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

Lengyel Ádám

NÉZETI MODELLEK EGYSZERŰSÍTETT, KÉTIRÁNYÚ SZINKRONIZÁCIÓJA

KONZULESEK

Dr. Horváth Ákos
Semeráth Oszkár
Debreceni Csaba

BUDAPEST, 2015

Tartalomjegyzék

Összefoglaló

A modellvezérelt fejlesztés (*Model Driven Development - MDD*) alapgondolata, hogy a rendszerfejlesztés folyamatát egy precíz modellezési fázissal indítjuk el, amely során a rendszerrel szemben támasztott követelményeket magas szintű modellekkel írhatjuk le. Ezt követően ezen magas szintű modellekből kiindulva, különböző finomítási lépések alkalmazásával származtathatjuk a megvalósítás részleteit is tartalmazó platform specifikus modelleket, amelyből az utolsó lépésként automatikusan generálhatjuk a célalkalmazás dokumentációját, konfigurációját vagy akár a komponensek forráskódját is.

Azonban, ahogy a fejlesztési folyamat során több különböző részletességű modellel dolgozunk felmerül, hogy mind méretükből, mind pedig bonyolultságukból fakadóan áttekinthetetlenné és kezelhetetlenné válnak. Emiatt kiemelten fontos a modellek feletti egyszerűsített nézetek létrehozása. Ezen megközelítés segítségével, egy megfelelően definiált nézeten keresztül, a mérnökök könnyebben felismerhetnek, azonosíthatnak egy-egy hibát a rendszertervekben, és akár a hiba kezelését is egyszerűbben elvégezhetik. Azonban, az elvégzett módosítások visszavezetése az absztrakciókon nem minden esetben határozhatóak meg egyértelműen, ezért feloldásuk jellemzően összetett, logikai következtetést igényel.

A *TDK* dolgozat keretében olyan, nézetek definiálását és vizualizálását megvalósító keretrendszert mutatok be, amely képes a nézeti modelleken végzett módosítások félautomatikus, forrás oldali visszavezetésére. A megközelítésem lényege, hogy a lehetséges forrás modellbeli változtatások összegyűjtésére korszerű *SMT* következtetőket alkalmazok, amelyek képesek hatékonyan megoldani a visszavezetés során előálló logikai problémát, és ennek megoldásaiból már közvetlenül származtatható a kívánt forrás modell.

Az elkészített keretrendszer az iparban de facto szabványként tekinthető *Eclipse Modeling Framework*-re épül, míg a nézeti modellek megjelenítését a *Sirius* platform segítségével valósítottam meg. A leképzés során használt transzformációs szabályokat *EMF-IncQuery* gráfminták segítségével definiálhatjuk. A felmerülő logikai feladatokat pedig a *Z3* és az *Alloy SMT* keretrendszerekkel oldottam meg. Dolgozatomban a rendszer működését a *CONCERTO Európai Uniós Artemis* projekt keretében megvalósított, telecare rendszerek modellvezérlet fejlesztését támogató nézeti modelljein keresztül mutatom be.

Abstract

The main concept of the *Model Driven Development (MDD)* approach is that all design decisions are captured using graph based models (e.g., *UML* class diagram etc.) Usually, an *MDD* development process starts with a precise modeling phase, where system requirements are specified using high-level models, from these high-level models through precise refinement steps detailed implementation specific models are derived, which serve as the input for the final source code and configuration generation steps.

However, as the complexity of the systems under design grows, models used for describing their internal details are becoming too large and complex to be handled effectively. To ease their understanding, we have to be able to define specific views (e.g., security, dependability, communication paths, etc.) to filter out all unnecessary details from the underlying model as it is easier to recognize and prune out design flaws and specification errors using these domain-specific abstract views. However, calculating the effects of the manipulation done on the view back to the underlying model cannot be always unambiguously defined, and may require complex logical reasoning that can result in multiple solutions.

In this report, I present an approach, that supports the creation and visualization of abstract views over graph models with support for the semi-automated propagation of manipulations made on the view to the underlying source model. The main idea of my approach is to calculate the possible valid states of the source model which correspond to the modified view that can be described as a complex logical formula, which can be effectively solved using *SMT* (*Satisfiability modulo theories*) reasoners.

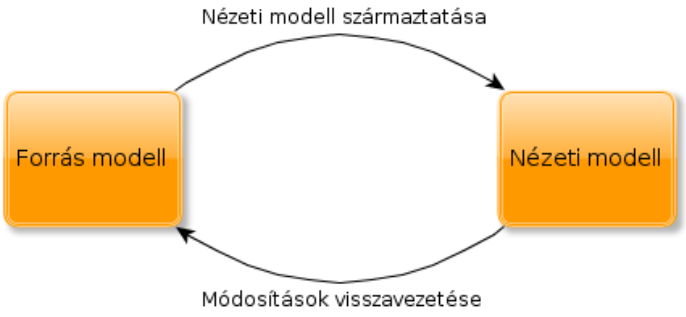
My approach is built upon the *Eclipse Modeling Framework*, considered as the de facto industry standard for modeling, while the visualisation of the views is created by the well-known Sirius platform. The rules that are used by the transformation can be defined by *EMF-IncQuery* graph patterns. Additionally, I used the *Z3* and the *Alloy SMT* solvers for the reasoning. Finally, the operation of the framework is demonstrated through an example from the *CONCERTO EU-Artemis* project that aims in providing model driven development techniques for designing telecare systems.

1. Bevezetés

Napjainkban egyre inkább alkalmaznak modellvezérelt fejlesztést (*Modell Driven Development - MDD*) különböző elosztott és / vagy beágyazott rendszerek tervezése esetén. A megközelítés alapgondolata, hogy a rendszerfejlesztés folyamatát egy precíz modellezési fázissal indítjuk el, amely során a rendszerrel szemben támasztott követelményeket magas szintű modellekkel írhatjuk le. Ezt követően ezen magas szintű modellekből kiindulva, különböző finomítási lépések alkalmazásával származtathatjuk a megvalósítás részleteit is tartalmazó platform specifikus modelleket, amelyből az utolsó lépésként automatikusan generálhatjuk a célalkalmazás dokumentációját, konfigurációját vagy akár a komponensek forráskódját is.

1.1. Problémafelvetés

Azonban, ahogy a fejlesztési folyamat során több különböző részletességű modellel dolgozunk felmerül, hogy ezek mind méretükből, mind pedig bonyolultságukból fakadóan áttekinthetetlenné és kezelhetetlenné válnak. Emiatt kiemelten fontos a modellek feletti egyszerűsített nézetek létrehozása (1.1 ábra). Ezen megközelítés segítségével, egy megfelelően definiált nézeten keresztül a mérnökök könnyebben felismerhetnek, azonosíthatnak egy-egy hibát a rendszertervekben, és akár a hiba kezelését is egyszerűbben elvégezhetik. Fontos ugyanakkor az is, hogy a definiált nézetek származtatása valamint karbantartása a modellek módosítása esetén automatikusan menjen végbe, mivel ezen feladatok manuális elvégzése igen bonyolult lehet, és számtalan hibalehetőséget rejthet magában. További problémát okoz, hogy nézeten elvégzett módosítások visszavezetése az absztrakciókon nem minden esetben határozhatóak meg egyértelműen, ezért feloldásuk jellemzően összetett feladat, melyre nem létezik általánosan jól használható megoldás.



1.1. ábra: Probléma szemléltetése

1.2. Célkitűzés

Kutatómunkám célja alapvetően az volt, hogy egy olyan keretrendszert tervezzek, mellyel lehetséges

- absztrakt nézeti modellek származtatása,
- a nézeti modellek automatikus karbantartása,
- valamint a nézeten végzett változtatások félautomatikus visszavezetése az eredeti (forrás) modellre.

A *TDK* dolgozatomban bemutatom a tervezett rendszer működésének fontosabb részleteit, kitérek a nézeti modell származtatásának módjára valamint az azzal kapcsolatos nehézségekre, továbbá ismertetem a visszavezetés során alkalmazott módszer elméleti hátterét illetve alkalmazásának korlátait.

