FELADATKIÍRÁS

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

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

MagicDraw állapottérképek formális verifikációja

Szakdolgozat

Készítette
Gáti László Dávid

Konzulens
Farkas Rebeka

2018. november 24.

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. november 24.

Gáti László Dávid
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

1. fejezet

Kapcsolódó munkák

2. fejezet

Háttérismeretek

2.1. Modellek

Modelleket az élet számos területén alkalmazunk, ez lehetővé teszi az előtt vizsgálni a megvalósítandó rendszert, hogy azt ténylegesen létre kellene hozni. A vizsgálatoknak számos módja és célja lehetséges. A látvány tervező 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.

2.2. Statechart formalizmus

Az állapottérkép egy diagram ami irányított gráfot tartalmaz, ahol a csomópontok állapotokat, az élek állapotátmeneteket definiálnak. Az állapotok és átmenetek egy állapot alapú viselkedést írnak le amit állapotgépnek nevezünk. 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 az állapot inaktívvá válik a cél pedig aktívvá (hurok él is lehetséges).

A viselkedés indulásakor 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.

2.2.1. Állapottérképek UML 2-ben

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

- *Állapot (State)*: az állapottérképek csomópontjai, a rendszer működésének egyfajta jól megkülönböztethető fázisai, amelyek valamilyen esemény bekövetkezésére várnak. 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. Szintaktikája fekete színezett kör (2.2 ábra).

- *Végállapot (Final State)*[1] (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).

- *Terminálo állapot (Terminal State)*: pszeudoállapot, ami az egész állapotgépet azonnal terminálja, ezt hibák lekezelésére lehet például alkalmazni, szintaktikája 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:

  - Változás esemény (Change Event): valamilyen értéknek a megváltozása.
  - Üzenet esemény (Message Event): valamilyen üzenet típusú objektumnak az érkezése, amit 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, egy jelnek (*Signal*) a 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[2] OpaqueBehavior[3]

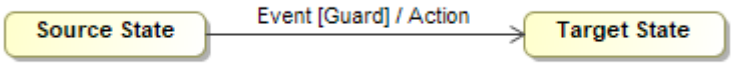
Gyakran előfordul, hogy általánosabb állapotot célszerű felbontani részállapotokra. Az egyszerű állapottérképek elemeivel, ez a fajta hierarchikus viszony az állapotok között nehezen ábrázolható, ezért célszerű további elemek használata.

- *Régió*: állapotokat tartalmazó egység, az állapottérkép mindig tartalmaz egy régiót amibe az állapotok definiálhatók. Régiók létezhetnek egymással párhuzamosan ilyenkor a végrehajtásuk párhuzamosan történik.

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

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

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



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

- *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, 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 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, Terminálló állapot és Végállapot

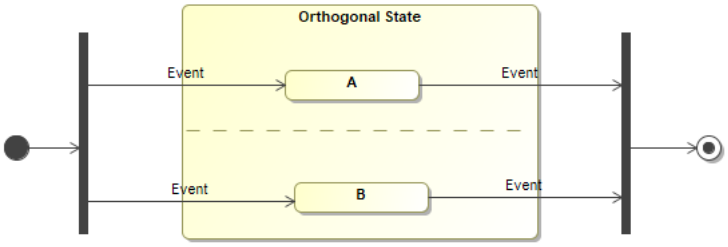
Rendszerünket egy időpillanatban több egymástól független állapot is jellemezheti. Ezt a viselkedést párhozamos régiók alkalmazásával lehet leírni.

- *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. Szintaktikájuk a 2.3 ábrán látható.

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

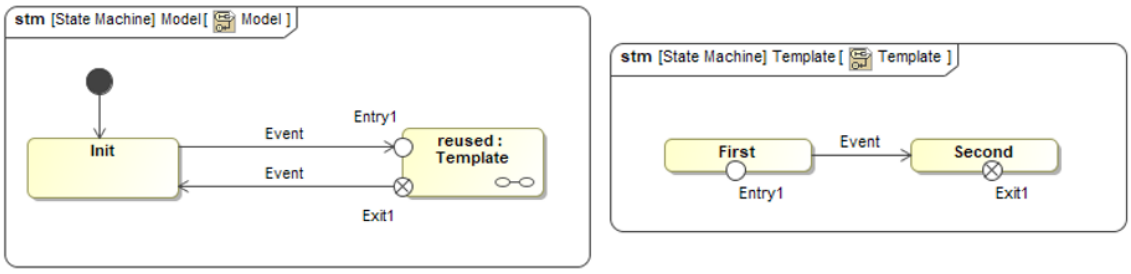
- *Submachine state*: Egy olyan állapot, ami egy állapottérképre hivatkozik, ez lehetővé teszi, hogy egy állapottérkép többszöri felhasználását, akár más-más kontextusban



