



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 3.

Tartalomjegyzék

Kivonat	i
Abstract	ii
1. Bevezetés	1
2. Kapcsolódó munkák	3
2.1. Gamma Statechart Composition Framework	3
2.2. TismTool	3
3. Háttérismeretek	4
3.1. Modellek	4
3.2. Állapot alapú modellezés	4
3.2.1. Állapottérképek UML 2-ben	4
3.3. Eclipse Modelling Framework	8
3.4. MagicDraw	9
3.4.1. Állapottérképek SysMLben	9
3.4.2. Plug-in fejlesztése MagicDrawhoz	9
3.5. Viatra	10
3.6. Formális verifikáció	10
3.7. Gamma Framework	10
4. MagicDrawToGamma plugin	13
4.1. Konceptió	13
4.2. Fejlesztőkörnyezet	13
4.3. MagicDraw - Gamma transzformáció	14
4.3.1. Gyökér elemek létrehozása:	14
4.3.2. Alap struktúra kialakítása:	15
4.3.3. Pszeudoállapotok átalakítása:	16
4.3.4. Állapotátmenetek leképzése:	16
4.3.5. Változók leképzése	17
4.3.6. Események, triggerek leképzése:	17
4.3.7. Örfeltételek leképzése:	17
4.3.8. Akciók leképzése:	18
4.3.9. Trace Modell	19
4.4. A Verifikáció végrehajtása	19
4.5. Esettanulmány	21
4.5.1. Szemléltető példa	21
4.5.2. Verifikáció menete	21
4.5.3. A probléma:	23
4.5.4. Megoldás	23

5. Értékelés	24
5.1. Alternatíva - kódgenerálás	24
5.2. Plug-in teljesítménye	24
6. Továbbfejlesztési lehetőségek	25
6.1. IBD - Composition Language	25
6.2. Szimuláció generálása	25
6.3. Validation kit	25
7. Összefoglalás	26
7.1. Jövőben elvégzendő munka	26
Köszönetnyilvánítás	27
Irodalomjegyzék	28
Függelék	29

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 3.

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 leképezést a két eszköz modelljei között.

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

Abstract

The development of complex systems makes it necessary to gain access modern modelling tools that support functionalities like validation, code generation and verification. One of the most well-known tool in the industry is MagicDraw: a tool for designing 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 has very rich a functionality 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 support 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. Furthermore I describe in details the concepts and technologies 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 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 széleskörű alkalmazott eszköz a MagicDraw, ami többek között UML és SysML nyelveken teszi lehetővé modellek készítését. Ezeken magas szinten leírható egy rendszer felépítése és működése. MagicDraw-t gyakran alkalmaznak kritikus rendszerek modellezésére melyek esetében elengedhetetlen, hogy matematikai precizitással megbizonyosodjunk működésének 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 nevű eszköz a tanszéken fejlesztett állapottérképek modellezésére és ezek fejlesztésének támogatására készült eszköz, ami elsősorban Eclipse 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 leképezést biztosítok 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 hasonló hiszen az eszköz már megvalósít egy leképezést saját nyelve: a már említett Yakindu - Gamma leképezést. A különbség, hogy a Gamma meta-modellje 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 Eclipse-től. Bár létezik MagicDraw - Eclipse integráció felhasználói szempontból előnyös lenne, az eszközt önmagában lehetne használni.

A dolgozat felépítése a következő: a *2. fejezetben* felsorolok pár eszközt ami hasonló funkcionalitással rendelkezik, mint amit a feladat során elkészítettem, a *3. fejezetben* bemutatom a munkám megértéséhez szükséges alapismereteket. A *4. fejezetben* ismertetem az alkalmazott megoldást elméleti és gyakorlati szempontból, továbbá egy esettanulmányon szemléltetem az eszköz működését. Az *5. fejezetben* értékelem az alkalmazott megoldásokat és mérésekkel ellenőrzöm a teljesítményt, a *6. fejezetben* ismertetem azokat a továbbfejlesztési lehetőségeket, amiket még a Gamma keretrendszer segítségével még implementálni lehetne végül a *7. fejezetben* összefoglalom a dolgozatban leírtakat.

2. fejezet

Kapcsolódó munkák

2.1. Gamma Statechart Composition Framework

A Gamma Statechart Composition Framework[6] egy tanszéken fejlesztett eszköz, amit komponens alapú viselkedés modellezést tesz lehetővé. Az eszköz képes kódot generálni és a formális verifikáció végrehajtására, továbbá képes Yakindu modelleket áttranszformálni Gamma modellekké. A Yakindu-Gamma transzformáció megvalósítása hasonló mint az általam implementált MagicDraw-gamma transzformációé. A Gamma keretrendszerrel a 3.7 szakasz foglalkozik részletesebben.

2.2. 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. A TismTool 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 fájljából amit egy futtatókörnyezettel kell végrehajtani továbbá nem formális alapokon hajtja végre a verifikációt, hanem ún. futás idejű verifikációt¹ használ.

¹<http://www.tismtool.com/tooling.html>

3. fejezet

Háttérismeretek

3.1. Modellek

Modelleket a tudomány számos területén alkalmazunk, ez lehetővé teszi az előtt vizsgálni a megvalósítandó rendszert, 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 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.

3.2. Állapot alapú modellezés

Az állapottérkép egy diagram ami irányított gráfot tartalmaz, ahol a csomópontok állapotokat, az élek állapotátmeneteket definiálnak. Az így kapott gráfot állapotgépnak 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.

3.2.1. Állapottérképek UML 2-ben

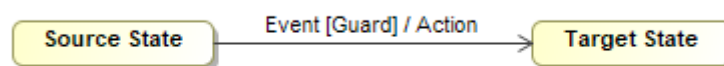
Előzőekben az állapot alapú modellezés alapjai kerültek bemutatásra. A következőkben az állapottérképek egy konkrét specifikációja kerül bemutatásra az UML 2 szerint.

