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

MagicDraw állapottérképek formális verifikációja

Szakdolgozat

Készítette
Gáti László Dávid

Konzulens
Farkas Rebeka

2018. december 6.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2018. december 6.

Gáti László Dávid
hallgató

Kivonat

Komplex rendszerek tervezése során szükségessé válik fejlett modellező eszközök különböző funkcióinak igénybevétele, pl. validáció, kódgenerálás, verifikáció. Az iparban az egyik legelterjedtebb eszköz a MagicDraw ami lehetőséget biztosít komplex rendszerek tervezésére és ezek viselkedésének modellezésére. MagicDraw-t gyakran használnak kritikus rendszerek tervezésére, melyek esetén elengedhetetlen, hogy megbizonyosodjunk a rendszer működésének helyességéről.

A MagicDraw számos hasznos funkcióval rendelkezik, azonban a formális verifikációt nem támogatja. Az ipari alkalmazhatóság szempontjából viszont jelentős előnyökkel járna ezeknek a módszereknek a támogatása. A Gamma egy, a tanszéken fejlesztett modellező eszköz, amely nem rendelkezik a MagicDraw-éhoz hasonlóan kiterjedt eszközkészlettel, azonban viselkedésmodellek formális verifikációját támogatja. A munkám során MagicDraw modellek formális verifikációját azáltal teszem lehetővé, hogy megvalósítok egy leképzést a két eszköz modelljei között.

Dolgozatomban bemutatom a Gamma és MagicDraw eszközöket és a leképzés megvalósításához szükséges háttérismereteket. Továbbá részletesen kitérek az implementáció során használt technológiákra és elvekre. A módszert egy példán szemléltem és elméleti megfontolások alapján értékelem. A bemutatott eszköz lehetővé teszi modellek egy tágabb halmazának formális verifikációját.

Abstract

The development of complex systems makes it neccessary to gain access to modern modelling tools that support functionalities like validation, code generation and verification. One of the most well-known tool in the indsutry is MagicDraw: a tool for desinging systems and to model their behavior. MagicDraw is often used during the development of fault tolerant systems where it is mandatory to ensure that the system operates correctly.

Although MagicDraw is very rich in functionalities it lacks the ability to perform formal verification. The use of such methods would be highly beneficial for industrial use. Gamma is a modelling tool developed at the Department which is not as extensive in functions as MagicDraw is but it does support the verification of behavioral models. In my work I supporting formal verification of MagicDraw by creating a transformation between models of the two tools.

In this thesis I introduce Gamma and Magicdraw tools and the background knowledge needed to understand the transformation method. Furhtermore I describe in details the concepts and techologies used in my implementation. I describe the method via an example and I evaluate the results based on theoretical considerations. The presented product enables the formal verifications on a wider set of models.

1. fejezet

Bevezetés

Nagyméretű és komplex rendszerek tervezése során szükségessé válik, hogy olyan fejlett modellező eszközök álljanak rendelkezésünkre, melyek nem csupán gördülékenyebbé teszik a modell alapú fejlesztés menetét, de funkcióikkal lehetővé teszik rendszerek átfogó vizsgálatát: *validációját* azaz szintaktikai ellenőrzését, illetve *verifikálását*: a működésének helyességének ellenőrzését. A modellekből lehet kódot generálni, ami meggyorsítja a fejlesztést és csökkenti az implementáció során vétett programozói hibák számát.

Modelleket érdemes rajzok, diagramok segítségével definiálni, ezek ugyanis könnyebben megérthetők és hibákat is könnyebb észrevenni, mint egy kód alapú leírás esetében. A legelterjedtebb diagram alapú modellezési nyelvek az UML és a SysML melyek absztrakciók alkalmazásával rendszerek implementációtól független leírását teszik lehetővé. Ezek vagy hasonló modellezési nyelvek alkalmazása egy ipari projekt esetében kiemelkedően fontosak, ugyanis lehetővé teszik a rendszerek elemzését hibák felderítése céljából. Ezek felfedezése és kiküszöbölése még a tervezési fázisban nem jár jelentős többletköltségekkel, hiszen nem kell az implementációt megváltoztatni, termékeket visszahívni egy esetleg utólag felfedezett tervezési hiba miatt.

Az iparban egy széles körben alkalmazott eszköz a MagicDraw, ami többek között UML és SysML nyelveken teszi lehetővé modellek készítését. Ezekkel magas szinten leírható egy rendszer felépítése és működése. MagicDraw-t gyakran alkalmaznak kritikus rendszerek modellezésére melyek esetében elengedhetetlen, hogy matematikai precizitással megbizonyosodjunk a működés helyességéről, hiszen egy meghibásodás komoly anyagi veszteségekkel vagy akár ember életek veszélyeztetésével járhat. Bár a MagicDraw már sok hasznos funkcióval rendelkezik, formális verifikációt végrehajtására még nincs lehetőség. Az ipar szempontjából jelentős előnyökkel járna viselkedésmodellek formális verifikációjának támogatottsága.

A Gamma egy tanszéken fejlesztett állapottérképek modellezésére és ezek fejlesztésének támogatására készített eszköz, ami elsősorban Eclipses környezetekben alkalmazható. Bár a Gamma eszközkészlete nem annyira kiterjedt mint a MagicDraw-é, de az állapottérképek formális verifikációja a funkcióinak részét képezi. A Gamma saját nyelvet alkalmaz az állapottérképek leírására ami kifejezetten úgy lett megalkotva, hogy támogassa más nyelvek leképezhetőségét például Yakinduét. Dolgozatomban állapotgépek formális verifikációját azáltal teszem lehetővé, hogy egy leképzést valósítok meg a MagicDraw és a Gamma modelljei között és a Gamma formális verifikációhoz kötődő funkcióit MagicDraw-n belül is végrehajthatóvá teszem.

Hasonló munka például maga a Gamma hiszen az eszköz már megvalósít egy leképzést saját nyelvére: a már említett Yakindu - Gamma leképzést. A különbség, hogy a Gamma metamodellje és szemantikája közelebb áll a Yakinduéhoz, mint a SysML állapottérképeihez: pl. felhasználhatók olyan elemek is amik Gammában nem részei. Másfelől a Yakindu

és a Gamma is Eclipses környezetekhez lettek fejlesztve még a MagicDraw önmagában nem függ az Eclipsetől. Bár létezik MagicDraw - Eclipse integráció felhasználói szempontból előnyös lenne, az eszközt önmagában lehetne használni.

A dolgozat feléptése a következő: *a 2. fejezetben* bemutatom a munkám megértéséhez szükséges alapismereteket és pár létező eszközt ami képes állapottérképeket verifikálni. *A 3. fejezetben* ismertetem az alkalmazott megoldást elméleti és gyakorlati szempontból, továbbá egy esettanulmányon szemléltetem az eszköz működését. *A 4. fejezetben* értékelem az alkalmazott megoldásokat és mérésekkel ellenőrzöm a teljesítményt, továbbá bemutatom azokat a lehetőségeket amikkel az eszköz kiegészíthető, végül *az 5. fejezetben* összefoglalom a dolgozatban leírtakat.

2. fejezet

Háttérismeretek

Ebben a fejezetben bemutatom a dolgozat megértéséhez szükséges háttérismereteket, köztük az állapottérképek formalizmusát és azokat a technológiákat amiket a feladat végrehajtásához használtam.

2.1. Kapcsolódó munkák

Legjobb tudomásom szerint még nem létezik olyan eszköz ami MagicDrawban képes formális verifikációt végrehajtani. Felhasználhatóság és megvalósítás szempontjából hasonló munkák közül az alábbiakat érdemes kiemelni.

2.1.1. Gamma Statechart Composition Framework

A Gamma Statechart Composition Framework[6] egy tanszéken fejlesztett eszköz, ami komponenst alapú viselkedés modellek fejlesztését teszi lehetővé olyan funkciókkal támogatva, mint a kód generálás és a formális verifikáció. Továbbá képes Yakindu modelleket áttranszformálni Gamma modellekké, melynek a megvalósítása hasonló mint az általam implementált MagicDraw-Gamma transzformáció. A Gamma keretrendszerrel a 2.8 szakasz foglalkozik részletesebben.

2.1.2. TismTool

TismTool[9] egy többszálú, komponens alapú rendszerek fejlesztését támogató eszköz, ami képes UML modellekből kódot generálni különböző nyelveken (C#, Java, C++ vagy C) és ezeket verifikálni. Az eszköz képes különböző UML modellező eszközök modelljeit feldolgozni, az egyetlen megkötés, hogy ezek képesek legyenek az UML model XML Metadata Interchange (XMI) szabvány szerint modellt sorosítani: a MagicDrawra ez teljesül.

Az eszköz azáltal végzi el a verifikációt, hogy kódot generál az XMI fájlokból amit egy futtatókörnyezettel kell végrehajtatni, továbbá nem formális alapokon hajtja végre a verifikációt, hanem ún. futás idejű verifikációt[1] használ.

2.2. Modellek

Modelleket a tudomány számos területén alkalmazunk. Ez lehetővé teszi a megvalósítandó rendszer vizsgálatát még azelőtt, hogy azt ténylegesen létre kellene hozni. A vizsgálatoknak számos módja és célja lehet. A látványtervező bemutat egy látványtervet és a megfelelő *stakeholderek* ezt értékelik, vagy egy rendszerről komplex matematikai módszerekkel kell

[1] http://www.tismtool.com/tooling.html

eldönteni, hogy stabil-e. A modellek leírása sokféle módon történhet, de célszerű egy olyan standardizált jelölési rendszert alkalmazni, hogy a többi szakember is értelmezni, vizsgálni tudja a modellt.

A jelölési rendszer vagy modellezési formalizmus lehet egy szöveges leírás is, például egy programkód, de sokszor célszerű vizuális megoldást használni, szimbólumokat tartalmazó diagramokat alkalmazni. Ezek sok esetben kifejezőbbek, lényegre törőbbek, és elsősorban könnyebben értelmezhetőek mint a szöveges leírások.

2.3. Állapot alapú modellezés

Az állapottérkép egy diagram ami irányított gráfot tartalmaz, ahol a csomópontok állapotokat, az élek állapotátmeneteket definiálnak. Az így kapott gráfot állapotgépnek nevezzük. Az állapotátmenetek állapotok közötti lehetséges átjárásokat definiálnak és általában feltételhez kötöttek, ami jellemzően valamilyen esemény bekövetkezése vagy logikai feltétel teljesülése. A feltételek teljesülésekor az állapotváltás bekövetkezik, ezt szokás *tüzelésnek* is hívni. Az állapotgép legfőbb jellemzője, hogy adott időpillanatban melyik állapotok aktívak. Állapotváltások aktív állapotból történnek, ilyenkor a kiindulási állapot inaktívvá válik a célállapot pedig aktívvá (a kiinduló és célállapot megegyezhet).

A működés kezdetekor az első aktív állapotot kezdő állapotnak hívják, ez egy különleges ún. pszeudoállapot, ami az állapotgép belépési pontja és általában azonnal átléptetésre kerül egy másik állapotba.

2.3.1. Állapottérképek UML 2-ben

Előzőekben az állapot alapú modellezés alapjai kerültek bemutatásra. A következőkben az állapottérképek egy konkrét specifikációja kerül bemutatásra az UML 2 szerint.

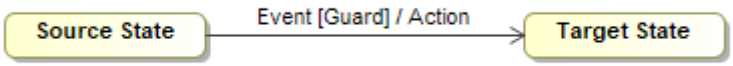
- *Állapot (State)*: az állapottérképek csomópontjai, melyeket UML 2-ben is állapotoknak hívnak. Az állapotok rendelkeznek:
  - név: az állapot neve
  - be/kilépési akció: be és kilépés során végrehajtandó cselekvés.
- *Kezdő állapot (Initial State)*: pszeudoállapot, régiónként egy szerepelhet belőle és a régió belépési pontjaként szolgál. Jelölése fekete színezett kör (2.2 ábra).
- *Végállapot (Final State)*[2] (final state): A végső állapot, a régió terminálasi pontja, ha egy állapotgép összes régiója egy végső állapotba ért akkor az állapotgép is terminál. Szimbóluma fehér körben egy kisebb színezett fekete kör (2.2 ábra).
- *Termináló állapot (Terminal State)*: pszeudoállapot, ami az egész állapotgépet azonnal terminálja. Ezt hibák lekezelésére lehet például alkalmazni, jelölése kis kereszt (2.2 ábra).
- *Állapot átmenet (Transition)*: az állapottérképek élei, a lehetséges állapotváltozásokat definiálják. Az állapotátmenetek rendelkeznek:
  - Triggerekkel
  - Őrfeltételekkel
- *Trigger*: Állapot váltást kiváltó esemény amely lehet:

[2]UML 2-ben a végső állapotot nem pszeudoállapotként hanem állapotként van definiálva https://www.omg.org/spec/UML/2.0

- Változás esemény (Change Event): valamely változó értéknek a megváltozása.
- Üzenet esemény (Message Event): üzenet típusú objektumnak az érkezése, ami ebben a kontextusban kérésnek felel meg. Az ilyen típusú kommunikáció kétféle eseménytől függ, az üzenet elküldésétől és annak küldőjétől és az üzenet fogadásától és fogadójától. A kérés lehet egy metódus hívás vagy egy jel (*Signal*) fogadása.
- Időzítés esemény (Time Event): idő változásához kötött esemény.

