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

Automatikus absztrakció a modellvezérelt fejlesztésben

TDK dolgozat

Készítette

Debreceni Csaba
II. éves mérnök-informatikus (MSc)

Konzulensek

dr. Horváth Ákos
tudományos munkatárs

Hegedüs Ábel
tudományos segédmunkatárs

Dr. Varró Dániel
egyetemi docens

2013. október 25.

Kivonat

A mérnöki megközelítés egyik legalapvetőbb eszköze a modellalkotás, melynek célja a feladat komplexitásának csökkentése, az átláthatóság növelése, vagy éppen a megoldás megtervezésének elősegítése. A szoftverfejlesztésben is ezen alapul a modellvezérelt fejlesztési paradigma, ami lehetővé teszi precíz modellekből, pontosan definiált absztrakciós vagy finomítási lépéseken keresztül az alkalmazás kódjának, dokumentációjának vagy konfigurációs leíróinak automatikus generálását.

A *Modellvezérelt fejlesztésben (MDE)* a részletes (forrás) rendszermodellekhez gyakorta szükséges különböző nézeti modelleket létrehozni (pl.: megbízhatósági modell, hardver architektúra stb), amelyek kizárólag az adott nézőpont szempontjából releváns információkat tartalmazzák *(model abstraction)*. A forrás modellek és nézeti modellek közötti szinkronizáció megvalósítására jellemzően batch transzformációkat használnak. Ez azonban nem inkrementális megoldás, hiszen bármilyen változás esetén a transzformációt újra le kell futtatni, és a semmiből kell felépíteni az egész nézeti modellt.

A TDK dolgozatom keretében sikerült egy olyan keretrendszert létrehoznom, mely képes lekérdezések által definiált absztrakt nézeti modelleket inkrementálisan szinkronban tartani. Az így létrejött modellekben olyan származtatott objektumok materializálódnak, melyek életciklusa szorosan összekapcsolódik a forrás modellel. Mivel a nézetek teljes értékű modellként viselkednek, így akár bemenetként szolgálhatnak bármilyen más modellekkel kapcsolatos operációknak is.

Továbbá fontos megjegyezni, hogy a keretrendszer egy olyan általános megoldást ad a modell karbantartásra, ami felhasználható az *MDE*-ben ismert más problémák esetén is, mint például *(i)* olyan nézeti modell készítésére, ami képes megjeleníteni több forrásmodelltől függő származtatott értékeket *(model merge)*, vagy *(ii)* kibővíteni egy már meglévő modell vázat *(model extension)*.

A keretrendszer az iparban *de facto* szabványnak számító *Eclipse Modeling Framework* (*EMF*)-re épül, ahol a lekérdezéseket deklaratív gráf mintákkal definiálhatjuk, amelyek a végrehajtásához az *EMF-IncQuery* inkrementális gráf minta-illesztőt használtam és bővítettem ki. Az elkészült rendszer használhatóságát egy releváns repülőgépipari kutatási projekt keretében értékeltem ki.

Abstract

Modeling is considered as one of the most elementary discipline in engineering. Its purpose is to overcome complexity, increase understandability or ease design. In software development, the model-driven engineering paradigm follows this concept by envisioning a process starting form high-level system models and through several well-defined abstraction and refinement steps automatically derive source code, documentation or configuration artifacts.

In *Model-driven Engineering (MDE)*, the concept of view-models (e.g., reliability model, hardware architecture etc.) automatically derived from detailed system models to highlight relevant, domain specific information is a widely used and accepted approach *(model abstraction)*. Synchronization and maintenance between these system (source) models and their different view-models are usually done by batch transformations. Unfortunately, as these transformations lack of incremental execution all the transformations have to be executed from scrap to rebuild the various view-models whenever a change happens in their corresponding system model.

In the current report, I specify a framework, that can incrementally synchronize abstract view-models defined by a set of queries. These view-models consist of derived objects automatically materialized from the source model tightly following its lifecycle, resulting in full featured (view-)models that can be inputs for any other model transformation.

Additionally, my approach offers a general solution for model synchronization and maintenance, which can be used for other well-known problems in MDE such as (i) *model merging*, where the defined view-model can have derived attributes, which depend on multiple source models, or (ii) *model extension*, where an already existing model is extended with derived objects.

My framework is built upon the *Eclipse Modeling Framework* (*EMF*) considered as the *de facto* industry standard for modeling, and *EMF-IncQuery* , an incremental graph pattern matching framework with declarative pattern definition capabilities. Finally, the approach is evaluated on an ongoing avionics research project.

Tartalomjegyzék

1. Bevezető **5**

2. Esettanulmány **8**

3. Háttértechnológiák **11**

3.1. Metamodellezés és modellezés 11

3.1.1. Eclipse Modeling Framework 12

3.2. Modelltranszformációk és lekérdezések 17

3.2.1. Lekérdezés gráfmintaillesztéssel 17

3.2.1.1. EMF-IncQuery technológia 18

4. A koncepció áttekintése **24**

5. Megvalósítás **26**

5.1. Műveletek definiálása mintákon keresztül 26

5.2. EMF alapú nyomonkövethetőség 27

5.3. Transzformációs szabályok IncQuery-vel 30

5.3.1. Alkalmazása a gyakorlatban 31

5.4. Modellek betöltésének felüldefiniálása 35

5.5. Inkrementális lekérdezés alapú erőforrás (QBR) 36

5.5.1. Műveletek végrehajtása a cél modellen 37

5.6. Kiegészítés: Dinamikus metamodell építés 38

6. Értékelés **42**

7. Kapcsolódó munkák **45**

7.1. Virtual EMF 45

7.2. Query/View/Transformation (QVT) 45

7.3. Változásvezérelt modell transzformációk 46

7.4. Triple Graph Grammar (TGG) 46

8. Összefoglalás és jövőbeli tervek **48**

8.1. Megvalósított célok 48

8.2. Származtatott objektumok további kutatásokban 49

8.3. Továbbfejlesztési irányok 50

Irodalomjegyzék **51**

Függelék **53**

F.1. Simulink minta modell (forrás modell) 53

F.2. Funkcionális minta modell (cél modell) 54

1. fejezet

Bevezető

A mérnöki alapú tervezés egyik legfontosabb eszköze a modellalkotás, melynek célja a feladat komplexitásának csökkentése, az átláthatóság növelése, vagy éppen a megoldás megtervezésének elősegítése. Ezen alapelvekre épül a *modellvezérelt fejlesztés (MDE)* [18] paradigma is, melynek célja, hogy már a tervezés korai fázisától kezdve, magas szintű szakterület specifikus modellekből kiindulva precízen definiált absztrakciós és finomítási lépéseken keresztül származtassa a részletes rendszermodelleket, amelyekből kiindulva lehetőség nyílik a megvalósítandó alkalmazás kódjának, dokumentációjának vagy konfigurációs leíróinak automatikus generálására.

Az MDE alapú fejlesztés egyik legnagyobb előnye, hogy a részletes rendszermodellekből kiindulva képes modelltranszformációk segítségével különböző szakterület specifikus nézeti modelleket származtatni (megbízhatóság, biztonság, stb.), amelyek így kizárólag az adott nézőpont szempontjából releváns információkat tartalmazzák *(model abstraction)*, ez által biztosítva, hogy a rendszertervező mérnököknek csak a saját területükre jellemző fogalmaival kell dolgozniuk.

Ezen nézeti modellek megvalósításának egyik lehetséges módja a széles körben alkalmazott *származtatott értékek (derived features)* használata, mely lehetőséget ad arra, hogy definiálhatunk olyan speciális a szakterület számára releváns attribútum értékeket vagy modell elemek közötti referenciákat, amelyek rendszermodellbeli attribútumokból és referenciákból közvetlenül számíthatóak.

Fontos jellemzője ezeknek a származtatott értékeknek (MDE környezetben), hogy pillanatnyi értékük nem transzparens módon számolódik ki minden egyes lekérdezésükkor, hanem a(z első) kiszámítása utána materializálódik a modellben és csak a kiszámításában résztvevő modellelemek változása esetén számolódik újra.

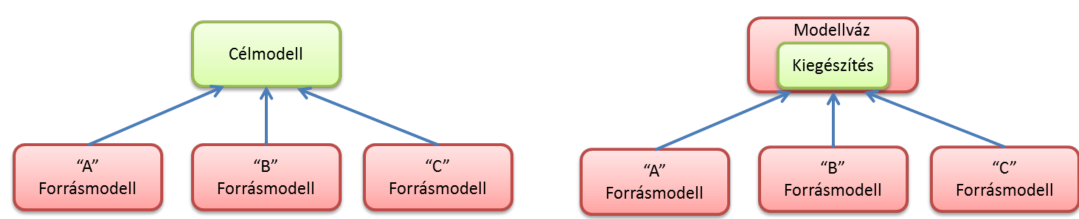
A TDK dolgozatom célja, hogy a származtatott értékek kiterjesztéseként definiálja, és megvalósítsa a *származtatott objektumok (derived object)* koncepcióját. Ezáltal képesek

legyünk nem csak attribútumokat és éleket definiálni, hanem teljes objektumokat is, melyek életciklusa más modellbeli objektumok pillanatnyi létezésétől, értékeitől függnek. Ez a megvalósítás így lehetőséget nyújt absztrakt nézeti modellek inkrementális szinkronizációjára azáltal, hogy a nézeti modellek olyan származtatott objektumokból épülnek fel, amik rendszermodellbeli elemtől függnek. Az így létrejött nézetek teljes értékű modellként viselkednek, amik akár bemenetként szolgálhatnak bármilyen más modellekkel kapcsolatos operációknak is.

Ezidáig jellemzően kötegelt (*batch*) modelltranszformációkat használnak a forrásmodellek és célmodellek közötti szinkronizáció megvalósítására, ami azt jelenti, hogy minden transzformációs szabályt egymástól függetlenül futtatnak le egy adott sorrendben. Ezek a szabályok különböző végrehajtandó műveletekből épülnek fel, amik módosítják a cél modellt.

Ez azonban nem inkrementális megoldás, mivel a célmodellben megjelenő új elemek életciklusa nincs szinkronban azokkal a forrásmodellben lévő elemekkel, amikből származnak, a transzformációk pedig nem építenek a korábbi lefutások eredményeire. Ez a gyakorlatban azt jelenti, hogy a forrásmodellen történő bármilyen módosítás után az összes szabályt újra végre kell hajtani, ezért a célmodell törlődik, majd a semmiből újra felépül.

Az általam megvalósított keretrendszer által viszont egy hatékonyabb megoldást adhatunk a modell-absztrakció problémájára. Ezenkívül a származtatott objektumokon keresztül egy olyan általános megoldást ad a modell karbantartásra, ami felhasználható az *MDE*-ben ismert más problémák esetén is, mint például *(i)* olyan nézeti modell készítésére, ami képes megjeleníteni több forrásmodelltől függő értékeket *(model merge)*, vagy *(ii)* kibővíteni egy már meglévő modell vázat *(model extension)*.



1.1. ábra. *(i) model merge*

1.2. ábra. *(ii) model extension*

A megvalósításom az iparban *de facto* szabványnak számító *Eclipse Modeling Framework* (*EMF*) modellező keretrendszerre épül. A következtetési szabályokat leíró deklaratív gráfminták végrehajtásához az *EMF-IncQuery* inkrementális gráf minta-illesztőt használtam. Az életciklusok kapcsolatának definiálására egy nyomonkövethetőségi modellt határoztam meg, a szabályok által definiált műveleteket pedig egy saját erőforrás implementáció hajtja végre.

A rendszer felhasználását, illetve előnyeit és hátrányait egy a *Méréstechnika és Információs Rendszerek Tanszéken* folyó repülőgépipari kutatási projekt részletén keresztül mutatom be. A projekt célja különböző hardver és szoftver modulok megfelelő allokációja a repülőgép architekúráján.

Kitűzött célok A keretrendszerrel kapcsolatban a következő célok lettek kitűzve, amiket a megvalósítás során el kellett érni:

1. A keretrendszer valósítsa meg a származtatott objektumok koncepcióját
2. A származtatott objektumok által létrehozott modellek teljes értékű *EMF* modellként viselkedjenek.

A dolgozat felépítése a következő struktúrát követi:

- A 2. fejezetben az esettanulmány bemutatására kerül sor.
- A 3. fejezetben a felhasznált technológiákat ismertetem.
  - 3.1. alfejezetben modellezésről,
  - 3.2. alfejezetben pedig a lekérdezésekről és transzformációkról szól.
- A 4. fejezet egy áttekintést nyújt a megközelítésről.
- Az 5. fejezet tartalmazza az implementációs részletek kifejtését.
- A 6. fejezetben találhatóak a mintapéldán elvégzett mérések eredményei.
- Kitekintésként a 7. fejezetben a kapcsolódó munkákról kesz szó.
- Befejezésként a 8. fejezetben a teljesített célok összefoglalása történik meg.

2. fejezet

Esettanulmány

A mai repülőgépekben számtalan biztonságkritikus, beágyazott rendszer található. Ezek megbízhatósága kulcskérdés, hiszen bármelyik elem kiesése végzetes eseményeket okozhat. Tervezésük során számtalan törvényben előírt feltételnek kell megfelelni. A hibák elkerülése mellett fontos szempont a bekövetkező hibahatások redukálása is.

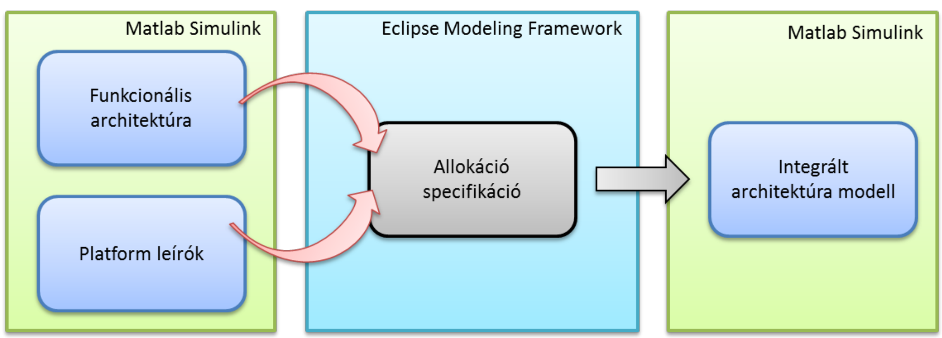
Az ilyen rendszerek feljesztése során fontos megoldandó feladat, hogy a rengeteg hardver és szoftver modult megfelelően allokálják, felkészülve a hardver hibák által okozott hatások csökkentésére. Ennek célja meghatározni, hogy mely szoftver modul, mely hardver modulon helyezkedjen el, illetve a szoftver modulok közötti kommunikáció milyen fizikai szintű kommunikációnak feleljen meg. Ezek tervezésére szigorú biztonsági kritériumok érvényesek (pl.: DC-178C [7]).

A fejlesztők a tervezéshez gyakran a *Matlab Simulink* alkalmazást használják, ami egy, a biztonságkritikus rendszerek fejlesztésénél széles körben elterjedt, grafikus modellező eszköz. Segítségével különböző szimulációkat lehet futtatni az elkészült modelleken, amik a különböző dinamikus tulajdonságokat képesek vizsgálni. A szimulációk eredményének analizálására többféle lehetőséget is biztosít az alkalmazás. Illetve képes automatikusan az adott szakterületre vonatkozó (esetünkben repülőgépipari) tanúsítványnak megfelelő kódot generálni.

A civil utasszállító repülőgépekhez (pl.: Airbus A380, Boeing 787, Dreamliner) gyakran alkalmazzák az *Integrated modular avionics (IMA)* architektúrát, ami egy partícionált környezetet definiál, mely képes különböző biztonságkritikus szinten lévő funkciókat működtetni egyetlen operációs rendszer felett.

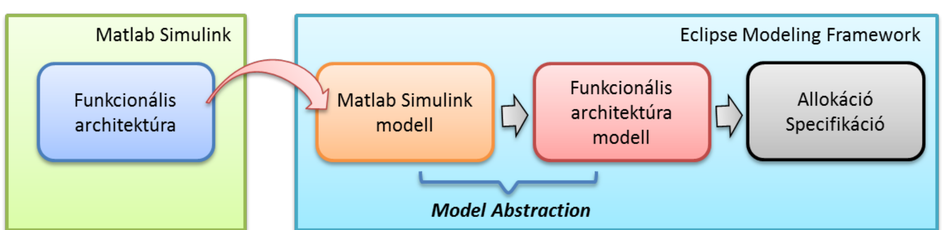
A tanszéken zajló kutatási projekt egy olyan szoftveres megoldást próbál adni az allokációs problémára *IMA* architektúra felett, ami a tervezőket segíti a megfelelő megoldás megtalálásában. Az alkalmazás elsősorban a kiválasztott allokáció modell-szintű validációját támogatja, azonban a végleges döntés minden esetben a felhasználó kezében van.

