workspace*ben helyezkednek el. A modell elemei a következők (3.3 ábra):



3.3. ábra. Infrastruktúra elemei

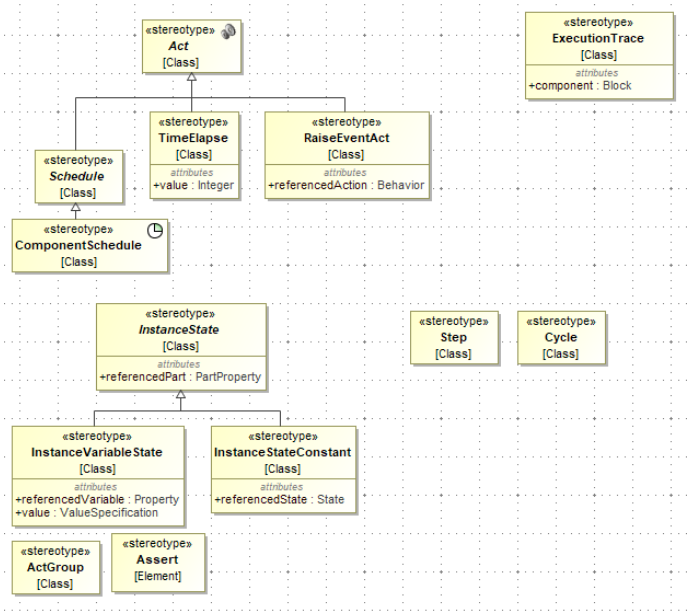
- **GammaWorkspace**
  Letranszformált modellek és egyéb konfigurációkat és köztes állapotokat tároló modell eleme a projektben.
  Ennek az elemnek három fontos attribútuma van. A *Target: Block[1]* kijelöli azt a blokkot amin a funkcionalitás végre szeretnénk hajtani. A *WorkspaceUri: String[1]* ami kijelöl egy könyvtárat a háttértáron, hogy oda sorosodjanak ki azok a modellek amiket az UPPAAL fog használni a futtatás során. Azt, hogy szernénk-e vissza an-

notálni az ellenpéldát a modellben a *Counter example: Boolean[1]* állításával tudjuk megadni. Ezeken kívül még referenciákat is tárol a majdani letranszformált modellekre.

- **GammaCheckExpression**
  Egy *Opaque Behavior* ami a törzsében(*body*) a Gamma *Property Language* segítségével definiált tulajdonságokat tárol a modellre vonatkozóan. Itt fontos megjegyezni, hogy ezekben az elemekben nyelvet is lehet specifikálni. Ezt viszont figyelmen kívül hagytam és mindenképp az előbbi nyelvtan szerint értelmezem a kifejezéseket.
- **GammaModel**
  Letranszformált Gamma modelleket tárol XMI-k formájában az elemhez csatolva kommentként. Ezen kívül *property* elemeket tartalmaz amelyek a Gamma - MagicDraw visszakereshetőségért felelnek a következőképp: a MagicDraw elemre egy Trace él mutat a *property*ből, a Gammabeli modell elem pedig EMF hivatkozás formájában kerül tárolásra mint az elem neve.
- **GammaWorkspaceFile**
  Hivatkozás egy Gamma modellre a háttértáron. A modell elem neve a fájl elérési útja.

3.2.3. Back-annotation modell

A back-annotation modell feladata visszacsatolni az eredeti modellbe a valamilyen külső, jellemzően szimulátorból származó időzítési adatokat. A Gamma is készít egy ilyen modellt. A megoldás amit kidolgoztam Gamma modell egy MagicDraw specifikus változatának létrehozásából és egy Gamma-MagicDraw transzformáció megtervezéséből és megvalósításából állt. Az UML profil egy szakterület specifikus nyelvet definiál. Elemei pedig a következők:



3.4. ábra. Back-annotation UML profilja

- **ExecutionTrace**
  Egy kompozit komponens végrehajtásának menetét rögzíti. A végrehajtás *Step*ekből illetve a végrehajtás végén egy ciklusból állhat. A ciklus pedig további *Step*ekből. Az *ExecutionTrace* egy hivatkozást is tárol az adott komponensre.

- **Step**
  A végrehajtás egy lépése. Ez három részből áll. Az elsőben leírja, hogy milyen állapotokat vettek fel a komponensek és milyen értékeken álltak a változók. A másodikban, hogy mely események érkeztek és vezetnek át majd a következő lépésbe. A harmadik rész pedig a kimenő eseményeket rögzíti.
- **Act**
  Absztrakt sztereotípia. Egy végrehajtott művelet.
- **ComponentSchedule**
  Komponensen egy kör végrehajtása (üzenetek kiolvasása az üzenetsorokból, állapotok léptetése)
- **TimeElapse**
  Várakozást, az idő múlását szimbolizálja. Ez az érték milliszekundumban egész számként van megadva az elem *Value* mezőjében.
- **RaiseEventAct**
  Egy esemény elküldése a komponens példánynak. Egy *Activiy* diagramot tartalmazó *Class*. Ez a diagram az adott szignált küldő egyetlen akcióból áll.
- **InstanceState**
  Absztrakt. Az egyes példányok, partok állapotainak leírása. Itt tárolódik a referencia arra a *Part*ra amelyiknek az állapotai rögzítésre kerülnek.
- **InstanceVariableState**
  Egy változónak adott lépésben felvett értékét tárolja.
- **InstanceStateConstant**
  Az tárolja, hogy adott lépésen belül milyen állapotban volt a *Part*.

3.3. Kompozíciók transzformációja

Egy nagy komplex rendszert célszerű nem egyben, hanem részekre bontva modellezni majd a részek egymáshoz illesztéséből, komponálásából képezni a teljes rendszert. Állapottérképek dekomponálását azaz részekre bontását a Gamma is támogatja. A kihívás a SysML és a Gamma közötti megfeleltetések megválasztása oly módon, hogy a szemantika ne sérüljön. A megfeleltetés két szempontól kell vizsgálni, egyszer az elemek tartalmazási hierarchiái szerint, egyszer pedig a köztük modellezett kommunikáció szerint.

3.3.1. Struktúra megfeleltetése

A Gammában nyelvi szinten elkülönül az állapottérkép (StatechartDefinition) és a kompozit komponens (Composite Component) fogalma. Állapottérképek a modell hierarchia gráfjában a levelekben helyezkednek el. SysML esetében a minden Blokknak lehet viselkedése, jelen esetben állapottérképe. A megfeleltetés elvégzéséhez megkötést fogalmaztam meg mely kétféle blokkot enged meg.

- a viselkedéssel rendelkezőket, amelyek nem tartalmaznak *Part*okat

és

- a viselkedés nélkülieket, melyek *Part*okat tartalmaznak.

Előbbiek a tartalmazási hierarchiában a levél elemek. Ezzel a módszerrel nem fordulhat elő, hogy egy Blokknak nem egyértelmű a viselkedése. Hiszen ezt mindig a tartalmazottjai adják. Fontos, hogy csak olyan modelleket lehet ellenőrizni amik legalább egy *Part*ot tartalmaznak. Azaz a legkisebb modellnek állnia kell legalább két blokk definícióból amelyek a két már fent említett esetek egyikébe esnek.

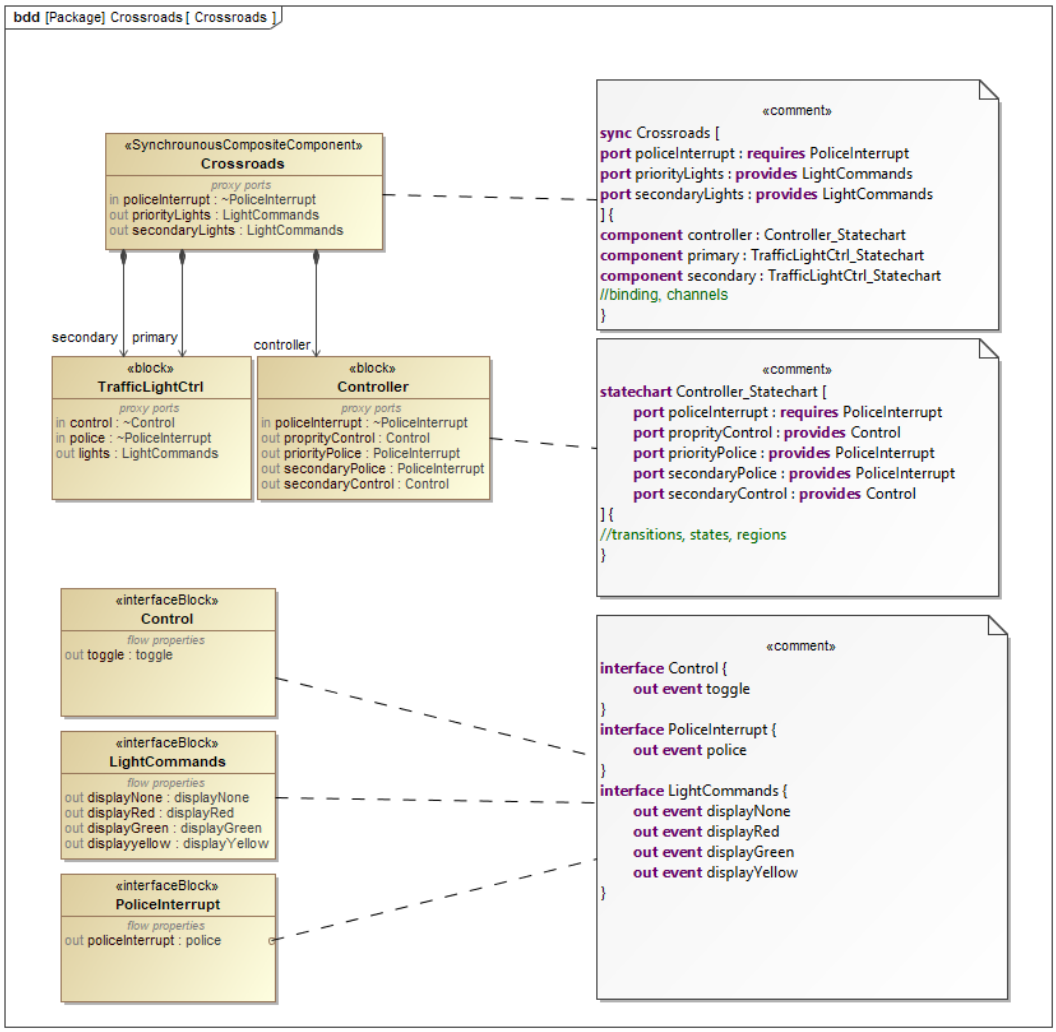
Az érdekes kérdés lehet, hogy ha egy magasabb szintű blokknak mégis lehetne viselkedése, azt szemantikailag hogyan kezelhetnénk. Elképzelhető olyan értelmezés, hogy ez a magas szintű állapottérkép a dekomponált részek együttes viselkedését írja le. Ez lehetőséget adna arra, hogy a magasabb és az alacsonyabb szintű viselkedés halmazt összehasonlítsuk működés szempontjából és ha nincsenek szinkronban akkor tervezési hibaként értelmezzük. Így a magasabb hierarchia szintek tulajdonképpen validálnák az alacsonyabb szinteket. Ennek a lehetőségnek a komolyabb kifejtése azonban nem célja a dolgozatnak.