A kidolgozott módszer segítségével lehetővé válik absztrakt nézetek definiálása, valamint – bizonyos korlátok mellett – a nézeten végzett módosítások visszavezetése a kiindulási modellre. Amennyiben több lehetséges megoldás is előáll a visszavezetés során, úgy a felhasználó döntésén múlik, hogy melyik kerül alkalmazásra. A megközelítés különösen nagy előnye abban mutatkozik meg, hogy csak kevés megkötést tesz a nézeti modellek származtatása során használható szabályokra, ezáltal nagy mozgásteret biztosít a transzformáció definiálása során.

1.3. A dolgozat felépítése

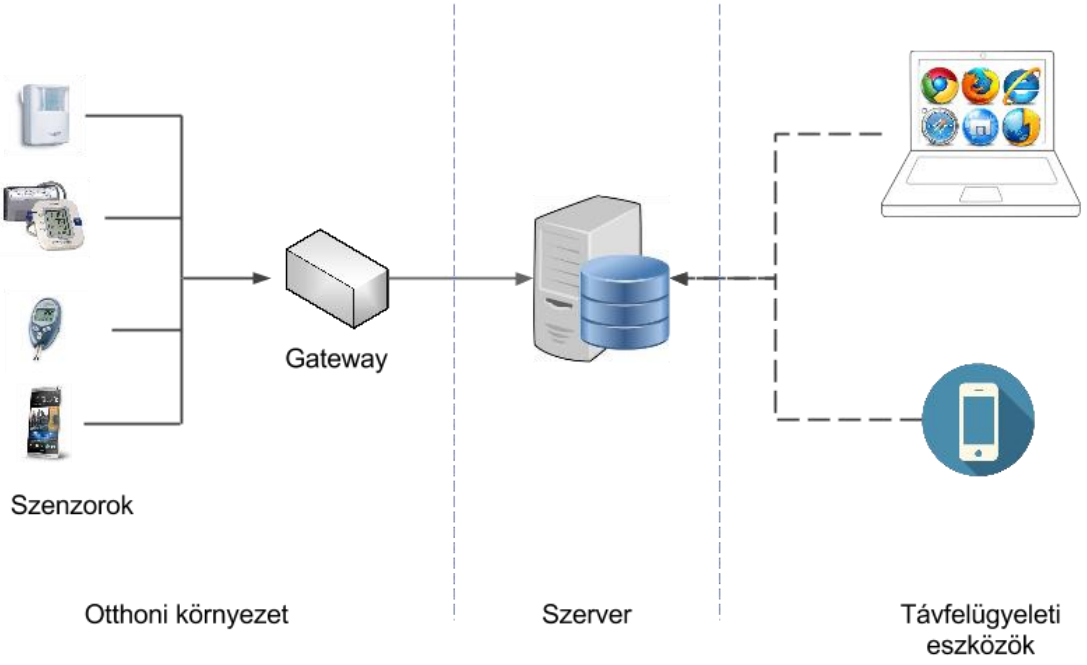
Dolgozatomban elsőként a munkám megértéséhez szükséges fontosabb ismereteket mutatom be kitérve a felhasznált technológiákra (2. fejezet). Ezt követően részletesen ismertetem a megoldandó feladatot (3. fejezet), majd bemutatom az általam kidolgozott megoldást (4. fejezet). Ebben a fejezetben először a nézeti modellek származtatásával kapcsolatos részletek kerülnek ismertetésre (4.1), majd ezt követően a módosítások visszavezetése során alkalmazott megközelítést tárgyalom (4.2). A(z) 5. fejezetben az elkészített rendszeren végzett mérések eredményeinek bemutatása következik, majd végül a(z) 7. fejezetben összegzem az elvégzett munkát és meghatározok néhány további fejlesztési irányt.

2. Háttérismeretek

Ebben a fejezetben első sorban a munkám megértéséhez szükséges ismereteket mutatom be. Elsőként az esettanulmányom ismertetése következik, majd bemutatom a munkám során felhasznált fontosabb technológiákat. Először ismertetem az *Eclipse Modeling Framework* modellező keretrendszert, valamint bemutatom az *EMF-IncQuery* gráfminta illesztő technológiát. Ezt követően kitérek a nézeti modellek származtatásának és használatának fontosabb kérdéseire, valamint bemutatom a feladat megvalósítása során használt logikai következtetőkkel kapcsolatos lényegesebb tudnivalókat. Végezetül pedig ismertetek egy általánosan használható grafikus nyelvek definiálására alkalmas eszközt (*Sirius*).

2.1. Távfelügyeleti rendszerek

Dolgozatomban a *CONCERTO Európai Uniós* projekthez[1] kapcsolódó példán keresztül fogom bemutatni a fontosabb eredményeket. A projekt célja egy egységes modell vezérelt megközelítés megvalósítása különböző kritikus rendszerek fejlesztéséhez. Ennek egyik lehetséges jövőbeli szakterülete a távfelügyeleti (úgynevezett *telecare*) *domain*. Ezen rendszerek általános felépítését mutatja az alábbi ábra.



2.1. ábra: Távfelügyeleti (*telecare*) rendszerek általános felépítése

Egy távfelügyeleti rendszer alapvetően három fő részre osztható fel:

1. O*tthoni környezet*: a felügyelet alatt álló személy otthona. Ide kerülnek telepítésre a távfelügyelet biztosításához szükséges eszközök.
    a. *Szenzorok*: egészségügyi valamint aktivitás adatok gyűjtésére használt eszközök (például vérnyomásmérő, vércukorszint mérő, mozgásérzékelő, okostelefon, stb.)
    b. *Gateway*: az adatok feldolgozásáért és továbbításáért felelős eszköz. Feladatai közé tartozik továbbá a felhasználó figyelmeztetése amennyiben egy-egy mérés esedékessé válik, valamint ez az eszköz végezheti a mért adatok előfeldolgozását illetve szűrését is.
2. *Szerver:* felelőssége az egyes felhasználóktól beérkező adatok összegyűjtése, feldolgozása illetve tárolása. A feldolgozás során a szerver képes kiértékelni a rendelkezésre álló információkat, észlelni a megszokott, normálisnak tekinthető értékektől való eltéréseket és ezek alapján meghatározni a felügyelt személy egészségi állapotát. Amennyiben ezen állapot azt indokolja, a szerver értesíti a felügyeletben részt vevő személyeket az esetleges problémáról. További feladata a szervernek a beérkezett adatok elérésének biztosítása az arra jogosult felhasználók számára.
3. *Távfelügyeleti eszközök*: segítségükkel lehetővé válik a felügyelt személyről gyűjtött adatok folyamatos nyomon követése. A rendszer által szolgáltatott információk többnyire böngésző segítségével vagy akár okostelefonon is elérhetőek.

2.2. Szakterület-specifikus nyelvek

Szakterület-specifikus nyelvek olyan, adott területhez (úgynevezett *domain*-hez) tartozó alapvető fogalmakat gyűjtik össze, melyek segítségével a kiválasztott területet érintő ismeretek / problémák megfogalmazhatók, felírhatók. Modellezés során ezen szakterület-specifikus nyelveket úgynevezett *metamodellek* segítségével definiálhatjuk, míg a nyelv használatával megfogalmazott információk az úgynevezett *példány modellben* kapnak helyet.

Az *Eclipse Modeling Framework (EMF)* egy *Eclipse*-alapú modellező keretrendszer, mely támogatást nyújt a modellvezérelt fejlesztéshez *(Model Driven Development - MDD)* különböző alapvető modellezési és kódgenerálási eszközök biztosításával.