A *Matlab Simulink*-ben felépített modellek viszont az allokáció számára rengeteg irreleváns információt tartalmaznak, ezért a projekt célja a tervező mérnök egy magasabb absztrakciós szinten tudja megoldani a feladatot. Mivel a *Matlab Simulink*-ben az absztrahálás egy rendkívül nehézkes, ezért a feladat *Eclipse*-es környezetben kerül megvalósításra. A projekt megvalósítási lépéseit, illetve az egyes komponenseket és a közöttük lévő platformhatárokat szemlélteti a 2.1. ábra:



2.1. ábra. *A kutatási projekt megközelítése*

- ***Funkcionális architektúra:*** A különböző szoftver modulok részletes leírását tartalmazó modell. *(pl.: robotpilóta, motorvezérlő)*
- ***Platform leírók:*** A különböző hardver modulok részletes leírását tartalmazó modell. *(pl.: router, chassis, RTOS)*
- ***Allokáció specifikáció:*** Tartalmazza a lehetséges allokációk közül az egyik megvalósítható lehetőséget modellként.
- ***Integrált architektúra modell:*** Az elkészült modell, amiben minden elem a helyén van, és megfelel minden előírt kényszernek.



2.2. ábra. *Modell-absztrakció a projektben*

Az importálás során először EMF modell épül fel a rendszerből (lásd 2.2. ábra), amely pontos mása a *Simulink*-ben lévőnek. Ezt a modellt absztraháljuk *funkcionális architektúra modellé (FAM)*, amiben már csak allokáció specifikus információk vannak. Az így kapott nézeti modell már sokkal felhasználóbarátabb eszközkészletet nyújt a tervező számára a tervezés végrehajtásához, továbbá az így kapott modellen sokkal könnyebben írhatóak fel validációs kényszerek is, amik szintén a tervező munkáját segítenek[5].

Azonban, ha a rendszerbe egy új funkció kerül, túlságosan időigényes az absztrakcióhoz szükséges transzformációkat újra és újra lefuttatni. Mivel folyamatban van a *Simulink* modellek inkrementális importálása az *EMF* modellekbe, ezért a *FAM* modelleket is célszerű lenne automatikusan inkrementálisan frissíteni, aminek megoldására felhasználhatóak az általam definiált *származtatott objektumok*.

3. fejezet

Háttértechnológiák

Ebben a fejezetben a keretrendszer elkészítéséhez felhasznált módszertanok technológiák kerülnek részletes bemutatásra. Először minden terület elméleti része jelenik meg, majd ezt követően a gyakorlati technológiák következnek.

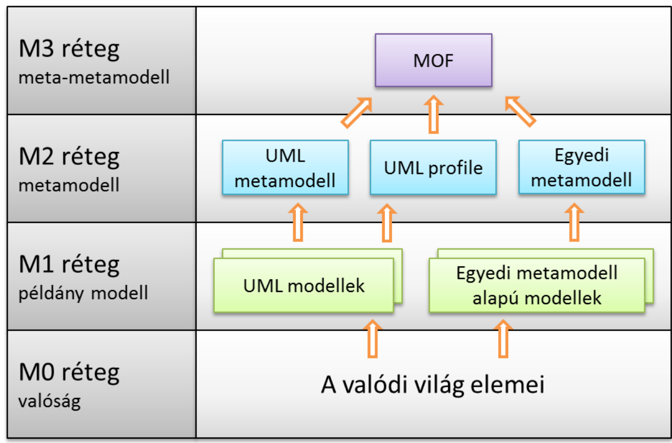
Mivel az esettanulmányban szereplő metamodellek több száz elemet tartalmaznak, az egyszerűbb érthetőség érdekében az csak egy részlete kerül bemutatásra.

3.1. Metamodellezés és modellezés

A *Metamodellezés* több területen is sokat használt kifejezés, de minden esetben a rendszerben tárolt adatok struktúráját fogalmazza meg. Maguk a *modellek* olyan elemei a rendszernek, melyek megfelelnek a hozzájuk tartozó metamodellek strukturális kényszereinek. Ezeket a modelleket *példánymodell*eknek is nevezzük.

- ***metamodell:*** a modellek strukturális felépítését és egyéb tulajdonságait leíró modell
- ***példánymodell:*** a metamodellnek megfelelő, konkrét adatokat tartalmazó modell

Az *Object Management Group (OMG)* szabványai egy négyszintű modellezési architektúrát határoznak meg az *MDE* területén, amit, a 3.1. ábra mutat. Egy legfelső szintű (M3) megvalósítás *Meta-Object Facility (MOF)*, ami megköti, hogy minden rétegben szereplő elemnek szigorúan meg kell felelnie egy az eggyel fentebbi szinten lévő elemnek. Egy *MOF*-t megvalósító (M2) réteg például az *Unified Notation Language (UML) 2.0*, ami meghatározza, hogy az őt megvalósító (M1) példánymodellekben (pl: osztály diagram, szekvencia diagram stb), milyen elemek lehetnek (attribútumok [*Attribute*], osztályok [*Class*], kapcsolatok [*Classifiers*], példányok [*Instance*] stb). Az utolsó rétegben (M0) pedig a valódi világ elemei találhatóak meg.



3.1. ábra. *Az OMG modellezési architektúrája*

2006-ban az *OMG* két új invariánsát is definiálta a *MOF*-nak: *Essential MOF (EMOF)*, *Complete MOF (MOF)*. Ezen kívül egy harmadik invariánsra is igény lett: *Semantic MOF (SMOF)*. Az *Eclipse Modeling Framework* egy saját *Ecore* meta-metamodellt definiál a strukturált modellek létrehozására, ami bár korábban kialakult, mint az *EMOF*, mégis annak egy variánsának tekinthetjük.

3.1.1. Eclipse Modeling Framework

Az *Eclipse Modeling Framework (EMF)* egy *Java* alapú modellező és kódgeneráló keretrendszer, mely segítségével könnyen készíthetünk strukturált modell alapú szoftvereket. A keretrendszer a benne létrehozott modelleket felhasználva biztosít eszközöket és futásidejű támogatást, hogy a *JVM*-ben a modellek reprezentációjára alkalmas *Java*-osztályokat létrehozza, emellett a modellek megjelenítését és az alapszintű szerkesztését lehetővé tegye.

Modellezés

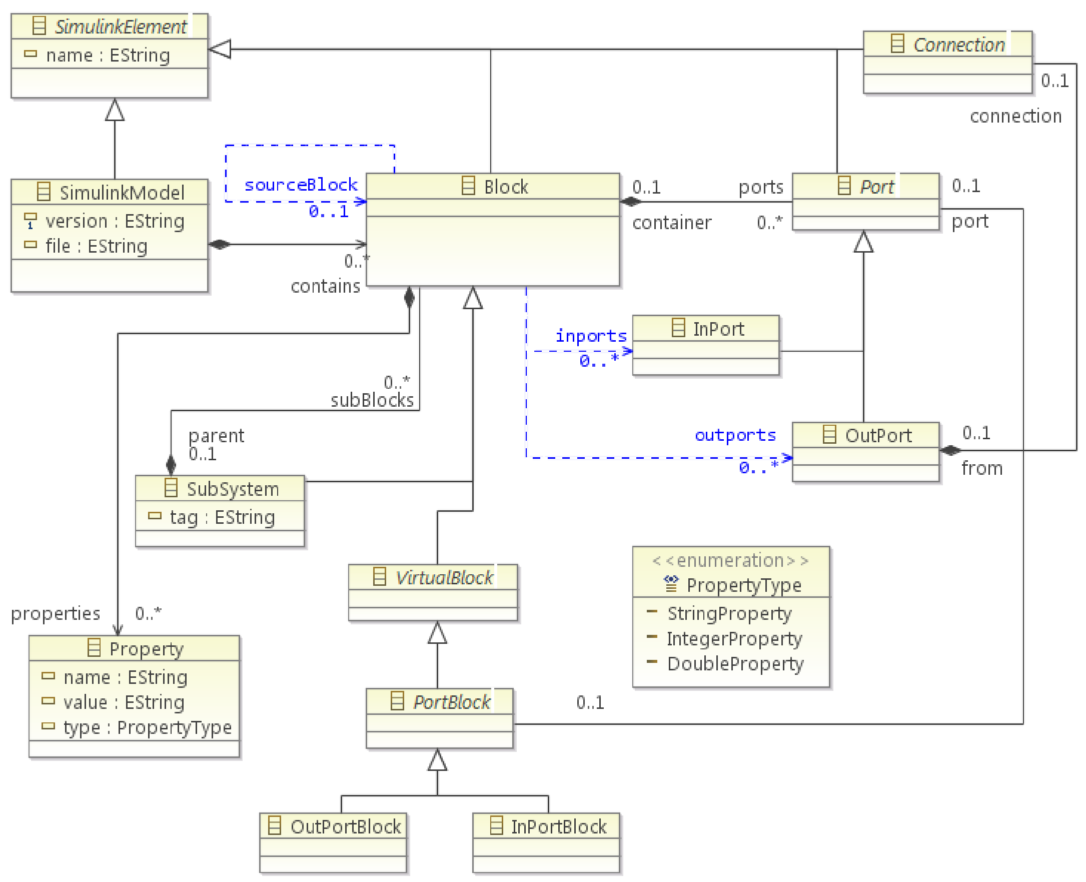
Az *EMF* által definiált *Ecore* meta-metamodell (M3 réteg) a hozzá konzisztens metamodelleken a következő főbb elemeket határozza meg[1]:

- ***EClass***: magukat az osztályokat modellezi.
- ***EEnum***: egy olyan érték típust határoz meg, aminek egy fix értékhalmaza van
- ***EAttribute***: modellezi az attribútumokat, az objektum adatainak komponenseit.
- ***EReference***: az osztályok közötti kapcsolatok modellezésére használható. Tulajdonságai között beállítható a multiplicitás és típus is, mint asszociáció, aggregáció.
- ***EInheritence***: osztályok közötti származást definiáló kapcsolat. Egy osztály több őstől is származhat.

[1] A lista nem teljes, csak a leggyakrabban használt elemeket tartalmazza

Az attribútumok és kapcsolatok lehetnek származtatottak is *(derived features)*, ami azt jelenti, hogy értékük más elemekből következik. Ehhez a metamodellen a megfelelő elem *derived* tulajdonságát kell beállítani[2].

A 3.2. ábrán látható a *Simulink* rendszerek leírását definiáló *EMF* metamodell. Jól látható, hogy ezek a modellek hierarchikusan épülnek fel, melyek gyökere a *SimulinkModel*, ami különböző blokkokból (*Block*) épül fel. Egy blokk lehet atomi, vagy alrendszer (*SubSystem*). A blokkokhoz tartoznak portok (*Port*) is, amik lehetnek ki- vagy bemeneti portok (*InPort, OutPort*). Ezeket a portokat pedig port blokkokba csoportosíthatjuk (*PortBlock* → *InPortBlock, OutPortBlock*). Ezen kívül kapcsolatok is definiálhatóak (*Connection*) egy-egy ki- és bemeneti port között[3]. A blokkokhoz definiálhatunk tetszőleges számú tulajdonságot is (*Property*). A tulajdonságok típusa három féle lehet (szöveg, egész vagy decimális szám), ami egy jó példa az *EEnum* használatára.



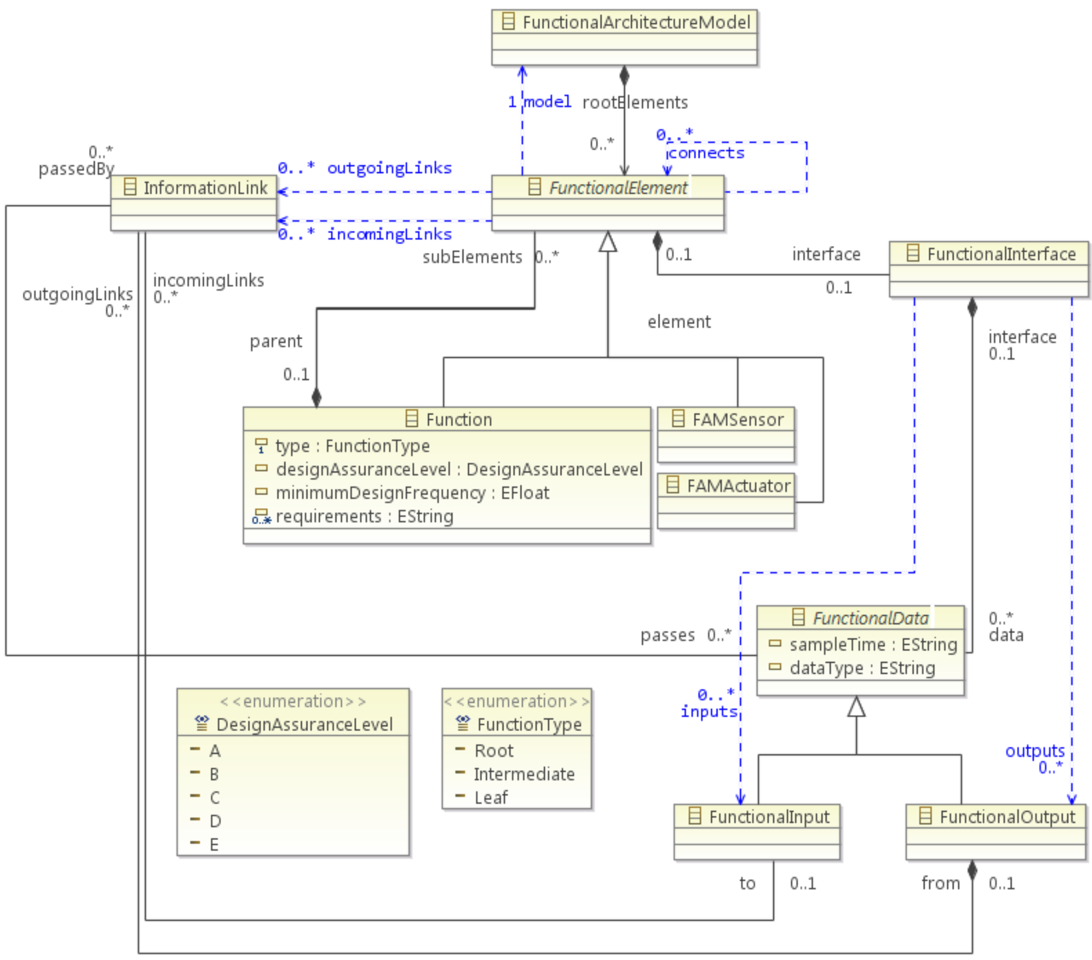
3.2. ábra. *A „simulink.ecore" metamodell részlete*

A szaggatottan kékkel jelölt relációk(sourceBlock, inports, outports) az ábrán származtatott értékeket jelölnek. Egy blokkhoz eltároljuk a modellben az összes hozzá kapcsolódó

[2] Azt, hogy pontosan miként származnak ezek az értékek, a modellezés szintjén nem lehet definiálni.
[3] Az ábrán a kommunikáció specifikus elemek a könnyebb érthetőség érdekében már nem jelennek meg.

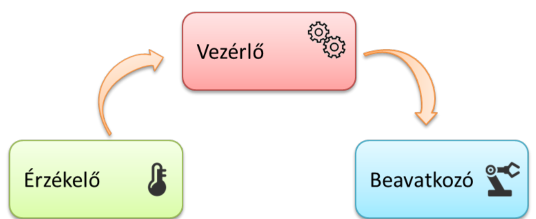
portot. Ezekből egyértelműen következik, hogy melyek lesznek a bemeneti portjai (azon portok halmaza, amik típusa *InPort*), illetve a kimeneti portjai (azon portok halmaza, amik típusa *OutPort*).