3.3.2. Kommunikáció megfeleltetése

A Gammában a komponensek kommunikációja kimondottan az eseményvezérelt állapot alapú rendszerek sajátosságain alapul, azonban SysML-ben a kommunikáció leírása sokkal általánosabb és többféle módon is modellezhető. Éppen ezért a feladat elvégzése során az események és az interfészek modellezésére megkötéseket kellett megfogalmazni. Az egyik ilyen megkötés szerint a portoknak interfész blokkokkal kell, hogy tipizálva legyenek.

Az eseményeknek mindig specifikálniuk kell, hogy melyik porton várják a szignál érkezését. Ez igaz az akciókra is. Az ő esetükben azt kell specifikálni, hogy milyen porton keresztül történjen a küldés.

Portok közül csak a *Proxy* portok támogatottak. Ennek oka, hogy talán ezek állnak legközelebb a Gammában használatos portokhoz szemantikailag szemben például a *Full Port*okkal. Utóbbiak a rendszer külön részeinek tekinthetők és saját tulajdonságokkal bírhatnak, míg a *Proxy* portok a blokkjuk tulajdonságaihoz nyújtanak hozzáférési pontot a "külvilágnak". Azt, hogy melyek pontosan ezek a tulajdonságok az *Interface Block* definiálja. A Proxy portok iránya származtatott a *Flow property* irányokból amik keresztül mehetnek rajta. Az irány az *isConjugated* flag igazra állításával változtatható.



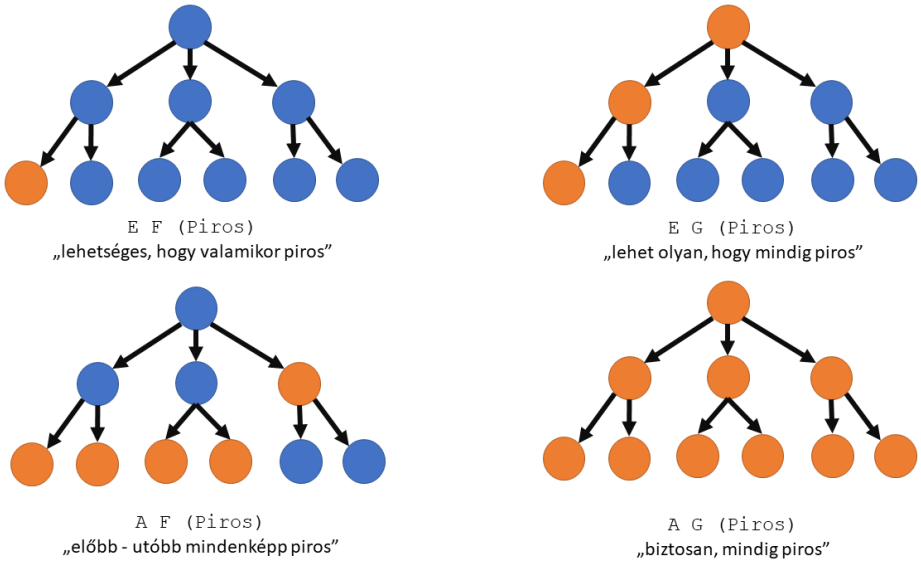
3.5. ábra. Struktúra megfeleltetése

3.4. Modell tulajdonságainak leírása

A modell felett meg kell tudnunk fogalmazni tulajdonságokat, amelyeknek a teljesülését formális verifikáció segítségével szeretnénk majd igazolni. Ezeknek a tulajdonságoknak sokféle módon le lehetne írni, például Object Constranint Language (OCL) segítségével, esetleg valamilyen szcenárió alapú leírással. A Gamma esetében lehetősége van a felhasználóknak elágazó idejű tempóralis logikai kifejezésekkel tulajdonságokat megfogalmazni (példa: 3.6 ábra). A feladat elkészítéséhez ezt a nyelvtant használtam fel.

Az elágazó idejűség azt jelenti, hogy a rendszernek nem csak egy, hanem az összes lefutását vizsgáljuk egyszerre. Éppen ezért meg kell tudnunk mondani azt, hogy egy tulajdonság teljesülését minden lefutás esetében elvárunk, vagy megelégszünk azzal, hogy van olyan lefutás ahol teljesül. Ezek jelölése a nyelvben az ”A” (for all futures) és az ”E” (exits future) kvantorok.

Az egyes útvonalakon is meg kell tudnunk mondani, hogy a tulajdonság teljesülését mikor várjuk. Valamikor a jövőben vagy esetleg minden lépésben azaz a tulajdonság invariáns. Ezek jelölései az ”F” (future) és a ”G” (globally). A nyelv ezeken felül támogatja még az ”X” (next), ”U” (until) és ”R” (release) operátorokat is.



3.6. ábra. Példa: CTL kifejezések

3.4.1. Kifejezések a definiálása és használata

A modellben a kifejezéseket *Check Expression*ök törzsében kell megadni. Ezeknek a modell elemeknek egy tetszőleges *package*ben kell lennie aminek viszont egy *Gamma-Workspace*ben. Ellenőrzés során a kifejezések kiolvasásra kerülnek és Xtext segítségével *parse*oldónak. Fontos, hogy ez a nyelvtan Gamma modellekre képes hivatkozni nevezetesen állapotokra és változókra, éppen ezért el kell végezni a MagicDraw - Gamma transzformációt mielőtt a *parse*olást elvégeznénk. A transzformáció után a hivatkozható elemek ugyan azon a néven szerepelnek és ugyan olyan hierarchiában a két modellben ezért nem volt szükséges a kettőt objektum szinten is összekapcsolni. Ez a kérdés azért fontos mert ezeket a tulajdonságokat a SysML modellen és nem pedig a Gamma modellen szeretnénk kimondani.

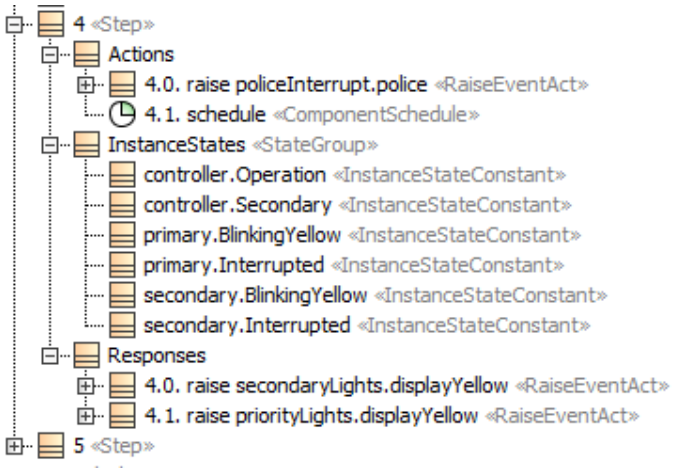
Érdemes megjegyezni, hogy mivel a Gamma a háttérben az UPPAALsegítségével végzi a formális verifikációt ezeket a kifejezéseket még át kell alakítani UPPAAL queryvé. Ezt a konverziót a Gamma elvégzi. Igazából ez az a lépés ahol a Gamma modellek megléte szükségessé válik, hiszen az állapot és változó neveknek a majdani UPPAALmodellben is helyesnek kell lenni.

3.5. MagicDraw modellek back-annotációja

Bizonyos tulajdonságok meglétét vagy éppenséggel hiányát ellenpéldák igazolják. Ezeket a Gamma back-annotation segítségével vezeti vissza az ellenőrzött modelljeibe. A fentebb ismertetett UML profilt azért hoztam létre, hogy ezeket az ellenpéldák, vagy hibautak könnyebben modellezhetőek legyenek.