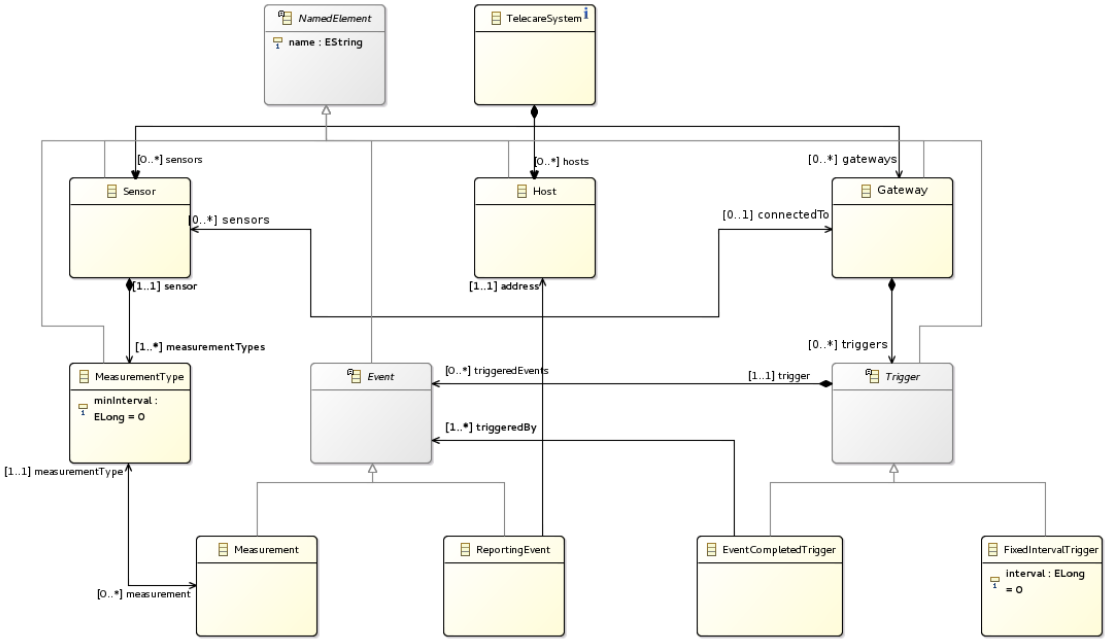
A keretrendszer segítségével definiálhatjuk egy adott szakterülethez tartozó *metamodellt,* melynek célja, hogy leírja az elkészítendő (*példány*) modell struktúráját (vagyis a modellezés során használt nyelvet). A metamodell definiálja a modellezés során előkerülő legfontosabb fogalmakat (az azokra jellemző tulajdonságokkal együtt), valamint a köztük fennálló kapcsolatokat. A metamodell alapján az *EMF* keretrendszer képes az őt reprezentáló *Java* osztályokat kigenerálni, melyeket már könnyen integrálhatunk bármilyen *Java* alapú rendszerhez. A metamodellek definiálásához az *EMF* az *ECore[2]* nyelvet biztosítja, amely az alábbi nyelvi konstrukciókat támogatja:

- ***Osztály (EClass)***: az adott terület egyes fogalmait reprezentálja. Az egyes osztályok definiálhatóak absztraktként, melynek következtében azok közvetlenül nem példányosíthatóak a példány modellben.
- ***Öröklés (Inheritance)***: segítségével az egyes osztályok között lehetőség van öröklési hierarchia definiálására.
- ***Attribútum (EAttribute)***: a modellben szereplő fogalmakat (*EClass*) jellemezhetjük további tulajdonságokkal. Az attribútumok típusa valamilyen egyszerű *Java* típus (például *Integer, String,* stb.) kell legyen.
- ***Referencia (EReference)***: a modellben szereplő fogalmak (*EClass*) közti kapcsolatok reprezentálására szolgálnak.
- ***Tartalmazás (Containment)***: *EMF* modellek esetén követelmény, hogy a modellben szereplő minden egyes elemnek (a gyökér elemet kivéve) kell legyen pontosan egy tartalmazó eleme, valamint az is, hogy a tartalmazási hierarchiában nem lehet kör. A tartalmazási hierarchia definiálására szolgálnak a *Containment* típusú referenciák.
- ***Típusok (EDataType):*** segítségével saját típust definiálhatunk a modellben, melyet használhatunk attribútumok típusaként (az *ECore* metamodellben szereplő *EString*, *EBoolean*, stb. típusok őse is az *EDataType* elem, lásd [2]).

A modellezés során az attribútumokra illetve referenciákra egységesen strukturális tulajdonságként (***EStructuralFeature***) hivatkozunk. Strukturális tulajdonságok esetén a fentieken túl definiálható azok multiplicitása is, melyet *[x..y]* formában adhatunk meg, ami azt jelenti, hogy minimum *x*, maximum *y* elem / érték található az adott referencián / attribútumon.

2.2.1. Telecare rendszerek modellezése (példa)

A következőkben bemutatom a *CONCERTO* projekt során definiált, távfelügyeleti rendszerek modellezését támogató metamodell egyszerűsített változatát (2.2 ábra). A példában első sorban a mérést illetve adatgyűjtést végző eszközök modellezésére törekedtem. Látható, hogy a bemutatott metamodell definiálja a távfelügyeleti rendszerek modellezéséhez szükséges főbb fogalmakat, kapcsolatokat és tulajdonságokat (vagyis a modellezési nyelvet), melyek segítségével leírható (megfogalmazható) ilyen rendszerek alapvető felépítése.



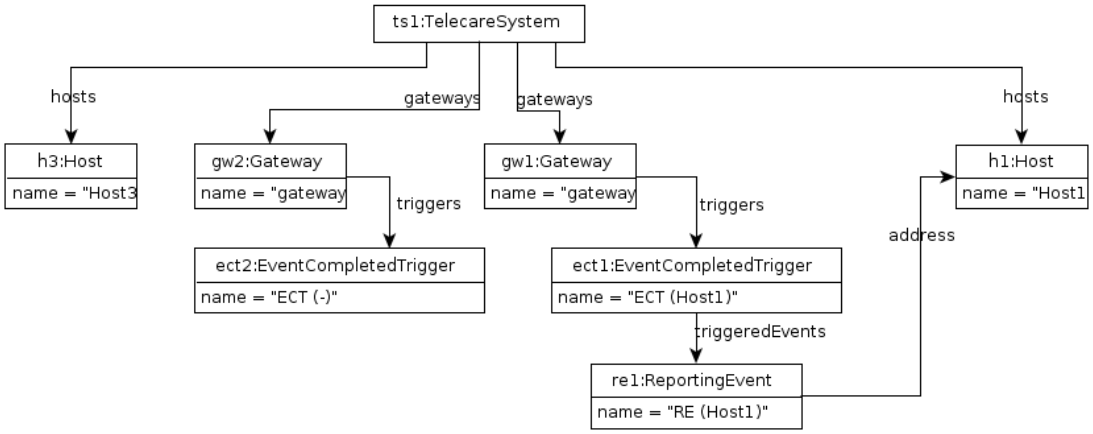
2.2. ábra: Telecare rendszer leírására szolgáló metamodell

A metamodell az alábbi főbb elemeket tartalmazza:

- ***TelecareSystem***: egy *távfelügyeleti* rendszert reprezentáló elem.
- ***NamedElement:*** absztrakt ősosztály név tulajdonsággal rendelkező elemekhez.
- ***Sensor***: az otthoni környezetben használt eszközök (pl. vérnyomásmérő, vércukorszint mérő, stb.) modellezésére szolgál.
- ***MeasurementType***: segítségével leírható, hogy egy-egy *Sensor* milyen típusú adatok mérésére képes (pl. vérnyomásmérő esetén ez a *systolé*, *diasztolé* és a pulzus). Minden *Sensor* elem rendelkezik egy *minInterval* tulajdonsággal, mely azt határozza meg, hogy egy adott mérést legalább milyen időközönként kell elvégezni.

- ***Host***: a rendszerhez tartozó adatgyűjtést végző szervereket reprezentáló elem.
- ***Gateway:*** az otthoni környezet adatgyűjtő egységét reprezentáló elem (lásd 2.1).
- ***Trigger***: absztrakt ősosztály *Trigger*-típusú elemek számára. A *triggeredEvents* referencián a trigger által kiváltott események találhatóak meg. Két típusa létezik: *FixedIntervalTrigger* valamint *EventCompletedTrigger*.
- ***FixedIntervalTrigger***: segítségével olyan akciók modellezhetőek, melyek bizonyos időközönként kerülnek végrehajtásra és hatásukra valamilyen esemény következik be (például tízpercenként, óránként, naponta, stb.). Az elemhez tartozik az *interval* attribútum, mellyel megadható, hogy milyen időközönként kerüljön végrehajtásra az adott akció.
- ***EventCompletedTrigger:*** segítségével olyan akciók modellezhetők, melyeket bizonyos események (pl. egy vérnyomás mérés) bekövetkezése vált ki. A kiváltó események a *triggeredBy* referencián találhatóak meg.
- ***Event***: absztrakt ősosztály különböző típusú események modellezésére. Két típusa létezik: *Measurement* valamint *ReportingEvent.*
- ***Measurement***: segítségével mérési események modellezhetőek. A *measurementType* referenciáján keresztül hozzá tudjuk rendelni a mérés során mért tulajdonságot (pl. pulzus, *systolé,* stb.).
- ***ReportingEvent***: mérési eredmények feltöltését reprezentálja a megadott *Host*-ra (*address* referencia).