A 3.3. ábra pedig a cél metamodelljét mutatja. Ez egy jóval absztraktabb modellt ad a funkciókról. Itt már nincsenek *Simulink* specifikus elemek, mint például portok vagy blokkok, csak az allokáció számára releváns funkciók és azok tulajdonságai. Ez a modell is egy gyökér objektummal rendelkezik (*FunctionArchitectureModel*), amihez különböző funkcionális elemek (*FunctionalElement*) kapcsolódnak. Ezek között kapcsolatokat definiálhatunk (*InformationLink*). A funkciók ki és bemenetéről egy interfész (*Functional Interface*) tartalmaz információkat. A funkciók lehetnek kiemelt szereplőek (pl.: szenzorok [*FAMSensor*], beavatkozók [*FAMActuator*] stb.), amik tartalmazhatnak egyéb funkciókat is (*Function*). A funkciók különböző biztonsági szinttel rendelkeznek (*DesignAssuranceLevel*), illetve elhelyezkedésük alapján lehetnek gyökér, levél vagy köztes szinten (*FunctionType*)



3.3. ábra. *A „functionalArchitecture.ecore” metamodell részlete*

Az esettanulmányhoz kapcsolódóan tekintsük azt a rendkívül egyszerű rendszert, ahol egy érzékelő (sensor) folyamatosan figyeli a külvilágot, és a mért adatait elküldi egy vezérlő egységnek (controller), amely feldolgozza azokat. Adott esetben, ha szükséges utasítja a rendszerben lévő beavatkozóegységet (actuator), hogy lépjen működésbe a célnak megfelelően. A szenzort és a beavatkozóegységet működtető alkalmazás, továbbá a vezérlőegység egy-egy funkciónak felel meg az allokációs feladat során. Ezeket a komponenseket és működésüket szemlélteti a 3.4. sematikus ábra.



3.4. ábra. *Egy egyszerű rendszer sematikus ábrája*

A minta rendszer *Simulink* metamodellben megvalósított példány modellje található meg a függelék F.1. ábráján. Ebben található egy *SimulinkModel (sb)*, aminek egyetlen gyökér szintű eleme van, ami egy *SubSystem (wrapper)*. Ez csomagolja be releváns rendszert egy alrendszerbe, ami tartalmazza az érzékelőt (sensor) definiáló *InPortBlock*-t és a beavatkozóegységet (actuator) reprezentáló *OutPortBlock*-t, a vezérlő (controller) elem pedig egy alrendszerként lett definiálva. Ezen elemekhez tartozik néhány *Property* objektum is, amik az adott elemhez kapcsolódó fontosabb attribútumok értékét definiálják. Ezen kívül egy *InPort (inport)* tartozik a beavatkozóegységhez, és egy *OutPort (outport)* az érzékelőhöz, amik a ki- és bemeneteket definiálják.

A *FAM* metamodellben megvalósított példánymodellt pedig a függelék F.2. ábrája írja le. Ebben a gyökér elem egy *FunctionalArchitectureModel (fam)*, amiben taláható egy *FAMSensor (sensor)*, ami az érzékelő, egy *FAMActuator (actuator)*, ami a beavatkozóegység, illetve egy *Function (controller)*, ami a vezérlőt reprezentálja. Mindegyik elemben található egy-egy *FunctionalInterface*. A *sensor* interfészéhez tartozik még egy *FunctionalOutput*, az *actuator* interfészéhez pedig egy *FunctionalInput* elem is.

Érdemes megjegyezni, hogy a két modell ugyanazon rendszert szemléltetik, mégis az absztraktabb *FAM* modell közel feleannyi elemet tartalmaz, mint a *Simulink* modell.

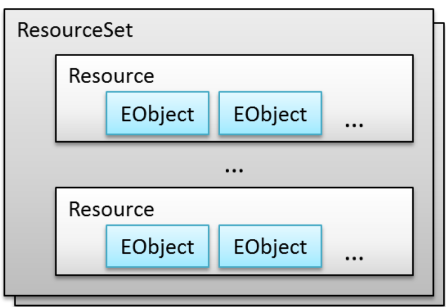
Integrálás az alkalmazásokba

Az *EMF* egyik fő előnye, hogy egy reflektív API-n keresztül dinamikus modellekhez is ad támogatást, ami azt jelenti, hogy az egyes osztályok kódgenerálás nélkül is példányosíthatóak. Az API-n keresztül továbbá elérhető minden strukturális tulajdonsága a

metamodelleknek (pl.: EClass-ok, azok referenciái, attribútumai stb.), amin keresztül a példánymodellek pillanatnyi értékei is generikusan elérhetőek és módosíthatóak.

Az *EMF* metamodellekben szereplő osztályok példánya mindig egy közös ősből, az *EObject* interfészből származnak. Az *EObject*-ek minden esetben egy ***Resource***-ban találhatóak, ami tartalmazza az egész példánymodellt. A *Resource*-ok pedig egy ***ResourceSet***-be kerülnek, ami tartalmazhat több *Resource*-t is. A *Resource*-okat egy-egy ***URI*** azonosíthatja, ami lehet *namespace*, *platform*, *file* URI is. Ezek a típusok az *Eclipse* keretrendszerben definiált fájl elérési struktúrát határozzák meg. A *ResourceSet*, *Resource* és *EObject* viszonyát az *EMF* keretrendszerben a 3.5. ábra mutatja.

- ***EObject:*** az *EClass* elemek példányai
- ***Resource:*** a példánymodellt tartalmazó objektum
- ***ResourceSet:*** *Resource*-ok gyűjteményét tartalmazó objektum
- ***URI:*** *Resource*-ok azonosítására szolgáló cím



3.5. ábra. *ResourceSet, Resource és EObject tartalmazási viszonya*

A modell betöltése egy adott *ResourceSet*-en keresztül történik. Ekkor, ha az adott *ResourceSet*-be még nem volt betöltve a *Resource*, meghívódik az ehhez tartozó *ResourceFactory* létrehozó metódusa, ami meghatározza, hogy mely *Resource* implementáció töltse be a modellt. Azt, hogy melyik *ResourceFactory* fog meghívódni a hivatkozott *URI* kiterjesztése dönti el.

Változáskövetés

Az *EMF* keretrendszerben a változáskövetés az *Adapter* interfész segítségével történik. Azon objektumok, amik megvalósítják ezt az interfészt feliratkozhatnak az *EObject*-re, ami folyamatosan értesíti őket minden egyes változásáról *Notification*-ök formájában. Egy ilyen *Notification* tartalmaz információt a módosítás típusáról (*add*, *remove*, *set*), illetve tartalmazza módosítás előtti és utána állapotot is.

3.2. Modelltranszformációk és lekérdezések

Az *MDE* egyik alap építőköve modellek automatikus származtatásához, módosításához, leképezéséhez a **modelltraszformáció** . Legfőbb célja, hogy csökkentse a hibák lehetőségét az automatizmusnak köszönhetően. A **lekérdezés**, olyan speciális modelltranszformációs művelet, mely kiválasztja azokat a részeit a modellnek, amikre teljesülnek a lekérdezés feltételei.

- ***transzformáció:*** modelleken végez műveleteket előre definiált szabályoknak megfelelően
- ***lekérdezés:*** a modell adott részeit kiválasztó művelet előre meghatározott minták alapján

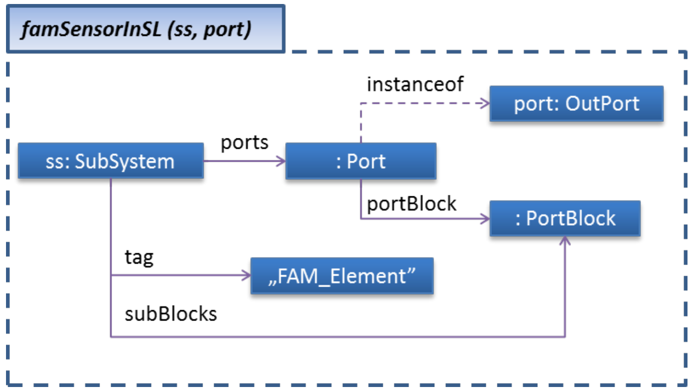
3.2.1. Lekérdezés gráfmintaillesztéssel

Az *MDE*-ben modellek jellemzően típusos gráfnak tekinthetőek, ahol az egyes csomópontok az objektumoknak, az élek pedig az ezek között lévő kapcsolatoknak felelnek meg. A gráfok módosításával kapcsolatos transzformációkat *gráftranszformációknak* nevezzük. Ezek alapeleme a gráf minta (*graph pattern*), amikkel lekérdezéseket (*queries*) fogalmazhatunk meg a gráfban. A gráf minták strukturális és egyéb feltételeket szabnak meg a gráfra nézve. Egy minta akkor illeszkedik egy modellre, ha képes kielégíteni a minta az összes feltételét. A *mintaillesztés* és a *mintafelismerés* között a lényegi különbség az, hogy az illeszkedés tartalmazza a konkrét csúcsokat is, amik esetén teljesülnek ezek a kényszerek, míg a felismerés csak jelzi, hogy van ilyen illeszkedés, de nem határozza meg a konkrét találatot.

- ***gráfminta:*** a modell struktúrájával kapcsolatos feltételeket követel meg
- ***illeszkedés:*** egy olyan csúcshalmaz, ami kielégíti a mintában szereplő feltételeket

Tekintsük a F.2. ábra mintamodelljét, amire definiáljuk a 3.6. ábrán látható *famSensorInSL* mintát, aminek két paramétere van: *ss* és *port*. Keressük azokat a *SubSystem* (*ss*) és *OutPort* (*port*) párokat, ahol a *SubSystem ports* referenciáján el lehet navigálni egy porthoz, ami egyben megvalósítja az *OutPort*-ot is, és a *portBlock* referenciája egy olyan *PortBlock*-ra mutat, ami a *SubSystem subBlocks* referenciáján elérhető. Ezen párokon belül nevezzük a *SubSystem* elemet *ss*-nek, az *InPort* elemet pedig *inport*-nak.

Ezeknek a feltételeknek csak egy páros felel meg a modellben, tehát a mintán egyetlen illeszkedése van. Ez a következő csomópontokat jelenti: *wrapper* és *inport*.



3.6. ábra. *Példa egy minta definiálására*

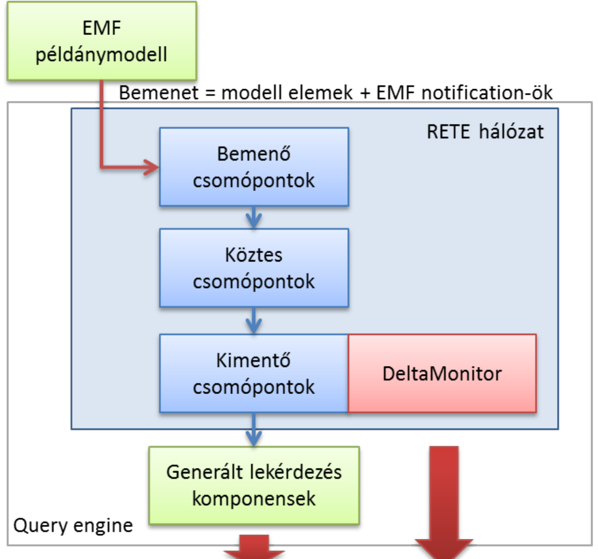
3.2.1.1. EMF-IncQuery technológia

Az *EMF-IncQuery* egy olyan keretrendszer, mely lehetővé teszi deklaratív gráfminta lekérdezések definiálását *EMF* modellek fölött. Ezen lekérdezések aztán hatékonyan végrehajthatóak egyéb kézzel írt kód nélkül is. A legnagyobb előnye a nagy futásidejű teljesítmény, amit inkrementális gráfminta illesztő technikákkal éri el [2].

A technológia belső architektúráját szemlélteti a 3.7. ábra, aminek felépítése a következő: A folyamat bemenete az EMF példánymodell, amire a mintát kell illeszteni. Az EMF-IncQuery egy saját *Adapter* implementációjával feliratkozik minden modellelem változására. Ezek után a lekérdezés leírásának függvényében létrehoz egy Rete szabály kiértékelő hálózatot [8], ami feldolgozza az elemeket, hogy megalkossa az eredményt, mint kimeneti csomópontot. A lekérdezéseket az automatikusan generált lekérdezés komponensek ezután újra feldolgozzák, hogy így biztosítsanak típushelyes hozzáférési réteget, megkönnyítve ezáltal az integrációt létező alkalmazásokhoz. Ezt a Rete hálózatot addig kell fenntartania, amíg a lekérdezésre szükség van: továbbra is megkapja az elemi változásokról az értesítéseket, és továbbterjeszti őket. Ennek következtében lekérdezés eredmény deltákat (query result delta) hoz létre a delta monitor lehetőség segítségével, amiket az eredmény inkrementális frissítésére használ fel[19].

Emiatt a megközelítés miatt a lekérdezés eredmények (például a gráfminta illeszkedések találatai) folyamatosan karban vannak tartva a memóriában, és direkt módon hozzáférhetőek. Ennek eredményeképp az *EMF-IncQuery* képes hatékonyan kiértékelni bonyolult lekérdezéseket nagy méretű modelleken is.

Egy másik nagy előnye ennek a technológiának, hogy képes statikus és dinamikus *EMF* módban is működni, ezáltal képes olyan modellekre is mintát illeszteni, amikhez nem tartozik generált *Java* osztály. Ebben az esetben az azonosítást az *EClass*-ok nevei alapján határozza meg.

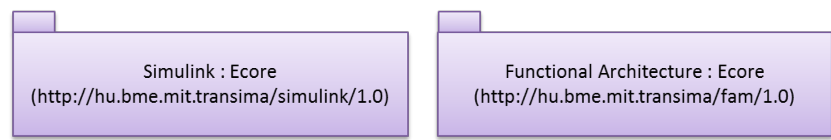


3.7. ábra. *EMF-IncQuery architektúrája[19]*

Minták megfogalmazása (pattern language)

A mintákat az *IncQuery* saját mintaleíró nyelvével (*pattern language*) lehet definiálni [3]. Mivel az általam elkészített keretrendszer ezt a nyelvet bővíti ki, ezért az alábbi szakasz a nyelv legfontosabb elemeit mutatja be. Tekintsük példaként az 3.6. ábrán található mintát.

Ahhoz, hogy a technológia értelmezni tudja az *OutPort*, *Block* stb. elemeket, referálni kell azokra a metamodellekre, amikre definiáljuk a mintákat. Ehhez első lépésként ezek *namespace URI*-ai alapján kell beimportálni őket:



3.1. Kódrészlet. *Metamodell importálása namespace URI-val*

```
1 import "http://hu.bme.mit.transima/fam/1.0"      //FAM model
2 import "http://hu.bme.mit.transima/simulink/1.0" //Simulink model
```

Egy minta definiálása a `pattern` kulcsszóval történik, ami után a minta neve következik, majd a paraméterváltozók felsorolása. Ezek típusát is meg lehet adni, de nem kötelező (`paramName : ParamType`). A strukturális kényszereket ezt követően lehet meghatározni `{` és `}` jelek között.

3.2. Kódrészlet. *Minták definiálása*

```
pattern famSensorInSL(ss : SubSystem, port) {
        // ... kényszerek ... //
}
```

A legalapvetőbb kényszerek a típusra, kapcsolatra és attribútumra vonatkoznak, amiket a következőképpen definiálhatunk:

- `OutPort(port)` → Az *port* legyen *OutPort* típusú.
- `SubSystem.tag(ss, "FAM_Element")` → Az *ss* elem *tag* attribútuma legyen „*FAM_Element*”.
- `SubSystem.ports(ss, inport)` → Az *ss* elem *ports* referenciái között legyen az *port*.
- `SubSystem.ports.portBlock.port(ss, inport)` → Az *ss* elem *ports* referenciáján van olyan *Port*, aminek van egy *PortBlock*-ja, aminek a *port* referenciája az *port*-ra mutat.

Ezek alapján a példa minta a következőképpen nézhet ki, aminek az illeszkedése természetesen a korábban említett *ss* és *inport* páros a példamodellben:

3.3. Kódrészlet. *Példa minta definiálása IncQuery-ben*

```
pattern famSensorInSL(ss : SubSystem, port) {
        SubSystem.tag(ss, "FAM_Element");
        SubSystem.ports(ss, iort);
        SubSystem.subBlocks(ss, portBlock)
        OutPort.portBlock(port, portBlock);
}
```

A kényszerek között „*vagy*” kapcsolatot az `or` kulcsszóval hozhatunk létre. Ebben az esetben a minta akkor fog illeszkedni, ha legalább az egyik kényszerekből álló blokk teljesül. Példaként tekintsük azt a mintát, ami az előző példamintát úgy bővíti ki, hogy az *InPort* típusú portok esetén is illeszkedni fog[4]:

3.4. Kódrészlet. *Kényszerek „vagy” kapcsolata*

```
pattern famSensorOrActuatorInSL(ss : SubSystem, port) {
        SubSystem.tag(ss, "FAM_Element");
        SubSystem.ports(ss, port);
        SubSystem.subBlocks(ss, portBlock)
        Outport.portBlock(port, portBlock);
} or {
        SubSystem.tag(ss, "FAM_Element");
        SubSystem.ports(ss, port);
        SubSystem.subBlocks(ss, portBlock)
        Inport.portBlock(port, portBlock);
}
```

[4] Természetesen a mintát egyszerűbben is lehetne definiálni, de a kapcsolat reprezentálása miatt lett így leírva.