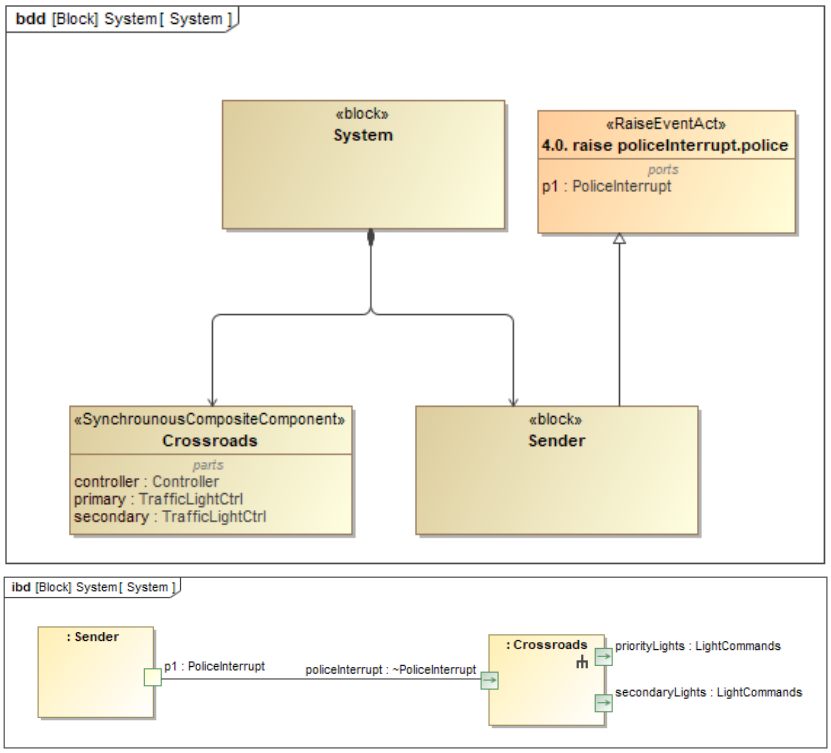
Ezeknek az előállítása most is modell transzformációkon keresztül történik, azonban ezek iránya itt megfordul nem MagicDraw modellekből állítok elő Gamma modelleket hanem épp fordítva Gamma modelleket transzformálok MagicDraw modellekké. Egyébként erre a modellre nem feltétlenül lenne szükség hiszen a további származtatásokat már a Gamma modellek segítéségével is meg lehetne valósítani. Viszont a végrehajtás új módon történő tárolása már nem függ a Gammától és általánosabb mint egy *Activiy* diagram

vagy egy szekvencia, ezért könnyebb bemenetként használni olyan transzformációkhoz ami ezeket állítja elő, esetleg valami dokumentációt generál.



3.7. ábra. Lépések felbontása

Egy ellenpélda lépéseket ír le amik a feltétel sérüléséhez vezetnek (3.7). A lépések rögzítik az aktuális állapotot, az akciókat amik a következő lépésbe vezetnek és a kimeneti eseményeket. Az lépések közti akciók és a kimenet *Activiy* diagramokra képződnek le. Ezek jellemzően egy akciót tartalmaznak például egy szignál küldést ami az őt tartalmazó osztály egy portján keresztül küld eseményeket. Ez azért jó mert így a rendszerünkből létre tudunk hozni egy *Part*ot amire ha rákötünk egy ezekből a *RaiseEvent*ekből leszármazó blokkal tipizált partot akkor ez képes lesz kommunikálni a rendszerünkkel (3.8 ábra).



3.8. ábra. Üzenetek küldése a rendszernek

3.6. Szimuláció

A fejlesztés során a futtatás megjelenítésének egyik módjának a szimuláció ígérkezett. Ez sajnos a szimulációs eszköz képességei miatt nem tudott maradéktalanul megvalósulni, viszont az előállított modellek mint megjelenítési forma egészen ígéretesnek bizonyult. Továbbá esetleg később ahogy fejlődik a szimulációs eszköz lehetséges, hogy kis alakításokkal lehetővé válik a modellek szimulációja is. A továbbiakban bemutatom a funkció megvalósítására tett próbálkozásokat és az akadályokat ami miatt nem tudott a funkció maradéktalanul megvalósulni.

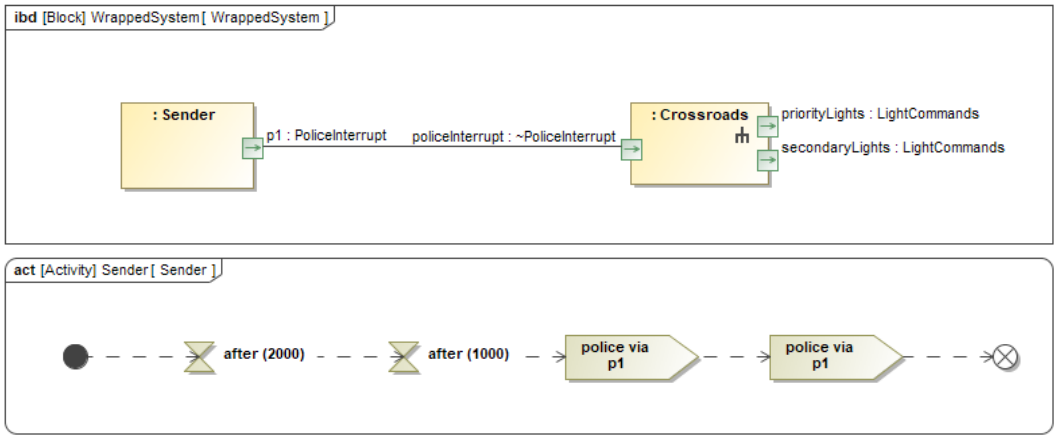
3.6.1. Szekvencia diagramok

Kezdetben a legígéretesebb lehetőségnek a szekvencia diagram bizonyult. Ahogy viszont elkezdtem belemélyedni ezek használatába elkezdtek a hiányosságok már a modellezés szintjén megmutatkozni azok iránt az aspektusok iránt, amiket modellezni szerettem volna. Ugyan a komponensek kommunikációja és annak sorrendisége leírható nincsen lehetőség egyéb strukturális tulajdonságok használatára mint például a portok.

A Cameo Simulation toolkit képes a szimulációt rögzíteni szekvencia diagramon azonban a szignál kommunikáció nem fog működni a hiszen az eseményeket nem maguk a komponensek, hanem a portjaik fogadják. Amit viszont nagyon jól lehet ezeken a diagramokon ábrázolni azok a rendszer állapotai *State Invariant*ok formájában. Ezekkel, ha működő szimulációt nem is, egy jól áttekinthető megjelenítését kaphatjuk a működésről.

3.6.2. Activity diagramok

Egy másik megközelítés amivel próbálkoztam, hogy a működést *Activity* diagramok segítségével írom le. Ezeken várakozni is lehetséges *Time Event* csomópontokkal és a *Send Signal* eventek képesek kijelölni a portokat amiken a kommunikáció végbemegy. Ezzel a módszerrel már el lehet küldeni a szignálokat a megfelelő időzítéssel az ellen példának megfelelően, sőt igény szerint a szimulátor szekvencia diagram generátorával fel lehet venni a végrehajtást (3.9 ábra). Az egyetlen hiányosság, az aktuális állapotok nyomon követhetősége. Alap beállítások mellett a szimulátor nem kezeli jól az időzítéseket. Például az egyes

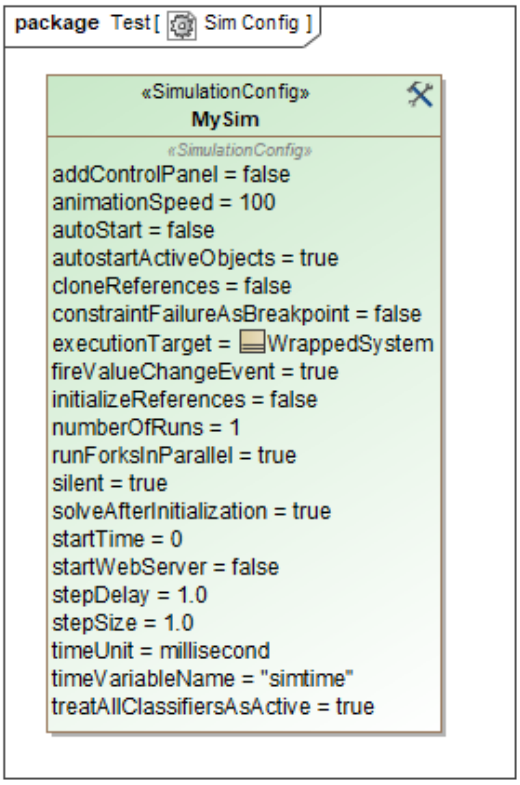


3.9. ábra. Ellenpélda végrehajtása

komponensek kezdő állapotai nem a nulla időpillanatba voltak aktívak, hanem később és a különböző esemény küldések is időbe teltek amik azt eredményezték, hogy az ellenpélda nem működött a példán, hiszen a modellek nem tartalmaztak ezekre vonatkozóan infor-

mációkat és a végrehajtások elcsúsztak. Szerencsére azt, hogy hogyan kezeli a szimulátor az idő múlását van lehetőség állítani *Simulation Config* segítségével (3.10 ábra).

A modell órájának megfelelő konfigurációja után már az ellenpéldában is szereplő időzítésekkel hajtódtak végre az állapotváltások. Ezek a *start time* nullára állítása. A *step delay* és *step size* 1-1-re állításai voltak. A konfigurációban az időzítéseken túl a végrehajtás



3.10. ábra. Szimuláció konfigurálása

más aspektusait is állítani lehet. Ami kiemelendő az a *slient* mód. Ez kikapcsolja a diagramokon való animációkat. A hivatalos dokumentáció azt ajánlja, hogy időzítés érzékeny modelleken ezt érdemes kikapcsolni.

3.7. Validáció

A modell ellenőrzés akkor lehetséges, ha a felhasználó betart bizonyos modellezési technikákat, amik lehetővé teszik a modell transzformáció végrehajtását. Ezek betartatására illetve az esetleges hibák megtalálásához készítettem egy validációs szabálykészletet.

3.7.1. Elnevezett régiók

Név:	RegionNamedRule
Azonosító:	GAMMA_REGION_NAMED
Súlyosság:	Error
Üzenet:	Region must have a name

Erre a szabályra azért van szükség mert a *Property* nyelvtanban az egyes állapotokra az őket tartalmazó régiókon keresztül lehet hivatkozni, ezért egy állapotgépen belüli el nem nevezett régiók validációs hibához vezetnek.

3.7.2. Régiók nevei egyediek

Név:	RegionNameUniqueRule
Azonosító:	GAMMA_REGION_UNIQUE
Súlyosság:	Error
Üzenet:	Region must have a unique name

Szintén a fent említett hivatkozások miatt a régiók neveinek egyértelműnek kell lennie állapotgépeken belül. Az azonos nevű régiók validációs hibát dobnak.

3.7.3. Szignál küldhető

Név:	SignalSendRule
Azonosító:	GAMMA_SIGNAL_SEND
Súlyosság:	Error
Üzenet:	Signal cannot be sent through target port

Ahhoz, hogy egy szignált el lehessen küldeni egy porton a port interfészén szerepelnie kell mégpedig a származtatott irányának ”ki” irányúnak kell lennie. A származtatott irány nem *isConjugate* esetében megegyezik a szignál interfészen feltüntetett irányával egyébként azzal ellentétes. Ez a szabály csak *Send Signal* akciókra érvényes.