Fontos megjegyezni, hogy a dolgozat nem tér ki részletesen az események kiváltásának kérdésére, valamint az események küldésénél és fogadásánál szerepet játszó portokra, és interfészekre, ezen elemek a dolgozat szempontjából irrelevánsnak tekinthetők.

- *Őrfeltétel (Guard)*: tágabb értelemben egy logikai kifejezés, melynek teljesülnie kell, hogy az adott állapotátmenet bekövetkezhessen. UML-ben ezek megszorírásként *(Constraint)* vannak értelmezve, ebben az értelemben a megszorításnak való megfelelés az állapotváltás feltétele.
- *Akció*: Különböző események bekövetkezésekor, mint állapotváltások, belépés állapotokba, kilépés állapotokból, vagy maga az állapotban maradás, lehetőségünk van viselkedéseket végrehajtani. UML szerint ezek lehetnek: Activityk, Állapotgépek, Interakció[3] OpaqueBehavior[4] Állapotváltáskor, állapotba való be-, kilépéskor a viselkedés végrehajtódik, míg állapotban maradáskor addig hajtódik végre, amíg az állapot aktív, egyébként megszakításra kerül.



2.1. ábra. Állapotok és köztük definiált állapotátmenet, triggerrel, őrfeltétellel és actionnel

Gyakran előfordul, hogy általánosabb állapotot célszerű felbontani részállapotokra. Az egyszerű állapottérképek elemeivel, ez a fajta hierarchikus viszony az állapotok között nehezen ábrázolható, ezért célszerű további elemek használata.

- *Régió*: állapotokat tartalmazó egység, az állapottérkép mindig tartalmaz egy régiót amibe az állapotok definiálhatók. Régiók létezhetnek egymással párhuzamosan ilyenkor a végrehajtásuk párhuzamosan történik.
- *Összetett állapot(Composite State)*: ha az állapotnak vannak további belső állapotai is, ha az állapot aktív akkor legalább egy belső is aktív, ha az állapotgép egy állapotváltás hatására kilép a kompozit állapotból akkor a belső állapotokból is kilép.
- *History State*: olyan pszeudo állapot, amely egy régióban megjegyzi az utolsó aktív állapotot kilépéskor. A régióba visszalépve a history state visszaállítja a megjegyzett állapotot. Amennyiben nincs előző állapot az ő belőle húzott állapotátmenet cél állapota lesz aktív. Kétféle History Statet különböztetünk meg Shallow és Deep Historyt. Előbbi csak adott régión belül jegyzi meg az állapotot míg utóbbi a tartalmazott régiók állapotait is megjegyzi és visszaállítja.

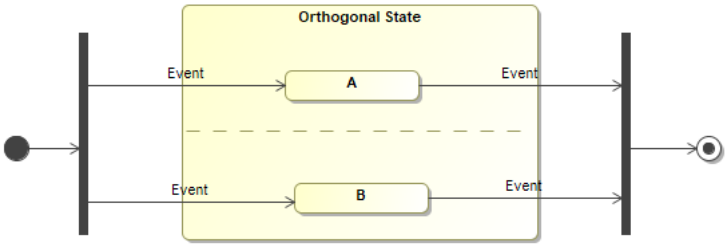
  A *HistroyStatek* szintaktikája a 2.2 ábrán látható.

2.2. ábra. Kezdőállapot, DeepHistory, ShallowHistory, Termináló állapot és Végállapot

Rendszerünket egy időpillanatban több egymástól független állapot is jellemezheti. Ezt a viselkedést párhozamos régiók alkalmazásával lehet leírni.

- *Ortogonális állapot (Orthogonal State)*: olyan összetett állapot ami két vagy több régiót tartalmaz.
- *Fork*: pszeudoállapot, ami egy beérkező átmenetet szétbont több átmenetre, amiknek a cél állapotuk ortogonális régiókban találhatók. A kimenő átmeneteken nem lehet se trigger, sem pedig őrfeltétel.
- *Join*: pszeudoállapot, ami több beérkező átmenetet kapcsol össze eggyé. Az átmenetek ortogonális régiókból kell, hogy induljanak és nem lehet rajtuk trigger vagy őrfeltétel. A join szinkronizációs funkcionalitással bír: addig nem lehet tovább lépni belőle amíg minden beérkező átmenet végre nem hajtódott.

  Fork/Join alkalmazásával ki tudjuk kényszeríteni, hogy részrendszereink elérjenek egy adott állapotot, mielőtt a végrehajtás folytatódhatna. Jelölésük a 2.3 ábrán látható.



2.3. ábra. Példa: fork-join

Rendszereink leírásakor előfordulhatnak ismétlődő részek, amiket célszerű egyszer leírni és újra felhasználni.

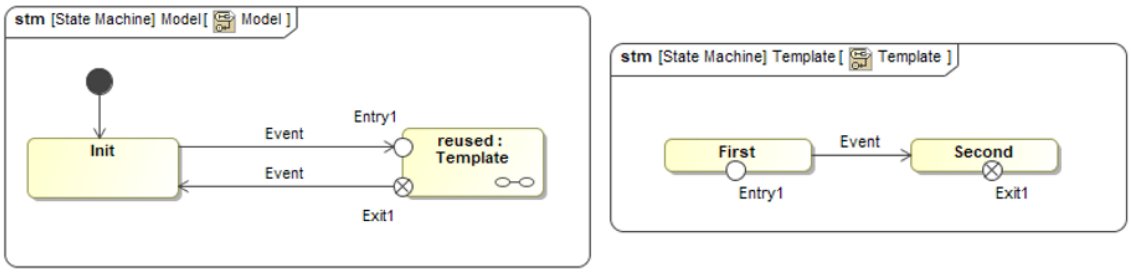
- *Submachine state*: Egy olyan állapot, ami egy állapottérképre hivatkozik, ez lehetővé teszi, hogy egy állapottérkép többszöri felhasználását, akár más-más kontextusban
- *Belépési pont(Entry Point)*: egy pszeudoállapot, ami egy állapotgép vagy egy kompozit állapot belépési pontját reprezentálja, célja egységbe zárni az állapotot vagy az állapotgépet. Továbbá léteznie kell egy állapotátmenetnek közte és egy az állapot vagy állapottérkép fő régiója között. Jele kis fehér kör.
- *Kilépési pont(Exit point)*: mint a belépési pont, de ez kilépési pontot reprezentál, jele kis fehér kör áthúzással.

[3]Interakció modell elemek között, leírásához a jellegétől függően többféle diagram használható(Szekvencia, Kommunikációs, Időzítés).

[4] szöveges, UML-től eltérő nyelvvel specifikált viselkedés.

- *Kapcsolódási pont referencia(Connection Point Reference)*: *Submachine Stateben* definiált be és kilépési pontokra tudunk vele hivatkozni, ez lehetővé teszi, hogy a *Submachine Stateben* leírt belső állapotokhoz is felvehessünk állapotátmeneteket.

Egy állapottérképen a belépési és kilépési ponttal élek lehetséges kezdő illetve végpontjait tudjuk definiálni. Újrafelhasználásnál a behivatkozott állapottérképen ezekre referálhatunk Kapcsolódási Pont Referenciákkal. Az ezekbe húzott állapotátmenetek úgy tekintendők mintha kezdő vagy végpontjuk az az állapot lenne amihez a belépési vagy kilépési rendelve van. A *Submachine State* szintaktikáját és használatát a 2.4 ábra szemlélteti.



2.4. ábra. Állapottérkép újrafelhasználása Submachine State segítségével

Logikailag állapotátmenetek is összetartozhatnak, ezért célszerű bizonyos esetekben egyesíteni, vagy szétbontani őket több alternatív átmenetre.

- *Csomópont (Junction)*: pszeudoállapot, több állapotátmenet összekapcsolása és egyként kezelése, például ha a cél állapotuk ugyan az és logikailag összetartoznak vagy egy beérkező átmenet szétbontása több átmenetre. Ilyenkor lehetőség van őrfeltételt rakni az átmenetekre, ezeknek a kiértékelése viszont még azelőtt történik, hogy bármelyik átmenet végrehajtásra kerülne, ezért egy ilyen ágat szokás *statikus feltételes ág*nak nevezni.

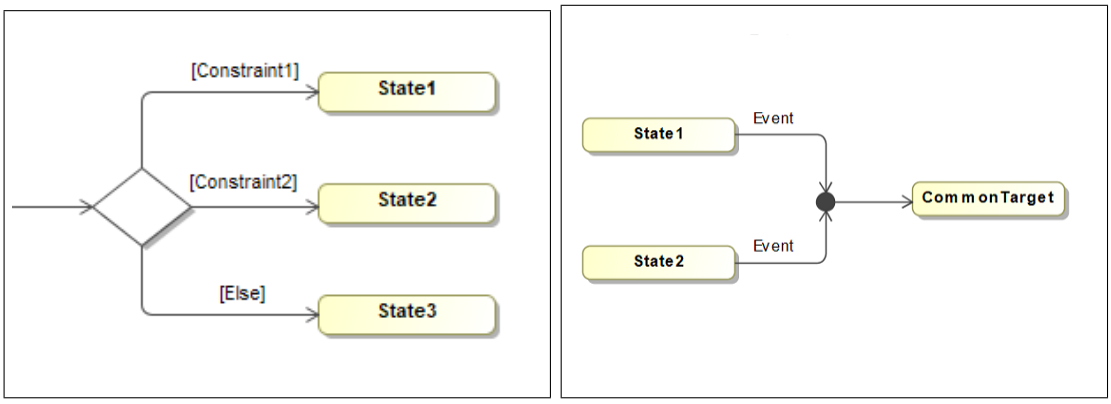
- *Döntés (Choice)*: hasonló mint a csomópont, viszont az őrfeltételek az elágazásba való belépéskor értékelődnek ki *dinamikus*an. Ezt jellemzően alternatív útvonalak megadására használjuk, hasonló mint a programozási nyelvekben *"if than else"* elágazás. Fontos megjegyezni, hogy az elágazás és a csomópont esetében is lehetséges, hogy több átmenet is tüzelhetne, ilyenkor az érvényre jutó átmenet kiválasztása nem determinisztikus módon történik ezért elkerülendő, például *else* őrfeltétel alkalmazásával.

  A csomópont szintaxisa egy a kezdő állapotnál kisebb színezett kör, a döntésé pedig egy rombusz. (2.5 ábra).

2.4. Eclipse Modelling Framework

Az Eclipse egy nyílt forráskódú platform független keretrendszer és fejlesztőkörnyezet, ami kiterjeszthető, hogy Java mellett más programnyelveket(pl. C, PHP, Ruby, Python, Erlang) is támogasson, de modellező eszközként is használható (pl. Yakindu, Papyrus, Gamma).

Az Eclipse Modelling Framework (EMF) Eclipse plug-inok egy halmaza, amik lehetővé teszik adatmodellek létrehozását és ebből kód generálását. Az EMF kétféle modellt különböztet meg a metamodellt és példány modellt. A példány modell struktúráját a metamodell írja le. A modell egy konkrét példánya a metamodellnek.



2.5. ábra. Döntés és Csomópont szintakitkája

Az Eclipse Modelling Framework működése meglehetősen komplex ezért részletes bemutatása nem célja a dolgozatnak (szakirodalom a keretrendszer kapcsán: *EMF: Eclipse Modelling Framework*[7]), csak azoknak az alap fogalmakat és működési mechanikáknak az ismertetésre amik szükségesek a dolgozat megértéséhez.

Egy EMF metamodell `ecore` és egy `genmodel` leíró fájlból áll. Előbbi magát a modellt utóbbi pedig a modell generálására vonatkozó információkat tartalmaz. Az *EMF Persistence Framework* lehetővé teszi modellek perzisztens tárolását XMI és XML alapokon. A fájlrendszerben található modelleket *Resource*-ok reprezentálnak melyeket egy URI séma azonosít. A *Resource*-ok *ResourceSet*-ekbe helyezhetők. A modellek közötti hivatkozások feloldásához a modelleket tartalmazó *Resource*-oknak azonos *ResourceSet*-ekben kell lenniük.