- *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, régióként egy szerepelhet belőle és a régió belépési pontjaként szolgál. Jelölése fekete színezett kör (3.2 ábra).
- *Végállapot (Final State)*¹ (final state): A végső állapot, a régió terminálási pontja, ha egy állapotgép összes régiója egy végső állapotba ért akkor az állapotgép is terminál. Szimbóluma fehér körben egy kisebb színezett fekete kör (3.2 ábra).
- *Termináló állapot (Terminal State)*: pszeudoállapot, ami az egész állapotgépet azonnal terminálja. Ezt hibák lekezelésére lehet például alkalmazni, jelölése kis kereszt (3.2 ábra).
- *Állapot átmenet (Transition)*: az állapottérképek élei, a lehetséges állapotváltozásokat definiálják. Az állapotátmenetek rendelkeznek:
 - Triggerekkel
 - Örfeltételekkel
- *Trigger*: Állapot váltást kiváltó esemény amely lehet:
 - 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.

- *Örfeltétel (Guard)*: tágabb értelemben egy logikai kifejezés, melynek teljesülnie kell, hogy az adott állapotátmenet bekövetkezhesen. UML-ben ezek megszorítás-kén (*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, Interakció² OpaqueBehavior³



3.1. ábra. Állapotok és köztük definiált állapotátmenet, triggerrel, örfeltétellel és actionnel

¹UML 2-ben a végső állapotot nem pszeudoállapotként hanem állapotként van definiálva <https://www.omg.org/spec/UML/2.0>

²Interakció modell elemek között, leírásához a jellegétől függően többféle diagram használható (Szekvencia, Kommunikációs, Időzítés).

³szöveges, UML-től eltérő nyelvvel specifikált viselkedés.

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 pseudo állapot, amely egy régióban megjegyzi az utolsó aktív állapotot, kilépéskor, ha a régióba visszalépünk a history state visszaállítja a megjegyzett állapotot. Amennyiben nincs előző állapot az ő belőle húzott állapotátmenet cél állapota lesz aktív. Két féle History State 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 *HistoryState* szintaktikája a 3.2 ábrán látható.



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

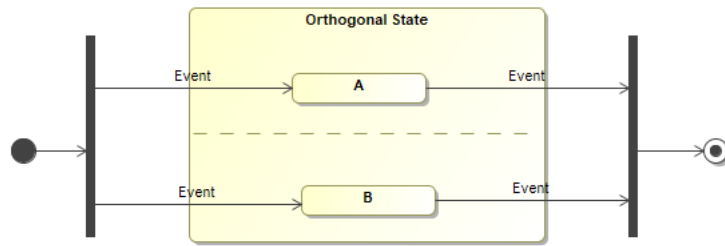
Rendszerünket egy időpillanatban több egymástól független állapot is jellemezheti. Ezt a viselkedést párhuzamos régiók alkalmazásával lehet leírni.

- *Ortogonalis állapot (Orthogonal State)*: olyan összetett állapot ami két vagy több régiót tartalmaz.
- *Fork*: pseudoállapot, ami egy beérkező átmenetet szétbont több átmenetre, amiknek a cél állapotuk ortogonalis régiókban található. A kimenő átmeneteken nem lehet se trigger, sem pedig őrfeltétel.
- *Join*: pseudoállapot, ami több beérkező átmenetet kapcsol össze egyé. Az átmenetek ortogonalis 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 3.3 ábrán látható.

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

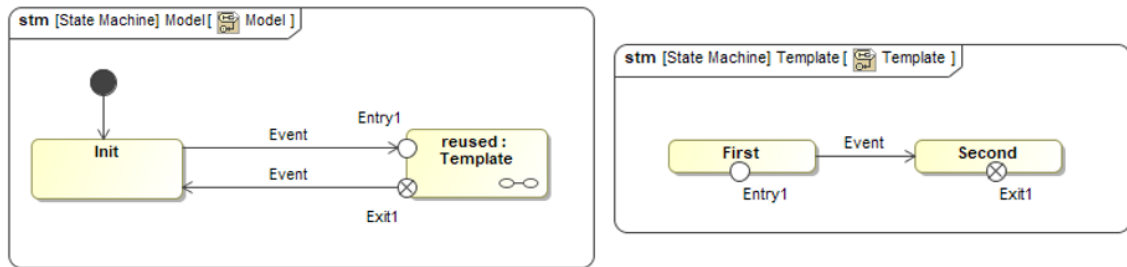
- *Submachine state*: Egy olyan állapot, ami egy állapottérképre hivatkozik, ez lehetővé teszi, hogy egy állapottérkép többszöri felhasználását, akár más-más kontextusban



3.3. ábra. Példa: fork-join

- *Belépési pont(Entry Point)*: egy pseudoállapot, 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 állapotterkép 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.
- *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 3.4 ábra szemlélteti.