3.7.4. Szignál fogadható

Név:	SignalReceiveRule
Azonosító:	GAMMA_SIGNAL_RECEIVE
Súlyosság:	Error
Üzenet:	Signal cannot be received from target port

Szignál fogadásához, hasonlóan mint a küldés esetében a szignálnak szerepelnie kell a megfelelő származtatott iránnyal a port interfészén. Ez a szabály *Triggerek*re fut le és akkor okoz hibát, ha nem szerepel a referált szignál az interfészen vagy nem megfelelő az iránya.

3.7.5. Szignál neve egyedi

Név:	SignalUniqueRule
Azonosító:	GAMMA_SIGNAL_UNIQUE
Súlyosság:	Error
Üzenet:	Flow property type name must be unique

Ahhoz, hogy szignálthoz irányt lehessen rendelni fel kell venni egy *Flow Property*t és tipizálni a szignállal. A MagicDraw kikényszeríti, hogy egyedi neve legyen a *Flow Property*knek az azonban lehetséges, hogy két különböző *Flow Property* ugyan azt a szignált használja típusnak vagy ugyan külön szignálra hivatkoznak viszont ezek nevei megegyeznek. Ez a transzformáció során hibához vezethet hiszen a majdani *Event*ek nevei ütközni fognak. Éppen ezért a *Flow Property*k által használt szignálok neveinek egyedinek kell lennie.

3.7.6. Kompozit blokkok

Név:	CompositeBlockRule
Azonosító:	GAMMA_COMPOSITE
Súlyosság:	Error
Üzenet:	Composite blocks cannot define classifier behaviors

Blokkok vagy kompozit blokkok vagy viselkedést, állapotgépeket definiáló blokkok lehetnek. Ez a szabály hibásnak jelöl minden blokkon ami vegyesen definiál partokat és viselkedéseket.

3.7.7. Statechart definíciók

Név:	StatechartBlockRule
Azonosító:	GAMMA_STATECHART
Súlyosság:	Error
Üzenet:	Statechart blocks cannot contain parts

Állapotgépet definiáló blokkok nem tartalmazhatnak partokat.

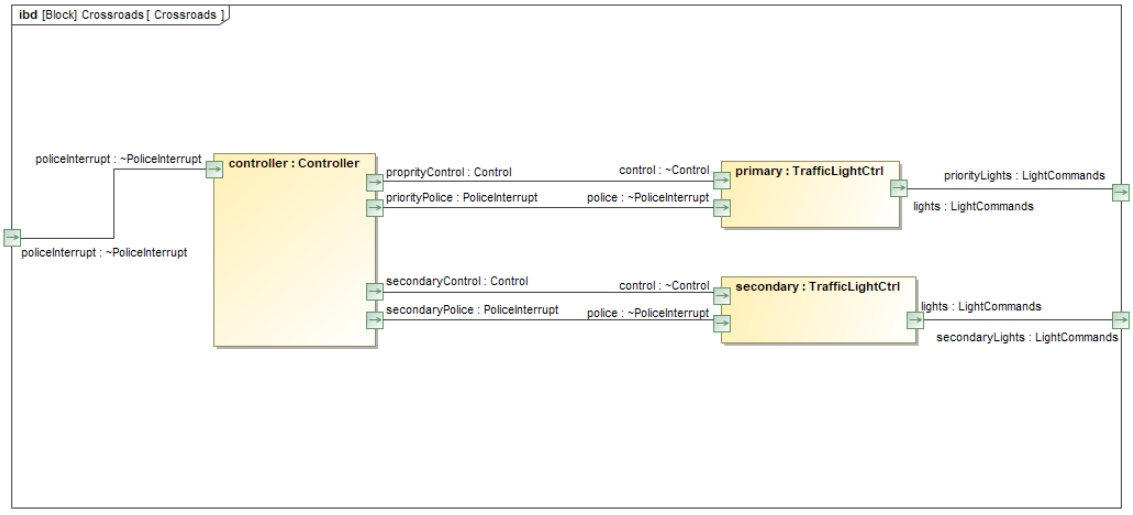
4. fejezet

Esettanulmány

Ebben a fejezetben bemutatom az eszköz használatát egy példamodellen. Ez a modell a Gamma tutorial csomagjában lévő közlekedési lámpákat tartalmazó modelljének SysML-be átemelt változata. A példán ismertetése közben bemutatom a az eszköz által támogatott modellezési módszertant is.

4.1. A példamodel

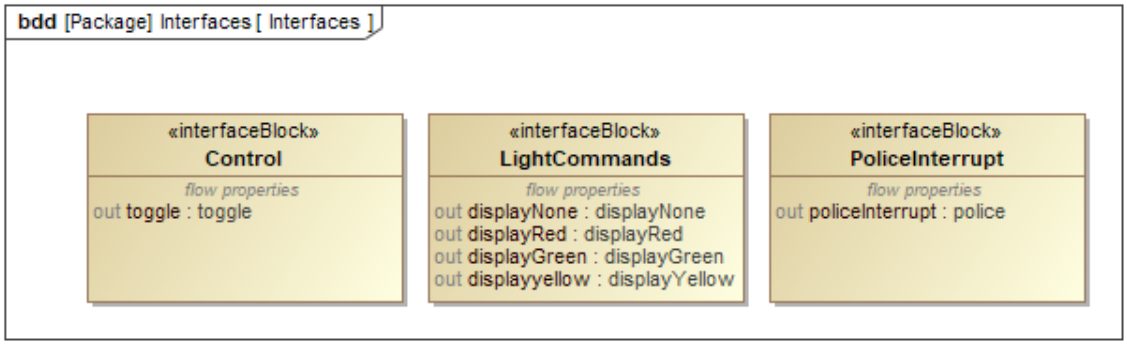
A modell három komponensből áll (4.1 ábra): két a lámpákat irányító (*primary*, *secondary*) és egy az ezeket szinkronban tartó *Controller* komponensből. A rendszernek van két kimeneti portja amellyel a két közlekedési lámpa jelzéseit tudja változtatni és rendelkezik egy bemeneti porttal amelyen keresztül a rendőrség *Interrupted* (”Sárga villogó”) állapotba tudja állítani a rendszert illetve vissza is tudja állítani a normál állapotba, ahol a piros - zöld - sárga jelzések váltakoznak a két lámpán mégpedig egymásnak ellentétesen.



4.1. ábra. Az útkereszteződés SysML modellje

A rendszer három interfészt definiál (4.2 ábra): *PoliceInterrupt*, *Control*, *LightCommands*. Ezek mindegyike *Interface Block*ként kell, hogy megjelenjen a modellben, és minden az állapottérképeken használt *Signal*hoz fel kell venni egy *Flow Property*-t a megfelelő iránnyal. Jelen esetben ez a következőként jelenik meg. A *Control* interfész tartalmaz egy *Flow Property*-t *out*, azaz kimenő iránnyal és egy ’toggle’ nevű *Signal*lal van tipizálva. Ez azért fontos mert az állapottérképen ezeket a *Signal*okat kell majd használni. A *LightCommands* négy kimenő *Property*t tartalmaz. Ezek a *displayNone*, *displayRed*, *displayGreed* és

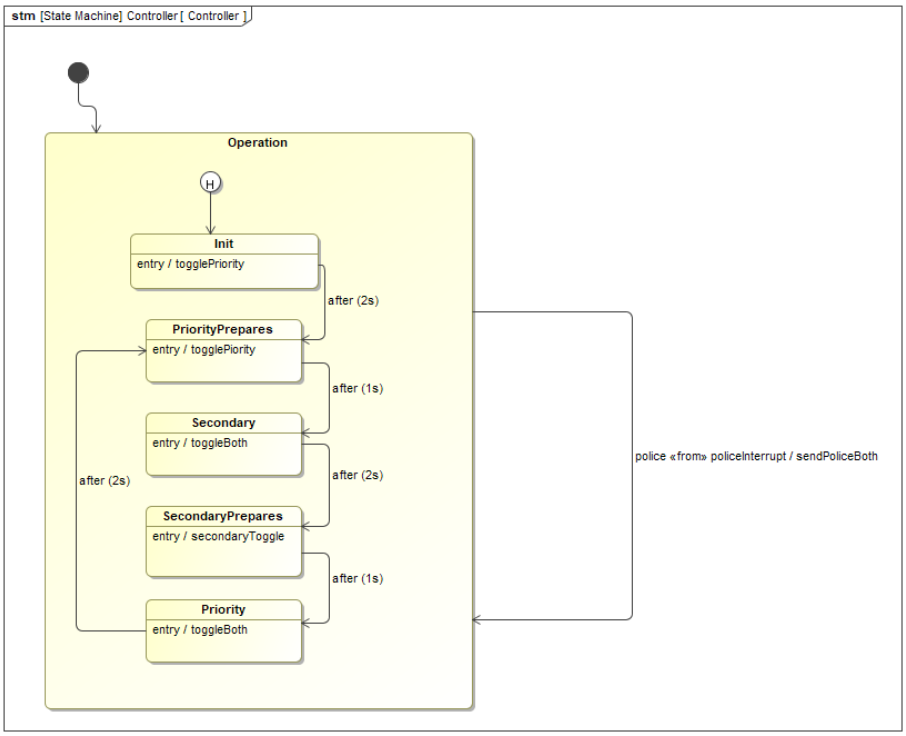
displayYellow szignálokkal vannak tipizálva. A *PoliceInterrupt* interfész csak egy *Property*t tartalmaz *out* iránnyal a *police* szignálhoz.



4.2. ábra. Interfészek a modellben

Most, hogy már az interfészek és a portok le vannak modellezve el ezeket fel lehet használni az állapottérképek leírásához. A modell két állapottérképet definiál egyet a két lámpa controllerjéhez (*LightCtr*-hez) és egyet a *Controller*hez.