Eclipses környezetekben az EMF-nek nagy jelentőséggel bír. Xtext[5]-el együtt alkalmazva saját szakterület specifikus nyelveket lehet fejleszteni saját nyelvtannal és absztrakt szintaxissal.

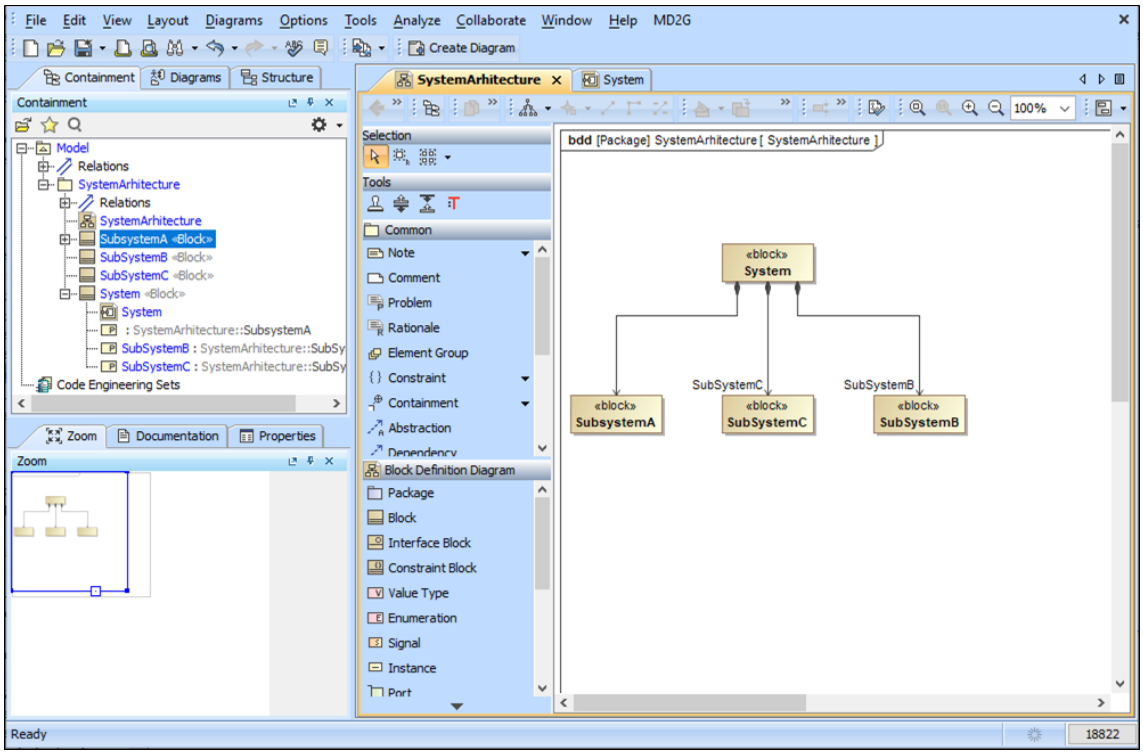
2.5. MagicDraw

A MagicDraw a No Magic [4] nevű cég által fejlesztett modellező eszköz, amivel a modellek előállításán kívül lehetőségünk van ezeket szimulálni, validálni vagy akár kóddá alakítani. Az eszköz első sorban UML modelleket lehet készíteni, de plug-innal lehetőségünk van SysML [3] modelleket is létrehozni. Ehhez egy az eszköz fejlett grafikus felhasználói interfészt biztosít (2.6. árba), amivel gyorsan és hatékonyan tudunk modelleket létrehozni.

A SysML egy általános-célú modellezési nyelv ami az UML egy részének kiragadásával és annak kibővítésével keletkezett. SysMLel struktúrát és viselkedést lehet leírni magas szinten. Alapeleme a Blokk, ami UML-ben a Classnak felel meg. A Blokk egy absztrakt egység ami bárminek megfeleltethető, így a modellezendő rendszer is általában egy Blokként jelenik meg. A Blokkok definícióját és tartalmazási hierarchiáját Block Definition Diagramokkal írhatjuk le. Egy másik megemlítendő diagramfajta az Internal Block Diagram, amivel a Blokkok belső felépítését és komponenseinek kapcsolatait lehet leírni portok segítségével.

Viselkedést Állapottérképekkel és Activity Diagrammokkal szokás leírni SysMLben. Utóbbival munka- és adatfolyamokat, ahol a folyamat lépései *Activity*k és *Action*ök. Állapottérképekkel reaktív rendszereket szokás leírni, a rendszer eseményekre reagál, ezek

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



2.6. ábra. MagicDraw felhasználói felülete

határozzák meg a viselkedését, szemben az Activity Diagrammokkal, amivel adott bemenetből valamilyen kimenet előállításának folyamatát írjuk le.

2.5.1. Állapottérképek SysMLben

Az állapottérképeket SysML modellek részeiként, más modell elemekhez vannak viselkedésként hozzárendelve, a triggereket aktiváló események a modell strukturális leírásában vannak definiálva, az elvégezhető akciók pedig más modell elemek által leírt viselkedések. Mivel a dolgozat kizárólag állapottérképekkel foglalkozik ezért implicit feltételezzük, hogy az állapottérképen használt szignálok definiálva vannak és ezeket az állapotgép fogadni tudja, azaz rendelkezik a megfelelő portokkal és interfészekkel amelyek ezt lehetővé teszik.

2.5.2. Plug-in fejlesztése MagicDrawhoz

A MagicDraw lehetővé teszi, hogy harmadik fél plusz funkciókat adhasson az eszközhöz plug-inok formájában. A MagicDraw Java nyelven íródott, plug-int is ezen a nyelven van lehetőség fejleszteni, ehhez egy Api-t kapunk amit Open Api-nak [5] hívnak, ez teszi lehetővé, a modell elemek kóddal történő manipulációját, és a grafikus interfész kiegészítését, saját funkcionalitással.

A MagicDraw indulásakor bejárja a plug-in könyvtárat plug-in leíró fájlokat tartalmazó könyvtárak után. Ezek írják le melyik Java osztály reprezentálja a plug-int, ennek le kell öröklődnie a *com.nomagic.magicdraw.plugins.Plugin* osztályból. A MagicDraw plugins managere meghívja ennek az osztálynak az init metódusát, amiben GUI elemeket tudunk regisztrálni, vagy egyéb funkcionalitást hozzáadni az eszközhöz.

```
public class MyPlugin extends com.nomagic.magicdraw.plugins.Plugin {
```

```
    public void init(){
        //plugin belépési pontja
    }

    public boolean close(){
        //plugin leáll
    }

    public boolean isSupported(){
        //feltételek teljesülése a plug-in betöltéséhez
        return true;
    }

}
```

A SysML elemei sztereotipizált UML elemek, a SysML plug-in SysML szintaktikát használ, de a létrehozott elemek a MagicDraw UML metamodellje szerint lesznek példányosítva megfelelően sztereotipizálva, ezért kód szinten is eszerint érhetőek el.

A modell elemek ugyan saját MagicDraws implementációval rendelkeznek, de EMF [6]-es interfaceket is realizálnak ezért lehetőségünk van EMF Apijának használatára a plugin fejlesztése során.

2.6. Viatra

Az Eclipse Viatra Framework[7] egy Eclipse alapú keretrendszer ami lehetővé teszi modellek hatékony transzformációját és lekérdezését. A modell lekérdezésekhez egy külön nyelvet Viatra Query Language(VQL)-t használ. VQL lekérdezésekből Java osztályok generálódnak: ezek a lekérdezek java kódbeli reprezentációi, melyek felhasználása lehetővé teszi modell transzformációk hatékony implementációját és a lekérdezésekre illeszkedő elemek lekérdezését a modellből.

Viatra használatában MagicDraw plugin fejlesztése során is van lehetőség. A Viatra for MagicDraw (V4MD) egy plug-in ami lehetővé teszi a Viatra Api használatát más plug-inokban. A V4MD projekt megnyitásakor létrehozza azokat a VIATRA specifikus objektumokat amelyeken keresztül lekérdezéseinkkel hozzá férhetünk a MagicDraw projekt modelljeihez.

2.7. Formális verifikáció

A modell alapú fejlesztés egyik nagy előnye, hogy már tervezési fázisban tudjuk vizsgálni rendszerünk egyes aspektusait. A rendszer helyes működésének ellenőrzését verifikációnak hívják. Ez történhet tesztekkel, szimulációval, vagy formális módszerekkel.

A formális módszerek előnye, hogy a rendszer helyességéről matematikailag precíz bizonyítást adnak, nem szükséges az elvárt kimenetek meghatározása, csak megkötések megfogalmazása. Továbbá teljesek ezért nem kell lefedettséggel foglalkozni mint tesztelésnél. Megkötés sérülésekor, vissza lehet követni, azt végrehajtási útvonalat(*execusion trace*) ami a megszorítás megsértéséhez vezetett. Hátrányuk, hogy nehezen skálázható, erőforrás igényes és egy összetett rendszert formális módszerekre való visszavezetése gyakran nehézkes.

[6] Eclipse Modelling Framework https://www.eclipse.org/modeling/emf/
[7] Viatra: https://projects.eclipse.org/projects/modeling.viatra

Állapottérképek formális verifikációjára már léteznek megoldások. A Gamma keretrendszer például képes állapottérképek verifikációjának elvégzésére. Ehhez az Uppaal [1, 2] nevű eszközt használja fel.

2.8. Gamma Framework

A Gamma Statechart Composition Framework komponens alapú reaktív rendszerek tervezésére, validálására, verifikálására és kód generálásra lett létrehozva. Az eszköz alap komponensei az állapottérlépek amiknek a leírásához egy saját nyelvet Gamma Statechart Languaget (2.7. ábra) definiál, ami lehetővé teszi külső modellező eszközök integrálását is. Az első integrált eszköz a Yakindu Statechart Tools [8], aminek a modelljeit Gamma képes a saját maga által definiált modellekké (2.8. ábra) transzformálni és feldolgozni.

Az eszköz egy másik komponensei az ún. Kompozit komponensek, amiket Gamma Composition Languagel lehet definiálni. Ezek írják le a rendszer felépítését komponensekre bontva, illetve az ezek portjai és interfészei között definiált kapcsolatokat.

A keretrendszer még két saját nyelvet definiál. Az egyik a Gamma Constraint Language, amivel megszorításokat lehet definiálni, ami egy általános megoldása típus definíciók, változók, függvények deklarálásához és kifejezések specifikálásának. A másik nyelv pedig Gamma Interface Language, ami interfészek definiálását teszi lehetővé, amik a kapcsolódó komponensek egymás felé nyújtanak. Az interfész határozza meg, hogy milyen események fogadhatóak, vagy küldhetőek. Ezek az interfészekben deklarálandók és irányuk lehet, *IN, OUT, INOUT*.

Munkámban a egy MagicDraw plug-int hozok létre ami a formális verifikáció végrehajtásához a Gamma keretrendszert használja azáltal, hogy *VIATRA* segítségével átranszformálja a MagicDraw modelljeit Gamma modellekké.