A(z) 2.3 ábra az F.1 függelékben szereplő, az előbbiekben ismertetett metamodellhez tartozó egy lehetséges példány modell egy részét ábrázolja (az ábra csak a példány modell szemléltetését szolgálja, így az átláthatóság érdekében nem tartalmaz minden elemet).

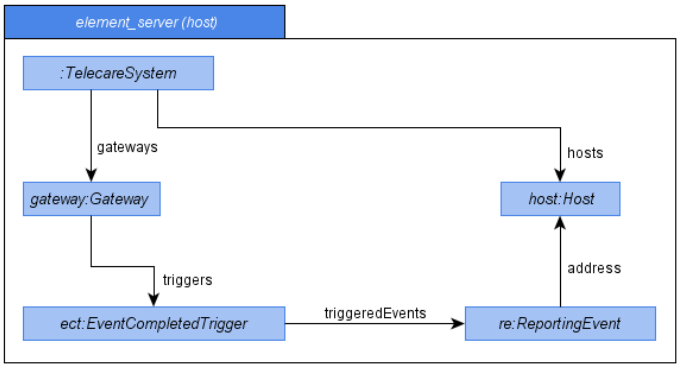


2.3. ábra: *Telecare* rendszer példánymodellje (részlet)

2.3. Gráfminta illesztés

A modellezés során létrehozott példány modellekre tekinthetünk olyan típusos gráfként, melynek egyes csomópontjai a modell elemeit, a csomópontok közötti élek pedig az elemek közti kapcsolatokat írják le (lásd 2.3).

A modell gráfként történő reprezentációja lehetővé teszi *gráfminta* alapú lekérdezések definiálását és végrehajtását ilyen modellek fölött. Gráfminták *(graph pattern)* segítségével egy adott modell struktúrájával és egyéb tulajdonságaival kapcsolatos megkötéseket *(constraint)* definiálhatunk. Egy minta abban az esetben illeszkedik *(match)* egy objektum halmazra, ha az illeszkedés minden, a mintában szereplő kényszert kielégít. A(z) 2.4 ábra egy példát mutat a(z) 2.2 ábrán bemutatott metamodell felett definiált gráfmintára, mely olyan *Host* elemekre fog illeszkedni, melyekre ugyan abban a távfelügyeleti rendszerben lévő eszközről adatokat küldünk fel. Az *element_server* minta megköveteli, hogy a gráfban létezzen egy *TelecareSystem* típusú elem, melynek *gateways* referenciáján található egy *Gateway* elem. Megkötés továbbá, hogy ezen elem *triggers* referenciáján keresztül el lehet jutni egy *EventCompletedTrigger típusú* elemhez, majd abból a *triggeredEvents* referencián keresztül elérhető egy *ReportingEvent* típusú elem. A minta végül azon *Host* elemre illeszkedik (*host* paraméter), mely az előbbi *ReportingEvent* elem *address* referenciáján található (ebből következik tehát az a további megkötés, hogy kell létezzen *Host* elem az *address* referencián). A példát megvizsgálva láthatjuk azt is, hogy egy-egy minta abban az esetben illeszkedik egy modellre, ha a minta által meghatározott gráfot (2.4 ábra) a vizsgált modell részgráfként tartalmazza (ez a tartalmazás akár többször is létezhet, ebben az esetben a mintának több illeszkedése van).



2.4. ábra: Az *element_server* gráfminta grafikus ábrázolása

A(z) 2.3 ábrán látható példa esetében az *element_server* mintának pontosan egy illeszkedése lesz, mégpedig a *h1 Host* típusú elem.

2.3.1. EMF-IncQuery

Az *EMF-IncQuery[3]* a *Budapesti Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek Tanszékén* kifejlesztett *Eclipse* alapú keretrendszer, melynek segítségével lehetőségünk van *EMF* modellek feletti lekérdezések definiálására deklaratív módon, gráfminták segítségével.

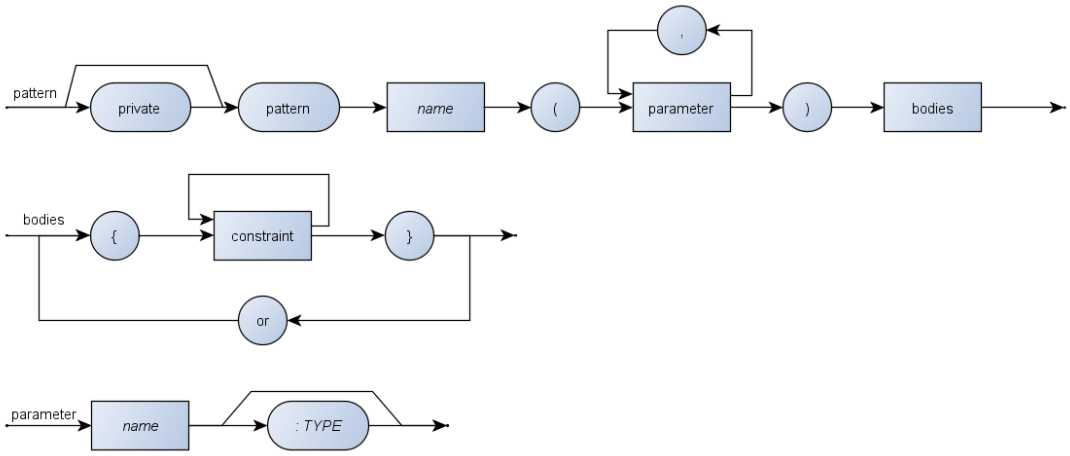
Az eszköz fejlett szerkesztő felületet biztosít *EMF* modellek feletti gráfminták definiálására. Támogatja a lekérdezések inkrementális illesztését, melynek köszönhetően a vizsgált modell módosítása után csak a módosítás miatt érintett lekérdezések újbóli kiértékelését végzi el, ezzel rendkívül gyors működést téve lehetővé (lásd 5.1).

2.3.1.1. IncQuery pattern language

A következőkben az *IncQuery* eszköz saját mintaleíró nyelvét (*IncQuery pattern language - IQPL*) fogom bemutatni a munkám szempontjából fontosabb részekre koncentrálva. Minta definiálásához minden esetben szükség van azon metamodellre / metamodellekre, melyek fölött a mintát definiáljuk. Egy-egy metamodell a hozzá tartozó egyedi azonosító alapján (*namespace URI*) az *import* kulcsszó használatával hivatkozható be a minta definiálása során (ezt mutatja a lenti kódrészlet).

```
import „http://hu.bme.mit.inf.concerto/telecare/example/tdk/1.0”
```

Egy minta definiálása az alábbi szintaxis gráffal leírt szabályok szerint kell történjen. A gráfban szereplő *name* elem helyére a minta neve kell kerüljön, míg a *TYPE* elem helyére az importált metamodellekben szereplő *EClass* vagy *EDataType* elemek helyettesíthetőek be típusként.



2.5. ábra: IncQuery gráfminta felépítése

A bemutatott szabályok alapján egy minta definíciója például az alábbiak szerint néz ki:

```
pattern exampleName(a : ElementA, b) {
	/* Kényszerek */
} or {
	/* Kényszerek */
}
```

A(z) 2.5 ábrán látható nyelvi szabályoknak megfelelően lehetséges olyan mintát definiálni, melynek több, egymással *vagy* kapcsolatban álló törzse van. Ilyen esetben *a minta bármelyik törzsében definiált kényszerhalmaz illeszkedése (akár egyszerre több is) esetén illeszkedik az adott modellre*.