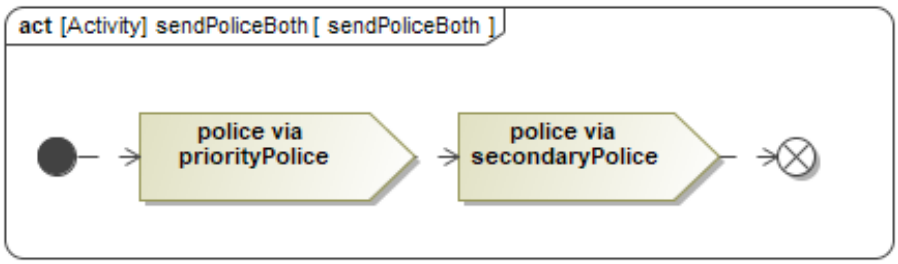
Tekintsük előbb a *Controller* állapottérképére (4.3 ábra). Az állapotgép egy kompozit állapotból és ennek négy belső állapotából áll. *Police Interrupt* hatására az állapotgép kilép a kompozit állapotból mindkét *PoliceInterrupt* típusú portján küld egy-egy *policeInterrupt* jelet, majd visszatér ugyan ebbe az állapotba. Amelyben a *History State* miatt abból az állapotba kerül vissza amiben a kilépés előtt volt.



4.3. ábra. Controller állapotai

Ami ennek az állapot átmenetnek a modellezése kapcsán izgalmas az *Signal* küldésnek a módja. A plugin jelenlegi formájában ezt Activity diagrammal kell leírni *Send Signal* akciók használatával. Ez jelen esetben egy két akciós *Activity*t fog eredményezni, amelyben

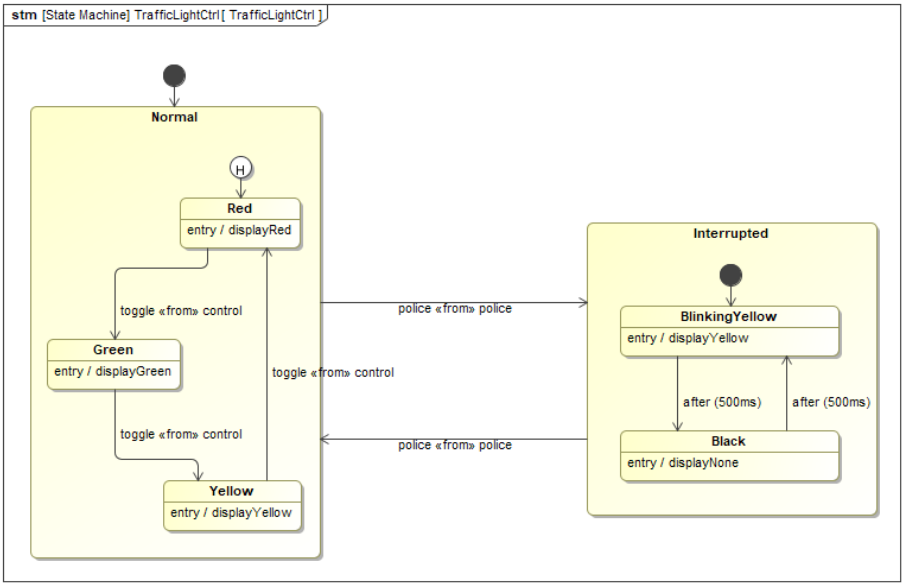
sorban a megfelelő portokon elküldésre kerülnek a szignálok (4.4 ábra). A *togglePriority*, *secondaryToggle* és *toggleBoth entry action*ok hasonlóan vannak modellezve.



4.4. ábra. *Signal*ok küldése Activityel

Amiről ennek az állapottérképnek a kapcsán még érdemes lehet beszélni ezek a bizonyos idő elteltével tüzelő átmenetek. Ezek a modellben *Time Event*ként jelennek meg. Fontos hogy ezeknek a *relative* attribútumát igazra kell állítani, hisz ez jelenti azt, hogy a forrás állapotba való belépéstől kell számítani az időt.

A teljesség kedvéért vizsgáljuk meg a másik állapottérképet is (4.5 ábra). Ez két kompozit állapotból áll. A *Normál* a szokásos működést jelenti és a *Red*, *Green*, *Yellow* állapotokból áll melyek a beérkező toggle szignálok hatására váltakoznak. Az *Interrupted* állapot a rendőrség által előidézett "sárga villogó" állapotot jelenti. Ebben a *BlinkingYellow* és a *Black* állapotok váltakoznak 500 milliszekundumonként.

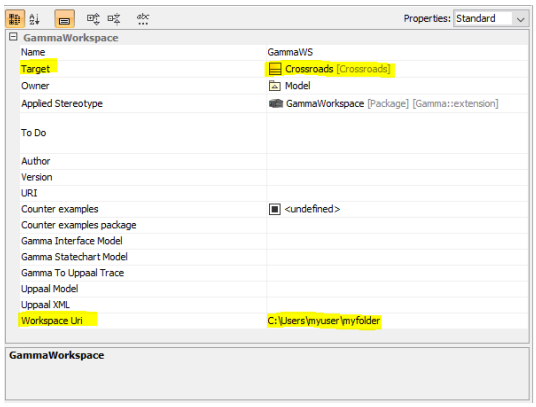


4.5. ábra. *TrafficLightCtrl* állapottérképe

4.2. Transzformációk végrehajtása

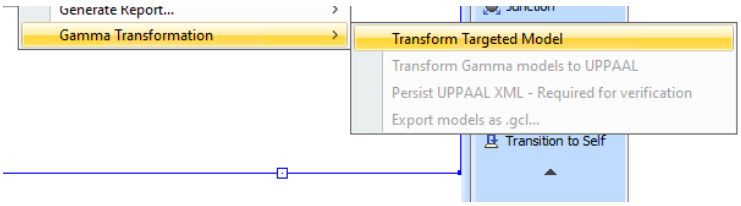
A modell összerakásánál egy dolog még nem lett lemodellezve, mégpedig, hogy melyik komponenst szemantikát szeretnénk alkalmazni. Jelen esetben a szinkron szemantikát szeretnénk. Ehhez jelen esetben elegendő a gyökér elem *Block* sztereotípiáját a specifikusabb *SynchronousCompositeComponent*re cserélni. Most, hogy a komponens szemantika specifikálva van a modellek transzformálhatók és ellenőrizhetők.

Első lépésként létre kell hozni egy *GammaWorkspace*t. Ez a modell elem fogja tárolni az ellenőrzés során szükséges adminisztrációs objektumokat. Azonban meg kell nevezni egy könyvtárat is a háttértáron amit ideiglenes fájlok tárolására fog a plugin használni. Ehhez egy *Workspace Uri*t kell megadni ami egy abszolút elérési út egy tetszőleges könyvtárhoz. Az ellenőrizendő modellt a *Target* mező megadásával lehet specifikálni.



4.6. ábra. *GammaWorkspace* specifikációja

A kötelező mezők specifikálása után a *Gamma Workspace*n nyitott gyorsmenüben a *Gamma Transformation* menüpont alatt található akciók segítségével tudjuk a transzformációkat végrehajtani (4.7). Az egymással függésben lévő lépések akkor válnak aktívvá, hogyha a bemenetük már előállt.



4.7. ábra. Gyormenü elérése a transzformációkhoz

Az akció végrehajtását követően két dolog történik. Először is megjelenik két *GammaModel* sztereotípiával ellátott osztály elem, a transzformált interfészeknek és a komponenst modelleknek illetve állapotgépeknek. Másfelől a *GammaWorkspace Gamma Statechart Model* és a *Gamma Interface Model* mezők ezekre fognak mutatni. A letranszformált modellek ezekhez a sztereotipizált osztályokhoz XMI formátumban kommentek formájában hozzá lesznek rendelve innen lehet őket *EMF Resource*okba visszaolvasni amennyiben erre szükség van, így nem kell újra letranszformálni a modelleket.

Az osztályok belsejében továbbá létrejön jó pár *property*. Ezek csomópont hivatkozások az XMI fájlok egyes elemeire. Belőlük pedig *Trace* élek futnak melyek MagicDraw elemekre mutatnak. Ez az a visszakövethetőségi modell melynek segítségével tudjuk, hogy melyik Gamma elem milyen MagicDraw elemekből állt elő. Ezt a 4.8 ábra szemlélteti, ahol a *Traced element* egy származtatott oszlop a *Trace* élek cél elemei.

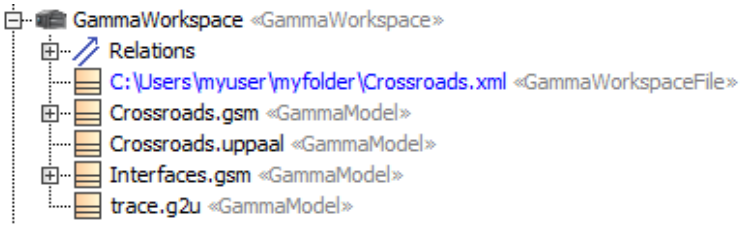
Következő lépés a az UPPAAL EMF speficikus modelljének előállítása ez ehhez tartozó *trace*kkel együtt amelyeket a Gamma majd a back-annotation előállításához használ majd. Ezeket a mostmár aktív "Transform models to UPPAAL" akció végrehajtásával tudjuk megtenni. Hasonlóan ez előbb szemléltetettekhez létrejön két sztereotipizált osztály egy az UPPAAL modellnek egy pedig az UPPAAL - Gamma *traceknek*. A modellek ugyan úgy mint az előbb XMI formátumban az elemekhez lesznek kapcsolva. Ebben az

#	Name	Traced element
1	//@components.0	TrafficLightCtrl
2	//@components.0/@ports.0	in control : ~Control
3	//@components.0/@ports.1	in police : ~PoliceInterrupt
4	//@components.0/@ports.2	out lights : LightCommands
5	//@components.0/@regions.0/@stateNodes.0	Interrupted
6	//@components.0/@regions.0/@stateNodes.0/@regions.0/@stateNodes.0	
7	//@components.0/@regions.0/@stateNodes.0/@regions.0/@stateNodes.1	BlinkingYellow
8	//@components.0/@regions.0/@stateNodes.0/@regions.0/@stateNodes.2	Black
9	//@components.0/@regions.0/@stateNodes.1	
10	//@components.0/@regions.0/@stateNodes.2	Normal
11	//@components.0/@regions.0/@stateNodes.2/@regions.0/@stateNodes.0	Red
12	//@components.0/@regions.0/@stateNodes.2/@regions.0/@stateNodes.1	
13	//@components.0/@regions.0/@stateNodes.2/@regions.0/@stateNodes.2	Green
14	//@components.0/@regions.0/@stateNodes.2/@regions.0/@stateNodes.3	Yellow
15	//@components.1	Controller
16	//@components.1/@ports.0	in policeInterrupt : ~PoliceInterrupt
17	//@components.1/@ports.1	out proprityControl : Control
18	//@components.1/@ports.2	out priorityPolice : PoliceInterrupt
19	//@components.1/@ports.3	out secondaryPolice : PoliceInterrupt
20	//@components.1/@ports.4	out secondaryControl : Control

4.8. ábra. Elemek visszakövethetősége MagicDrawban (részlet)

esetben nem fognak *property*k megjelenni hiszen ezek a modellek tulajdonképpen nem függnek a MagicDraw modelltől hanem funkcionálisan a Gamma részei.