Ebben az esetben már két illeszkedést is találunk a példamodellben: *ss* - *inport* és *ss* - *outport*.

Mintákat egymásba is ágyazhatunk. Ehhez a `find` kulcsszót kell használni, majd mint egy metódushívást lehet meghívni más mintákat. Azt is gyakran szeretnénk megfogalmazni, hogy egy minta akkor illeszkedjen, ha egy másik pont nem illeszkedik. Ehhez a `neg` kulcsszót kell a `find` elé helyezni. Ez pontosan azt az esetet defininálja, hogyha a *neg*-ben szereplő objektumok és a közöttük létező kapcsolatok nem pont a mintában leírt a konstellációban szerepelnek, a minta illeszkedik.

Másik eset, hogy az illeszkedések számára szeretnénk megfogalmazni kényszert. Az illeszkedések számát a `count` kulcsszó elhelyezésével lehet elérni a `find` előtt. A nyelv lehetőséget ad arra is, hogy ha egy olyan mintát szeretnék meghívni, ami leköt olyan csúcspontokat is, amikre valójában nincs szükségünk, akkor ezt kiválthassuk bármilyen elemmel. Erre a _ joker karaktert használhatjuk.

A most bemutatott kulcsszavak használata előtt, definiáljuk az előző példa minta „vagy” kapcsolatának azon blokkját, ami az *InPort* típusú portokat kérdezi le, egy új mintaként:

```
1 pattern famActuatorInSL(ss : SubSystem, port) {
2         SubSystem.tag(ss, "FAM_Element");
3         SubSystem.ports(ss, outport);
4         SubSystem.subBlocks(ss, portBlock)
5         InPort.portBlock(port, portBlock);
6 }
```

Ezek után tekintsük azt a mintát, ami olyan *SubSystem*-ekre teljesül, amikben a *famActuatorInSL* minta nem teljesül egyetlen elem esetén sem, viszont a *famSensorInSL* minta legalább két különböző port esetén illeszkedik. Ezt a mintát grafikusan már rendkívül nehézkes lenne ábrázolni, ezért tekintsük csak a nyelvben használt kódrészletet:

3.5. Kódrészlet. *Példa a find, neg, count és _ nyelvi elemek használatára*

```
1 pattern noFamActuatorAndAtLeastTwoFamSensorInSL(ss : SubSystem) {
2     neg find famActuatorInSL(ss, _);
3     M == count find famSensorInSL(ss, _);
4     M > 2;
5 }
```

A nyelv szintén egy *EMF* metamodell példányának minősül. Ezért amellett, hogy ezekből a mintákból *Java* osztályok generálódnak, a mintákat tartalmazó fájlokat *EMF* modellként is be lehet tölteni. Emellett mindent minta is egy-egy részmodellnek tekinthető.

Adatkötés (Databinding)

A technológia integrálása kapcsán előkerül az az igény is, hogy értesítést kapjunk az illeszkedések megszűnéséről és a keletkezéséről, illetve egy illeszkedéshez tartozó objektumok változásáról is.

- ***adatkötés (databinding):*** két adatforrás összekötése, és az adatok szinkronizálása

Az *EMF-IncQuery* egy *Data Binding API*-t nyújt az adatkötésre. Ez elsősorban felhasználói felületek *(UI elements)* elemeihez kapcsolható megvalósítást ad. A minták változásaira az *IncQuery* által implementált *DatabindingAdapter* osztály példányaival lehet feliratkozni. Ezektől az adapterektől pedig egy-egy *IObservableValue* objektumot lehet elkérni a minta adott paraméteréhez. Ezek az objektumok tartalmazzák a paraméter jelenlegi értékét, és képesek értesíteni, ha ez az érték megváltozott. A minták illeszkedésvel kapcsolatban pedig *IObservableList*-eket vagy *IObservableSet*-eket lehet elkérni az adaptertől, amik tartalmazzák a pillanatnyi illeszkedéseket, és értesítést küldenek, amikor egy illeszkedés eltűnik, vagy megjelenik az adott kollekcióban.

Megjelenítők (Viewers)

Az adatkötést használják fel az *EMF-IncQuery Viewers* megjelenítők is. Feladatuk elsősorban a fejlesztés segítése azon tervező mérnökök számára, akik ezt a technológiát szeretnék integrálni alkalmazásukban. Használatuk során a mintákat különböző annotációkkal lehet megjelölni. Három ilyen annotáció tartozik ehhez a komponenshez:

- *Item*: a minta egyik paraméteréből hoz létre egy „elemet”.
- *Edge*: a minta két paraméteréből létrehozott „elem” között ad meg egy „élt”.
- *ContainsItem*: hasonló az „élhez”, de ezzel egy tartalmazási viszonyt lehet leírni.

Az annotációkban egy megjelenítési szöveget is meg lehet adni. Ehhez a paramétereket is fel lehet használni `$` és `$` karakterek között, amiken keresztül navigálni is lehet különböző referenciákon, attribútumokon át. Erre példát a 3.6. kódrészlet mutat.

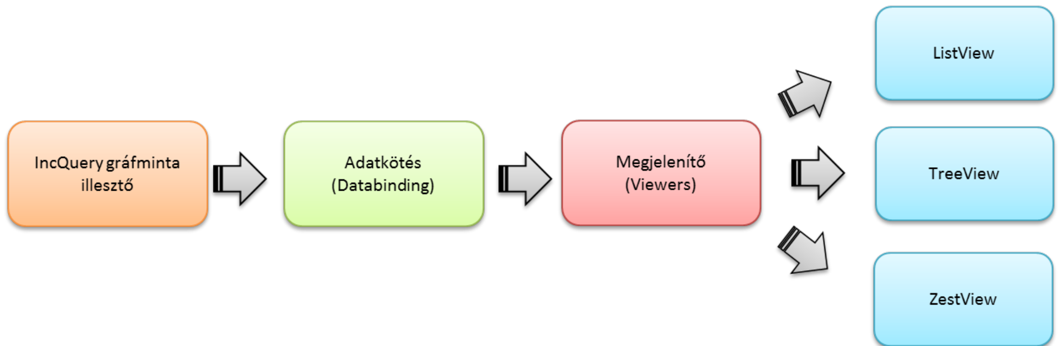
3.6. Kódrészlet. *Viewers annotációk a mintákon*

```
1 @Item(item = model, label = "SimulinkModel: $model.name$")
2 pattern getSimulinkModel(model) { ... }
3
4 @Edge(src = port, trg = portBlock)
5 pattern getPortAndPortBlockConnection(port, portBlock) { ... }
6
7 @ContainsItem(src = parent, trg = child)
8 pattern getChildBlocks(parent,child) { ... }
```

Jelenleg három nézet került implementálásra a keretrendszerben:

- *List Viewer*: egy egyszerű listát tud megmutatni az elemekről, a közöttük lévő kapcsolatokat nem tudja ábrázolni
- *Tree Viewer*: egy fa struktúrát mutat, az éleket a tartalmazás viszony tárolja, magát a tartalmazást viszont nem tudja ábrázolni
- *Zest viewer*: egy grafikus gráfmegjelenítő, ami minden kapcsolatot képes leírni (használatához a *GEF4 Zest*[5] technológiára van szükség)

A *Viewers* struktúrája a következő elemekből épül fel: *(i)* A *ViewerDataModel* tartalmazza azt az *Observable* listát vagy halmazt, ami a létrejött elemeket, éleket, tartalmazásokat tárolja; illetve azoknak a mintáknak a listáját, amik illeszkedését kell figyelni. Ezek azok a minták, amiket a megfelelő annotációkkal elláttunk definiálásuk során. *(ii)* A *ViewerDataModel* egy példányához, megadhatunk egy *ViewerDataFilter* objektumot, ami néhány általunk kiválasztott mintailleszkedését nem figyeli a futás során. *(iii)* A *ViewerState* pedig tartalmazza a *ViewerDataModel* példányt; nyomonköveti, hogy mely objektumokból mely *Item* példányok jöttek létre, kik között van kapcsolat stb. Ezen kívül egy *IViewerStateListener* interfészt megvalósító osztályok feliratkozhatnak a *ViewerState* példány változásaira.



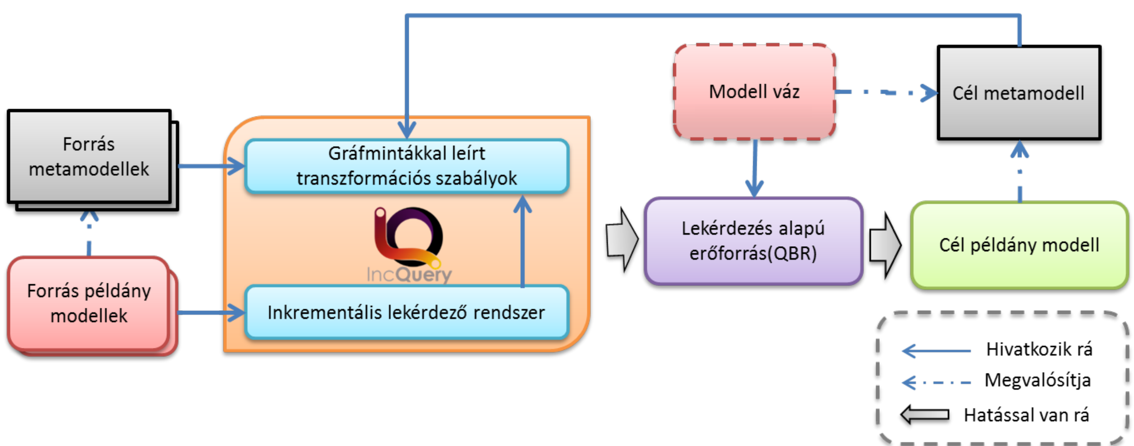
3.8. ábra. *A mintaillesztő, adatkötés és megjelenítők hatáslánca*

[5]További információk a Zest weboldalán: http://wiki.eclipse.org/Zest

4. fejezet

A koncepció áttekintése

Jelen fejezetben a rendszer működését mutatom be, illetve a megoldás ötleteit vezetem be a részletes ismertetésük előtt. Az elkészült keretrendszer folyamatát szemlélteti a 4.1. ábra.



4.1. ábra. *A megvalósítás áttekintése*

A folyamat bemenete a *forrás* és *cél metamodellek*, a *forrás példánymodellek*, illetve opcionálisan egy *modellváz*. A *cél* metamodellből alapvetően[1] csak egy lehet, míg *forrás* metamodellből akár több is. A cél modellben megjelenő származtatott objektumokat definiáló szabályokat, amik a metamodellek között teremtenek kapcsolatot, *EMF-IncQuery* gráfmintákkal lehet leírni. Ezekben a mintákban új és módosított annotációkon keresztül lehet meghatározni a létrehozandó objektumok típusát és tulajdonságaik beállításához szükséges műveleket. A forrás metamodelleket megvalósító példány modellek változásait az inkrementális lekérdező rendszer követi nyomon. Az opcionálisan definiálható modellváz használata során, ez a váz egy statikus részét képezi a célmodellnek, és ezt egészíti ki a rendszer származtatott objektumokkal a szabályoknak megfelelően. A 3.1.1. fejezet alapján a cél példánymodellt egy *Resource*-nak kell tartalmaznia, ami ebben az esetben

[1] Megjegyzés: abban az esetben, ha a cél metamodell referál más metamodellekben lévő elemekre, akkor célmetamodellből is több létezik

egy saját implementációt jelent, ennek a neve *Query Based Resource (Lekérdezés alapú erőforrás)* lett. Ez az erőforrás tartja naprakészen a célmodellben lévő származtatott objektumokat, illetve hozzájuk kapcsolódó attribútumokat és referenciákat. A hatáslánc pedig úgy működik, hogy ha egy forrás példány megváltozik, ami egy adott illeszkedést módosít, megszüntet vagy létrehoz, akkor arról a mintaillesztő jelzést küld az erőforrásnak, hogy a mintában definiált műveletek alapján megfelelő módon módosítsa a cél modellt.

A rendszer működését leíró példában a *forrás* a *Simulink metamodell*, a *cél* pedig a *Functional Architecture metamodell* (lásd 3.2. ábra, illetve 3.3. ábra), míg egy lehetséges forrás példány modellnek a F.1. ábrán látható modellt tekinthetjük. Miközben a forrásmodell módosul a példákhoz kapcsolódó minták (lásd 5.3. fejezet) illeszkedése változik, amiről a mintaillesző üzenetet küld az erőforrásnak. Az erőforrás pedig végrehajtja a mintában leírt műveleteket. A módosítások végére a célmodell a F.2. ábrán található modellel fog megegyezni.

Származtatott objektumok definiálása: Mivel az *EMF-IncQuery Viewers* is hasonló fogalmakkal dolgozik, mint elemek és élek, ezért célszerűnek tűnt ezt kibővíteni úgy, hogy támogassa a származtatott objektumok definiálását és a hozzájuk kapcsolódó attribútumok beállítását is. Továbbá azt is meg kell oldani, hogy a létrehozott elemek *EMF*-beli elemek legyenek, ne szimpla *POJO*[2] objektumok (lásd 5.1. fejezet).

Nyomonkövethetőség a szinkronizációban: Fontos feladat nyílvántartani azt is, hogy mely elemekből mely elemek jöttek létre az új modellben. Ezeket az információkat el kell tudni érni a mintákon keresztül is, ezért egy *EMF* alapú nyomonkövethetőségi modell lett definiálniálva, amihez könnyedén lehet *EMF-IncQuery* mintákat definiálni, és felhasználni őket a transzformációs minták specifikációjához (lásd 5.2. fejezet).

Modell betöltésének felüldefiniálása: A megvalósítás alapötlete az volt, hogy válasszunk ki egy speciális kiterjesztést (*„.qbm”*), amihez egy saját *ResourceFactory*-t regisztrálhatunk be az *EMF*-en belül (*QueryBasedResourceFactory*). Ez a factory osztály példányosít egy saját *Resource* implementációt, ami a *QueryBasedResource* (lásd 5.4. fejezet).

Műveletek végrehajtása a célmodellen: Miután sikeresen definiáltuk a gráfmintákon, hogy az illeszkedések esetén milyen transzformcáiós szabályok fussanak le, a létrehozott erőforrás objektumnak ezeket végre is kell hajtania. A végrehajtás során a nehézséget az okozza, hogy a frissített célmodellnek konzisztensnek és érvényesnek kell lennie a hozzákapcsolódó metamodellben (lásd 5.5.1. fejezet).

[2]POJO: „plain old java object” - egyszerű java objektum

5. fejezet

Megvalósítás

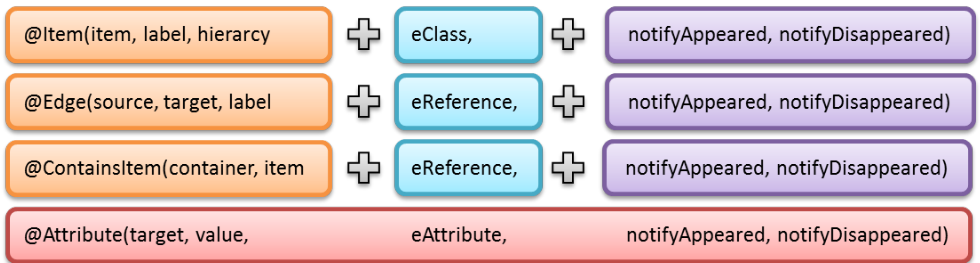
A megvalósítás során a cél az *EMF-IncQuery* technológia kibővítése úgy, hogy a mintákon keresztül származtatott objektumokat lehessen definiálni, amik származtatott objektumokat határoznak meg; majd következő feladatként az erőforrás ezeket a műveleteket hajtsa végre úgy, hogy a célmodell minden esetben egy konzisztens és érvényes modell maradjon; mindezt egy nyomonkövethetőségi modell segítségével, a forráselemek és célelemek életciklusát kapcsolja össze.

5.1. Műveletek definiálása mintákon keresztül

A származtatott objektumok definiálásának alapötlete az volt, hogy szinkronizációs szabályokat határozzunk meg gráfmintákon keresztül. A minta illeszkedése előfeltétele (*precondition*) a szabály végrehajtásának, maguk a szabályok pedig az annotációkban lesznek meghatározva.