Az alábbi példa a(z) 2.5 ábrán bemutatott mintát írja le az *IncQuery* mintaleíró nyelvet használva. A példában található kényszerek a modellben szereplő elemek típusára, valamint az egyes elemek közti kapcsolatokra vonatkoznak.

```
10 pattern element_server(host : Host) {
11     TelecareSystem.gateways(_, gateway);
12
13     Gateway.triggers(gateway, ect);
14
15     EventCompletedTrigger(ect);
16     EventCompletedTrigger.triggeredEvents(ect, re);
17     ReportingEvent.address(re, host);
18 }
```

2.6. ábra: Az *element_server* minta *IQPL* leírása

A fenti kényszerek általános módon az alábbiak szerint fogalmazhatóak meg:

Típuskényszer: definiálása ***Type*** *(variable)* alakban történik. Az ilyen típusú kényszerekkel megköveteljük, hogy a modellben kell létezzen legalább egy *Type* típusú elem

(vagy annak leszármazottja; természetesen ebben az esetben is a típus lehet *EClass* vagy *EDataType* egyaránt). Amennyiben létezik ilyen elem, azt a *variable* változóba kerül behelyettesítésre. A példában a 15. sorban szereplő megkötés ilyen, melyet a(z) 2.3 ábrán bemutatott modell esetén például az {ect1} elem kielégít.

Útvonalkényszer: létezik referenciákra és attribútumokra vonatkozó útvonalkényszer is.

- *referenciákra vonatkozó útvonalkényszer:* definiálása ***Type.referenceName****(var1, var2)* alakban történik. Az ilyen típusú kényszerekkel megköveteljük, hogy a modellben kell létezzen *Type* típusú elem, valamint ennek az elemnek *referenceName* referenciáján található legalább egy elem. Amennyiben létezik a kényszernek megfelelő struktúra a modellben, úgy az a *var1 és var2* változókba kerül behelyettesítésre. A példában a 13. sorban található kényszer ilyen, melyet a(z) 2.3 ábrán bemutatott modell esetén például a {gw1, ect1} pár kielégít. *Megjegyzés: a referenciákra vonatkozó megkötések láncolhatók a következő minta szerint (amennyiben az a metamodell szerint megengedett):* ***Type.ref$_1$.ref$_2$…ref$_N$****(var1, var2);*
- *attribútumra vonatkozó útvonalkényszer:* definiálása ***Type.attributeName****(var1, var2)* alakban történik. Az ilyen típusú kényszerekkel azt követeljük meg, hogy a modellben kell létezzen *Type* típusú elem, valamint annak *attributeName* attribútumán található valamilyen érték. Amennyiben létezik a kényszernek megfelelő struktúra a modellben, úgy az a *var1 és var2* változókba kerül behelyettesítésre, utóbbiban az attribútum értéke lesz.

A fentieken túl általánosan elmondható az is, hogy amennyiben ugyan azt a változót használjuk egy-egy kényszerben, úgy ezen kényszerek mindegyikének meg kell feleljen a közös változóba behelyettesített elem. Ahogy az egyes változók esetében, úgy a teljes mintára is igaz, hogy a minta csak abban az esetben fog illeszkedni egy adott modellre, ha létezik annak minden kényszerét kielégítő változó behelyettesítés (vagyis létezik olyan részgráf az eredeti modellben, melynek egyes csomópontjai (és attribútum értékei) a megfelelő változókba behelyettesíthetők úgy, hogy azok a megadott kényszereknek eleget tesznek).

Megjegyzés: speciális, úgynevezett anonim változónak tekinthetőek a nyelvben azok, melyeknek neve „_” karakterrel kezdődik. Ezek a változók csak azokban a kényszerekben

érvényesek, amelyekben használjuk őket (tehát hiába egyezik meg két ilyen típusú változó neve, a behelyettesítésük eltérő lehet).

Egyenlőségi kényszer: a mintán belül bevezetett változók nem csak a fent ismertetett módon használhatóak, mivel lehetőségünk van további, a változókra vonatkozó kényszerek definiálására az == és a *!*= operátorok segítségével. Amennyiben az operátor két oldalán szereplő operandusok *EClass* példányok, úgy referenciális egyenlőséget (vagy egyenlőtlenséget) vizsgálunk, míg *EDataType* leszármazottak esetén az implementáló osztály *equals* metódusa alapján történik a vizsgálat.

Mintahivatkozás: a fentieken túl a nyelv (hasonlóan a programozási nyelvekben megszokott függvényhívásokhoz) lehetővé teszi más minták hívását egy minta törzsén belül (*pattern call*), melyhez a ***find*** kulcsszó használható. Erre láthatunk példát a(z) 2.7 ábrán.

```
47 pattern element_trigger(trigger : FixedIntervalTrigger) {
48     FixedIntervalTrigger.triggeredEvents(trigger, measurement);
49     Measurement(measurement);
50
51     find eventCompletedTriggerWithMeasurement(_, measurement);
52 }
53
54 private pattern eventCompletedTriggerWithMeasurement(ect : EventCompletedTrigger, m : Measurement) {
55     EventCompletedTrigger.triggeredBy(ect, m);
56     EventCompletedTrigger.triggeredEvents(ect, re);
57
58     ReportingEvent(re);
59 }
```

2.7. ábra: Az *element_trigger* minta (find kulcsszó használata)

Általánosan elmondható, hogy egy ilyen minta abban az esetben fog illeszkedni, ha az összes, a minta törzsében szereplő kényszer kielégíthető úgy, hogy a hívott mintá(k)nak is van illeszkedése (a változókra vonatkozó korábban ismertetett szabályok mintáknak átadott paraméterek esetén is igazak). Az *element_trigger* minta a(z) F.1 függelékben bemutatott modell esetén illeszkedni fog a {fit_5}, {fit_10}, {fit_1440} elemekre, viszont nem fog illeszkedni a {fit_180} elemre.

- *Negatív mintahivatkozás:* a nyelv lehetőséget biztosít minta hívása esetén a hívott minta negálására (*negative pattern call*). Ehhez a ***neg find*** *pattern_name(...)* hívást kell kiadni egy minta törzsén belül. Az ilyen módon meghívott minta abban az esetben fog illeszkedni, ha van olyan kényszere, mely nem elégíthető ki. Ilyen hívások esetén a hívás kontextusától függően eltérő viselkedés várható. A(z) 2.7 ábrán bemutatott példa illeszkedése negatív hívás esetén az átadott változók függvényében az alábbiak szerint alakulna:

- (_, measurement) változók esetén: olyan *FixedIntervalTrigger* elemekre illeszkedne a minta, melynek *triggeredEvents* referenciáján van *Measurement* elem; ugyanakkor ez az elem nem lehet olyan *EventCompletedTrigger* elem *triggeredBy* referenciáján, melynek *triggeredEvents* referenciáján létezik *ReportingEvent* elem.

- *Tranzitív lezárt:* minta hívással kapcsolatos további funkció a tranzitív lezárt meghatározása (*transitive closure*). Ezzel a funkcióval lehetséges egy meghatározott referencián keresztüli elérhetőségi vizsgálat elvégzése a modellben két elem között. Ehhez természetesen metamodell szinten léteznie kell egy megfelelő referenciának, mely mentén a tranzitív lezárt meghatározható, valamint definiálni kell egy speciális, két paraméterű mintát, mely ezen referencia bejárását végzi. Tranzitív lezárt meghatározásához az alábbi kódrészletnek megfelelően kell a bejárást végző mintát meghívni (a + operátor hatására történik a tranzitív lezárt számítása).

```
find calculate_transitive_closure+(v1, v2);
```

- *Illeszkedés számosság*: további lehetőség a minta hívások esetén, hogy a ***count*** kulcsszó használatával összeszámolhatjuk a hívott minta különböző illeszkedéseit, valamint arra további kényszereket definiálhatunk. A(z) 2.8 ábrán a ***count*** kulcsszó használatát bemutató minta látható. Az *at_least_three_sensor* minta pontosan akkor fog illeszkedni, ha a vizsgált modellben legalább háromszor illeszkedik az *element_sensor* minta.

```
68 private pattern at_least_three_sensor(N) {
69     N == count find element_sensor(_);
70     check (N >= 3);
71 }
```