Az eddigiekben nem volt szükség a háttértárra sorosítani, minden a MagicDraw modell belsejében került tárolásra. Azonban a következő lépés már használni fogja a behivatkozott könyvtárat is. Ez az utolsó lépés amit el kell végeznünk ahhoz, hogy olyan leírás álljon elő amit már az UPPAAL ellenőrzője be tud olvasni. Erre a *Persist UPPAAL XML akció szolgál*. Ez szintén létrehoz egy sztereotipizált osztály elemet ez azonban már nem a *GammaModel* hanem a *GammaWorkspaceFile* sztereotípiát fogja megkapni. Ez utal arra, hogy ez egy külső hivatkozás (4.9 ábra).



4.9. ábra. Workspace tartalma a transzformációk után

Léteznek bizonyos felhasználási módok, hogy ezeket a modelleket MagicDraw-n kívül szeretnénk használni. A modelleket az *Export models as .gcl...* akció végrehajtásával tudjuk kimenteni. Ezek a Gamma nyelvtanára fognak sorosodni. A korábban bemutatott interfészek például ilyen formában fognak megjelenni:

```
package Interfaces
interface LightCommands {
  out event displayNone
  out event displayRed
  out event displayGreen
  out event displayYellow
}
interface PoliceInterrupt {
  out event police
}
interface Control {
  out event toggle
}
```

4.3. Formális verifikáció

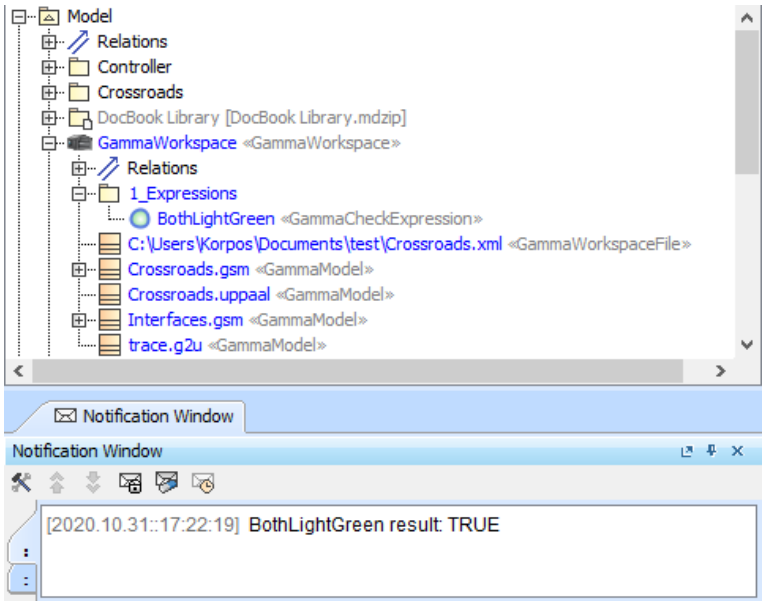
A letranszformált modellek mellet meg kell tudni fogalmazni az ellenőrizendő feltételeket is. Ehhez előbb létre kell hozni egy *Package*t a GammaWorkspaceben tetszőleges néven. Ebben tudunk létrehozni *GammaCheckExpression*öket. Ezek fogják leírni az ellenőrizendő tulajdonságait a modellnek. Erre a Gamma Property nyelvtanát tudjuk használni. Tegyük fel, hogy elő szeretnénk írni, hogy a két *TrafficLightCtrl* primary és secondary ne kerülhessenek egyszerre a "Green" állapotba. A kifejezést a *GammaCheckExpression* body mezőjébe meg kell megadni. Sajnos ennek a megadása kissé kényelmetlen, mert nincs *content assist* sem semmilyen támogató funkció.

```
E F [{state primary.NormalRegion.Green and state secondary.NormalRegion.Green}]
```

Az alábbi kifejezés azt írja le követelményként, hogy létezik-e út az állapottérben olyan állapotban, hogy mind két lámpa zöld. Ebben az esetben azt várjuk, hogy ez ne legyen igaz.

4.4. Eredmények kiértékelése

A *GammaCheckExpression*öket szintén a gyors menüben a *Gamma Transformation* alatt található *Execute* paranccsal tudjuk futtatni. Az eredményt a 4.10 ábrán láthatjuk.



4.10. ábra. Verifikáció futtatásának eredménye

Tehát a feltétel teljesül. Ezt az eredményt nem kielégítő hiszen azt szeretnénk, ha ez soha nem forduljon elő. Vizsgáljuk meg az példát amit kaptunk, hogy hogyan jutottunk el ebbe az állapotba (3.8 ábra). A végrehajtás a harmadik lépésig a megszokott viselkedést produkálja azonban a negyedik lépésben érkezik egy *police* szignál. Ennek határára a negyedik lépésben a két lámpa *Interrupted* állapotba vált. Ugyan ebben a lépésben érkezik még egy *police* szignál. Ezután az ötödik lépésben már hibás állapotba kerül a rendszer. Ennek az oka talán elsőre nem annyira látszik, de ha közelebbről megvizsgáljuk az állapotokat azt láthatjuk, hogy a harmadik, negyedik és ötödik lépésben is a *controller* a *Secondary* állapotba lép bele. Ennek az állapot belépéskor végrehajtja a *toggleBoth* (4.4 ábra) viselkedést, ami először a *priority* majd a *secondary* partoknak küldi el a *toggle* szig-

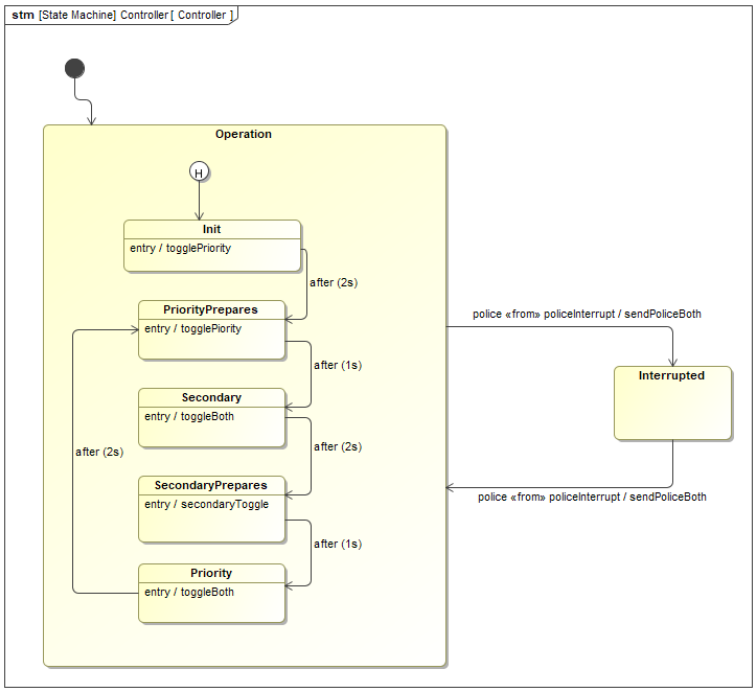
nálokat ebben a sorrendben. A negyedik lépésben *primary* piros a *secodnary* épp zöld így amikor a következő lépésben megérkezik előbb a *primary*ba a *toggle* szignál, így ő átlép a zöldbe még mielőtt a *secondary* váltana így teljesül a tulajdonság.

#	Instance States	Actions
1	controller.Operation controller.Init primary.Normal primary.Red secondary.Normal secondary.Red	
2	controller.Operation controller.PriorityPrepares primary.Normal primary.Green secondary.Normal secondary.Red	1.0. after 2000 1.1. schedule
3	controller.Operation controller.Secondary primary.Normal primary.Yellow secondary.Normal secondary.Red	2.0. after 1000 2.1. schedule
4	controller.Operation controller.Secondary primary.Normal primary.Red secondary.Normal secondary.Green	3.0. raise policeInterrupt.police 3.1. schedule
5	controller.Operation controller.Secondary primary.Interrupted primary.BlinkingYellow secondary.Interrupted secondary.BlinkingYellow	4.0. raise policeInterrupt.police 4.1. schedule
6	controller.Operation controller.Secondary primary.Normal primary.Green secondary.Normal secondary.Green	5.0. schedule

4.11. ábra. Verifikáció futtatásának eredménye

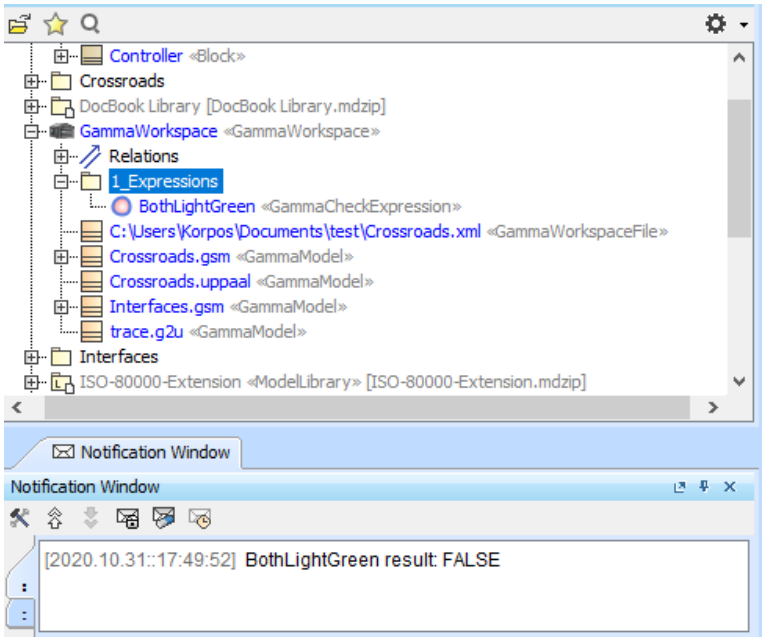
4.5. Modell javítása

A problémát úgy lehet kiküszöbölni talán a legkönnyebben, hogy a *Controller*-be is felveszünk egy olyan állapotot amikor várakozik (4.12 ábra). így már nem történik meg a többször egymás utáni szignál elküldés. Legalább is ezt feltételezzük.