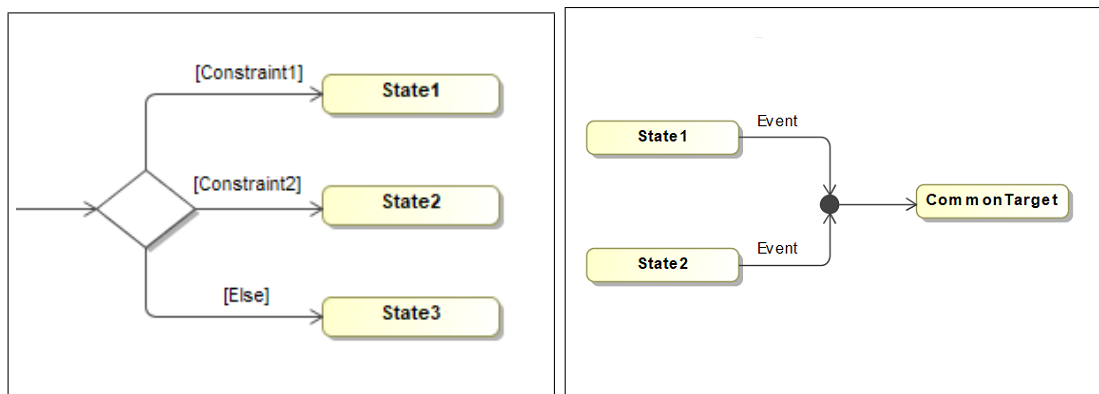
3.4. ábra. Állapottérkép újrafelhasználása Submachine State segítségével

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

- *Csomópont (Junction)*: pseudoállapot, 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 ágnak nevezni.
- *Döntés (Choice)*: hasonló mint a csomópont, viszont az őrfeltételek az elágazásba való belépéskor értékelődnek ki dinamikusán. 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* örfeltétel alkalmazásával.

A csomópont szintaxisa egy a kezdő állapotnál kisebb színezett kör, a döntése pedig egy rombusz. (3.5 ábra).



3.5. ábra. Döntés és Csomópont szintaktikája

3.3. 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 meta-modellt és példány modellt. A példány modell struktúráját a meta-modell írja le. A modell egy konkrét példánya a meta-modellnek.

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 meta-modell **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*-ekben találhatók. A modellek közötti hivatkozások feloldásához a modelleket tartalmazó *Resource*-oknak azonos *ResourceSet*-ekben kell lenniük.

Eclipses környezetekben az EMF-nek nagy jelentősége van. Xtext⁴-el együtt alkalmazva saját szakterület specifikus nyelveket lehet fejleszteni saját nyelvtannal és absztrakt szintaxissal.

⁴Xtext: <https://www.eclipse.org/Xtext/>

3.4. 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.

A SysML egy általános-célú modellezési nyelv ami az UML egy részének kiragadásával és annak kibővítésével keletkezett. SysML-el 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 megemlíthető diagramfajta az Internal Block Diagram, amivel a Blokkok és portjaik között tudunk kapcsolatokat definiálni.

Viselkedést Állapottérképekkel és Activity Diagrammokkal szokás leírni SysMLben. Utóbbival munka- és adatfolyamokat, ahol a folyamat lépései activityk és actionök. Állapottérképekkel reaktív rendszereket szokás leírni, a rendszer eseményekre reagál, ezek határozzák meg a viselkedését, szemben az Activity Diagrammokkal, amivel adott bemenetből valamilyen kimenet előállításának folyamatát írjuk le.

3.4.1. Állapottérképek SysMLben

Az állapotterképeket SysML modellek részeként, más modell elemekhez vannak viselkedésként hozzárendelve, a triggereket aktiváló események a modell strukturális leírásában vannak definiálva. Mivel a dolgozat kizárólag állapotterképekkel foglalkozik ezért implicit feltételezzük, hogy minden állapotterképhez tartozik egy Blokk, egy Blokk viselkedését pontosan egy állapotterkép írja le. A Blokkon definiálva van egy Port ami az események fogadásáért felelős. Minden állapotátmenet ettől a Porttól várja az eseményeket.

3.4.2. 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 plug-ins 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 meta-modellje szerint lesznek példányosítva megfelelően sztereotipizálva, ezért kód szinten is eszerint érhetőek el.

A modell elemek ugyan saját MagicDraws implementációval rendelkeznek, de EMF⁵-es interfaceket is realizálnak ezért lehetőségünk van EMF Apijának használatára a plugin fejlesztése során.

3.5. Viatra

Az Eclipse Viatra Framework⁶ egy modell transzformációs eszköz, ami lehetővé teszi modellek hatékony eseményvezérelt transzformációját és lekérdezését. A modell lekérdezésekhez egy külön nyelvet Viatra Query Language(VQL)-t használ. VQL lekérdezésekből Java osztályok generálódnak, melyek segítségével a lekérdezések és transzformációk futtathatók.

Viatra használatában MagicDraw plugin fejlesztése során is van lehetőség. A Viatra for MagicDraw egy plug-in ami lehetővé teszi a Viatra Api használatát más plug-inokban.

3.6. 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(*excursion trace*) ami a megszorítás megsértéséhez vezetett. Hátrányuk, hogy nehezen skálázható, erőforrás igényes és egy összetett rendszert formális módszerekre való visszavezetése gyakran nehézkes.

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

3.7. Gamma Framework

A Gamma Statechart Composition Framework komponens alapú reaktív rendszerek tervezésére, validálására, verifikálására és kód generálásra lett létrehozva. Az eszköz egyik alap komponensei az állapotterlépek amiknek a leírásához egy saját nyelvet Gamma Statechart Language-t (3.6. 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 (3.7. ábra) transzformálni és feldolgozni.

⁵Eclipse Modelling Framework <https://www.eclipse.org/modeling/emf/>

⁶Viatra: <https://projects.eclipse.org/projects/modeling.viatra>

Az eszköz egy másik komponensei az ún. Kompozit komponensek, amiket Gamma Composition Language lehet definiálni. Ezek írják le a rendszer felépítését az állapotterképek portjait, az azok által realizált interfészeket és a közöttük 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 deklarálандók 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.