A minták definiálásához az *IncQuery* mintanyelvét használtam fel. Mivel már így is rengeteg annotációt használt a nyelv, ezért a már meglévő *Viewers* technológiát bővítettem ki (lásd 5.1. ábra). A meglévő *Item*, *Edge*, *ContainsItem* annotációkat úgy kellett kiegészíteni, hogy tartalmazzanak információt arról, hogy milyen *EClass* példányt hoznak létre, illetve milyen *EReference* kapcsolatot állítanak be. Emellett egy új annotációra is szükség volt, ami az attribútumok leképezését valósítja meg. Ezen keresztül lehet beállítani különböző *EAttribute* értékeket adott elemekhez. Minden annotációhoz létrejött még két új paraméter: $notifyAppeared$, $notifyDisappeared$, amikkel meg lehet határozni, hogy frissüljön-e a modell az adott annotációban meghatározott szabály szerint, ha a minta egy illeszkedése megjelenik, vagy eltűnik. Alapértelmezésként mindkét paramétert értéke *true* vagyis igaz.

Az implementálás időszakában éppen fejlesztés alatt volt az *IncQuery* nyelvben az `eval` kulcsszó bevezetése, aminek segítségével egyszerűbb aritmetikai műveleteket (összeadás,



5.1. ábra. *Annotációk kibővítése az EMF-IncQuery Viewersben*

kivonás stb) értékeltethetünk ki az illesztett attribútumok fölött. A dolgozat írása során el is készült belőle egy alfa verzió, azonban használata során néhány korlát is előkerült: Például az esettanulmányban szereplő transzformációk között feladat egy *String* érték `','` karakterek mentén történő feldarabolása, és az így kapott tömb értékül adása. Ezt az `eval` kulcsszóval jelenleg nem lehet definiálni, ezért ahhoz, hogy ilyen komplexebb szabályokat is létrehozhassunk, elengedhetetlen egy komolyabb szkript nyelv támogatása az annotációkban. Első körben a *JavaScript* nyelvet választottam, de rövidtávú terveink között szerepel az *eval* kiértékelőjének továbbfejlesztése és teljes integrációja a keretrendszerbe.

Az *Attribute* annotációban lévő `value` paraméternek két fajta értéke lehet: *(i)* a minta egy paramétere, vagy *(ii)* egy *String*. A *(ii)* esetben a szöveg tartalmazhat `$` és `$` karakterek között paramétereken navigációt is (lásd 3.2.1.1. fejezet). Ha a szöveg `@` karakterrel kezdődik, akkor a *Viewers* ezek után *JavaScript* kódként fogja értelmezni. Emellett a `$` és `$` karakterek közé beírt kifejezések ugyanúgy be fognak helyettesítődni, és a kód a behelyettesített értékekkel fog kiértékelődni.

5.1. Kódrészlet. *Attribute annotáció használata szkript kóddal kombinálva*

```
1  @Attribute(target = function, value = "@$reqIds$.split(',')",
2            eAttribute = "requirements")
3  pattern functionReqIds(function : Function, reqIds : EString) {
4          find propertyValueForElement(function, "reqId", reqIds);
5  }
```

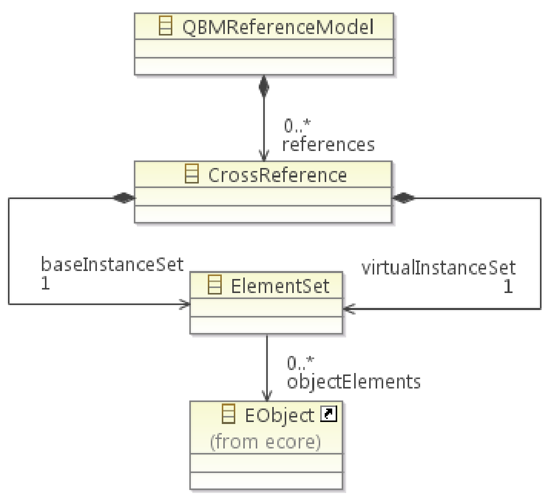
Ezekkel a kiegészítésekkel már képesek lehetünk néhány bonyolultabb származtatott objektum definiálására is, amik létrehozásához minden szükséges információ megjelenik a mintákban és az annotációkon.

5.2. EMF alapú nyomonkövethetőség

A minták definiálása során fontos, hogy tudjunk hivatkozni azokra az elemekre, amik egy bizonyos elemből jöttek létre, illetve azokra az elemekre, amikből egy bizonyos elem létrejött. Ehhez egy nyomonkövethetőségi modellt kell létrehozni, összeköti a forrás elemek

és cél elemek életciklusát.

A nyonomkövethetőség metamodellje látható a 5.2. ábrán, melynek gyökér eleme a *QBMReferenceModel*, amely a kapcsolatokat tartalmazza. A kapcsolatokat az *CrossReference* osztály írja le különböző elemhalmazok között. Mivel egy elemből több, illetve egy elem több elemből is létrejöhet, ezért az életciklusok kapcsolata elemhalmazok kapcsolataként írható le. Az elemhalmazokat az *ElementSet* definiálja, amiben tetszőleges típusú *EMF* objektum található, amit, mivel az *EMF*-ben minden osztálya megvalósítja, az *EObject*-tel lehet azonosítani. A *CrossReference* osztály *virtualInstanceSet* referenciája tehát azokat a tetszőleges típusú elemek halmazát határozza meg, amik a *baseInstanceSet* referencián található elemek halmazából jött létre.



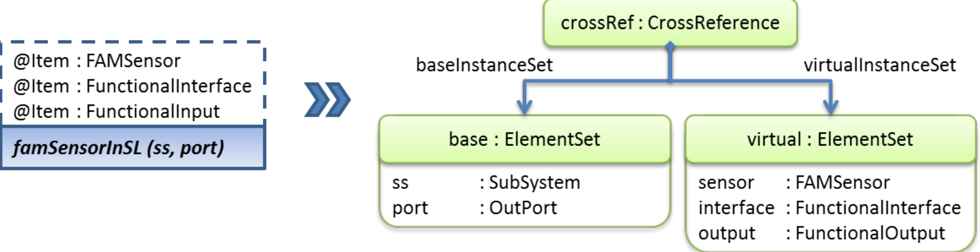
5.2. ábra. *Nyomonkövethetőségi metamodell*

A modellnek olyan gráfminták illeszkedésének változása esetén kell módosulnia, amik rendelkeznek *@Item* annotációval. Ebben az esetben a minta illeszkedésénél létre kell jönnie egy *CrossReference* példánynak, aminek a *baseInstanceSet* halmazában a minta összes paramétere megtalálható lesz, a *virtualInstanceSet* halmazában pedig a mintához tartozó összes *@Item* annotáció alapján létrejött elem kerül. Ha több minta számára is megfelelő ugyanaz az illeszkedés, a modellben minden minta esetén külön referencia példánynak kell keletkeznie.

Példa Tekintsük a 3.6. ábrán látható mintát, amit három *@Item* annotációval láttunk el, ahogy az az 5.3. ábra bal oldalán is látható[1]. A minta illeszkedése során a nyomonkövethetőségi modellben egy olyan *crossRef : CrossReference* példánynak kell létrejönnie, melynek *baseInstanceSet* halmazában az illeszkedés paraméterei találhatóak az *ss* és *port* objektumok, a *virtualInstanceSet* halmazban pedig az *@Item*-ben definiált osztályok egy-egy pél-

[1] Az *@Item* annotációk mögött az *eClass* paraméter értéke van megadva

dánya, úgy mint *FAMSensor (sensor)*, *FunctionalInterface (interface)* és *FunctionalInput (input)*.



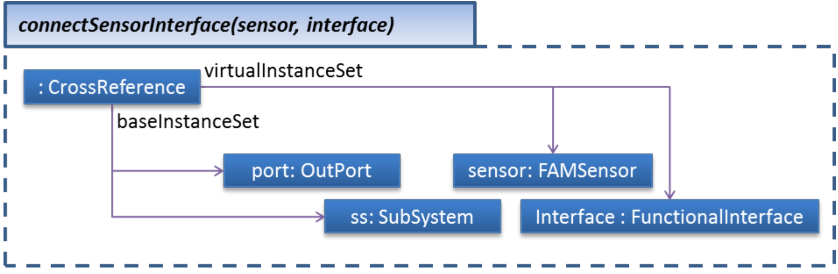
5.3. ábra. *Nyomonkövethetőségi modell bővítése illeszkedés esetén*

Az így elkészített modellen definiálhatjuk a `trace` mintát, amin keresztül elérhetjük a kapcsolatot az eredeti és a belőlük létrehozott elemek között (5.2. kódrészlet). Ez a minta statikusnak tekinthető, de sajnos jelenleg az *IncQuery* még nem tudja az *Eclipse*-be telepített mintákat elérni, ezért projektjeink során egyszer mindenképp definiálni kell.

5.2. Kódrészlet. *Nyomonkövethetőség* `trace` *mintája*

```
1 pattern trace(source, target) {
2         CrossReference.baseInstanceSet.objectElements(cRef, source);
3         CrossReference.virtualInstanceSet.objectElements(cRef, target);
4 }
```

Olyan minták esetén, ahol éleket szeretnénk definiálni cél modellbeli elemek között, rendkívül hasznos a *trace*. Tekintsük azt az esetet, ahol azok között a *sensor : FAMSensor* és *interface : FunctionalInterface* objektumok között szeretnénk kapcsolatot definiálni, amik ugyanazon *ss : SubSystem* és *port : OutPort* párosból jöttek létre. Ehhez a következő mintának kell teljesülnie:



5.4. ábra. *Nyomonkövethetőségi modell bővítése illeszkedés esetén*

Az 5.4. ábrán látható *connectSensorInterface* minta *EMF-IncQuery*-ben meghatározva az 5.3. kódrészleten olvasható.

5.3. Kódrészlet. *A `trace` minta használata*

```
pattern connectSensorInterface(sensor, interface) {
        find trace(ss, sensor);
        find trace(ss, interface);
        find trace(port, sensor);
        find trace(port, interface);

        FAMSensor(sensor); FunctionalInterface(interface);
        SubSystem(ss); InPort(port);
}
```

5.3. Transzformációs szabályok IncQuery-vel

A gyakorlatban tehát *EMF-IncQuery* mintákkal definiálhatjuk a származtatott objektumokat. Ehhez a mintákon az *@Item*, *@Edge*, *@Attribute* annotációkat használhatjuk fel elsődlegesen. Mivel az *EMF* metamodellekben található kapcsolatok egyértelműen tárolják azt az információt, hogy ők egy tartalmazási viszont írnak-e le vagy sem, ezért a *@ContainsItem* annotáció funkcionalitása ebben az esetben egyenértékű az *@Edge* annotációval.

A minták illeszkedéséhez származtatott objektum példányokat az *@Item* annotációval definiálhatunk. Az annotáció *target* paraméterének a minta valamelyik tetszőlegesen kiválasztott paraméterét kell megadni. Ennek értéke voltaképp érdektelen a keretrendszer számára, mivel mindenképp a minta összes paraméterével, mint összetett kulccsal fogja azonosítani az így definiált példányokat. Az *eClass* paramétere *String* típusú és annak a metamodellbeli osztálynak a nevét kell tartalmaznia, amit a minta illeszkedésének létre kell hoznia.

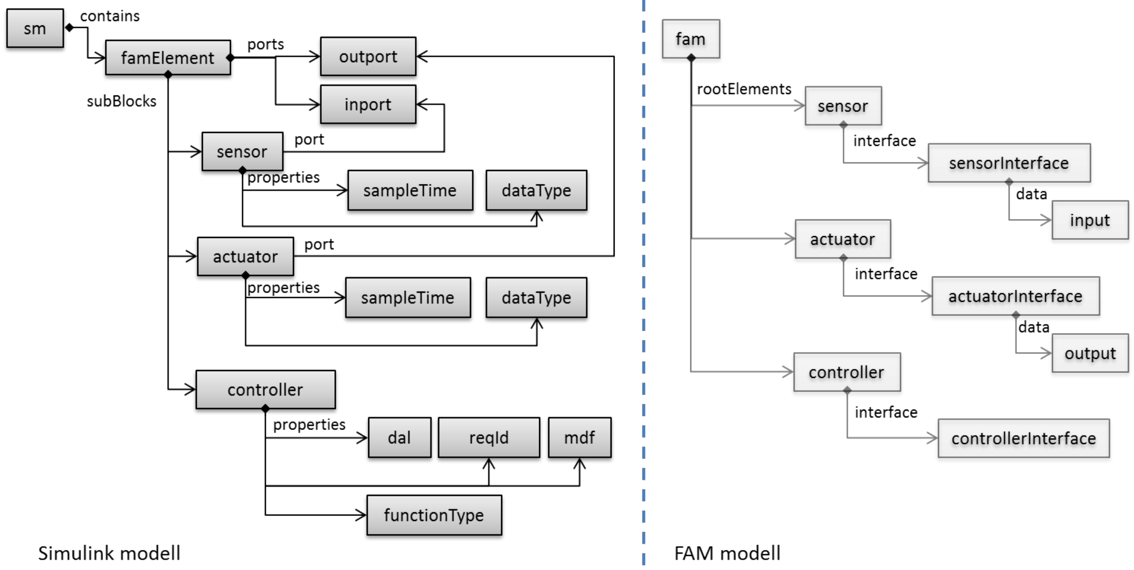
Az *@Edge* annotáció a kapcsolatok leírására szolgál. Ehhez olyan mintákat kell definiálni, amik paraméterei között cél modellbeli elemek is vannak, ugyanis az annotáció *source* paraméterének a minta azon paraméterét kell megadni, ami azt a cél modellbeli elemet jelenti, aminek a referenciáját szeretnénk beállítani. A *target* pedig a minta azon paramétere legyen, amit szeretnénk beállítani a referenciához. A referencia azonosításához az *eReference* paraméternek kell beállítani a referencia nevét.

Az attribútumok beállítására tehát az *@Attribute* annotáció szolgál. Az attribútum azonosításához az *eAttribute* paraméternek kell beállítani az attribútum nevét. A *target* paraméternek azt a cél modellbeli elemet kell beállítani, aminek a kiválasztott attribútumát szeretnénk beállítani. A *value* pedig meghatározza, hogy milyen értéket kell az attribútumhoz rendelni.

5.3.1. Alkalmazása a gyakorlatban

A fejezet folytatásként a kiterjesztett gráfminta nyelv használatát a már korábban ismertet mintapéldán mutatom be.

A forrás és cél metamodell tehát legyenek a 3.2. és 3.3. ábrákon láthatóak, a forrás modell és a cél modell pedig 5.5. ábra bal illetve jobb oldalán láthatóak, amik leegyszerűsített változatai a F.1. és F.2. függelékben található példánymodelleknek. Az ábrán az egyszerűbb vizuális megjelenítés miatt nem jelennek meg az objektumok típusa és attribútumaik értéke, csak a nevük alapján azonosítjuk őket. A dolgozat jelen fejezete azzal a színezési konvencióval él, hogy a fekete színnel jelölt elemek már léteznek a megadott modellben, míg a szürkével jelöltek még nem jöttek létre, de a példa során materializálódni fognak. Szaggatott vonallal és piros színnel lesznek jelölve az adott lekérdezésekhez tartozó illeszkedések elemei, míg szintén szaggatott vonallal, de zöld színnel az adott illeszkedés hatására létrejövő elemek és élek.

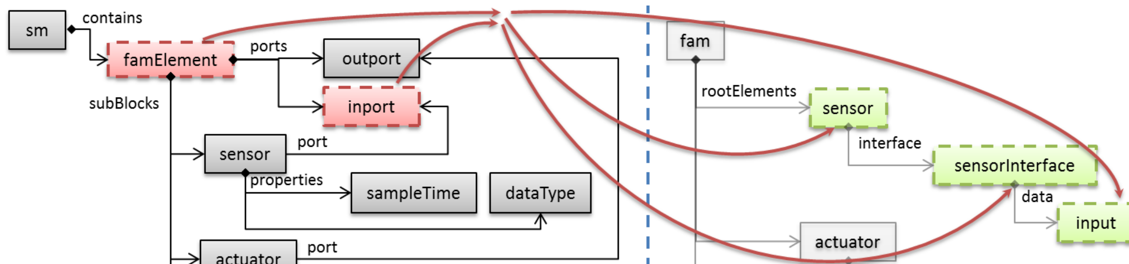


5.5. ábra. *Forrás és cél modell objektumai és azok kapcsolatai*

Első lépésként definiáljuk azt a mintát, ami a cél modellben létrehozza a *sensor*, *sensorInterface* és *output* elemeket. Ez a minta egyébként a 3.2.1.1. ismertetett minta annotációkkal ellátva:

```
1 @Item(item = inport, eClass = "FunctionalInterface")
2 @Item(item = inport, eClass = "FAMSensor")
3 @Item(item = inport, eClass = "FunctionalInput")
4 pattern famSensorInSL(ss : SubSystem, inport) {
5         SubSystem.tag(ss, "FAM_Element");
6         SubSystem.ports(ss, inport);
7         SubSystem.subBlocks(ss, portBlock)
8         InPort.portBlock(inport, portBlock);
9 }
```

A minta illeszkedést és a hozzá kapcsolódó szabály végrehajtásának eredményét mutatja a 5.6. ábra. A nyilak kezdőpontjai az illeszkedés elemeiből indulnak, a végük pedig a végrehajtás eredményeként létrejött elemekre mutat.