2.3. ábra. Példa: fork-join

- *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. Szintaktikája kis fehér kör.
- *Kilépési pont(Exit point)*: mint a belépési pont, de ez kilépési pontot reprezentál, szintaktikája 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 2.4 ábra szemlélteti.



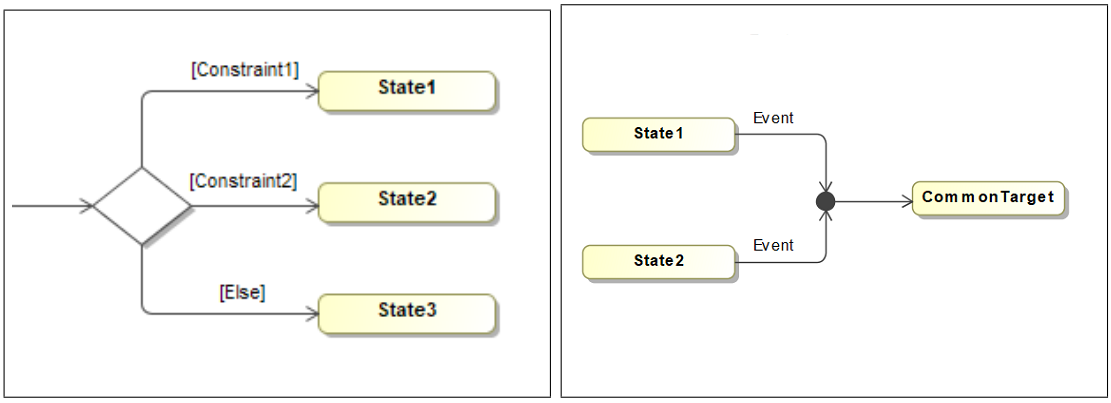
2.4. ábra. Állapottérkép újrafelhasználása Submachine State segítségével

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 ágnak nevezni.
- *Döntés (Choice)*: hasonló mint a csomópont, viszont az őrfeltételek az elágazásba való belépéskor értékelődnek ki dinamikusan. 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).



2.5. ábra. Döntés és Csomópont szintakitkája

2.3. 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. 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 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.

2.3.1. Állapottérképek SysMLben

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

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

A modell elemek ugyan saját MagicDraws implementációval rendelkeznek, de EMF [4]-es interfaceket is realizálnak ezért lehetőségünk van EMF Apijának használatára a plugin fejlesztése során.

2.4. Viatra

Az Eclipse Viatra Framework[5] 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.

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

2.5. 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, szimulációkkal, vagy formális matematikai módszerekkel.

A formális módszerek előnye, hogy a helyességük matematikailag garantált, nem szükséges az elvárt kimenetek meghatározása, csak megkötések megfogalmazása. Továbbá teljesek ezért nem kell lefedettséggel foglalkozni mint tesztelésnél. Megkötés sérülésekor, vissza lehet követni, azt az utat ami a megszorítás megsértéséhez vezetett.

Hátránya, 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 állapottérképek verifikációjának elvégzésére. Ehhez az Uppaal [1] [2] nevű eszközt használja fel.