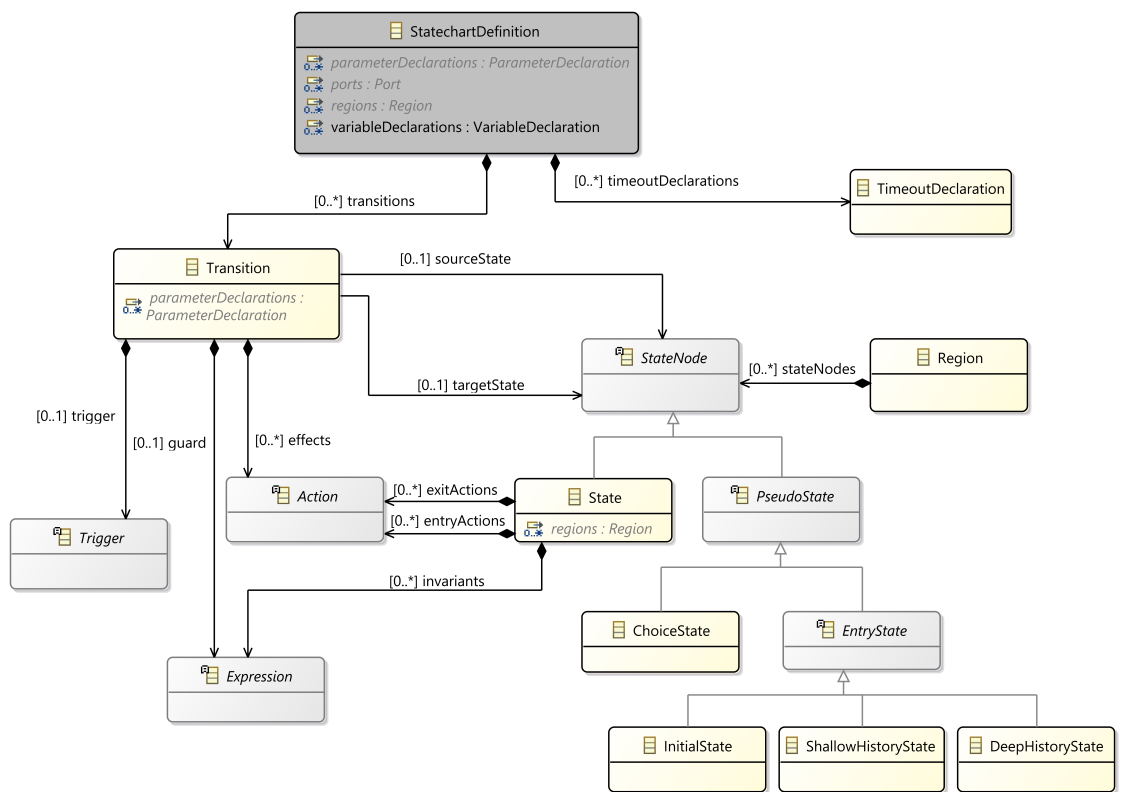
```
package Exapmle

statechart MonitorStatechart [] {

transition from Red to Blue
transition from Entry0 to Red

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

3.6. ábra. Gamma Statechart konkrét szöveges szintaxisa



3.7. ábra. Gamma állapottérkép absztrakt szintaxisa

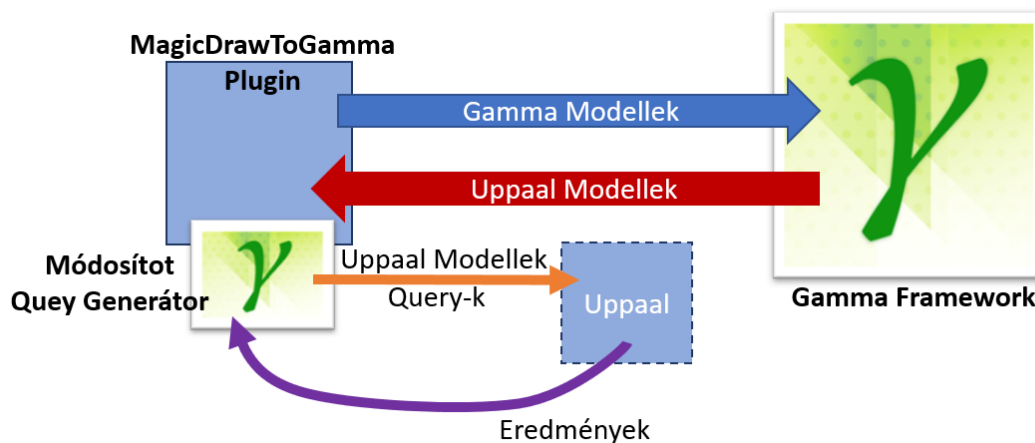
4. fejezet

MagicDrawToGamma plugin

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

4.1. Konceptió

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 az keretrendszer kezelni tudja, a verifikáció elvégzéséhez az eszköznek csak egyes részei szükségesek. A megoldást a 4.1 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.



4.1. ábra. Konceptió

4.2. Fejlesztőkörnyezet

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

talos *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 project, viszont Gradlet használ a projekt fordításához és a dependenciák kezeléséhez. Ez sokszor inkonzisztenciákhoz vezetett az egyik legnagyobb probléma a Viatra Querik generálása csak Eclipse-vel lehet generálni. A kódbasis 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 kódbasis azonban kompatibilis lehet még az újabb verziókkal is, amennyiben az állapottérképeket érintő meta-modellek nem változnak.

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

4.3. MagicDraw - Gamma transzformáció

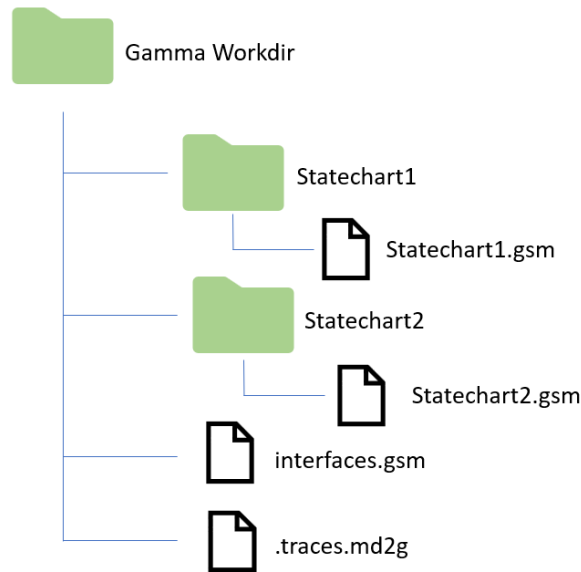
A MagicDraw - Gamma transzformációt egy menü elemmel lehet elindítani. Ehhez szükséges, hogy a projekthez hozzá legyen rendelve egy külső könyvtár (Gamma Workspace) ami a leképzett és perzisztensen eltárolandó modellek helyét jelöli. A hozzárendelt könyvtár abszolút elérését String formában a projekt 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 állapottérkép el lett távolítva a MagicDraw modellből és utána újra le lett képezve. Ebben az esetben a régebben leképzett Gamma állapottérképek is megmaradnak.

4.3.1. Gyökér elemek létrehozása:

A plugin a transzformáció elején létrehoz egy *ResourceSet*-et, amibe 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 visszakereshetőségül szolgál (ld. 4.3.9. szakasz), 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 Signal Eventeket és létrehoz a Signaléval azonos néven egy Eventet, a Gamma meta-modelljének megfelelően. A létrehozott eventek egy Interfacen kerülnek definiálásra aminek a neve megegyezik az állapottérképével. Az interface ezután belekerül az interfaceket tároló *Resource*-ba. Az eventek iránya INOUT ez lehetővé teszi azt is, hogy az állapotgép magának küldjön eseményt.

A leképzett állapottérképek külön Resourceokba kerülnek amik a Gamma Workspace-ben egy külön az állapottérkép nevével megegyező könyvtárba kerülnek, ugyan ezen a néven *.gsm* kiterjesztéssel (4.2-es ábra). (A *Resource* gyökere nem egy *StatechartDefinition*, hanem egy *Package*, ennek a neve szintén megegyezik az állapottérkép nevével)



4.2. ábra. Létrehozott fájlstruktúra