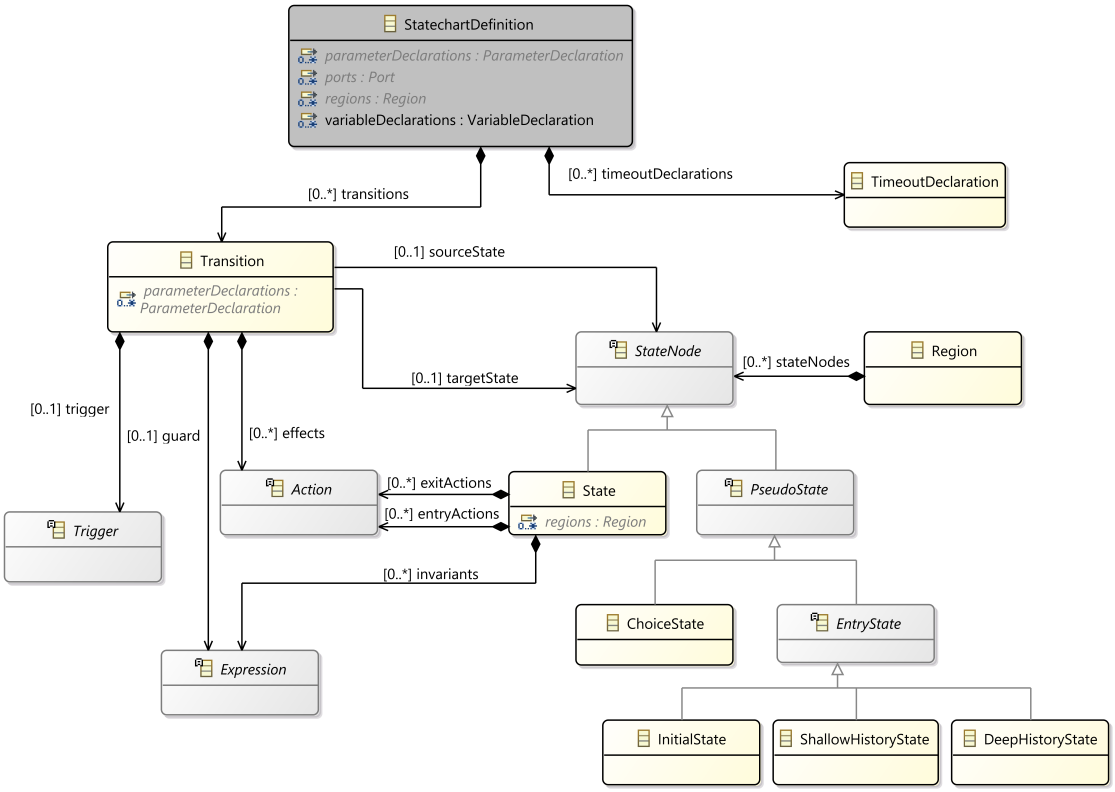
```
package Exapmle

statechart MonitorStatechart [] {

transition from Red to Blue
transition from Entry0 to Red

   region main_region {
      initial Entry0
      state Red
      state Blue
   }
}
```

2.7. ábra. Gamma Statechart konkrét szöveges szintaxisa



2.8. ábra. Gamma állapottérkép absztrakt szintaxisa

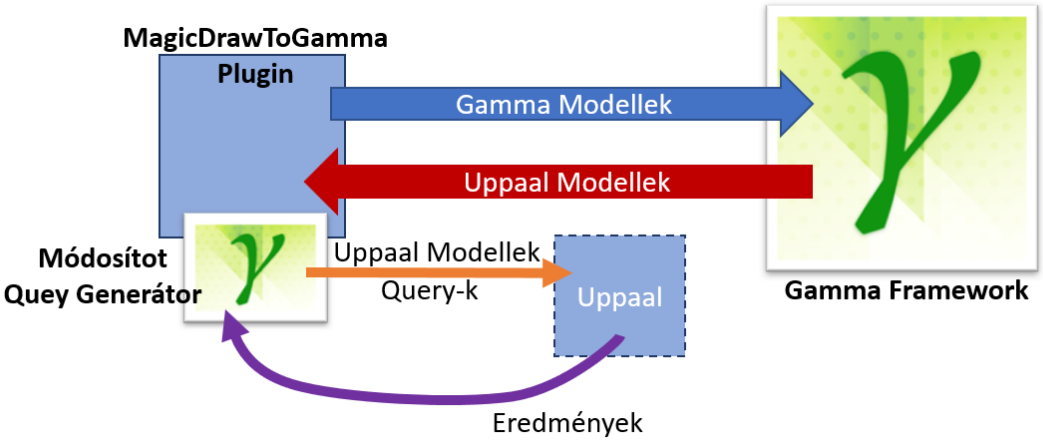
3. fejezet

MagicDrawToGamma plugin

Ebben a fejezetben bemutatom a munkám során elkészített alkalmazást kitérve, az alapkoncepcióra, az elméleti és gyakorlati problémákra és megoldásaikra. Továbbá kitérek a fejlesztés során felmerülő szoftver-infrastruktúra és implementáció okozta komplikációkra.

3.1. Koncepció

Állapottérképek formális verifikációjának támogatása MagicDraw-ban, egy plug-in fejlesztésével lett megvalósítva. A plug-in függ a Viatra For MagicDraw-tól, ami lehetővé teszi modellek transzformációját Viatra segítségével. A plug-in legfontosabb funkciója MagicDraw modellek Gamma modellekké való transzformációja. A letranszformált Gamma nyelvű modelleket a keretrendszer kezelni tudja. A verifikáció elvégzéséhez az eszköznek csak egyes részei szükségesek. A megoldást a 3.1 ábra szemlélteti. A felhasználónak lehetősége van megkötések megfogalmazására a plugin-al és elvégezni a verifikációt és megtekinteni az eredményt azaz, hogy teljesülnek-e a megkötések vagy sem.



3.1. ábra. Koncepció

3.2. Fejlesztőkörnyezet

A fejlesztés megkezdéséhez szükséges volt összeállítani egy olyan fejlesztőkörnyezetet amivel hatékonyan lehet plug-int fejleszteni. A MagicDraw biztosít egy ún. *skeletont* Eclipsehez és IntelliJhez is plug-in fejlesztéséhez, de a fejlesztés nem ezek segítségével hanem az *IncQuery Labs* által készített skeleton felhasználásával valósult meg. Ennek oka, hogy a

hivatalos *skeletonok* nem vagy csak részben működtek, a mögöttes infrastruktúra megismerése és javítása pedig túl hosszadalmas és a feladat szempontjából irreleváns lett volna.

A skeleton egy Eclipse projekt, ami Gradlet használ a projekt fordításához és a dependenciák kezeléséhez. Ez sokszor inkonzisztenciákhoz vezetett melyek közül az egyik legnagyobb problémát a Viatra Query-k okozták, mivel csak Eclipsel generálhatók. A kódbázis egy része nem Javában hanem Xtendben íródott, a Viatra modell transzformációk implementálása ezzel a nyelvvel egyszerűbb. Azok az osztályok melyeknél nem volt indokolt, jellemzően a MagicDraw felhasználói felületeinél, azok Java 8-ban lettek implementálva.

A dolgozat elkészítése idején a MagicDraw 19-es verziója is elérhető volt a plug-in azonban még nem ehhez, hanem a 18.5-ös verziójához készült. A kódbázis azonban kompatibilis lehet még az újabb verziókkal is, amennyiben az állapottérképeket érintő metamodellek nem változnak.

Gamma(2.0) a verifikációt Uppaal segítségével végzi el. Ehhez előállít egy formális leírást a rendszerről és egy *query*-t ami a rendszerrel szemben támasztott követelményeket írja le. A követelmények megírásához biztosít egy UI elemet amivel a felhasználó az Uppaal ismerete nélkül is képes a követelmények definiálására. Ez a funkció teljesen átkerült a Gammából módosított implementációval a MagicDrawToGammába. A Gamma az Uppaalra az operációs rendszeren keresztül hív át, ezért az Uppaalt külön kell telepíteni és konfigurálni.

3.3. MagicDraw - Gamma transzformáció

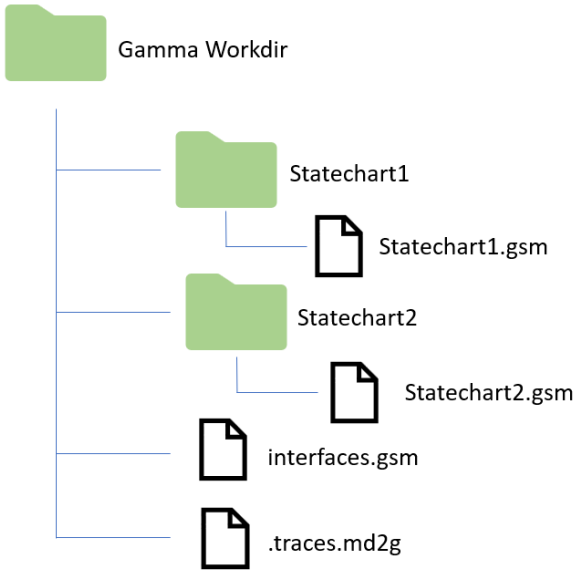
A MagicDraw - Gamma transzformációt egy menüponttal lehet elindítani a MD2G kategória alatt. Ehhez szükséges, hogy a projekthez hozzá legyen rendelve egy külső könyvtár (Gamma Workspace) ami a leképzett és a háttértáron eltárolandó modellek helyét jelöli. A hozzárendelt könyvtár abszolút elérését String formában a projekt tárolja, emiatt sem a projekt nem migrálható, sem pedig maga a Gamma Workspace. Továbbá a könyvtár karbantartása ebben a verzióban még a felhasználó felelőssége, ugyanis a plug-in nem töröl a könyvtárból csak hozzáad és módosít. Ennek akkor van jelentősége, ha leképzés után egy állapottérképet eltávolítanak a MagicDraw modellből és utána újra leképzik az egész modellt. Ebben az esetben a régebben leképzett Gamma állapottérképek is megmaradnak.

3.3.1. Gyökér elemek létrehozása:

A plug-in a transzformáció kezdetekor létrehoz egy *ResourceSet*-et, amivel készít egy *Resource*-t a *Gamma Workpace* gyökerébe .s.md2g néven, továbbá egy Resource-t `interfaces.gms` néven. Az előbbi Resource egy segédstruktúrát tartalmaz ami a leképzett elemek visszakereshetőségéül szolgál (ld. 3.3.9. alszakasz), utóbbi pedig a modellben definiált interfaceket tárolja.

Az eszköz ezután Viatra Queryk segítségével megkeresi az állapotgépeket a MagicDrawban és végig iterál rajtuk. Minden állapottérképen kigyűjti az éleken használt *SignalEventek*et és létrehoz a *Signal*éval azonos néven egy Eventet a Gamma metamodelljének megfelelően. Ez az Event a *Signal* Gammabeli megfelelője. A létrehozott eventek egy Interfacen kerülnek definiálásra aminek a neve megegyezik az állapottérképével. Az interface ezután belekerül az interfaceket tároló *Resourceba*. Az eventek iránya INOUT ez lehetővé teszi, hogy az állapotképek küldhessenek és fogadhassanak.

A leképzett állapottérképek külön Resourceokba kerülnek amik a Gamma Workspaceben egy külön az állapottérkép nevével megegyező könyvtárba kerülnek, ugyan ezen a néven .gsm kiterjesztéssel (3.2-es ábra). (A Resource gyökere nem egy StatechartDefinition, hanem egy Package, ennek a neve szintén megegyezik az állapottérkép nevével)



3.2. ábra. Létrehozott fájlstruktúra

3.3.2. Alap struktúra kialakítása:

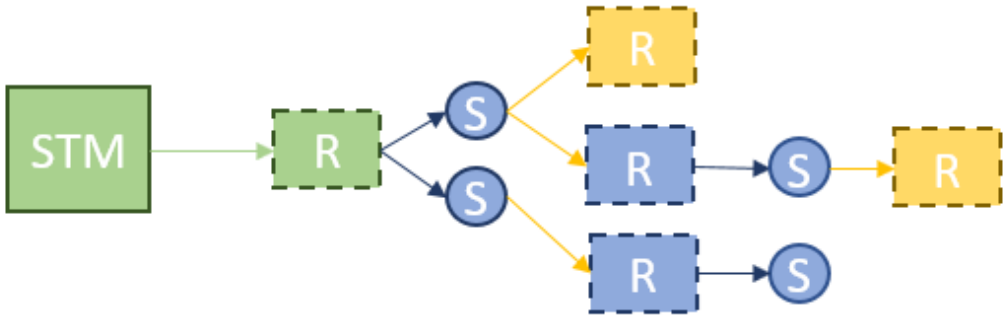
Az előző lépések során azok az elemek állnak elő, melyek a modellek gyökér elemeiként fognak szolgálni. A következő lépés létrehozni az állapottérképek egy belső, alap struktúráját, amit az állapotok és állapotátmenetek határoznak meg. A Viatra for MagicDraw projekt megnyitásakor készít egy ViatraQueryEngine-t aminek a *Scope*-ja a megnyitott modellre terjed ki. A transzformációs szabályok regisztrálása ezen az engine-en történik, létrejön azonban még egy aminek a *Scope*-ja magába foglalja az első lépésben létrehozott *Resource*okat és a MagicDraw modellt is. A készülő modellben végzett keresések és a és a visszakövetéseket ez az *engine* végzi el. A struktúra kialakításának lépcsői a következők:

1. Fő régiók leképzése (olyan régió aminek a szülője állapotgép).
2. Régiókban található állapotok leképzése.
3. Állapotokban található régiók leképzése.