2.8. ábra: Minta a *count* kulcsszó használatára

*Megjegyzés: a **check** kulcsszó segítségével olyan kényszer írható fel, mely a zárójelek között található logikai kifejezés igaz értéke esetén van kielégítve. A **check** kulcsszó mellett létezik az **eval()** utasítás is, mellyel a zárójelek közé írt Java kifejezés kiértékelése végezhető el. A példában szereplő **check** kényszer a következőképp írható át **eval**-t használó kényszerre: true == eval (N >= 3);*

Végezetül pedig fontos kitérnem arra a speciális esetre, amikor egy minta definíciójában szereplő változók között van olyan, amelyik nincs kivezetve a minta paraméterei közé (erre jó példa a(z) 2.7 ábrán bemutatott *element_trigger* minta, mivel ennek *measurement* változója ilyen). Az ilyen változókat *rejtett változóknak* nevezzük, a minta

illeszkedése során a változóba behelyettesített elemet pedig *rejtett elemnek* (ezek alapján a minta paraméterlistájára kivezetett változókat *publikus változóknak,* az azokba behelyettesített elemeket pedig *publikus elemeknek* nevezzük). Az F.1 függelékben található példát megvizsgálva látható, hogy az {fit_10} elemhez négy *Measurement* elem is tartozik a *triggeredEvents* referencián, melyek mindegyikére illeszkedik a hívott *eventCompletedTriggerWithMeasurement* minta is. Ez tehát azt jelenti, hogy a példány modellben pontosan négy olyan illeszkedése van a mintának (négy, a minta definíciójának megfelelő részgráf található a példány modellben), melyben szerepel a {fit_10} elem. Az *IncQuery* keretrendszer viszont az eredményhalmazt vetíti a minta paraméterlistája szerint, *így csak az ott felsorolt változók szerint vett különböző illeszkedések jönnek létre*. Ennek következtében (a példánál maradva) habár négy olyan részgráf van a modellben, melyre illeszkedik a minta, mégis csak egyetlen találatot kapunk (a {fit_10} elemre vonatkozóan).

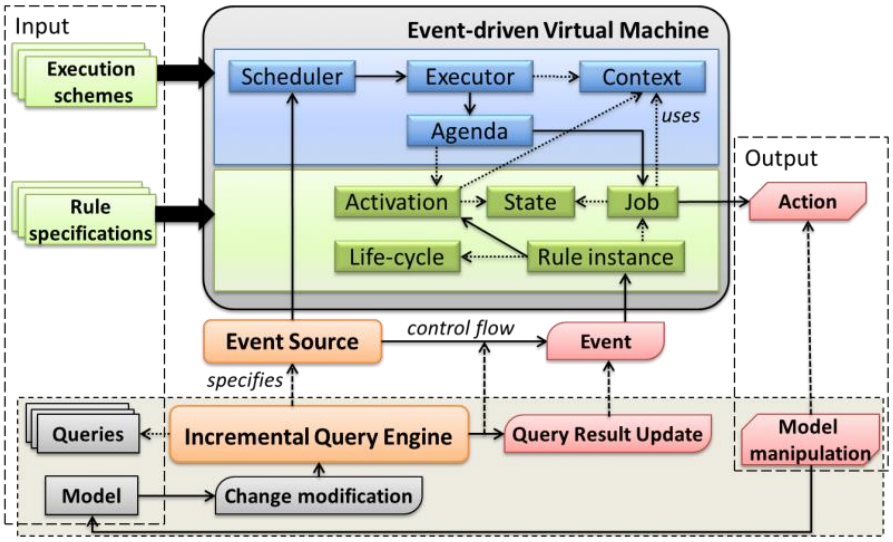
2.3.1.2. IncQuery EVM

Az *IncQuery EVM[4]* (*Event-driven Virtual Machine*) egy olyan reaktív keretrendszer, mely képes előre definiált feladatok (*job*) prioritásos végrehajtására meghatározott (minták illeszkedésével összefüggő) események bekövetkezése esetén. A következőkben röviden bemutatom az eszköz főbb komponenseit valamint azok működését, első sorban azokra a részekre koncentrálva, melyek ismerete a munkám megértéséhez elengedhetetlen. Tekintettel a rendszer bonyolultságára további részletek ismertetésével nem foglalkozom.

A(z) 2.9 ábra a rendszer architektúráját mutatja. Az *EVM*-ben alapvetően szabályok (*rule*) segítségével definiálhatjuk az elvárt viselkedést. Egy szabály segítségével meghatározható, hogy a rendszer egy adott *IncQuery* mintával (*query*) kapcsolatban bekövetkezett esemény (*event*) hatására milyen feladatot hajtson végre (*job*). Az eszköz három féle eseményt különböztet meg:

- *Illeszkedés megjelenéséről (match appeared)* akkor beszélhetünk, ha a modell változásának hatására olyan új illeszkedése van egy mintának, mely a változás előtt nem létezett.
- *Illeszkedés megszűnése (match disappeared)* abban az esetben következhet be, ha egy létező illeszkedés a modell változtatásának hatására érvénytelenné válik.

- *Illeszkedés frissülése (match updated)* abban az esetben történik, ha egy meglévő illeszkedésben lévő (paraméterek közé kivezetett) tetszőleges elem egy tulajdonsága megváltozik úgy, hogy ez a változás nem befolyásolja az illeszkedést.



2.9. ábra: Az *IncQuery EVM* architektúrája[5]

Másik fontos bemenő paramétere az eszköznek az úgynevezett *Execution schema.* Ezen paraméter segítségével az szabályozható, hogy az *EVM* milyen események hatására futtassa a végrehajtásra váró feladatokat (*enabled activation*). Lehetséges például a modell minden egyes módosítását követően végrehajtani az engedélyezett aktivációkat, ugyanakkor a végrehajtás kiváltó eseménye lehet *EMF* tranzakciók[6] sikeres befejeződése is. Az aktivációk végrehajtásának ütemezéséért felelős komponens (*scheduler*) további feladata az is, hogy az egyes engedélyezett aktivációkat a szabályokban definiált prioritás szerinti sorrendben hajtsa végre. Ennek a funkciónak köszönhetően lehetséges a különböző szabályokhoz tartozó feladatok végrehajtásának sorrendezése.

A fentieken túl elmondható az is, hogy az *EVM* az *IncQuery* keretrendszer által biztosított inkrementális mintaillesztő szolgáltatásra (*IncQuery Engine*) épít, így a minták illesztésével, az illeszkedések inkrementális karbantartásával, stb. kapcsolatos feladatokért kizárólag az *IncQuery Engine* felelős.

2.4. Logikai következtetők használata

Az *elsőrendű logika* a matematikában és informatikában is széleskörűen használt formális nyelv. A nyelv típusokból, függvényekből, állandókból és relációkból épül fel, valamint az ezekből alkotott logikai kifejezésekből. A logikai kifejezésekből állításokat fogalmazhatunk meg (axiómák). Olyan kiértékelését keressük ezeknek a relációknak,

amely minden axiómát teljesít. Amennyiben létezik ilyen megoldás, azt modellnek nevezzük, amennyiben ilyen nincs, úgy azt mondjuk, hogy ellentmondásos az axiómarendszer.

Kielégíthetőségi problémák esetén (*Satisfiability Problem - SAT*) azt vizsgáljuk, hogy egy adott logikai formulának létezik-e olyan változó-behelyettesítése, mely esetén a formula igaz lesz. *SAT* problémák definiálása során csak logikai változók, NEGÁCIÓ valamint logikai ÉS (AND) illetve VAGY (OR) műveletek használhatók, kvantorok és egyéb típusok nem.

Kényszerkielégítési problémák (*Constraint satisfaction problem - CSP*) esetén alapvetően azt vizsgáljuk, hogy a formulában megadott változóknak létezik-e olyan behelyettesítése, mely minden definiált kényszernek eleget tesz. Ilyen problémák esetén definiálni kell a változók, valamint a rájuk vonatkozó kényszerek halmazát, továbbá meg kell adni az egyes változókhoz a felvehető értékek véges tartományát.