4.12. ábra. Javított *Controller* állapottérképe

A transzformációkat újra végre kell hajtani, hiszen a modell megváltozott. Viszont ha újra futtatjuk a verifikációt akkor már a tulajdonság nem lesz igaz, tehát valóban sikerült javítani a modellt (4.13 ábra).



4.13. ábra. Verifikáció futtatásának eredménye a javított modellen

5. fejezet

Teljesítmény mérés, értékelés

Ebben a fejezetben megvizsgálom az eszköz teljesítményét és azt, hogy hogyan skálázódik a funkcionalitás a modell elemszámának növekedésével. Ez azért fontos, mert hiába van megoldásunk egy problémára, ha az a gyakorlatban az erőforrások korlátossága miatt nem képes időben eredményt adni. A két legfontosabb tényező az idő és a memóriafogyasztás. Megfelelő gyorsítótárak alkalmazásával az idő mértéke drasztikusan csökkenthető, viszont ez memóriába kerül. Ha a memória elfogy a szemét gyűjtő algoritmus agresszívabban próbálhatja összeszedni az elengedett objektumokat, ez szintén megnövelheti a végrehajtási időt, hiszen a rendszer szemétgyűjtéssel és nem pedig a működés elvégzésére használja az erőforrásokat.

A plugin VIATRA-t használ mégpedig inkrementális konfigurációval. Ez azt jelenti, hogy minták inicializációjuktól folyamatosan figyelik a modell változásait és módosítják a lekérdezések eredményhalmazát. Ez szintén memóriát vesz igényben viszont az elemek lekérdezése egy táblázat sorának kiolvasásának megfelelő komplexitással bír és nagyon gyors. Azt, hogy mennyire nő nagyra a VIATRA memória igénye leginkább a minták befolyásolják ezért különösen fontos, hogy ezek minél hatékonyabbak legyenek, mert nagy modell esetén nagyon nagy memória fogyasztást eredményezhetnek.

5.1. Módszertan

A teljesítmény mérés során az egyes transzformációk futási idejét és a VIATRA memória fogyasztását mértem. Magát a formális verifikációt nem vizsgáltam, mert az főként az UPPAAL teljesítményétől függ. Továbbá nem foglalkoztam különösebben az állapotgépek transzformációjának mérésével, mert ezt már vizsgálva volt a plugin korábbi változatában. A mérésekkel a kompozíciók skálázhatóságát vizsgáltam főként.

A méréseket a VisualVM[1] nevű java profilozó eszköz segítségével végeztem. A futtatást mintavételezés segítségével profiloztam. Ez a módszer kevesebb beavatkozással jár a tényleges teljesítményt tekintve a futtatásba, viszont nem szolgáltat teljesen pontos képet, például a metódus hívások pontos számának megállapítására nem alkalmas. A másik lehetőséghez az instrumentális profilozás ami, nagyobb *overhead*del jár viszont pontosabb képet adhat.

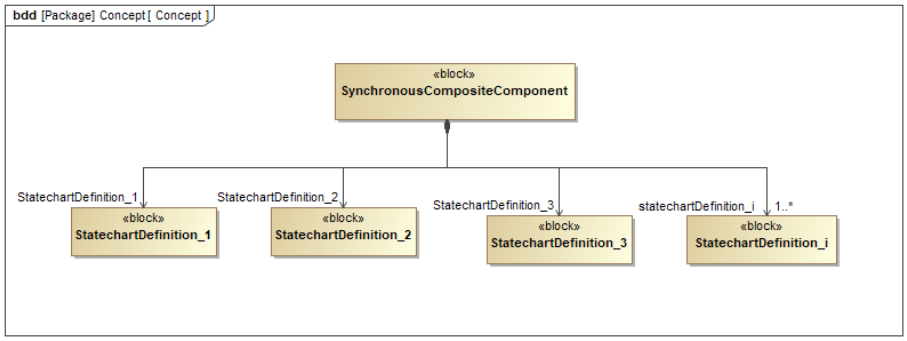
A mérések a következő specifikációk mellett lettek elvégezve:

MagicDraw verzió	19
Heap méret	10G
Operációs rendszer	Windows 10
CPU	Intel i5-10600K

[1] https://visualvm.github.io/

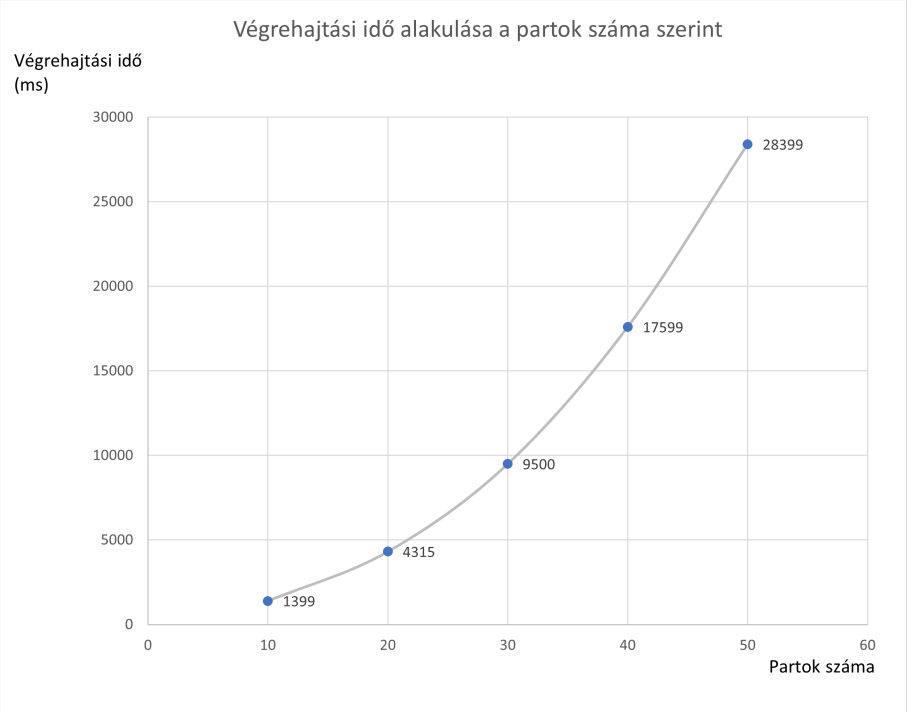
5.2. Végrehajtási idő változása az elemek számával

Az első mérés során azt vizsgáltam, hogy a partok és az őket tipizáló blokkok számának növelésével milyen mértékben növekszik a végrehajtási idő. Ehhez készítettem egy pár állapotból és címkézetlen átmenetekből álló állapotgép definíciót és ezt másolgatva növeltem a állapotgép definíciók és az velük tipizált partok mennyiségét (5.1 ábra).



5.1. ábra. Méréshez használt modell

A méréseket tíz, húsz, harminc, negyven és ötven part méretek mellett végeztem el. Az eredmények kezdetben elég lesújtóak voltak, nem árulkodtak túl jó skálázhatóságról és a végrehajtási idők is nagyon magasak voltak. A mérési eredmények az 5.2 ábrán láthatók.



5.2. ábra. Első mérés eredményei

Behatósabban vizsgálva a méréseket kiderül, hogy a legtöbb időt nem magának a transzformációnak a végrehajtása hanem az elemek visszakereshetőségét biztosító *trace* modellek létrehozása emészti fel [2]. Ez jelen esetben minden parthoz 18 *property* létrehozását jelenti a modellben. A legnagyobb modell esetében 900 modell *property*t kell létrehozni.

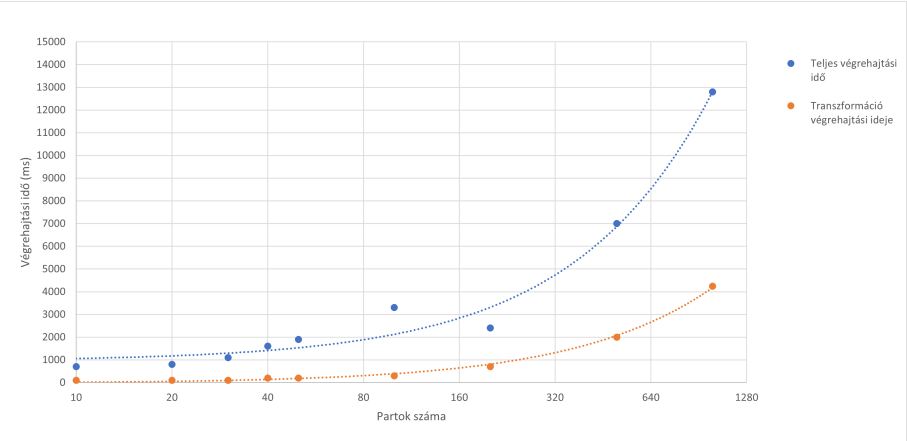
[2]Ez a visualvm-ben a modell session lázárásaként jelenik meg, ekkor véglegesülnek a modellen végzett módosítások

Ennek fényében úgy tűnik, hogy a *trace* modell ilyen módon történő tárolása nem a legjobb mérnöki megoldás.