A második lépésben a *Trace Modell* alapján megkeressük a már leképzett régiót a Gamma Modellben és beletesszük az újonnan létrehozott és a MagicDraw modellnek megfelelően elnevezett állapotot, ha a régió még nincs leképezve akkor létrejön, és abba kerül bele az állapotot. Ez a lépés olyan részgráfokat is eredményezhet amikbe nem vezet út a gyökér elemekből. Ennek kiküszöbölése a harmadik lépés ami, ugyanezt a műveletet hajtja végre csak a másik irányból, tehát az állapotok párjait keressük meg, amikbe régiókat helyezünk el. Ezek a régiók már létezhetnek ilyenkor nem új régió jön létre hanem a már meglévő kerül az állapotba. A működés során a MagicDraw modell jól formáltsága elvárt, azaz régió csak *State*ben és *StateMacine*-ban fordulhat elő. A régiókat tartalmazó állapotok kompozit és ortogonális állapotnak tekintendők.

A tartalmazási gráf összefüggővé válását a 3.3 ábra szemlélteti.

Az irányított nyilak tartalmazást jelölnek, a bekeretezett téglalap *StatechartDefinition*, a szaggatott vonallal körbevett *Region* és a körök *Statek*.



3.3. ábra. Állapotok és régiók leképzésének menete, 1. lépcső: zöld, 2. lépcső: kék, 3. lépcső: sárga

3.3.3. Pszeudoállapotok átalakítása:

Az állapotátmenetek leképzése előtt a pszeudoállapotok kerülnek leképzésre. Ezen a ponton már létezik minden régió amit tartalmazhatja őket. A leképzés legtöbb esetben támogatott viszont, egyes elemek tartalmazása esetén nem lehet a verifikációt végrehajtani. Az elemeket és párjaikat a 3.1 táblázat mutatja.

MagicDraw	Gamma	verifikálható
InitialState	InitialState	igen
Chioce	Choice	igen
Junction	Merge	nem
Fork	Fork	nem
Join	Join	nem
TerminalState	nincs	-
Conn. PointReference	nincs	-
EntryPoint	nincs	-
ExitPoint	nincs	-

3.1. táblázat. Pszeudoállapotok párosítása.

3.3.4. Állapotátmenetek leképzése:

A következő lépés az állapotátmenetek átalakítása. Ezen a ponton már az összes olyan elem leképzésre került, amely az állapotátmenetek kezdő, vagy végpontjaként szolgálhat. Egy MagicDraw modellben az állapotátmenetek régiók tartalmazzák, szemben Gammával, ahol a StatechartDefinition közvetlen gyerekei. A tartalmazó-tartalmazott, Statemahcine - Tranisiton párok megkeresése a következő patternekkel történik.

```
pattern RegionsInRegion(container: Region, region: Region){
    Region.subvertex(container, vertex);
    State.region(vertex, region);
}
pattern RegionsInStatemachine(stateMachine: StateMachine, subregion: Region){
    find MainRegions(stateMachine, subregion);
} or {
    find RegionsInRegion+(region, subregion);
    StateMachine.region(stateMachine, region);
```

```
}
pattern TranisitonsInStateMachine(stateMachine: StateMachine, transition:
    Transition){
    find RegionsInStatemachine(stateMachine, region);
    Region.transition(region, transition);
}
```

A *StateMachine* Gamma modellbeli párját a *Trace modell* segítségével lehet megtalálni és hozzáadni a megfelelő állapotátmenetet.

Az átmenetek leképzése után már elérhetőséget lehet is vizsgálni.

3.3.5. Változók leképzése

Változókat MagicDraw-ban attribútumként lehet definiálni. Ezeknek a következő tulajdonságait lehet beállítani: láthatóság, típus, statikusság, alapértelmezett érték.

A láthatóság és a statikusság, a jelen verzióban nem bírnak jelentőséggel, hiszen minden állapottérkép *singleton*nak tekinthető és egymástól független léteznek. A típus lehet bármilyen beépített vagy felhasználó által definiált típus vagy valamilyen primitív. A leképzés során kétféle primitív típus támogatott: *Integer* és *Boolean*, egyéb típusok leképzése még nem támogatott. A változók tulajdonságaira vonatkozó megkötéseket a 3.2 táblázat foglalja össze.

Tulajdonság	Lehetséges érték
Name	tetszőleges, de egyedi az állapotgépben
Type	Integer, Boolean
Static	tetszőleges, de mindig statikusnak tekintendő
Visibility	tetszőleges, de mindig private-nak tekintendő

3.2. táblázat. Változók lehetséges értékei.

3.3.6. Események, triggerek leképzése:

A MagicDrawban definiálható *Triggerek* pontosabban az őket kiváltó események közül jelenleg kettő támogatott. Egyik a *SignalEvent* a másik pedig *TimeEvent*. Előbbi *EventTriggerre* képződik le. A felhasználónak a *SignalEvent* forrásával most még nem kell foglalkoznia, hiszen ezekhez automatikusan generálódik egy *Interface* és egy *Port*.

A *Time Eventek* MagicDraw-ban kétféle típusúak lehetnek: relatív és abszolút. Utóbbit a plugin-in még nem támogatja. A relatív típusú *Time Eventek* a Gamma *Timeout* mechanizmusának feleltethető meg. Ez három részből áll: *StatechartDefinition*-ön definiált *TimeoutDeclaration*, akció ami ennek beállítja az értékét (ami lehet szekundumban, vagy milliszekundumban mért) és maga a *Trigger* ami hivatkozik a deklarációra. A *TimeoutDeclaration* és az érték beállítása implicit történik. A felhasználónak csak az időt kell megadnia trigger felvételekor a MagicDraw modellben.

3.3.7. Őrfeltételek leképzése:

Őrfeltételek definiálása MagicDraw állapottérképeken Opaque Expressionökkel[1] történik, ezért ezt le kell fordítani és modell alapú leírássá konvertálni. Kifejezéseket Gammában a Constraint modellel lehet leírni. Ehhez tartozik egy nyelvtan is, ami Xtext segítségével képes a Gamma Constraint nyelvből EMF alapú *Constraint* példánymodellt fordítani.

A leképzéshez a *Constraint Model Expressio*n szabálya lett felhasználva, amely alapján az Xtext parser előállítja a szöveges bemenetből a megfelelő *Expression* példányt.

[1] Szöveges nyelvel leírt kifejezés

```
Injector injector = new StatechartLanguageStandaloneSetup()
                  .createInjectorAndDoEMFRegistration();

ParserRule rule = injector.getInstance(StatechartLanguageGrammarAccess.class)
                  .getExpressionRule();

IParseResult result = injector.getInstance(StatechartLanguageParser.class)
                  .parse(rule, new StringReader(guardString));
```

A referenciák viszont nem lesznek feloldva ezért ezek feloldása *VIATRA* segítségével utólagosan történik a deklarációk név szerinti megkeresésével az előálló Gamma modellekből.

```
pattern DeclarationsByName(statechartDefinition: StatechartDefinition, name: java
     String, declaration: Declaration){
    StatechartDefinition.variableDeclarations(statechartDefinition, declaration);
    Declaration.name(declaration, name);
}
```

Mivel az őrfeltételek megírása a gamma nyelvével történik azért ugyan azokat az operátorokat támogatja, mint a gamma.

Név	Operátor	leírás	Csoportosítás
Imlikáció	imply	implikáció	jobbról balra
Diszjunkció	or	logikai vagy	n-szeres
Konjunkció	and	logikai és	n-szeres
Negálás	not	logikai nem	jobbról balra
Egyenlőség	=, /=	egyenlőség, egyenlőtlenség	jobbról balra
Reláció	<, >, <=, >=	relációs operátorok	balról jobbra
Hozzáadás	+, -	hozzáadás, kivonás	balról jobbra
Skálázás	*, /	szorzás, osztás	balról jobbra
Maradék	mod	maradékos osztás	balról jobbra
Egész osztás	div	egész osztás	balról jobbra
Előjel	+, -	egyváltozós +, egyváltozós -	balról jobbra

3.3. táblázat. Támogatott operátorok.

3.3.8. Akciók leképzése:

Az akciók viselkedések amiket négyféle helyen lehet végrehajtatni egy állapottérképen:

1. Állapotvátáskor: a viselkedés állapotváltáskor hajtódik végre.
2. Állapotba való belépéskor (*Entry*): a viselkedés akkor hajtódik végre amikor az állapot aktívvá válik.
3. Állapotban maradva (*Do*): a viselkedés addig hajtódik végre amíg az állapot aktív.
4. Állapotból kilépve (*Exit*): a viselkedés állapotból való kilépéskor hajtódik végre.

A MagicDraw akciói viselkedések (*Behavior*) lehetnek: pl. *StateMachine*, *Activity*, **OpaqueBehavior**. Az OpaqueBehavior pontosabban ennek egy része az OpaqueExpression lehetővé teszi, hogy valamilyen szöveges szintaxissal írjunk le viselkedést. Ennek felhasználásával, hasonlóan az őrfeltételek esetében Xtext segítségével elő tudjuk állítani a megfelelő EMF modellt a Gamma szintaxisa szerint. Ehhez a `ActionRule` szabályra van szükség.

Jelenleg kétféle akció támogatott: értékadás és az eseményküldés. Ezek szintaxisa az alábbi:

```
//assignment action
variableName := 1;
//raise event action
raise All.Event
```

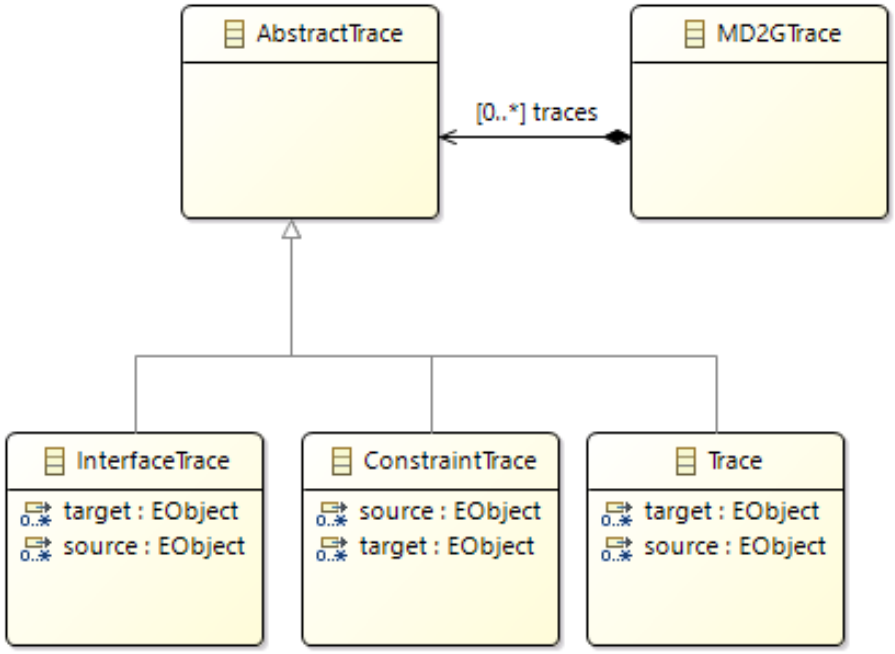
Az *All* nem Gamma specifikus jelölése az össze portnak, ebből jelenleg csak egy van egy implicit generált az események fogadáshoz, így jelen esetben az *All* megjelölés csak ezt a portot választja ki.

Megjegyzés: a küldött eseményeknek nincs hatása, nem fogadja őket az állapottérkép, még akkor sem ha az interfészén az események *INOUT*-ként vannak definiálva.

3.3.9. Trace Modell

A leképzések végrehajtása során fontos követni, hogy mely elemek képződtek le és mely elemekké. Erre a célra a MagicDrawToGamma bevezet egy Trace modell[2] nevű segédstruktúrát, ami lehetővé teszi a megfeleltetések visszakereshetőségét VIATRA segítségével, továbbá a modell sorosítása és háttértáron való tárolása is megoldott. A modell EMF-ben van definiálva és háromféle *Trace*-t különböztet meg. A Gamma állapottérképek alap elemeit `Trace`-ek, az interfészek elemeit `InterfaceTrace`-k és a *Contraint* modell elemeit `ConstraintTracek` kötik össze MagicDraw párjukkal, amiből le lettek képezve.

A három `Trace` típust egy `AbstractTrace` osztály fogja össze, és `MD2GTrace`-ben tárolhatók. (3.4 ábra)



3.4. ábra. Trace modell EMF definíciója

[2]Nem összekeverendő a a 2.7 fejezetben említett *Execusion Tracel*

3.4. A Verifikáció végrehajtása

A verifikáció végrehajtásához két dologra van szükség. A modellnek egy leírására amit az Uppaal képes értelmezni és előállítani egy időzített automatát belőle és az Uppaal Query nyelvén megfogalmazott feltételekre. A MagicDrawToGamma kétféle lehetőséget biztosít a verifikáció végrehajtására.