Munkám során használt *Alloy[7]* eszköz egy olyan speciális nyelvet biztosít, mellyel támogatást nyújt gráf problémák megfogalmazására. Ezeket a problémákat korlátos tartományban (*scope)* próbálja megoldani (például olyan, legfeljebb 10 elemű gráfot keresve, mely megfelel a kritériumoknak).

Ismert problémakörök és megoldási módszerek gyűjteménye az *SMT* (*Satisfiability modulo theories*). *SMT* megoldók használata során megfogalmazhatóak gráf problémák is tetszőleges méretben, azonban arra nincsen garancia, hogy az sikeresen lefut. Korszerű *SMT* megoldó például a *Z3[8]* eszköz.

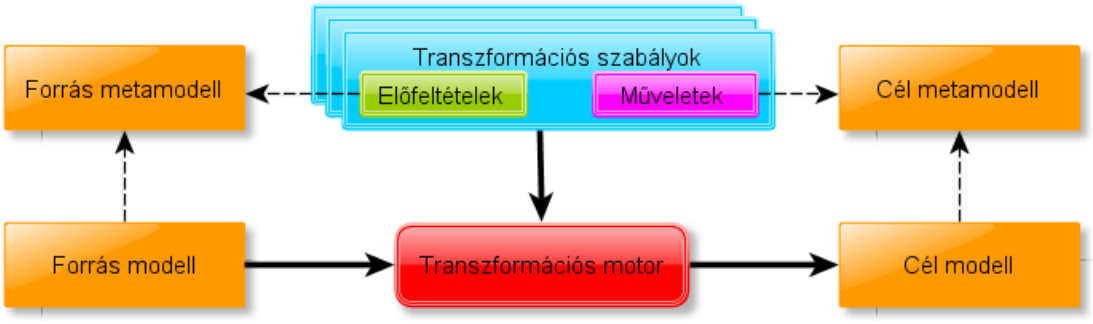
2.5. Modell transzformáció és nézeti modellek

Modellvezérelt fejlesztés során gyakran több, különböző részletességű modellel kell dolgozunk, így gyakran előforduló probléma, hogy ezek mind méretükből, mind pedig bonyolultságukból fakadóan áttekinthetetlenné, kezelhetetlenné válnak. Emiatt kiemelten fontos a modellek feletti egyszerűsített nézetek létrehozása. Ezen megközelítés segítségével egy megfelelően definiált nézeten keresztül az egyes hibák könnyebben felismerhetőek, azonosíthatóak, továbbá a hiba kezelése is gyakran egyszerűbben elvégezhető. Azonban, az elvégzett módosítások visszavezetése az absztrakciókon nem minden esetben határozhatóak meg egyértelműen, ezért feloldásuk jellemzően összetett, logikai következtetést igénylő feladat.

Nézeti modellek származtatása alapvetően modelltranszformációs feladatnak[9] tekinthető. A(z) 2.10 ábra modelltranszformációs rendszerek általános felépítését mutatja. Az ábra alapján látható, hogy a művelet bemeneteként szolgál a forrás modell, valamint a transzformációs szabályok, míg a folyamat eredménye (kimenete) a cél modell lesz.

A transzformáció definiálása szabályok segítségével történik: minden szabályhoz tartoznak bizonyos előfeltételeket, valamint műveletek. Az előfeltételek olyan (a forrás modellhez tartozó metamodell felett definiált) kényszerek, melyek teljesülése szükséges a szabály végrehajtásához. A műveletek olyan (a cél modellhez tartozó metamodell felett definiált) lépések, melyek a cél modell módosítását végzik. Egy szabály aktivációja azt jelenti, hogy a szabályhoz tartózó előfeltételek teljesülnek, így a definiált műveletek végrehajthatók. A műveletek konkrét végrehajtását a szabály tüzelésének is nevezzük.

A transzformáció során a transzformációs motor felelőssége az egyes szabályok előfeltételeinek vizsgálata, valamint az aktivált szabályok meghatározott sorrendben történő eltüzelése. A transzformációs motor további feladata lehet, hogy nyomon kövesse az egyes tüzelések hatását a cél modellben, és azokat összekapcsolja az előfeltételeket kielégítő forrás modellbeli részgráffal (ezek az információk az úgynevezett *traceability* modellben kerülnek tárolásra). Ezen információk felhasználásával lehetővé válik inkrementális modelltranszformációs eszköz készítése is, mivel így pontosan meghatározható, hogy egy-egy forrás modellbeli elem változása milyen cél modellbeli elemek módosítását (vagy törlését) vonhatja maga után.



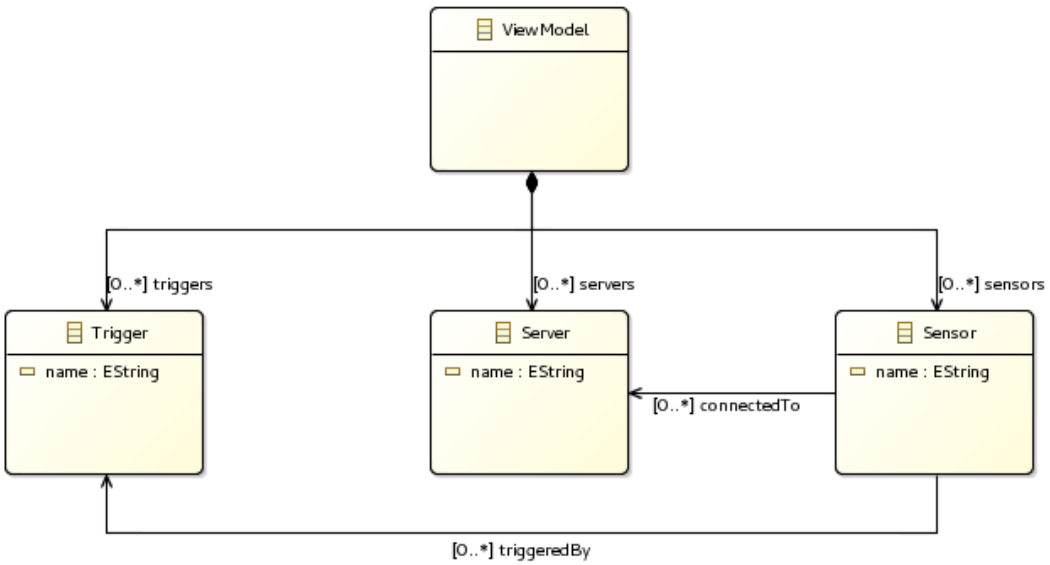
2.10. ábra: Modelltranszformáció (általánosan)

Inkrementalitás szempontjából az alábbi kategóriákat különböztetjük meg modelltranszformációk esetén:

- *nem inkrementális (batch transformations)*: ebben az esetben a forrás modell(ek) változása a teljes transzformáció újbóli végrehajtását vonja maga után. Előnye, hogy egyszerű implementálni, ugyanakkor működését tekintve nem hatékony.
- *részben inkrementális (dirty incrementality):* amennyiben a forrás modellek valamelyike megváltozik, a hozzá kapcsolódó transzformációs folyamat újból végrehajtásra kerül.
- *traceability-n alapuló (incrementality by traceability):* az inkrementális transzformáció a *traceability* modellben tárolt információk alapján biztosított. Amennyiben léteznek olyan forrás modellbeli elemek, melyekhez nem tartozik információ a *traceability* modellben, akkor ezekre az elemekre a transzformáció végrehajtásra kerül.
- *esemény vezérelt (event driven transformations):* a forrás modell(ek) módosításának hatására a transzformációs motor újra ellenőrzi az egyes szabályok előfeltételeinek érvényességét, és amennyiben léteznek aktivált szabályok, azok tüzelését is elvégzi. Ezzel a megközelítéssel biztosított, hogy csak azok a szabályok tüzelnek a módosítás hatására, melyek újbóli végrehajtása ténylegesen indokolt és lehetséges.