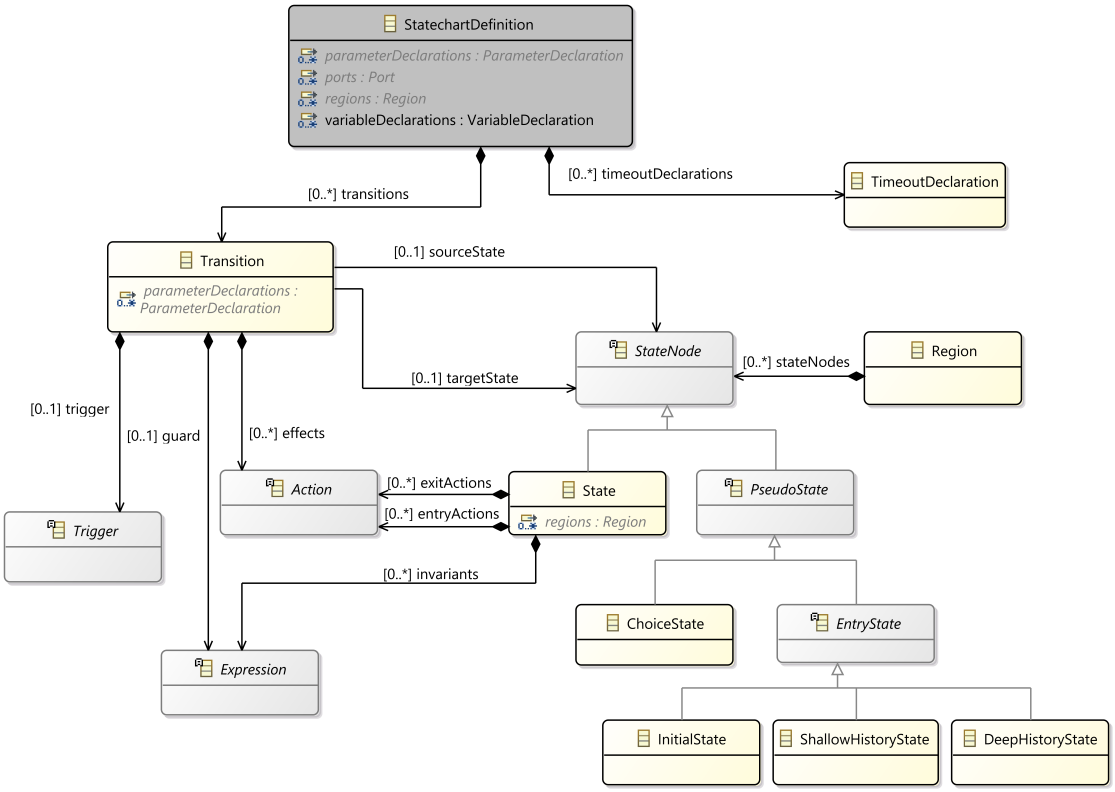
2.6. Gamma Framework[6]

A Gamma Statechart Composition Framework komponens alapú reaktív rendszerek tervezésére, validálására, verifikálására és kód generálásra lett létrehozva. Az eszköz egyik alap komponensei az állapottérlépek amiknek a leírásához egy saját nyelvet Gamma Statechart Languaget definiál, ami lehetővé teszi külső modellező eszközök integrálását is. Az első integrált eszköz a Yakindu Statechart Tools [7], aminek a modelljeit Gamma képes a saját nyelvére (2.6 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 az állapottérképek portjait, az azok által realizált interfészeket és a közöttük definiált kapcsolatokat.

A keretrendszer még két saját nyelvet definiál. Az egyik a Gamma Constraint Language amiven constrainteket lehet definiálni, ami egy általános módja típus definíciók, változók, függvények deklarálásához és kifejezések specifikálásához.

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*.



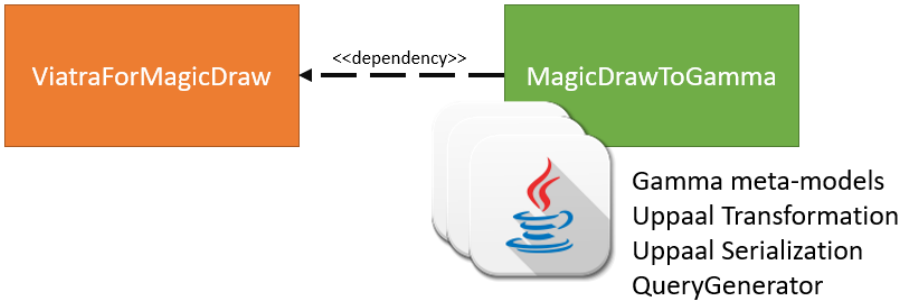
2.6. ábra. Gamma állapottérkép absztrakt szintaxisa

3. fejezet

Elvégzett munka

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 aminek a 2.0.1-es verziója van használva. A plug-in legfontosabb funkciója MagicDraw modellek Gamma modellekké való transzformációja. A letranszformált már Gamma nyelvű modelleket az keretrendszer kezelni tudja, a verifikáció elvégzéséhez az eszköznek csak egyes részei szükségesek (3.1 ábra). A plug-in MagicDrawToGammának lett elnevezve.