1. Uppaal közvetlen használata
2. Uppaal Query Generator

Első lehetőség időzített automaták leírásának előállítása amit a felhasználó megnyithat *UPPAAL*-al. Ennek végrehajtása két részből áll. Először a Gamma modellt át kell alakítani, ez a `StatechartToUppaalTransformer` osztályon keresztül történik, ami a *Gamma része*.

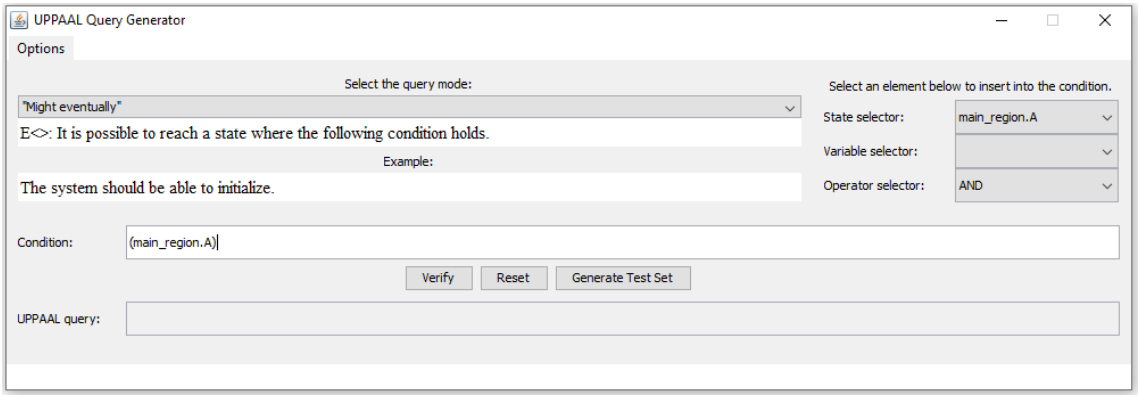
```
//initialize with a Gamma Package
StatechartToUppaalTransformer transformer = new StatechartToUppaalTransformer(p);
SimpleEntry<NTA, G2UTrace> entry = transformer.execute();
```

Ennek a kimenete két EMF alapú modell, egy *trace modell* (`GU2Trace`) és egy időzített automata (`NTA`). Az **időzített automata**[3] modelljét ezután egy formális leírásként kell szerializálni amit az UPPAAL képes beolvasni. Ez a Gamma `UppaalModelSerializer` osztálya állítja elő.

```
//NTA, parentFolder, fileName
UppaalModelSerializer.saveToXML(entry.getKey(), selected.getPath() , name
    +".xml");
```

A kimenetek a *Gamma Workspace-ben* az állapottérkép könyvtárába kerülnek, ahonnan a felhasználó megnyithatja őket *UPPAAL*-ban.

A második lehetőség az *Uppaal Query Genertor* (3.5 ábra) használata. Ez egy segédablak amivel a felhasználó egy grafikus felhasználói interfészen tud *UPPAAL Query*-ket definiálni, így a felhasználónak nem szükséges elsajátítania az *UPPAAL Query*-k szintaxisát.



3.5. ábra. UPPAAL Query Generator

A Query Generátor implementációja a Gammából lett átemelve a MagicDraw pluginba és az implementáció módosítva lett, hogy ebben a környezetben is tudjon működni. A Gamma egyik funkciója, hogy a formális verifikáció során esetleg keletkező ellenpéldákból

[3] a formális verifikáció az időzített automaták formalizmusán történik

képes Java nyelvű teszteseteket és szimulációt generálni Yakinduhoz. A MagicDrawToGammában a kódgenerálást még nem támogatja és a modellek nem Yakinduból származnak ezért ez a funkció nem használható a jelenlegi verzióban.

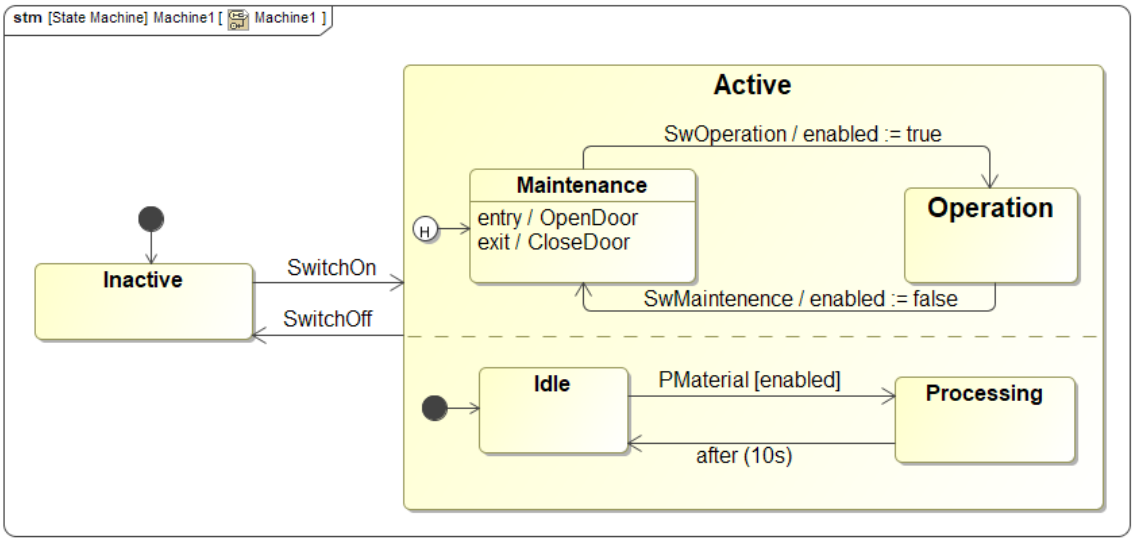
3.5. Esettanulmány

Az előző alfejezetek a MagicDrawToGamma főbb funkcióinak ismertetéséről és ezek megvalósítását mutatták be. Ebben az alfejezetben a tényleges működéséről lesz szó egy példán keresztül.

3.5.1. Szemléltető példa

Példa specifikációja: egy munkagépbe valamilyen anyagot lehet tölteni amit a gép tíz másodperc alatt feldolgoz. A gép az első bekapcsoláskor előbb szerviz állapotba kapcsol, ilyenkor kinyílik egy ajtó ami a gép szervizelését teszi lehetővé. Ahhoz, hogy a munkafolyamat elindulhasson a gépet üzem állapotba kell helyezni. Ilyenkor a gép ajtaja becsukódik és egy jel hatására feldolgozza a bemenetén elhelyezett anyagot. A gép kikapcsolásnál megjegyzi hogy szerviz vagy üzem állapotban volt-e és abba kapcsol vissza.

Rendszer állapot alapú definíciója: A rendszer két jól megkülönböztethető állapotból áll `Acitve` és `Inactive`. Az `Active` állapot felbontható két belső állapotra, hogy szerviz vagy üzem módba van: `Maintentece`, `Operation`. Ezeken kívül bevezethető még két állapot, hogy feldogoz-e épp anyagot a rendszer vagy sem: `Idle`, `Processing`. `Maintenance` állapotban a `Processing` állapotba való belépést egy logikai változóval `enabled` tiltjuk és engedélyezzük. Ezek alapján a rendszer egy lehetséges leírása lehet:



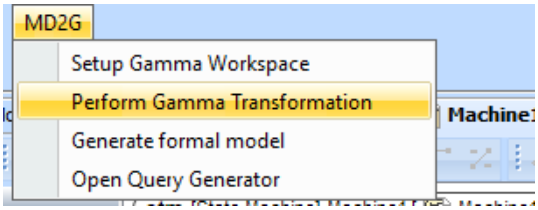
3.6. ábra. Lehetséges leírás

Megjegyzés Az `OpenDoor` és `CloseDoor` akciók csak szemléltetés jellegűek, ezek nem kerülnek leképzésre.

Feladat: Ellenőrizzük, hogy a `Maintentance` és `Processing` állapotok kizárják egymást. Azaz a rendszer ne kerülhessen olyan állapotba, hogy nyitva van az ajtó miközben még feldolgoz a rendszer, hiszen ez veszélynek tenné ki a dolgozókat.

3.5.2. Verifikáció menete

Ahhoz, hogy le tudjuk ellenőrizni, hogy a leírt működés megfelel-e a támasztott feltételeknek végre kell hajtanunk a MagicDraw - Gamma transzformációt. Ezt a MD2G menüpont alatt található `Perform Gamma Transformation` kiválasztásával lehet megtenni (3.7 ábra).



3.7. ábra. MD2G menüpontok

A menüpontot kiválasztva előállíthatjuk az általunk kiválasztott Gamma Workspacebe a Gamma példánymodelleket, ez viszont önmagában még nem elég. Ezeket formális modellekké kell alakítanunk, hogy UPPAAL-al is fel lehessen őket dolgozni. Ezt a `Generate formal model` menüponttal tudjuk végrehajtani. Ez feldob egy könyvtárválasztó ablakot amiben a Gamma Workspace-ben található könyvtárak közül ki kell választanunk a megfelelő modellt tartalmazót (jelen példánál maradva a könyvtár neve Machine1 ez látható a 3.6. ábrán címkeként is).

A formális modell előállítása után a `Open Query Generator` menüponttal tudjuk megnyitni a *Query Generator*-t, hasonlóan mint az előző lépésben ki kell választanunk újra a megfelelő könyvtárat a Gamma Workspace-ünkből, ami a formális modellt tartalmazza.

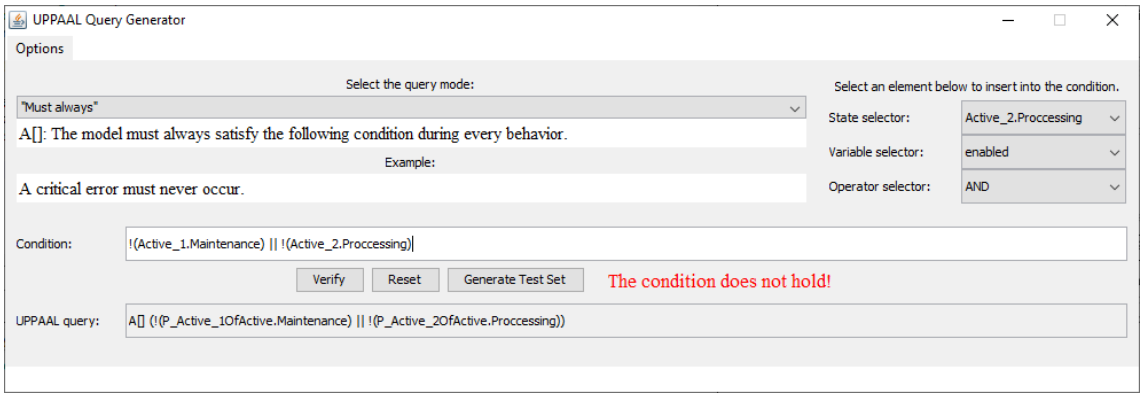
Feltétel megfogalmazása: a feltétel amit megszeretnénk fogalmazni, hogy nem lehet a rendszer egyszerre `Maintenance` és `Processing` állapotban.

$$\neg(Maintenance \wedge Processing) = \neg Maintenance \vee \neg Processing$$

A kifejezést a Gamma *Query Generátor*ba a következő alakba kell bevinnünk:

```
!(Active_1.Maintenance) || !(Active_2.Processing)
```

Ennek megírásához a *Query Generator* segítséget nyújt a hivatkozható elemek és a köztük megfogalmazható relációk felsorolásával. A feltételtől azt várjuk, hogy mindig teljesüljön ezért a `query mode`-ot *Must Always*ra állítjuk. A verifikációt a `verify` gomb megnyomásával hajthatjuk végre az eredményt a 3.8 ábra szemlélteti.