5.6. ábra. *A famSensorInSL minta illeszkedése és az ez általt létrejött elemek*

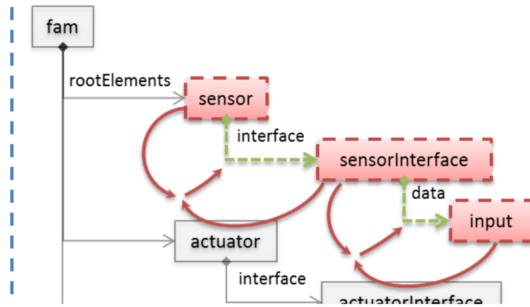
Az elemek létrehozása után automatikusan egy új *CrossReference*-nek is keletkeznie kell nyomonkövethetőségi modellben, melynek a *baseElementSet* halmazába az *SubSystem* és az *InPort* elemek kiválasztott példányai kerülnek, míg a virtualElementSet halmazába az újonnan létrehozott *FunctionalInterface*, *FunctionalInput* és *FAMSensor* elemek kerülnek.

Következő lépésként definiáljuk azt a mintát, ami kiválasztja az előbb létrehozott elemeket, és a megfelelő hierarchiába rendezi őket. A *interface* elem *data* referenciáját az *input*-ra állítja, illetve a *sensor* elem *interface* referenciáját az *interface*-re állítja. Ehhez a *trace* lekérés lett meghívva, aminek első eleme mindig a forrás elem, amiből létrejöttek az új elemek. Ez a forrás elem azonban nem minden esetben a forrás modellből származik, ugyanis egy cél modellbeli elem életciklusa függhet más cél modellbeli elemektől is.

```
1  @Edge(source = sensor, target = interface, eReference = "interface")
2  @Edge(source = interface, target = input, eReference = "data")
3  pattern connectfamSensorElementsInSL(sensor : FAMSensor, input, interface) {
4          FunctionalInput(input);
5          FunctionalInterface(interface);
6
7          find famSensorInSL(_, inport);
8
9          find trace(inport, sensor);
10         find trace(inport, input);
11         find trace(inport, interface);
12 }
```

A lekérdezés definiálása során éltünk azzal az egyszerűsítéssel, hogy a *Simulink* modellben lévő *SubSystem*-hez tartozó *InPort* elemekből csak egyetlen *FAMSensor*, *FunctionalInput* és *FunctionalInterface* jöhetett létre. Ellenkező esetben a *SubSystem* alapján is szűrni kéne az lekérdezés eredményét.

A minta illeszkedése ebben az esetben csak cél modellbeli elemeket választ ki, és köt össze az annotációknak megfelelően. Ezt mutatja a 5.7. ábra.



5.7. ábra. *A connectfamSensorElementsInSL minta illeszkedés, és az így létrejött élek*

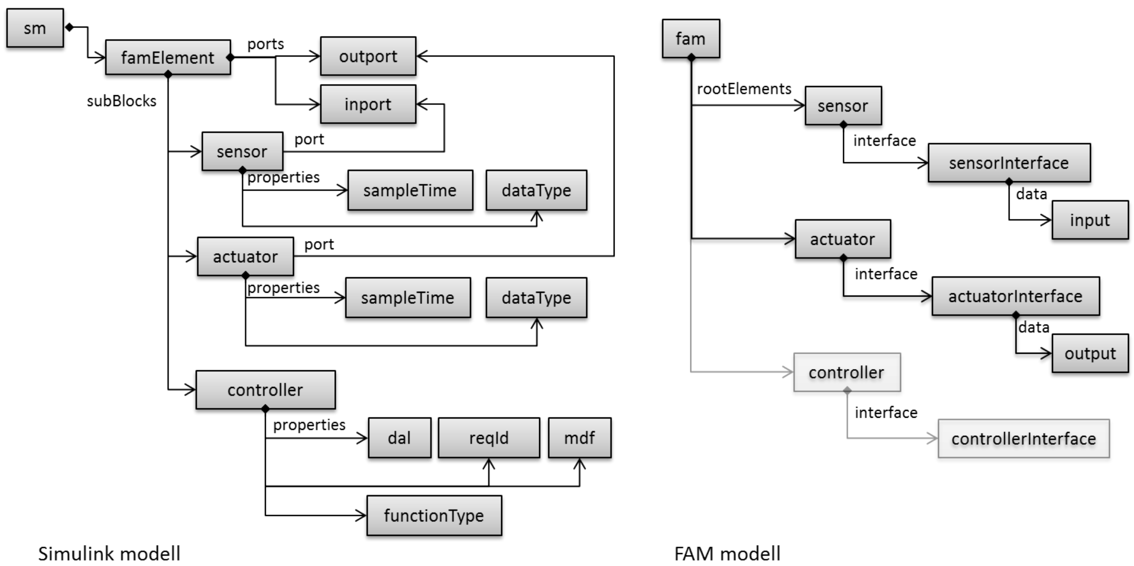
A *FAMActuator*-ral kapcsolatos minták pedig pontosan ugyanazok lennének mint az előbbi *FAMSensor*-hoz tartozó minták, annyi különbséggel, hogy *InPort* helyett *OutPort*-okat, *FunctionalInput* helyett *FunctionOutput*-okat kell használni. A *FunctionalArchitectureModel* elem létrehozására mindössze annyit kell definiálni, hogy mindent *SimulinkModel* elemből jöjjön létre egy.

```
1  @Item(item = slModel, eClass = "FunctionalArchitectureModel")
2  pattern famModel(slModel) {
3          SimulinkModel(slModel);
4  }
```

A *rootElements* referenciához azokat a *FunctionalElement* elemeket kell beállítani, amik forrása abban a *SimulinkModel*-ben található, amiből a *FunctionArchitectureModel* is létrejött. A következő minta ezt definiálja:

```
1  @Edge(source = famModel, target = famElement, eReference = "rootElements")
2  pattern connectFamModelAndFamElements(famModel, famElement) {
3          FunctionalArchitectureModel(famModel);
4          FuntionalElement(famElement);
5
6          find famSensorInSL(slElem,_);
7          SimulinkModel.contains(slModel, slElem);
8
9          find trace(slElem, famElement);
10         find trace(slModel, famModel);
11 }
```

Ezeket a mintákat felhasználva a célmodell már majdnem tartalmazza az összes szükséges elemet. A jelenleg létező elemeket mutatja a 5.8. ábra.

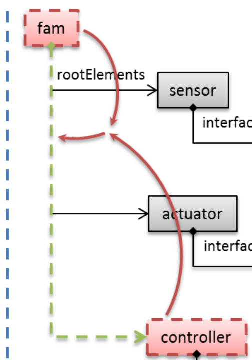


5.8. ábra. *Gyökér elemmel, és FAMActuatorral kiegészített cél modell*

A következő lépésben definiáljuk azt a mintát, ami létrehozza a *Function* elemet és a hozzá kapcsolódó *FunctionalInterface*-t. Ehhez a modellben lennie kell egy olyan *SubSystem*-nek, aminek a *tag* attribútuma *„FAM_Root_Leaf"* értékre van állítva.

```
1 @Item(item = ss, eClass = "FunctionalInterface")
2 @Item(item = ss, eClass = "Function")
3 pattern famFunctionInSL(ss : SubSystem) {
4         SubSystem.tag(ssParent, "FAM_Element");
5         SubSystem.subBlocks(ssParent, ss);
6         SubSystem.tag(ss, "FAM_Root_Leaf");
7 }
```

Miután a minta illeszkedéséhez kapcsolatos műveletek lefutnak, a *connectFamModelAndFamElements* minta is automatikusan egy új illeszkedést kap, aminek hatására triggerelődik az a műveletsor, ami beállítja az újonnan létrejött *Function* elemet a gyökér elem *rootElements* referenciái közé.



5.9. ábra. *A minta végrehajtása újabb mintát triggerel*

Utolsó lépésként a *Function* attribútumait definiáló mintát kell megalkotni. Ezeket a forrás *SubSystem*-hez kapcsolódó *Property* elemekből lehet kinyerni:

5.4. Kódrészlet. *Attribútumok beállítása annotációkkal*

```
1  @Attribute(target = f, value = "$dal.value$",
2                                      eAttribute = "designAssuranceLevel")
3  @Attribute(target = f, value = "$mdf.value$",
4                                      eAttribute = "minumumDesignFrequency")
5  @Attribute(target = f, value = "$type.value$",
6                                      eAttribute = "functionType")
7  @Attribute(target = f, value = "@$reqId.value$.split(\",\")",
8                                      eAttribute = "requirements")
9  pattern functionAttributes(f : Function, dal, mdf, type, reqIds) {
10                 find trace(ss, f); SubSystem.properties(ss, dal);
11                 SubSystem.properties(ss, mdf);
12                 SubSystem.properties(ss, type);
13                 SubSystem.properties(ss, type);
14
15                 Properties.name(dal, "dal");
16                 Properties.name(mdf, "mdf");
17                 Properties.name(type, "type");
18                 Properties.name(reqId, "reqId");
19 }
```

5.4. Modellek betöltésének felüldefiniálása

A keretrendszer megvalósításának egyik fontos feladata az volt, hogy képes legyen felüldefiniálni egy modell betöltődésének a folyamatát. Ehhez az *EMF org.eclipse.emf.ecore* pluginjének *extension_parser* kiterjesztési pontján keresztül kellett meghatározni, hogy egy *„.qbm”* kiterjesztéssel rendelkező *URI* betöltése esetén az általam implementált *QueryBasedResourceFactoryImpl*-t használja a rendszer. Ez a *factory* osztály mindössze átírja az alap *ResourceFactoryImpl* `Resource createResource(URI uri)` metódusát, így visszatérési értékként példányosít egy *QueryBasedResource* osztályt, ami már képes felügyelni a modell betöltődését.

5.5. Kódrészlet. *QueryBasedResourceFactory implementáció*

```
1  public class QueryBasedResourceFactoryImpl extends ResourceFactoryImpl {
2          @Override
3          public Resource createResource(URI uri) {
4                  return new QueryBasedResource(uri);
5          }
6  }
```

Mivel a forrásmodellek, metamodellek és minták különböző fájlokban találhatóak, ezért az operációs rendszer fájlrendszerén egy *„.qbm”* kiterjesztésű fájl is materializálódik, ami tartalmazza a szükséges elemek *URI*-ját. A nézeti modellek betöltését ezen fájlok *URI*-ján keresztül lehet elérni.

Megjegyzendő, hogy a forrás modellek metamodelljét nem szükséges meghatározni, mivel a transzformációs minták egyértelműen hivatkoznak rájuk.

5.5. Inkrementális lekérdezés alapú erőforrás (QBR)

A *QueryBasedResource* osztálynak nem felelőssége, hogy minden *EMF*-fel kapcsolatos műveletet felüldefiniáljon, ezért mindössze a már meglévő *XMIResourceImpl* erőforrás-implementáció `doLoad` és `doSave` metódusait definiálja felül.

Betöltés (doLoad) A betöltés során a következő lépések történnek meg:

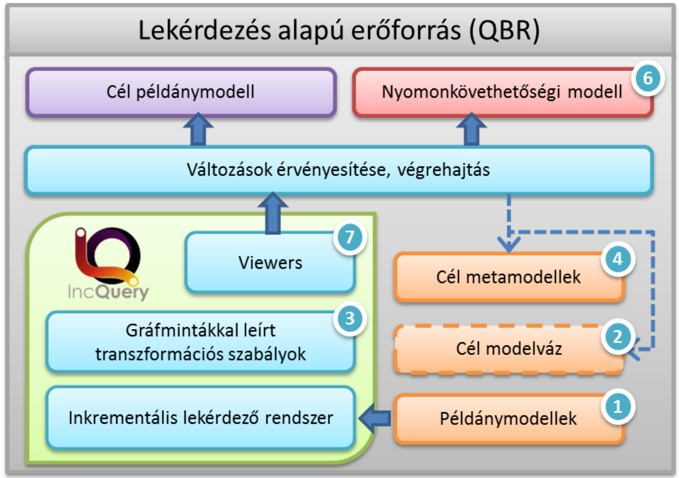
1. Forrásmodellek betöltése
(2.) Célmodell váz betöltése
3. Mintákat tartalmazó modellek betöltése
4. Cél metamodellek betöltése
5. Dinamikus metamodellek felépítése
6. Nyomonkövethetőségi modell létrehozása
7. EMF-IncQuery Viewers inicializálása

(4–5.: opcionális (lásd 5.6. fejezet))

Mentés (doSave) Mentés esetén az forrásmodellekben történt változást menti el az osztály. Az alapértelmezett metóduson kívül egy `doSaveSnapshot` metódussal is rendelkezik az erőforrás, amin keresztül a cél modell pillanatnyi állapotát lehet kisorosítani egy megadott fájlba.

A betöltés után az erőforrás tartalmát és feladatait a 5.10. ábra mutatja. A struktúra a példánymodellek változásán alapul. Változás esetén az inkrementális lekérdező rendszer aktiválódik, és a módosítások alapján a beregisztrált gráfminták illeszkedéseit újra megvizsgálja. Ha a minták illeszkedésében változás történt, akkor a *Viewers*-en keresztül értesül az errőforrás. Az értesítésben megtalálható maga az illeszkedés; újonnan létrejött, vagy megszűnt illeszkedésről van szó; illetve magát a végrehajtandó szabályt. A következő lépésben az erőforrás végre is hajtja ezeket a szabályokat, aminek hatására a célmodell módosul; illetve elemek létrehozása, törlése során a nyomonkövethetőségi modellt is megfelelően frissíti.

Azáltal, hogy a *QBR* erőforrás meglévő *EMF* interfészeket és osztályokat bővít ki, a modellező keretrendszer szempontjából egy egyenértékű modellként fog működni a cél példány



5.10. ábra. *A „Lekérdezés alapú erőforrás" tartalma*

modell. Ezzel biztosítva azt a célt, hogy minden *EMF* technológiát használó alkalmazásba beépíthető legyen, esetlegesen a korábbi modellek lecserélhetőek legyenek, vagy akár más *EMF* alapú modelltranszformációk bemeneteként szolgáljon.

5.5.1. Műveletek végrehajtása a cél modellen

Miután a *QBR* használja a *Viewers* technológiát, következő fontos feladat feliratkozni a változásokra, és végrehajtani az előírt szabályokat, amiket a minta egy új illeszkedése vagy egy régi eltűnése definiál. Ehhez az erőforrásnak implementálnia kell az *IViewerStateListener* interfészt, amiben minden annotáció típushoz található egy illeszkedés megjelenését (*appeared*) és eltűnését (*disappeared*) jelző metódus.

5.6. Kódrészlet. *IViewerStateListener interfész*

```
1  public interface IViewerStateListener {
2      void itemAppeared(Item item);
3      void itemDisappeared(Item item);
4      void containmentAppeared(Containment containment);
5      void containmentDisappeared(Containment containment);
6      void edgeAppeared(Edge edge);
7      void edgeDisappeared(Edge edge);
8      void attributeAppeared(Attribute attribute);
9      void attributeDisappeared(Attribute attribute);
10 }
```

Elemek megjelenésénél egyértelműen egy megadott típusú *EObject* példányt kell létrehozni, majd gyökérelemként hozzáadni a modellhez. Utóbbira azért van szükség, mert az *IncQuery* csak az erőforrásban lévő elemekre képes a mintákat illeszteni (pl.: tartalmazási éleket definiáló minták). Eltűnés során az a kérdés merült fel, hogy mi történjen az elem tartalmával. Erre két lehetőség adódott: *(i)* az elem gyerekei gyökérelemekké váljanak a

modellben, vagy *(ii)* a gyerekelemek is törlődjenek. Az *(i)* esetben a gyerekelemekre vonatkozó minták továbbra is illeszkednek, és végrehajthatóak a további szabályok, míg a *(ii)* esetben ezekkel a transzformációkkal a továbbiakban nem kell foglalkozni. Tervezői döntés alapján a *(ii)* változat került implementálásra.