4.3.2. Alap struktúra kialakítása:

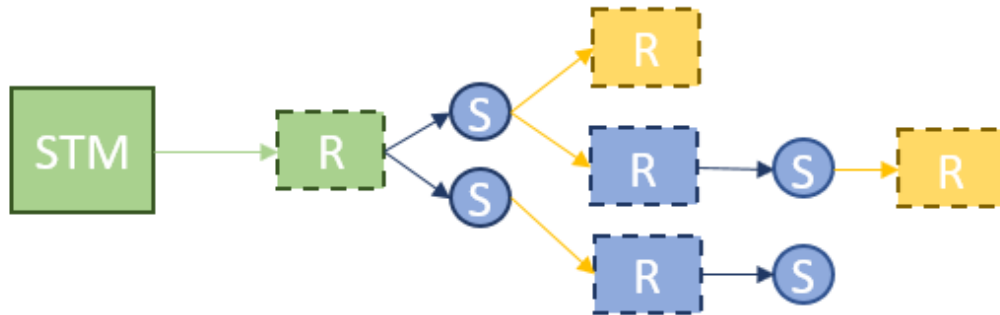
Az előző lépések során azok az elemek állnak elő, melyek a modellek gyökér elemeiként fognak szolgálni. A következő lépés létrehozni az állapotterképek egy belső, alap struktúráját, amit az állapotok és állapotátmenetek határoznak meg. A Viatra for MagicDraw projekt megnyitásakor készít egy ViatraQueryEngine-t 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 történő keresések, és a visszakövetések ezzel az engine-el történnek. Alap struktúra kialakításának lépései:

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

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, ugyan ezt 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 kihasznált és elvárt, továbbá az is ki van használva, hogy régió csak *State*ben és Állapotgépben lehet. A régiókat tartalmazó állapotok kompozit és ortogonális állapotnak tekintendők.

A tartalmazási gráf összefüggővé válását a 4.3 ábra szemlélteti.

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



4.3. ábra. Állapotok és régiók leképezésének menete, 1. lépcső: zöld, 2. lépcső: kék, 3. lépcső: sárga

4.3.3. Pszeudoállapotok átalakítása:

Az állapotátmenetek leképezése előtt a pszeudoállapotok kerülnek leképezésre. Ezen a ponton már létezik minden régió amit tartalmazhatja őket. A leképezés legtöbb esetben támogatott viszont, egyes elemek tartalmazása esetén nem lehet a verifikációt végrehajtani. Az elemeket és párjaikat a 4.1 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	-

4.1. táblázat. Pszeudoállapotok párosítása.

4.3.4. Állapotátmenetek leképezése:

A következő lépés az állapotátmenetek átalakítása. Ezen a ponton már az összes olyan elem leképezésre 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 patternekkal 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 TransitionsInStateMachine(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.

4.3.5. 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 során kétféle primitív típus esetén támogatott: *Integer* és *Boolean*, egyéb típusok leképzése még nem támogatott. A változók tulajdonságaira vonatkozó megkötéseket a 4.2 táblázat foglalja össze.

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

4.2. táblázat. Változók lehetséges értékei.

4.3.6. Események, triggerok leképzése:

A MagicDrawban definiálható *Triggerok* pontosabban az őket kiváltó események közül jelenleg kettő támogatott. Egyik a *SignalEvent* a másik pedig *TimeEvent*. Előbbi *Event-Triggerre* 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 két féle 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 az időt kell megadnia trigger felvételekor a MagicDraw modellben.

4.3.7. Örfeltételek leképzése:

Örfeltételek definiálása MagicDraw állapottérképeken Opaque Expressionökkel¹ 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 Modell* példánymodellre fordítani.

¹Szöveges nyelven leírt kifejezés

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