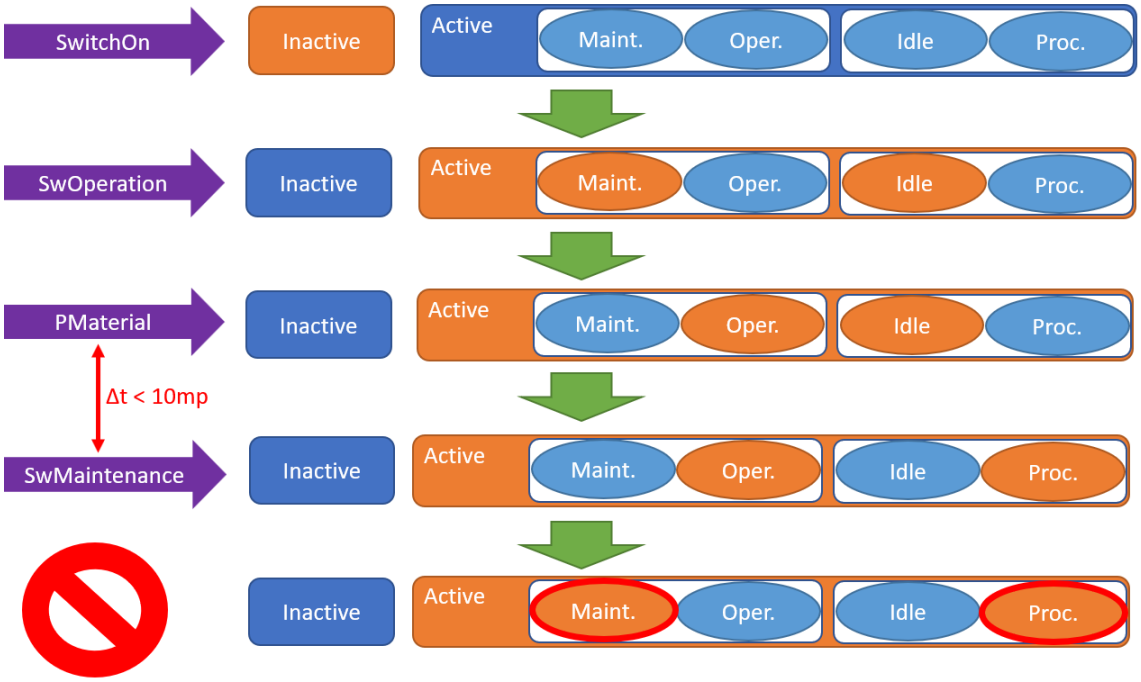


3.8. ábra. Verifikáció eredménye

Eredmény: A megszorítások nem teljesülnek. Ennek okát sajnos a plug-in még nem képes levezetni, ezt nekünk kell kitalálni vagy az UPPAAL segítségével megnyithatjuk a formális modellt és az generált UPPAAL query segítségével azon belül is el tudjuk végezni a verifikációt és tanulmányozni a generált végrehajtási útvonalakat. Ehhez azonban az UPPAAL és a leképzések ismerete szükséges.

3.5.3. A probléma:

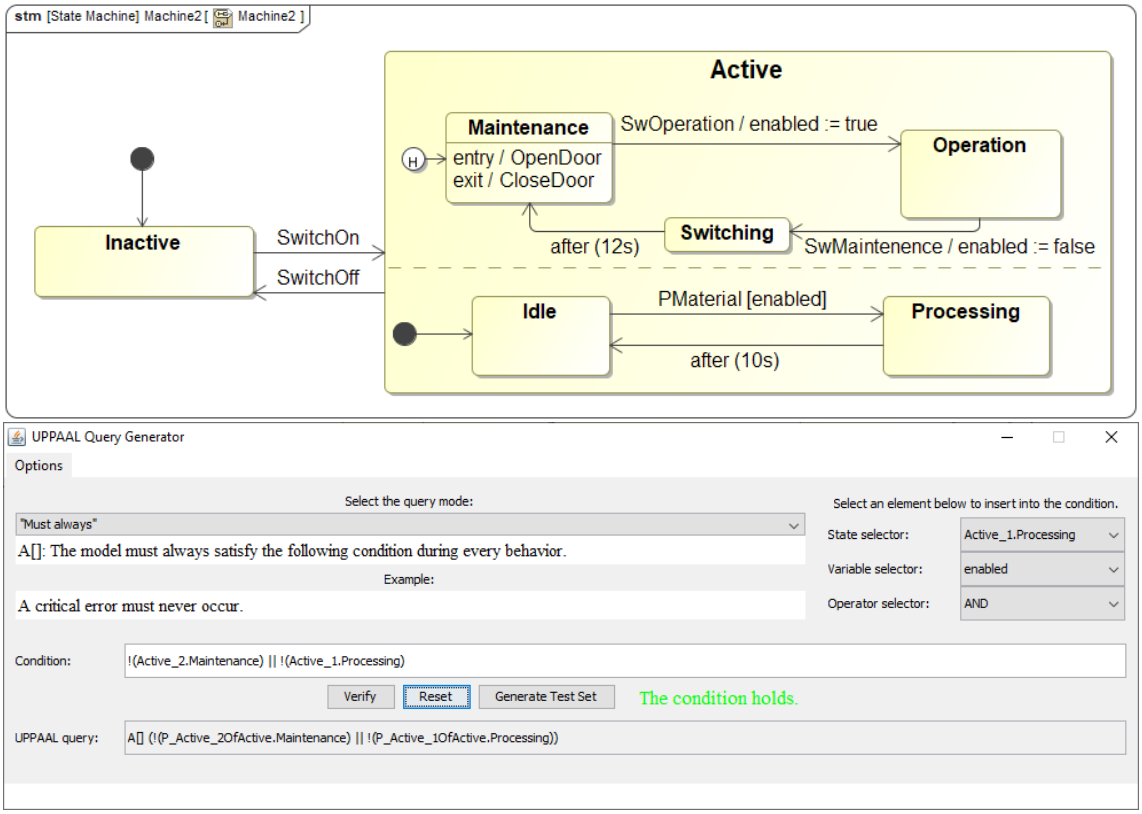
A példa esetében a megszorítások megsértését a következő okozza: az állapotgép belép `Operation` állapotba, ilyenkor az ajtók zárva vannak a működés pedig engedélyezve. Megkezdődik a feldolgozás, viszont mielőtt letelne a tíz másodperc a gépet szerviz állapotba kapcsolják, így egyszerre lesz `Maintenance` és `Processing` állapotban amíg a feldolgozásra szánt idő le nem telik.



3.9. ábra. A megkötés megsértésének lépési

3.5.4. Megoldás

Lehetséges megoldás lehet a probléma kiküszöbölésére egy köztes állapot bevezetése az `Active` ortogonális állapot felső régiójába. A gép szerviz állapotba kapcsoláskor előbb ide lép át, ilyenkor a feldolgozás letiltódik. Itt várakozik, méghozzá annyi időt, amennyi a feldolgozáshoz legalább szükséges. Ez a működés tehát megvárja a még folyamatban lévő feldolgozás befejezését és csak utána lép át a `Maintenance` állapotba. A módosított állapottérképet az új állapottal (`Switching`) és a verifikáció eredményét a 3.10. ábra mutatja.



3.10. ábra. Módosított állapottérkép és a verifikáció eredménye

4. fejezet

Értékelés

Ebben a fejezetben megvizsgálom a plug-in teljesítményét a transzformáció végrehajtási idejét alapul véve, továbbá bemutatok egy alternatív megvalósítási lehetőséget amit a modell transzformációk helyett lehetne alkalmazni. Végül ismertetek pár funkciót amivel az eszköz továbbfejleszthető.

4.1. Alternatíva - kódgenerálás

A Gamma saját nyelvtanokkal rendelkezik, melyek Xtext segítségével vannak implementálva. Ez a technológia lehetővé teszi, hogy saját szintaxis alapján, lehessen kódot írni és EMF példánymodellt generálni a leírásból. Ezt a mechanikát ki lehet használni a modell transzformáció alternatívájaként: a MagicDraw modellből *Gamma Statechart Language* szintaxisának megfelelő kódot lehetne generálni és ezt Xtext segítségével leparse-olni.

A visszakövethetőség is megoldható, ehhez a nyelvtant annotációkkal kéne kibővíteni, amik jelölnék az eredeti elemeket. Ennek előnye, hogy a kimeneteket utána tovább lehetne importálni Eclipse-be és abban folytatni a fejlesztést. Hátránya viszont, hogy a skálázhatóságot sokkal nehezebb megoldani, és kevésbé flexibilis mint a transzformáció.

4.2. Plug-in teljesítménye

A plug-in teljesítményét leginkább az jellemzi, hogy mennyi idő alatt képes végrehajtani a MagicDraw - Gamma transzformációt. Ennek meghatározására szolgálnak az alábbi mérések. Ezek célja a modell elemek számának és a végrehajtáshoz szükséges idő kapcsolatának vizsgálata.

4.2.1. Mérések elvégzéséhez használt specifikációk

Processzor	Intel i7-4770 @ 3.40GHz 3,40GHz
Ram memóra	8Gb
Operációs rendszer	Windows 10 Pro
Java verzió	Java 8 (181)
MagicDraw verzió	18.5
MagicDraw maximális heap méret	2433Mb

4.1. táblázat. Méréshez használ specifikációk

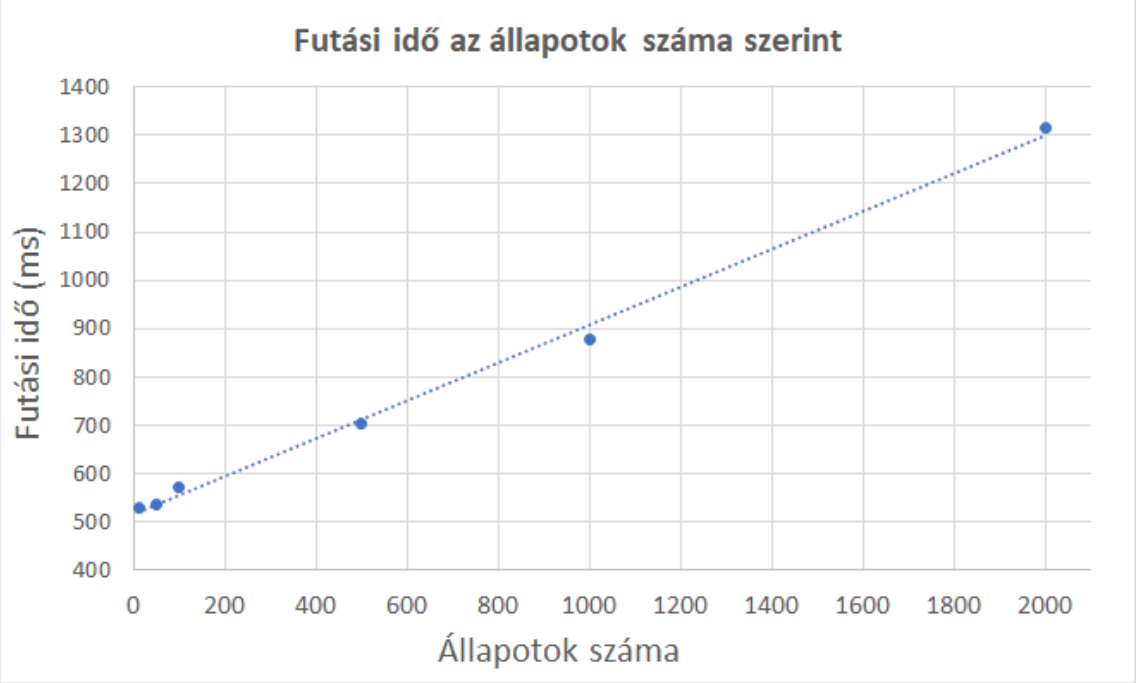
4.2.2. Egyszerű állapotgépek triggerrel

Az egyszerű állapotgépek nem tartalmaznak csak állapotokat állapotátmeneteket és *Signal Event Trigger*eket. A mérések hat különböző mintán lettek futtatva ezeket a 4.2. táblázat szemléltei.

Név:	m1	m2	m3	m4	m5	m6
Állapotok száma:	10	50	100	500	1000	2000
Állapotátmenetek száma:	10	46	91	461	821	1641
Signal Event Triggerek száma:	9	45	90	460	820	1640

4.2. táblázat. Első méréshez használt modellek

A mérések egymás után lettek futtatva ugyan azon a modellen új elemek felvételével az eredményeket a 4.1. ábra szemlélteti. Minden elemet ugyan az a régió tartalmazott és csak egy állapottérkép volt a modellben. Az állapotok és állapotátmenetek tíz hosszú láncokat alkottak. Minden leképzés után a leképzett modell kiírásra került a háttértárra.



4.1. ábra. Első mérés: transzformációk egymás után növekvő modellen

A grafikonról leolvasható, hogy az végrehajtási idő közel ugyan olyan arányban nőtt mint ahogy a modell mérete növekedett.