Az élek és a tartalmazások jelenleg ugyanazt az implementációt használják, mivel *EMF*-ben maguk a kapcsolatok már tárolják azt az információt, hogy ők tartalmazási vagy nem tartalmazási viszonyt képeznek-e két elem között. Élek megjelenése során tehát vizsgálni kell ezt az információt, mivel előbbi esetben meg kell keresni a beállítandó elem eredeti szülő objektumát, és közöttük lévő kapcsolatot meg kell szüntetni. Továbbá fontos azt is megvizsgálni, hogy megfelelő elemek kerülnek-e a kapcsolatok két végére. Eltűnés során a referencia alapértelmezett értéke lesz beállítva minden esetben.

Attribútumok beállítása során különböző típusú értékeket kell típushelyesen beállítani az elemeken. Ehhez először meg kell vizsgálni, hogy az attribútum milyen típust vár értékül. Ha nem ilyen jön, hanem egy *String*, akkor megpróbálja az adott típusként parse-olni. Minden beépített *EMF* adattípust képes létrehozni a szövegekből, illetve *Enum*-okat is ki lehet választani. Az *Enum*-okat számokból *Integer* is képes beállítani. Egy kollekció esetében sem triviális az értékadás, ugyanis egy kollekciót nem lehet értékül adni, minden elemet egyesével kell hozzáadni az attribútumhoz. Végső esetben a megfelelő típusra kasztolással próbálkozik a keretrendszer. Az eltűnés az élekhez hasonlóan a típus alapértelmezett értékét állítja be az attribútumhoz.

Érdemes megfigyelni a 5.4. kódrészletben meghatározott mintában, hogy a *DesignAssuranceLevel* és a *FunctionType* attribútumok *EEnum* típusúak, mégis a *Simulink* modellben *String* értékként vannak letárolva. A *QBR* azonban képes ezeket feloldani és értékül adni. A *MinimunDesignFrequency* pedig *Integer*-ként található meg a modellben, de a *QBR* a *FAM* metamodellnek megfelelően egy *EFloat* típusú értéket hoz létre. A *requirements* attribútumról már korábban is volt szó, ami miatt szükségessé vált a *JavaScript* nyelv integrálása a rendszerbe. A *Simulink* modellben egy vesszőkkel elválasztott *String* érték, amit *Integer* listaként kell a *FAM* modellben kezelni.

Az így definiált műveletek biztosítják azt, hogy egy-egy szabály végrehajtása után továbbra is konzisztens maradjon a célmodell.

5.6. Kiegészítés: Dinamikus metamodell építés

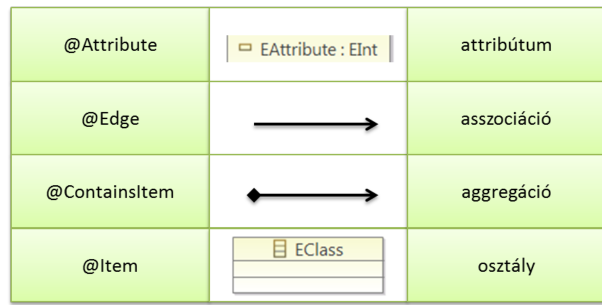
Több metamodellre történő fejlesztés esetén, ahol a fejlesztés kezdeti fázisaiban a metamodellek folyamatos finomítására van szükség, hasznos lehet egy olyan funkcionalitás, mely során nincs szükség a konkrét cél metamodellre, hanem az annotációkon keresztül de-

finiálhatjuk azok struktúráját, majd a nézeti modell betöltődésekor dinamikusan épülnek fel a memóriában. Az általam megvalósított keretrendszer támogatja ezt a funkcionalitást, ami bár tartalmaz bizonyos megkötésekkel, azonban képes szemléltetni azokat a modell elemeket, amik szerepelni fognak egy adott metamodellben, és azok példánymodelljeiben. Hasznos lehet abban az esetben is, ha a célmodell metamodellje nem áll rendelkezésünkre, de a felépítését szeretnénk szemléltetni.

EMF Dynamic EClass Az *EMF* keretrendszer nemcsak a példánymodellekből képes létrehozni dinamikus objektumokat, hanem magát a metamodellt is létrehozhatjuk vele. Az így létrehozott dinamikus metamodell elemekből pedig szintén példányosíthatunk különböző elemeket.

Nyelvi támogatás Az *IncQuery Viewers* annotációit, már a 5.1. fejezet kibővítette különböző paraméterekkel. Ahhoz, hogy a metamodelleket ezeken keresztül felépíthessük még szükség van az *Edge*, *ContainsItem*, *Attribute* annotációk paraméterei közé egy *eClass* paramétert is definiálni.

Az annotációkon keresztül tehát le tudunk írni különböző osztályokat, az *Item* annotáció *eClass* paraméterén keresztül, ahol a paraméter értéke az osztály neve lesz. Az attribútumokat az *Attribute* annotációk *eAttribute* paramétere adja, az *eClass* paraméter pedig meghatározza, hogy melyik osztályhoz fog tartozni. Míg a szinkronizációs szabályok leírásánál nem volt különbség az *Edge* és a *ContainsItem* annotáció között, a metamodellek felépítésénél viszont már lesz jelentősége. Az *ContainsItem* egy tartalmazási referenciát, vagyis aggregációt, míg az *Edge* asszociációt definiál az *eClass* paraméterben meghatározott osztályhoz. A dinamikus metamodellezés során szemlélteti az annotációk jelentését a 5.11. ábra.



5.11. ábra. *Annotációk jelentése dinamikus metamodellezés során*

A *QBR* erőforrás alapértelmezett működése során is felolvassa a szabályokat definiáló *Pattern* modelleket, amit aztán átad a *IncQuery* mintaillesztő motorjának. Ezért a felolvasás közben képes kiolvasni a modellen található összes annotációt is. Az erőforrás úgy lett

implementálva, hogy ha nincs definiálva cél metamodell a *.qbm* fájlhoz, akkor megpróbálja felépíteni az annotációk alapján a metamodell.

A metamodellek felépítése során néhány egyszerűsítéssel élünk. Mivel típusosságot nem lehet definiálni jelenleg az élekhez és az attribútumokhoz, ezért minden ilyen elem típusa a legalapvetőbb *Java* objektum lett, ami a `java.lang.Object`. Egy másik korlát a keretrendszerben, hogy jelenleg származtatást nem képes kezelni, ami rendkívül hasznos lenne, de ez a funkcionalitás is a rövid távú célok között szerepel (lásd 8.3. fejezet).

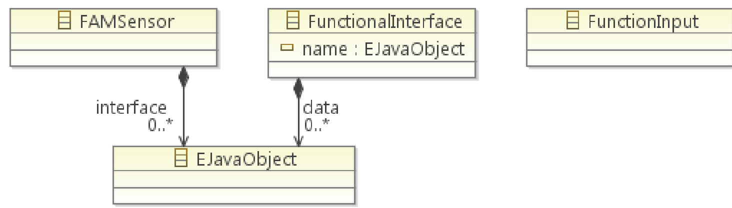
Az így definiált nézeti modellek semmiben sem viselkednek másképpen, mint a cél metamodellel rendelkezőek. Az automatizmus továbbra is fennáll, és az alkalmazások is ugyanolyan *EMF* példány modellnek tekintik, mint társaikat.

Példa Jelen funkcionalitás szemléltetésére tekintsük a korábban definiált *famSensorInSL* és a *connectFamSensorElementsInSL* mintákat, de az annotációjukat cseréljük le a következőképpen:

```
1 @Item(item = inport, eClass = "FunctionalInterface")
2 @Item(item = inport, eClass = "FAMSensor")
3 @Item(item = inport, eClass = "FunctionalInput")
4 pattern famSensorInSL(ss : SubSystem, inport) {
5         // ...
6 }
```

```
1 @ContainsItem(source = sensor, target = interface,
2                         eReference = "intreface", eClass = "FAMSensor")
3 @ContainsItem(source = interface, target = input,
4                         eReference = "data", eClass = "FunctionalInterface")
5 @Attribute(target = interface, value = $sensor.name$.interface,
6                         eAttribute = "name", eClass = "FunctionalInterface")
7 pattern connectfamSensorElementsInSL(sensor : FAMSensor, input, interface) {
8         //...
9 }
```

A *famSensorInSL* mintán lévő annotációk meghatározzák, hogy a készítendő metamodellben a következő osztályok fognak szerepelni: *FAMSensor*, *FunctionalInterface*, *FunctionalInput*. A hozzájuk tartozó kapcsolatokat és a megjelenő attribútumokat a *connectfamSensorElementsInSL* annotációi definiálják. Eszerint a *FAMSensor* és a *FunctionalInterface* egy-egy aggregált referenciával rendelkezik, illetve az *FunctionalInterface* egy *name* attribútumot is kap. Mindent referencia és attribútum típusa *EJavaObject*, ami *EMF*-ben a `java.lang.Object` osztályt reprezentálja. Az így keletkezett metamodellt a 5.12. ábra szemlélteti.



5.12. ábra. *Az annotációk alapján felépülő metamodell*

6. fejezet

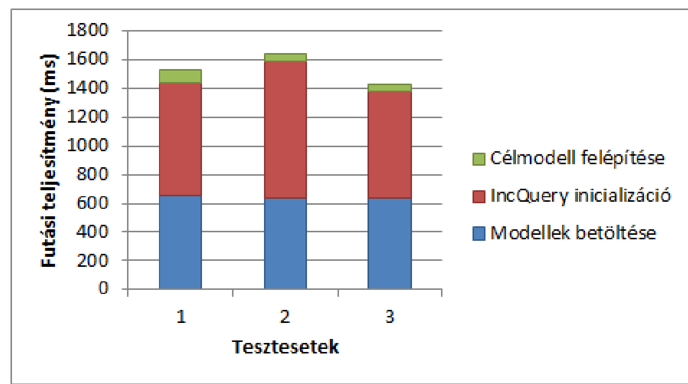
Értékelés

Az elkészült keretrendszert az esettanulmányban szereplő releváns repülőgépipari projekt keretében lehetett letesztelni. A tesztelés során elsősorban azt a futás teljesítményt vizsgáltam, ami megadja, hogy a kezdeti betöltés során mennyi idő alatt hajtódnak végre a szinkronizációs szabályok, illetve a forrásmodellek változásának bekövetkezésétől a változás hatásának megjelenéséig mennyi idő telik el.

A teszteket 8 különböző, de közel azonos bonyolultságú minta példán végeztem el, ezek mindegyikén 10-10 független mérést hajtottam végre, majd ezek átlagát vettem. A forrás modellekben külön-külön közel 400 elem volt, az ezekhez tartozó cél modellen közel 100 elem jött létre, szintén ennyi kapcsolat lett beállítva és kicsivel több mint 180 attribútum értéke lett meghatározva.

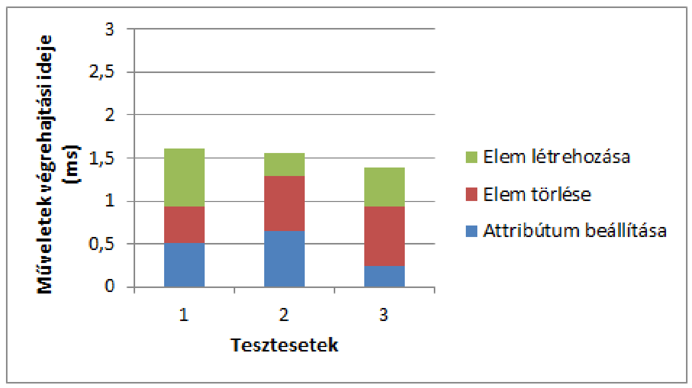
A mérések elvégzéséhez egy Intel Core i7-3770K 3,50 GHz-es processzorral és 16 GB RAM-mal rendelkező gépet használtam, amin 64-bites Windows7 operációs rendszer volt, keretrendszerem pedig az Eclipse 4.3 Kepler környezetbe volt telepítve.

A modellek betöltésével kapcsolatos eredmények egy részét a 6.1. ábra szemlélteti, amin egy bonyolultabb (1), közepesen bonyolult (2) és egy egyszerűbb (3) modellt használtam példának. A diagram vízszintes tengelyén a tesztesetek azonosítói láthatóak, míg a függőleges tengelyén a futási idő milliszekundumban van ábrázolva. Az egyes példákhoz tartozó oszlopok három részre vannak osztva, ami szemlélteti a teljes futási időn belül a forrásmodellek, metamodellek és gráfminta modellek betöltésén idejét (kék); a *EMF-IncQuery*-hez kapcsolódó RETE-háló felépítésének idejét (piros); illetve a minták kezdeti illeszkedéséhez kapcsolódó végrehajtási művelet lefutási idejét, aminek hatására felépül a célmodell (zöld). Az ábráról leolvasható, hogy a modellek betöltése és az *EMF-IncQuery* inicializálása közel ugyanannyi időt vett igénybe, míg a modellek felépítésére jóval kevesebb idő szükséges.



6.1. ábra. *Betöltés végrehajtási ideje*

A 6.1. ábra szintén ugyanazon a három példamodellen alapul, viszont itt a forrásmodelleken történő változás bekövetkezésétől lettek számítva az értékek, és a célmodellen történő módosítás befejezéséig tartott. Az ábrán látható, hogy a tesztelés során egy attribútum módosítását (kék), egy elem törlését (piros) és egy elem létrehozását (zöld) váltottam ki a célmodellen. A diagramon található értékekből azt a következtetést vonhatjuk le, hogy a módosítások lefutásainak ideje között nincs nagyságrendi különbség, és értékük is rendkívül alacsony lett



6.2. ábra. *Módosítás végrehajtási ideje*

A mérések során a vártaknak megfelelő eredmények születtek. A célmodellek felépítéséhez átlagosan 1,17534 másodperc volt szükséges. Ennek nagyjából 41,15%-a volt a forrás modelleket, metamodellek és gráfmintákat tartalmazó fájlok felolvasása és modellként történő betöltése. Az *EMF-IncQuery* inicializációja az idő 49,07%-át tette ki. A transzformációs műveletek, és a cél modell felépítése pedig 9,78%-át jelentette. A forrás modellek változása során bekövetkező attribútumok, élek és elemek beállításához/létrehozásához szükséges idő között nem volt mérhető eltérés. Ezek átlagosan 0,0005 másodpercet vett igénybe. A rendszer jóval nagyobb méretű modellek esetén is jól skálázódik köszönhetően a *EMF-IncQuery*-nek, ami több százezres nagyságrendű modellen megfelelő skálázódást mutatott [2].

A kapott eredményekből látható, hogy bár a célmodellek inicializációja időigényes, a változások során bekövetkező származtatott objektumok frissentartása rendkívül gyorsnak, közel azonnalinak tekinthető. Emiatt a gyorsaság miatt akár felhasználói felületekhez is lehetne kötni, hiszen ott felhasználók számára azonnalinak is tűnhet egy-egy módosítás következtében a céllmodell frissítése.

7. fejezet

Kapcsolódó munkák

7.1. Virtual EMF

Az inkrementális modell-szinkronizáció problémájára már más technológia is próbált megoldást találni az *EMF* keretrendszer felett. Ilyen például a *VirtualEMF*, ami hasonló elven működik az megalkotott megoldáshoz, csak más technológiákat felhasználva. Ezt a projektet a francia *AtlanMod* kutató csapat fejleszti.

A *VirtualEMF* annyiban különbözik az általam megvalósítottaknál, hogy nem hoz létre új modell-objektumokat a nézeti modellben, hanem saját *EMF Resource* és *EObject* implementációval rendelkezik, amelyek egyfajta proxy funkciót látnak, és transzparensen olvassák ki az alap modell értékeit minden egyes lekérdezés során[6].

Szerettem volna ki is próbálni ezt a lehetőséget, de jelenleg nincs működőképes állapotban. Miután a kapcsolatot felvettem a francia kutatókkal, elmondták, hogy egyhamar nem is lesz használható állapotban.

7.2. Query/View/Transformation (QVT)

A *Query/View/Transformation (QVT)* egy *OMG* által meghatározott deklaratív nyelv, mellyel egyirányú és kétirányú transzformációt is definiálhatunk. Transzformációkat modellek közötti relációk halmazával írhatunk le, amiket meghívhatunk *check only* módban, amivel a modellek közötti konzisztenciát vizsgálhatjuk, illetve *enforce* módban, amivel az egyik modellt módosítjuk a másiknak megfelelően a konzisztencia elérése érdekében [10].

Bár a *QVT* kibővítéseként már létezik inkrementális megvalósítása *EMF* felett is [17], azonban ebben az esetben a modell változásának hatására ismét az összes transzformációt

check only módban le kell futtatni, majd ezek eredményei alapján lehet kiszűrni a szükséges változásokat. Emiatt az általam megvalósított keretrendszer ennél egy hatékonyabb megoldást ad.