```
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 példány-modellekből.

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

Mivel a guardok megírása implicit a gamma nyelvvel történik azért ugyan azokat az operátorokat támogatja, mint a gamma.

Név	Operátor	leírás	Csoportosítás
Implikáció	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

4.3. táblázat. Támogatott operátorok.

4.3.8. Akciók leképzése:

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.

Kétféle akció támogatott: értékadás és event küldés. Ezek szintaxisa az alábbi:

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

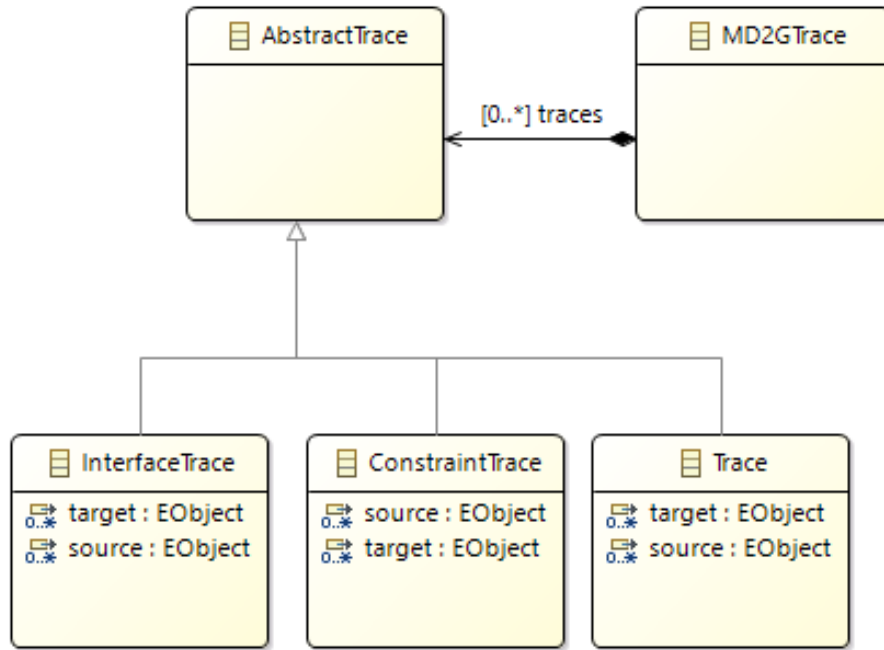

Az *All* nem Gamma specifikus jelölése az össze portnak, ebből jelenleg csak egy van egy implicit generált üzenet fogadáshoz, így ennek nem szükséges tudni a nevét.

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

4.3.9. Trace Modell

A leképezések végrehajtása során fontos követni, hogy mely elemek képződtek le és mely elemekké. Erre a célra a MagicDrawToGamma bevezet egy Trace modell² nevű segédstruktúrát, ami lehetővé teszi a megfeleltetések visszakereshetőségét Viatrával, továbbá a modell sorosítása és háttértáron való tárolása is megoldott. A modell EMF-ben definiálva és háromféle *Trace*-t különböztet meg. Az Gamma állapottérképek alap elemeit *Trace*-ek az interfészek elemeit *InterfaceTrace*-k és a *Constraint* modell elemeit *ConstraintTrace*-ek kötik össze MagicDraw párjukkal, amiből le lettek képezve.

A három *Trace* típust egy *AbstractTrace* osztály fogja össze, és MD2GTrace-ben tárolhatók. (4.4 ábra)



4.4. ábra. Trace modell EMF definíciója

4.4. 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

²Nem összekeverendő a a 3.6 fejezetben említett *Excursion Tracel*

2. Uppaal Query Generator

Első lehetőség időzített automaták leírásának előállítását 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.

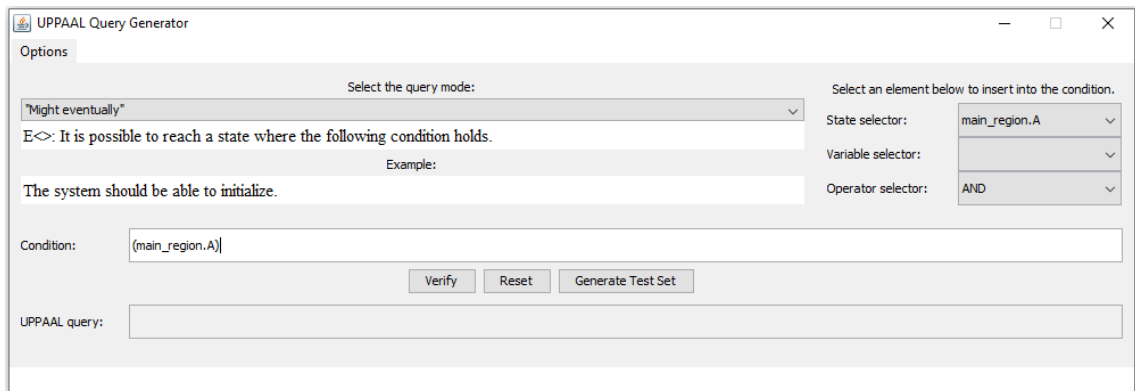
```
//initialize with a Gamma Package
StatechartToUppaalTransformer transformer = new StatechartToUppaalTransformer(p);
SimpleEntry<NTA, G2UTrace> entry = transformer.execute();
```

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 serializálni 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 állapotterkép könyvtárba kerülnek, ahonnan a felhasználó megnyithatja őket *UPPAAL*-ban.

A második lehetőség az *Uppaal Query Generator* (4.5 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.