4.2.3. Egyszerű állapotgépek őrfeltétellel

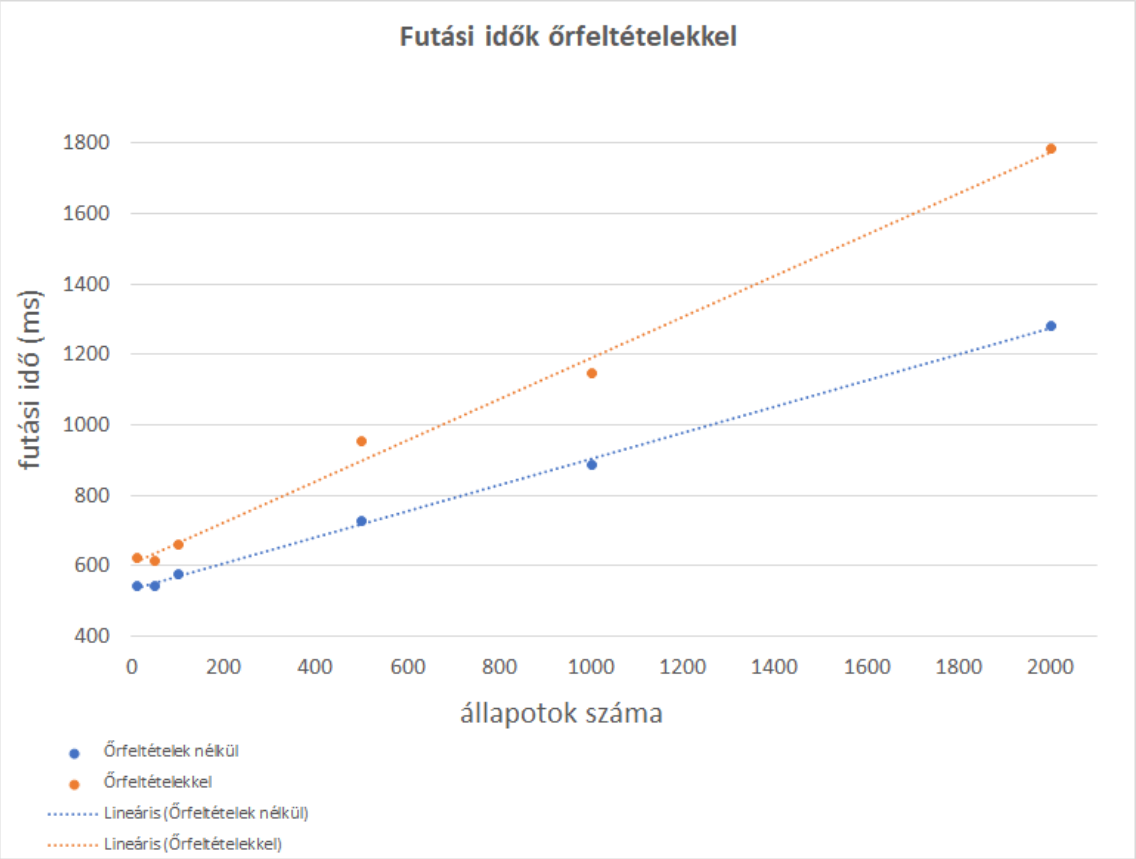
A következő mérések során az állapotátmenetekre őrfeltételek kerülnek. Ezek leképzése Xtext parsert használ. Ennek a méréseknek a célja megvizsgálni az *parse*-olás okozta teljesítmény romlást. Az új modellek és elemszámaikat a 4.3. táblázat tartalmazza, az őrfeltétel a következő kifejezés:

```
true = (true and true) or a > 12
```

Megjegyzés: az őrfeltételben használt `a` változó deklarálva lett az állapotgépen ennek a leképzése is megvalósul, azonban feltételezhető, hogy ez nem befolyásolja érdemben a mérést.

Név:	m1	m2	m3	m4	m5	m6
Állapotok száma:	10	50	100	500	1000	2000
Állapotátmenetek száma:	10	46	91	461	821	1641
Signal Event Triggerek száma:	9	45	90	460	820	1640
Őrfeltételek száma:	9	45	90	460	820	1640

4.3. táblázat. Második méréshez használt modellek



4.2. ábra. Második mérés: futási idők őrfeltételekkel

Az őrfeltételek alkalmazásával keletkezett méréshez tartozó lineáris regresszió meredekebb, minta az őrfeltételek nélküli esetben. Ebből arra lehet következtetni, hogy az őrfeltételek alkalmazása nagyobb mértékben lassítja a leképzést. A mérések alapján megállapítható, hogy a mért végrehajtási idők a modellek méreteinek függvényében lineárisan növekszenek, tehát a transzformáció minden esetben jól skálázódik.

4.3. Továbbfejlesztési lehetőségek

4.3.1. IBD - Composition Language

A Gamma Statechart Composition Framework egyik célja, hogy lehetővé tegye állapottérképekből, mint komponensekből egy komplett rendszer leírását és ennek vizsgálatát. Ilyesfajta leírás SysML-ben az Internal Block Diagram(IBD), amivel egy Block felépítését lehet leírni, és a blokk részei között fennálló kapcsolatokat lehet ábrázolni.

Az IBD-k leképzésének támogatásával a felhasználók képessé válnának komponens alapú modelleket definiálni és ezeket formálisan verifikálni, ami különösen hasznos ilyen modellek esetében hiszen a különböző viselkedések külön-külön diagramokon vannak: egymásra gyakorolt hatásuk nehezen kielemezhető.

4.3.2. Szimuláció generálása

Az UPPAAL opcionálisan előállítja azokat az utakat melyek sértik a megkötéseket. A Gamma Framework képes ezekből kódot és Yakindu szimulációt előállítani. A MagicDraw is rendelkezik egy szimulátorral Cameo Simulation Toolkit[1] néven amely plug-in a No Magic terméke. Segítségével modelleket lehet debugolni, szimulálni és UI prototyping funkcionalítással rendelkezik. A szimulációkat modell elemekkel is fel lehet konfigurálni Execution Configuration Classok segítségével, ezért potenciálisan modell transzformációkkal elő lehet állítani szimulációt.

A megkötéseket sértő utak megjelenítése a szimulátor segítségével hasznos lenne, a modell debugolását illetően.

4.3.3. Validation kit

A VIATRA-va inkrementális és reaktív tulajdonsági lehetővé teszik modell transzformációk végrehajtását, ha a modellt változik. Ezt a funkcionalitást kihasználva lehetőséget kapunk, hogy létrehozzuk validációs szabályok VQL-ben leírt halmazát és ezeket futás időben folyamatosan ellenőrizve a MagicDraw API-ján keresztül felannotálhatjuk azokat az elemeket amik nem felelnek meg a ezeknek szabályoknak. Ezeket a MagicDraw megjeleníti a GUI-ján.

A validációs szabályok lehetnek figyelmeztetések, hogy melyik elemek nem képezhetőek le, vagy leképezhetőek de nem támogatott a verifikációjuk, ezzel növelve a felhasználói élményt, hogy ne a transzformációk végrehajtása alatt értesüljenek a potenciális hibákról.

[1] Cameo simulation toolkit: https://www.nomagic.com/product-addons/magicdraw-addons/cameo-simulation-toolkit

5. fejezet

Összefoglalás

Dolgozatomban a feladatnak megfelelően bemutattam az állapottérképek formalizmusát és megterveztem egy folyamatot melynek segítségével támogatni lehet ezek formális verifikációját a MagicDraw modellező eszközben. A tervezett eszköz prototípusát megvalósítottam egy plug-in formájában melynek a működését egy esettanulmányon demonstráltam, végül pedig értékeltem az elvégzett munkát és megvizsgáltam a továbbfejlesztési lehetőségeket.

Az elvégzett munka fontosabb kontribúciói:

- Megfeleltetések megtervezése
  - Összevetettem a két eszköz Metamodelljét
  - Kiválasztottam az egymásnak megfeleltethető elemeket
  - Odafigyeltem a szemantikai különbségekre
- Leképzés implementációja
  - modell transzformációkra specializált technológiát használtam
  - az eszközt MagicDraw plug-in formájában valósítottam meg
- Fejlesztőkörnyezet kialakítása
  - összegyűjtöttem a szükséges dependenciákat
  - odafigyeltem a tranzitív dependenciák helyes menedzselésére
- Lehetővé tettem, hogy a MagicDraw-n belül elvégezhető legyen a verifikáció
  - átvettem és módosítottam a Gamma Query Generátor funkcióját
- Elméleti megközelítésből értékeltem a munkám
  - esettanulmányon mutatom be az elkészült eszközt
  - áttekintetem az alternatív megvalósítási lehetőségeket

Munkám eredményéül létrejött egy olyan MagicDraw plug-in amivel lehetőség nyílik állapottérképek formális verifikációjának végrehajtásába.

5.1. Jövőben elvégzendő munka

Az eddigi munkám során létrejött eszköz továbbfejleszthető, hogy lehetővé tegye komponens alapú modellek verifikációját is. A jövőben a leképzéseket kiterjesztem, hogy ezeket a modelleket is Gamma modellé lehessen transzformálni és elvégezni a formális verifikációt a rendszeren. Továbbá megtervezek egy olyan új funkciót ami megjeleníti a felhasználóknak azokat az utakat, amik a megszorításaik megsértéséhez vezettek. A felhasználói élmény javítása érdekében validációs szabályokat hozok létre, amik futásidőben figyelmeztetik a felhasználókat azoknak az elemeknek a használatára, amelyek felhasználása nem teszi lehetővé a leképzést, vagy a formális verifikáció végrehajtását. Továbbá megfontolom olyan elemek támogathatóságát, amik az állapottérképek újrahasznosítását teszik lehetővé (*SubmachineState*).

Köszönetnyilvánítás

Szeretnék köszönetet mondani mindazoknak a feladat elvégzése során segítették a munkám: Farkas Rebekának, aki konzulensemként mindig segítőkész, pozitív hozzáállásával, és szakmai tanácsival, kritikáival segítette a dolgozat létrejöttét. Továbbá Molnár Vincének aki a Gamma Statechart Composition Framework használatát javasolta és ötleteket adott a megvalósítható funkciókra vonatkozóan. Köszönöm továbbá az IncQuery Labsnak, hogy rendelkezésemre bocsátották a plug-in skeletonjukat, és a náluk töltött szakmai gyakorlatom során szerzett gyakorlati tapasztalatokat, amik jelentősen megkönnyítették az implementáció elkészítését.

Irodalomjegyzék

[1] Johan Bengtsson – Kim Larsen – Fredrik Larsson – Paul Pettersson – Wang Yi: Uppaal—a tool suite for automatic verification of real-time systems. In *International Hybrid Systems Workshop* (konferenciaanyag). 1995, Springer, 232–243. p.

[2] Johan Bengtsson – Kim G Larsen – Fredrik Larsson – Paul Pettersson – Wang Yi – Carsten Weise: New generation of uppaal. In *Proc. Int. Workshop on Software Tools for Technology Transfer (STTT'98)* (konferenciaanyag). 1998, 43–52. p.

[3] Object Management Group: *OMG System Modeling Language*. OMG, 2017. 05. `https://www.omg.org/spec/SysML/About-SysML/`.

[4] NoMagic Inc. URL `https://www.nomagic.com/`.

[5] NoMagic Inc.: *MagicDraw Open API javadoc*. NoMagic Inc., 2017. 04. `http://jdocs.nomagic.com/185/`.

[6] Vince Molnár – Bence Graics – András Vörös – István Majzik – Dániel Varró: The gamma statechart composition framework. 2018, ICSE.

[7] Dave Steinberg – Frank Budinsky – Ed Merks – Marcelo Paternostro: *EMF: eclipse modeling framework*. 2008, Pearson Education.

[8] Yakindu Statechart Tools: Yakindu.
URL `https://www.itemis.com/en/yakindu/state-machine/`.

[9] Leonardus MM Veugen: The framework of an embedded software controller using the transaction mechanism. 2012.

Függelék

F.1. Elemek megfeleltetése

MagicDraw	Gamma	Verifikálható
Vertex (absztrakt)	StateNode (absztrakt)	-
State	State	igen
CompositeState	State, egy belső régió	igen
OrthogonalState	State, több belső régió	igen
InitialState	InitialState	igen
TerminalState	nincs	-
FinalState	állapot 'FinalState' néven	igen
EntryPoint	nincs	-
ExitPint	nincs	-
Conn. Point Reference	nincs	-
SubmachineState	nincs	-
ForkState	ForkState	nem
JoinState	JoinState	nem
Choice State	Choice State	nem
Junction	Merge	nem
DeepHistory	DeepHistory	igen
ShallowHistory	ShallowHistory	igen
Region	Region	igen
Transition	Transition	igen
Signal, SignalEvent	Event	igen
Trigger, SignalEvent	EventTrigger	igen
Trigger, TimeEvent(rel)	TimeoutDeclaration, SetTimeoutAction, TimeoutTrigger	igen
Tigger, TimeEvent(abs)	nincs	-
Property(Integer)	VariableDeclaration(IntegerSpec.)	igen
Property(Boolean)	VariableDeclaration(BooleanSpec.)	igen
Action, FunctionBehavior	SendSignalAction, AssignMentAction	igen
Guard(OpaqueExpression)	Expression	igen