3.1. ábra. Architektúra koncepció

3.2. Fejlesztőkörnyezet

A fejlesztés megkezdéséhez szükséges volt összeállítani egy olyan fejlesztőkörnyezetet amivel hatékonyan lehet plug-int fejleszteni. MagicDraw biztosít egy ún. skeletont Eclipsehez és IntelliJhez is plug-in fejlesztéséhez, de a fejlesztés nem ezek segítségével hanem az IncQueryLabs által készített skeleton felhasználásával valósult meg. Ez már elő volt készítve V4MD használatához.

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 amit Gradleel nem csak Eclipsel lehet generálni. 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 meta-modellek nem változnak.

Gamma a verifikációt Uppaal segítségével végzi el, ehhez előállít egy leírást a rendszerről és egy queryt ami a rendszerrel szemben támasztott követelményeket írja le. Utóbbi 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 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 transzfromációt 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 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 a modell le lett képezve, ezután egy állapottérkép el lett távolítva utána újra le lett képezve. Ebben az esetben a régebben leképzett Gamma állapottérképek is megmaradnak.

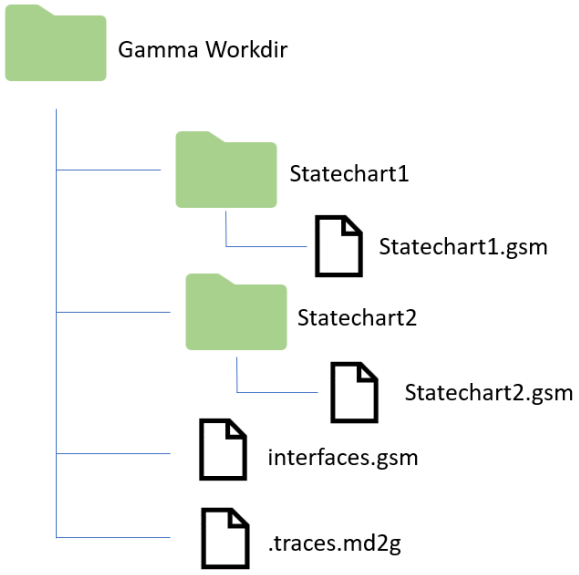
Előkészítés: A plugin a transzformáió elején létrehoz egy ResourceSet-et, amibe készít egy Resource-t a Gamma Workpace gyökerébe .traces.md2g néven, továbbá egy Resource-t interfaces.gms néven. Az előbbi Resource egy segédstruktúrát tartalmaz ami a leképzett elemek visszakereshetőségéül szolgál (ld. 3.3.1. 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 Signal Eventeket és létrehoz a Signaléval azonos néven egy Eventet, a Gamma metamodelljének megfelelően. A létrehozott eventek egy Interfacen kerülnek definiálásra aminek a neve megegyezik az állapottérképével. Az interface ezután belekerül az interfaceket tároló Resourceba. eventek iránya INOUT ez lehetővé teszi azt is, hogy az állapotgép magának küldjön eseményt.

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)

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 ás állapotátmenetek határoznak meg. Ehhez a plug-in a Viatra Batch Transformationját használta fel. 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 Scopje magában foglalja az első lépésben lérehozott Resourceokat és a MagicDraw modellt is. A készülő modellben történő keresések, és a visszakövetések ezzel az engine-el történnek. Alap struktúra kialakításának lépcsői:

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



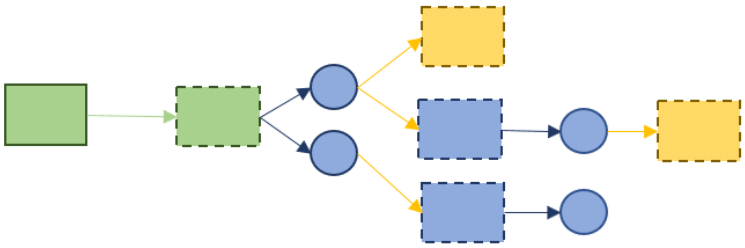
3.2. ábra. Létrehozott fájlstruktúra

3. Állapotokban található régiók leképzése.

A 2. 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 3. 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 Stateben és Állapotgépben lehet. 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 szagatott vonallal körbevett Region és a körök Statek.



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

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.

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 StatechartDefinitionban 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.

Triggerek leképzése:

Guardok leképzése:

Akciók leképzése:

3.3.1. Trace Model

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

Függelék