4.5. ábra. UPPAAL Query Generator

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

³a formális verifikáció az időzített autómáták formalizmusán történik

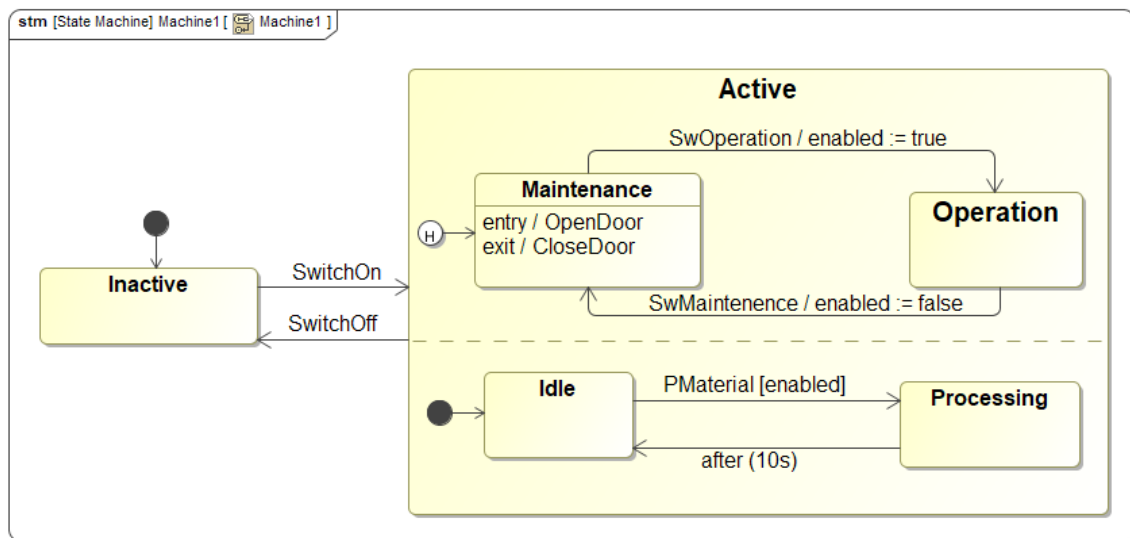
4.5. Esettanulmány

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

4.5.1. Szemléltető példa

Példa specifikációja: egy munkagépbe valamilyen anyagot lehet tölteni amit a gép tíz másodperc alatt feldolgoz. A gép az első bekapcsoláskor előbb szervíz 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 szervíz vagy üzem állapotban volt-e és abba kapcsol vissza.

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



4.6. ábra. Lehetséges leírás

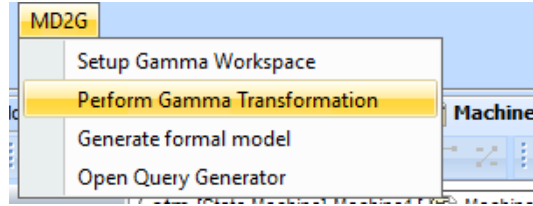
Megjegyzés Az **OpenDoor** és **CloseDoor** akciók csak szemléltetés jellegűek, ezek nem kerülnek leképzésre.

Feladat: Ellenőrizni szeretnénk, hogy ne forduljon elő, hogy egyszerre van **Maintenance** és **Processing** állapotban a rendszer, tehát nyitva van az ajtó miközben még feldolgoz a rendszer, hiszen ez veszélynek tenné ki a dolgozókat.

4.5.2. Verifikáció menete

Ahhoz, hogy le tudjuk ellenőrizni, hogy a leírt működés megfelel-e a támasztott feltételeknek végre kell hajtánunk a MagicDraw - Gamma transzformációt. Ezt a MD2G menüpont

alatt található **Perform Gamma Transformation** kiválasztásával lehet megtenni (4.7 ábra).



4.7. ábra. MD2G menüpontok

A menüpontot kiválasztva előállíthatjuk az általunk kiválasztott Gamma Workspace-be 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 nevem most Machine1 ez látható a 4.6. áran címkeként is).

A formális modell előállítás 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.

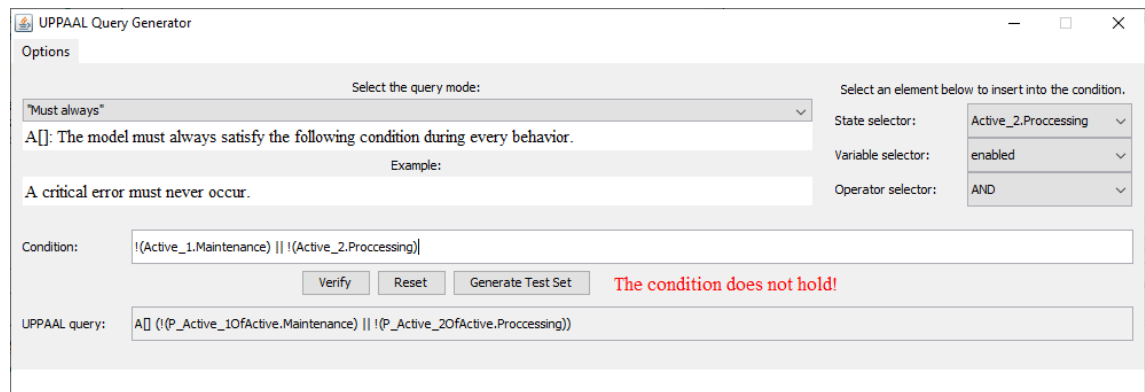
Feltétel megfogalmazása: a feltétel amit megszeretnénk fogalmazni, hogy nem lehet a rendszer egyszerre **Maintenance** és **Processing** állapotban.

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

A kifejezést a Gamma Query Generátorba 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 . ábra szemlélteti.