Mivel a lassulás a MagicDraw működéséből keletkezik megpróbáltam másféle modell elemeket létrehozni *property*k helyett remélve így kisebb lesz az elemek létrehozásának a működésre gyakorolt hatása. *Property*k helyett az EMF modellt szimbolizáló linkeket *Class* példányokra cseréltem és újból elvégeztem a mérést. Az eredmények jelentős teljesítmény javulást mutatnak melyet az alábbi táblázat és az 5.3 ábra mutat.

Partok száma	10	20	30	40	50	100	200	500	1000
Teljes futási idő	699	798	1098	1599	1899	3300	2400	7000	12799
Transzformáció	100	99	98	199	200	301	700	2000	4240

A diagramon a függőleges tengely logaritmikus. A diagramról megállapítható, hogy a végrehajtási idő lineárisan változik a *part*ok darabszámának növekedésével. Ugyanakkor az is látszik, hogy ugyan a korábbi implementációkhoz képest sokkal kevésbé jelentős a *trace* modell létrehozásának az ideje, de az elemszám növekedésével rosszabban skálázódik, mint maga a transzformáció, így kellően nagy modelleken ezzel a megoldással is dominálni fog.



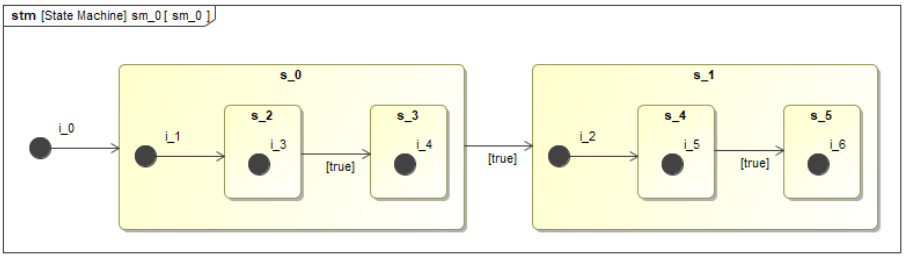
5.3. ábra. Végrehajtási idők *Class*ok használatával

Értékelés: A mérések alapján az derül ki, hogy a *trace* modellek az ismertetett módon történő tárolása nagyon erős mértékben rontja az eszköz transzformációs teljesítményét. Egy lehetséges alternatíva egy EMF alapú modell létrehozása lehetne, amit szöveges formában le lehetne tárolni egy modell elemben - hasonlóan az Gamma állapottérképek XMI formában történő tárolásához - és ezzel összekapcsolni a két modellt.

5.3. Memória igény változása az elemek számával

A plugin projekt betöltésekor inicializálja a VIATRA mintákat, melyeket a transzformáció használ. Ez jelentősen meg tudja gyorsítani magát a transzformációt és azokat a validációs szabályokat melyek használják, viszont a találatok tárolási tárigényes. Ráadásul első induláskor be kell indexelni a modellt, ami megnöveli a projektek betöltésének idejét is. Mivel a tesztek során ebből fakadó problémát nem észleltem az indexelési időt nem vizsgáltam.

A memória használat méréséhez generált modelleket használtam. Minden állapotgép ”n” hosszú állapot láncokból állt plusz egy kezdő állapotból. Az állapotok további ”n” hosszú állapotláncokat tartalmaznak, melyek ismét tartalmaznak ”n” hosszú láncokat és így tovább. Minden állapotátmeneten egy *true* őrfeltétel van (példa: 5.4 ábra).



5.4. ábra. Generált állapotgép 12+1 elemmel és 2 hosszú láncokkal

Minden végzett méréshez egy azonosítót rendeltem *m-n-k* alakban. A mérések azonosítójában szereplő számpár (n és k) a generált modell paramétereire utalnak. Az első szám a csomópontok száma (állapotok és kezdőállapotok) a második szám pedig a láncok hosszát jelenti. Fontos, hogy az állapotok egymásba ágyazottságának a mértéke is növelheti a VIATRA által karbantartott táblák számát. Az alábbi minta például tranzitív lezárt segítségével adja vissza egy állapotgép régióinak számát.

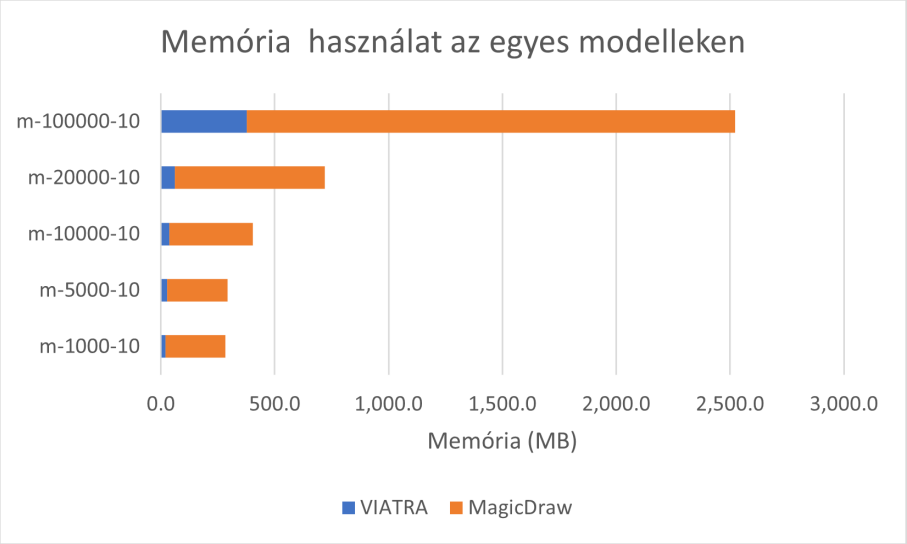
```
pattern RegionsInStatechart(stmt: StateMachine, region: Region){
   find MainRegions(stmt, region);
} or {
   find MainRegions(stmt, outerRegion);
   find InnerRegion+(outerRegion, region);
}
```

A beágyazottság mértéke az elemszám és a láncok hosszának hányadosa.

A mért értékeket a következő táblázat tartalmazza. A MagicDraw által használt memória méretét az indexelés előtt mértem.

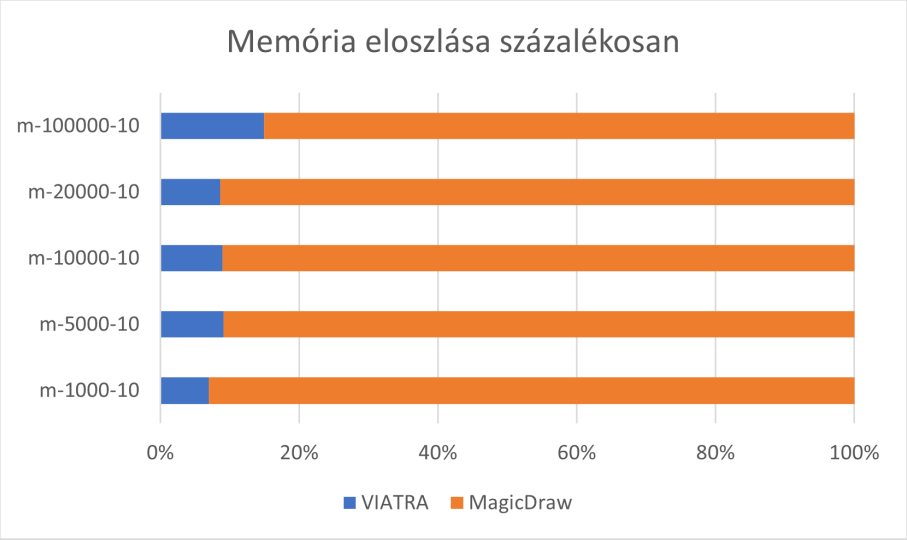
azonosító:	m-1000-10	m-5000-10	m-10000-10	m-20000-10	m-100000-10
kezdő állapotok:	111	1111	1111	8891	11111
állapotok:	890	3890	8890	11110	88890
régiók:	111	1111	1111	8891	88890
átmenetek:	890	3890	8890	11110	88890
VIATRA (MB)	19,8	26,7	36,3	62,2	377,4
MagicDraw (MB)	264	267	368	657	2144,5

Az 5.5 ábrán az egész heap látható. Ebből az látszik, hogy ugyan a legnagyobb modellen már egészen nagy helyet foglal a VIATRA. A beágyazottság viszont ezen a modellen már egész magas, öt mély. A gyakorlatban egy állapotgépen az állapotok túlzott egymásba ágyazottsága nem annyira valószínű, hiszen ezzel együtt a komplexitása is nagyon magas lesz a modellnek ami általában kerülendő.



5.5. ábra. Heap mérete és eloszlása

A meómria használat eloszlását picit részletesebben nézzük (5.6 ábra) akkor azt látjuk, hogy a legnagyobb és a legkisebb modell között a VIATRA nagyobb arányban használja a memóriát majdnem kétszer annyit. Ennek két fő oka lehet. Az egyik, hogy a mérések



5.6. ábra. Heap méretének százalékos eloszlása

során a MagicDraw teljes memóriahasználatát néztem a mibe beletartoznak az importált modellek is mint például a SysML projekt és a Gamma profil is. Ez szám ugyan állandó, de a kisebb méreteken dominálhat, míg a nagyobb méreteken már az a generált állapogép mérete lesz a meghatározó. A másik a már említett egymásba ágyazottság növekedésének mértéke.

Értékelés: a mérések során nem tapasztaltam kiugró mértékű teljesítmény csökkenést, ami azt jelenti, hogy a vizsgált minták között nem volt olyan, ami nem lett elég jól megírva. Ugyanakkor a közel 18%-os eredmény a legnagyobb modellen indokolttá teheti, hogy nagy modelleken a VIATRA ne fusson folyamatosan, hanem ez kikapcsolható legyen.

6. fejezet

Összefoglalás