7.3. Változásvezérelt modell transzformációk

A változásvezérelt modell transzformációk (*Change-Driven Model Transformations*) megközelítés egy jóval komplexebb és általánosabb megoldást ad a származtatott objektumok koncepciójára. Mivel egy teljes transzformációs nyelvet használ fel, nincsenek nyelvi határai a szinkronizációs szabályok definiálására. Alapötlete, hogy a változások által triggerelt transzformációkat aszinkron módon hajtódjanak végre. Ezeket a végrehajtandó transzformációkat egy *Change History* modellben tárolja, és sorrendben egymás után hajtja végre azokat.

Az így meghatározott szabályok szakítanak a tradicionális transzformációs szabályokkal, ugyanis míg hagyományos esetben a transzformáció bemenete és kimenete is egy-egy modell, addig az új szabályok modelleken végrehajtható operációkat várnak a bemenetükön, és produkálnak a kimenetükön[16].

A nehézsége, hogy az összekapcsolt életciklusok elszakadhatnak egymástól, emiatt sok esetben jóval bonyolultabbak a szabályok definiálása. Ezen kívül mindent változás fajtára külön kell definiálni a szabályokat, ezért használata elég nehézkes.

Jelenleg ebből a technológiából sem áll rendelkezésre *EMF* felett hatékonyan működő megoldás.

7.4. Triple Graph Grammar (TGG)

Az inkrementális szinkronizáció már korábban is létező megközelítés volt a modellből modellt készítő transzformációk között. Egy ilyen megközelítés a *Triple Graph Grammar (TGG)* is, ami képes konzisztens állapotban tartani a forrás és cél modellt bármelyik változása esetén is, vagyis kétirányú transzformáció is megvalósítható vele. Előnye, hogy képes meglévő nézeti modelleket is összekapcsolni forrásmodelleket, és az azok közötti kapcsolatot is, majd ezután inkrementálisan szinkronizálni [13].

A *TGG* koncepciójához hasonló megközelítést használnak *UML* modell alapú eszközök integrálásához is. Ebben a cél, hogy támogatást biztosítson a fejlesztés különböző fázisaiban felhasznált nyelvek szinkronizációjára[1]. Itt azonban a nyomonkövethetőséget explicit definiálni kell.

A *TGG*-hez készült egy általános interpreter implementáció, aminek segítségével bármilyen modellen lehet *TGG* szabályokat futtatni[12], azonban minden egyes metamodellhez meg kell valósítani egy adapter osztályt, amin keresztül az interpreter végre tudja hajtani a műveleteket.

EMF felett jelenleg ehhez a koncepcióhoz sem létezik hatásos megvalósítás.

8. fejezet

Összefoglalás és jövőbeli tervek

A *Modellvezérelt fejlesztés (MDE)* egyre szélesebb körben használt koncepció beágyazott és biztonságkritikus szoftverek tervezésénél és fejlesztésénél, mely során precíz modelleket definiálunk, amikből automatikusan generálhatunk alkalmazás kódot, konfigurációs leírókat stb. Gyakorta válik szükségessé az így felépített bonyolult rendszermodellekhez különböző nézeti modellek definiálása, mivel az adott felhasználási esetben több irreleváns információt is tartalmazhat (*model abstraction*).

A *modell szinkronizáció* feladata, hogy az így definiált nézeti modellek (cél) mindig konzisztensek legyenek az alapul szolgáló rendszermodellel (forrás). Általában ilyen feladatok elvégzésére kötegelt *(batch)* transzformációkat alkalmaznak, ami nem inkrementális megoldás, vagyis minden változás esetén újraépítik a cél modellt.

Jelen dolgozatban erre a problémára adtam egy inkrementális megoldást az *Eclipse Modeling Framework (EMF)* keretrendszerben definiált modellek fölött, ahol a szinkronizációs szabályok lekérdezés alapú deklaratív gráfmintákkal adhatóak meg. Az így definiált lekérdezéseket az *EMF-IncQuery* inkrementális gráfmintaillesztő hajtja végre.

8.1. Megvalósított célok

A bevezető 1. fejezetben definiált célok között az elsődleges feladat a *származtatott objektumok* koncepciójának megvalósítása volt *EMF* modellek felett, aminek segítségével elérhető az inkrementális *modell-absztrakció*.

A feladat megoldásához a következő részfeladatokat kellett megvalósítani az implementálás során:

(i) Az *EMF-IncQuery* technológia deklaratív lekérdező nyelvét ki kellett bővíteni, hogy képes legyen származtatott objektumok definiálására.

(ii) A lekérdezésekben definiált szabályok végrehajtásához, és az inkrementális viselkedés megvalósításához egy saját *EMF* erőforrás implementációt alkottam meg.

(iii) Az egyes célmodellbeli elemek életciklusához kapcsolódó forrásmodellbeli elemek összekapcsolasára egy nyomonkövethetőségi modellt definiáltam, amit a saját erőforrás implementáció tart karban.

(iv) Kiegészítésként az erőforrást kibővítettem úgy, hogy szükség esetén képes legyen a szabályoknak megfelelő dinamikus metamodellt létrehozni a cél modell számára.

Emellett a kialakított keretrendszer képes más *MDE*-ben fontos feladatot is megoldani, mint például

(i) olyan nézeti modell készítésére, ami képes megjeleníteni több forrásmodelltől függő értékeket *(model merge)*, vagy

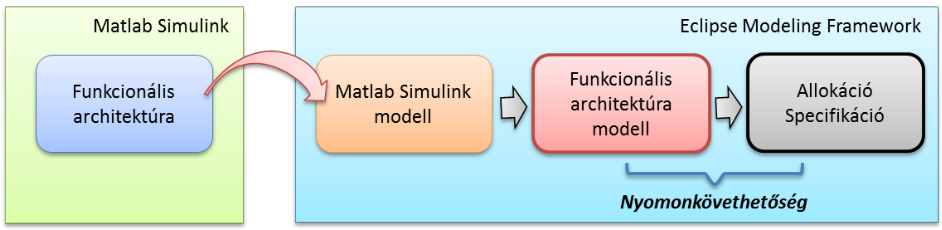
(ii) kibővíteni egy már meglévő modell vázat *(model extension)*.

Az így létrehozott modellek teljesen szabványos *EMF* modellként viselkednek, ezért minden *EMF* technológiát használó alkalmazás képes beépíteni őket, vagy a korábbi modelleket lecseréli ezekkel a nézeti modellekkel. Emellett más *EMF* alapú modelltranszformációk bemeneteként is szolgálhat. Így a másik kitűzött célt is sikerült teljesíteni.

8.2. Származtatott objektumok további kutatásokban

A megvalósított származtatott objektumok felhasználására egy másik keretrendszer fejlesztésénél is igény mutatkozott, ami pilot jelleggel fel lett használva a dolgozatban szereplő kutatási projektben.

A kialakított *allokáció specifikáció (AS)* felhasználja a *FAM* modellekben található adatokat és új példányokat épít belőle. Azonban az *AS* már nem egy nézeti modellt definiál a *FAM* modellekhez, hanem egy teljesen új szemantikát ír le. Az így létrejött modellek szinkronizációjához viszont szintén szükséges a *FAM* és *AS* elemek közötti nyomonkövethetőségi kapcsolat meghatározása.



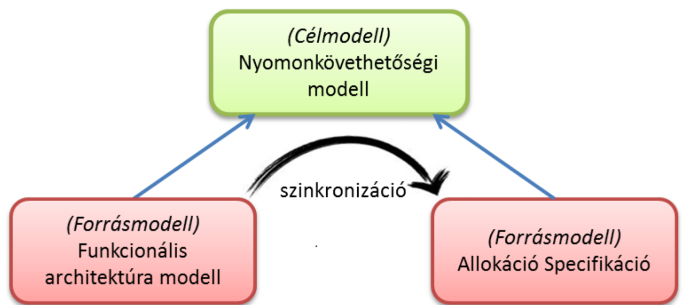
8.1. ábra. *Nyomonkövethetőség a kutatási projektben*

Dóczi Róbert III. éves mérnök-informatikus hallgató (BSc) szakdolgozata egy olyan általános keretrendszert valósít meg *EMF* felett, ami képes már meglévő modellek közötti

nyomonkövethetőség meghatározására, majd ezen keresztül a szinkronizáció megvalósítására. A modellelemek közötti összekapcsolásra egy új nyomonkövethetőségi metamodellt definiált, aminek példánymodelljei különböző *EMF-IncQuery* mintanyelvével meghatározott gráfminták illeszkedéseinek alapján épülnek fel.

Az általam megvalósított keretrendszer ezen a ponton kapcsolódik hallgatótársam szakdolgozatához. Tekintsük *forrásmodellnek* azokat a modelleket, amik között a szinkronizációt szeretnénk megvalósítani, a *célmodell* az elemeket összekötő nyomonkövethetőségi modell legyen, a gráfmintákat pedig csak egészítsük ki megfelelő annotációkkal. Ezáltal a nyomonkövethetőségi modell automatikusan és inkrementálisan működhet a különböző modellek között a származtatott objektumoknak köszönhetően.

A projektben, ahogyan azt 8.2. ábra is mutatja, *forrásmodellnek* tekintjük a *FAM* és *AS* példánymodelleket, amik szinkronizációjához a célmodellként a *Nyomonkövethetőségi modell* épül fel.



8.2. ábra. *AS, FAM és a nyomonkövethetőségi modell kapcsolata*

8.3. Továbbfejlesztési irányok

Jelen megvalósítással egy egyirányú szinkronizációt lehet definiálni a modellek között, ezért hasznos továbbfejlesztési irány a *kétirányú szinkronizáció*, ahol a nézeti modellek változása is megfelelően módosítja a forrás modell(eke)t.

Másik fejlesztési irány az lehet, *EMF-IncQuery* nyelvben található *eval* kulcsszó továbbfejlesztése, hogy képes legyen bonyolultabb kifejezések kiértékelésére, amik segítségével bármilyen transzformációs szabályt definiálhassunk.

A metamodellt készítő szakemberek számára hasznos lenne a dinamikus metamodell építés fejlesztése származtatással, esetleg típusos referenciák és attribútumok meghatározásával, *EEnum* típusok definiálásával.

Irodalomjegyzék

[1] Simon M. Becker, Thomas Haase, and Bernhard Westfechtel. Model-based a-posteriori integration of engineering tools for incremental development processes. *Software Systems Modeling*, 4(2):123–140, 2005.

[2] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental evaluation of model queries over emf models. *Model Driven Engineering Languages and Systems, 13th International Conference.*, October 2010.

[3] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for emf models. *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland*, October 2011.

[4] Frank Budinsky. The Eclipse Modeling Framework: Moving into model-driven development. `http://www.ddj.com/184406198?pgno=1`.

[5] Márton Búr. Matlab-Simulink rendszerek modell-alapú validációja. *Scientific Students' Associations of BUTE*, October 2012.

[6] Cauê Clasen, Frédéric Jouault, and Jordi Cabot. Virtualemf: A model virtualization tool. *ER 2011 Workshops - 30th International Conference on Conceptual Modeling.*, October 2011.

[7] RTCA Radio Technical Commission for Aeronautic. Software Considerations in Airborne Systems and Equipment Certification (DO-178C). 2012.

[8] Charles L. Forgy. A network match routine for production systems. Working paper, 1974.

[9] Steve Easterbrook Shiva Nejati Nan Niu Mehrdad Sabetzadeh Greg Brunet, Marsha Chechik. A manifesto for model merging. *GaMMa'06*, May 2006.

[10] Object Management Group. Documents Associated With Meta Object Facility (MOF) 2.0 Query/View/Transformation, V1.1. `http://www.omg.org/spec/QVT/1.1/`.

[11] Alexander Königs Johannes Jakob and Andy Schürr. Non-materialized model view specication with triple graph grammars. *Lecture Notes in Computer Science. SPRINGER*, (321-335), 2006.

[12] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. An adaptable tgg interpreter for in-memory model transformation. *Proc. of the Fujaba Days*, pages 35–38, 2004.

[13] Ekkart Kindler and Robert Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Depart-

ment of Computer Science University of Paderborn, D-33098 Paderborn, Germany, June 2007.

[14] Budapest University of Technology and Fault Tolerant Systems Research Group. Economics. EMF-IncQuery. `http://incquery.net/`.

[15] Eclipse project. Ecore API. `http://download.eclipse.org/tools/emf/2.0.5/javadoc/org/eclipse/emf/ecore/package-summary.html`.

[16] István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations. *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009*, October 2009.

[17] Hui Song, Gang Huang, Franck Chauvel, Wei Zhang, Yanchun Sun, Weizhong Shao, and Hong Mei. Instant and Incremental QVT Transformation for Runtime Models. In *MODELS 2011*, Wellington, Nouvelle-Zélande, October 2011.

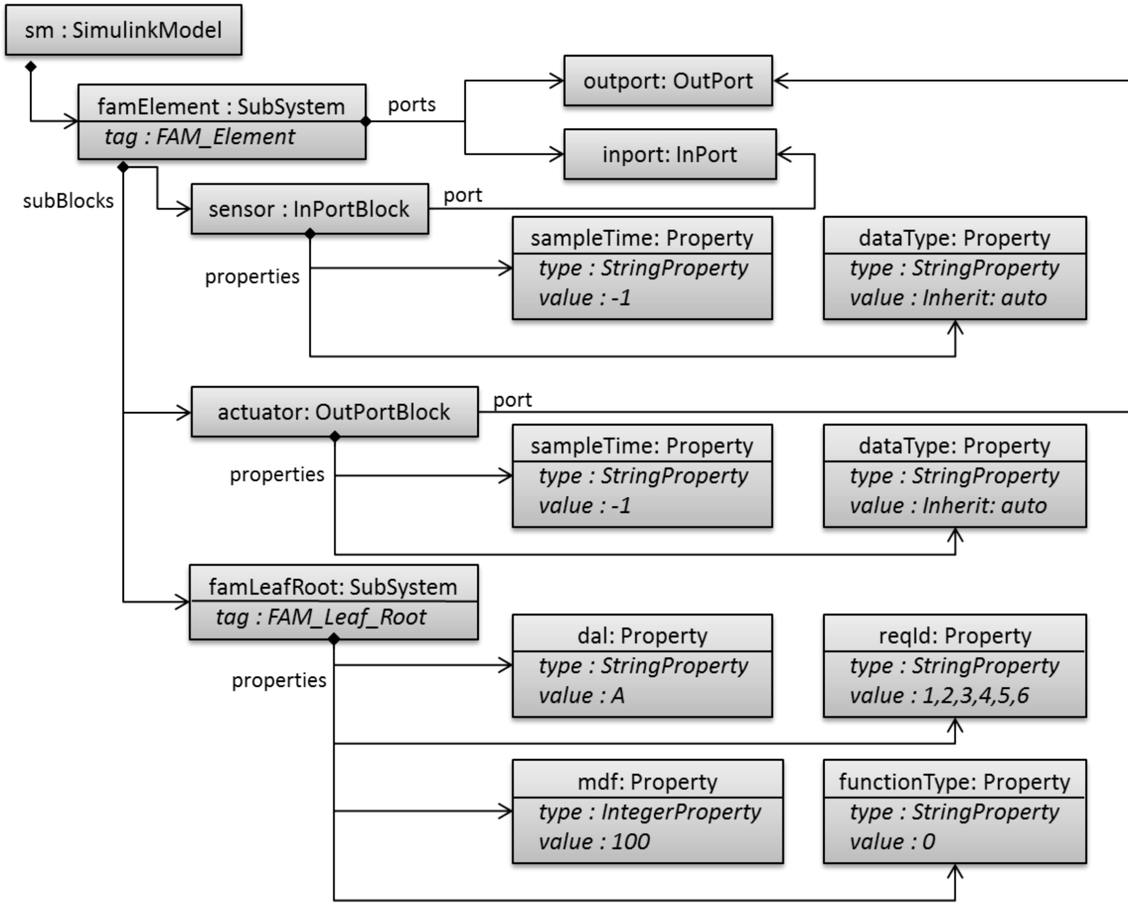
[18] Thomas Stahl and Markus Voelter. *Model-Driven Software Development: Technology, Engineering, Management (Wiley Software Patterns Series)*. John Wiley and Sons Ltd., 2006.

[19] Dániel Varró, István Ráth, and Ábel Hegedüs. Derived Features for EMF by Integrating Advanced Model Queries. *8th European Conference on Modelling Foundations and Applications.*, June 2012.

[20] Yinghui Wu Wenfei Fan, Xin Wang. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS).*, January 2013.

Függelék

F.1. Simulink minta modell (forrás modell)



F.2. Funkcionális minta modell (cél modell)