4.8. ábra. Verifikáció eredménye

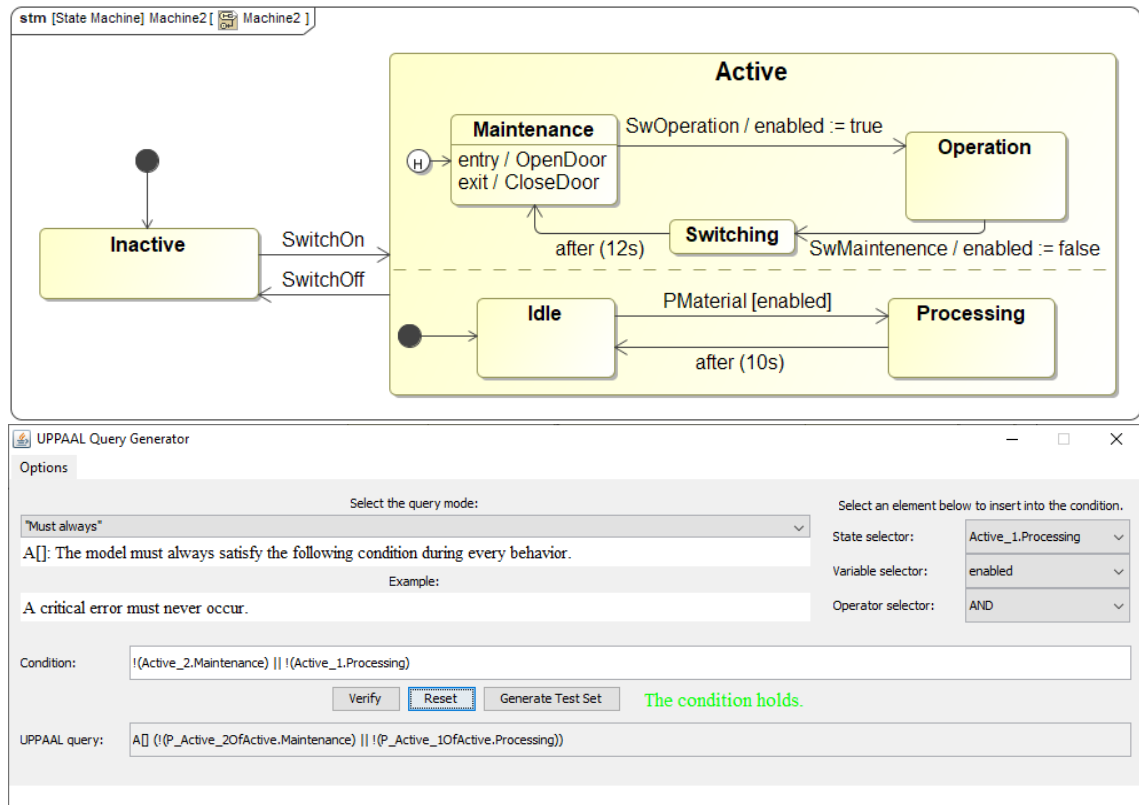
Eredmény: A megszorítások nem teljesülnek, ennek okát sajnos a plug-in még nem képes levezetni, ezt nekünk kell kitalálni vagy az UPPAAL segítségével megnyithatjuk a formális modellt és az generált UPPAAL query segítségével azon belül is el tudjuk végezni a verifikációt és tanulmányozni a generált végrehajtási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

4.5.3. 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 a tíz másodperc letelte előtt a gépet szerviz állapotba kapcsolják így egyszerre lesz Maintenance és Processing állapotban amíg a feldolgozásra szánt idő le nem telik.

4.5.4. 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 az állapotgép, ilyenkor a feldolgozás leáll. 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 állapotterképet az új állapottal (**Switching**) a 4.9. ábra mutatja.



4.9. ábra. Módosított állapotterkép és a verifikáció eredménye

5. fejezet

Értékelés

5.1. 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.

5.2. Plug-in teljesítménye

6. fejezet

Továbbfejlesztési lehetőségek

6.1. IBD - Composition Language

A Gamma Statechart Composition Framework egyik legfontosabb funkciója, hogy lehetővé teszi állapottérképekből, mint komponensekből egy komplett rendszer leírását. Ilyesfajta leírás SysML-ben az Internal Block Diagram (IBD).

Az IBD-k leképezésének támogatásával a felhasználók képessé válnának komplex reaktív rendszereket leírni és verifikálni.

6.2. 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 Toolkit¹ néven. A Cameo szimulátor plug-in a No Magic terméke. Segítségével modelleket lehet debugolni, szimulálni és UI prototyping funkcionálitással rendelkezik.

A szimulációkat modell elemekkel is fel lehet konfigurálni Execution Configuration Classok segítségével, ezért potenciálisan modell transzformációkkal elő lehet állítani szimulációt.

6.3. 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 létrehozunk 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.

Ezek 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.

¹Cameo simulation toolkit: <https://www.nomagic.com/product-addons/magicdraw-addons/comeo-simulation-toolkit>

7. fejezet

Összefoglalás

- Megfeleltetések megtervezése
 - Összevettem a két eszköz Meta-modelljét
 - Kiválasztottam az egymásnak megfeleltethető elemeket
 - Odafigyeltem a szemantikai különbségekre
- Leképezés 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 eredményeül létrejött egy olyan MagicDraw plug-in amivel lehetőség nyílik állapottérképek formális verifikációjának végrehajtására.

7.1. Jövőben elvégzendő munka

1. Komponens alapú modellezés támogatása
2. Megkötések megsértéséhez vezető utak megjelenítése
3. Leképezhető elemek validációja
4. Újrahasznosíthatóság támogatása (*Submachine State*)

Köszönetnyilvánítás

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

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