2.5.1. Nézeti modell származtatása (példa)

A következőkben a dolgozatomban ismertetett esettanulmányon keresztül szemléltetem a modelltranszformáció működését néhány egyszerűbb példa bemutatásával. Mivel nézetek származtatása modelltranszformációnak tekinthető, így olyan leképzés definiálása a feladat, melynek cél modellje a kívánt nézetnek felel meg. A korábban leírtaknak megfelelően a transzformációhoz szükség van mind a forrás modell, mind pedig a cél (nézeti) modell struktúráját definiáló metamodellre. A forrás metamodell legyen a 2.2 ábrán, a cél metamodell pedig a(z) 2.11 ábrán bemutatott.



2.11. ábra: *TeleCare* rendszerekhez definiált nézet metamodellje

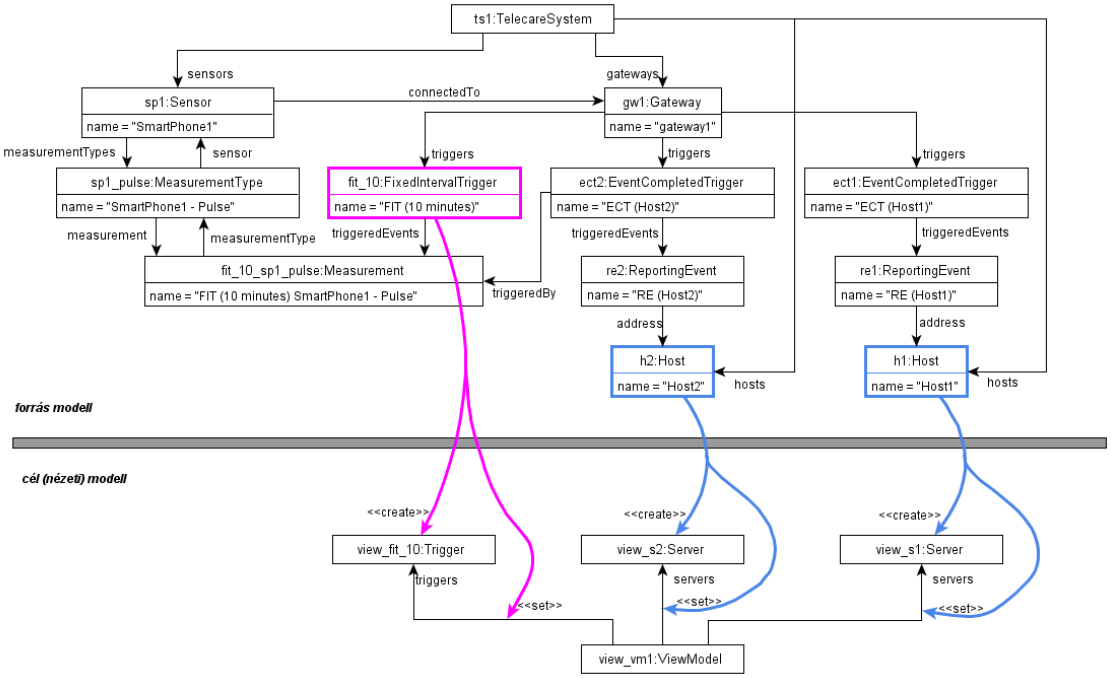
Az ábrán látható metamodell a korábban ismertetettnek egy lényegesen egyszerűbb változata, melynek köszönhetően a forrás modellben található bizonyos információk könnyebben megtalálhatóak, felismerhetőek a létrehozott nézet segítségével. Erre lehet példa a *Sensor* és *Server* elem közti kapcsolatot reprezentáló *connectedTo* referencia, melynek felismerése az eredeti modell alapján jelentősen nehezebb.

Definiáljuk úgy a transzformációs szabályokat, hogy azok előfeltételei egy-egy gráfmintával legyenek megadva. Amennyiben a mintának van illeszkedése a forrás modellen, úgy a megfelelő szabály aktívvá (tüzelhetővé) válik annyi alkalommal, ahány illeszkedése van az előfeltételként megadott mintának. Ebben az esetben a nézetben szereplő elemek az alábbi módon származtathatóak a forrás modellből:

- *Server elem*: amennyiben az *element_server* minta (2.6) illeszkedik a forrás modellen, hozzunk létre egy *Server* elemet a cél (nézeti) modellben. Ha létezik *ViewModel* elem, annak *servers* referenciájához adjuk hozzá a létrehozott elemet (ezzel biztosítva a szabályos tartalmazási hierarchiát).
- *Trigger elem*: amennyiben az *element_trigger* minta (2.7 ábra) illeszkedik a forrás modellen, hozzunk létre egy *Trigger* elemen a cél (nézeti) modellben. Ha létezik *ViewModel* elem, annak *triggers* referenciájához adjuk hozzá a létrehozott elemet.

A fentiek alapján látható, hogy további megfelelően definiált szabályokkal a nézeti modell minden eleme, az elemek közötti kapcsolatok valamint az elemekhez tartozó tulaj-

donságok is egyaránt származtathatóak a forrás modellből. A(z) 2.12 ábra a transzformáció működését szemlélteti az F.1 függelékben bemutatott modell (mint forrás) valamint az F.2 függelékben bemutatott modell (mint cél / nézeti modell) segítségével (az átláthatóság kedvéért az ábra csak a modellek fontosabb részeit tartalmazzák).



2.12. ábra: Transzformáció mintaillesztéssel

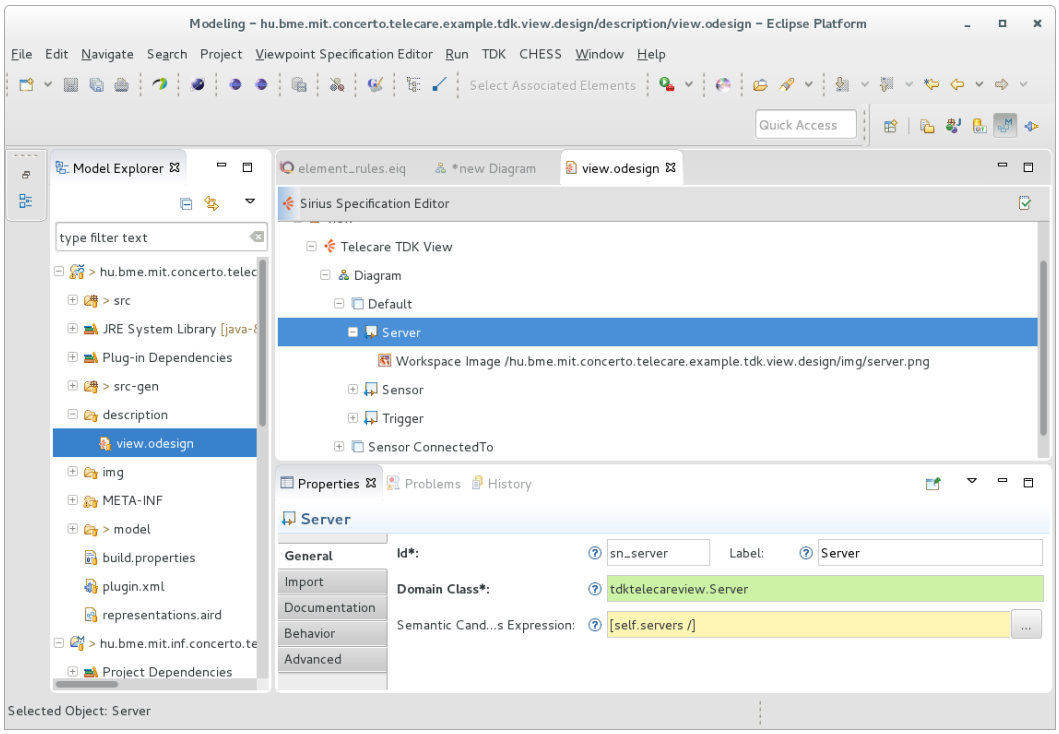
Az ábrán különböző színekkel jelölt nyilak egy-egy szabály tüzelésekor végrehajtott műveleteket jelölik a cél (nézeti) modellen. Látható, hogy a példa esetén az *element_trigger* minta a {fit_10} forrás modellbeli elemre illeszkedik, melynek hatására a nézeti modellben létrejön a {view_fit_10} elem és beállításra kerül a *triggers* referencia a {vm1, view_fit_10} elemek között. Ehhez hasonlóan az *element_server* minta illeszkedik a {h1}, {h2} elemekre, melynek következtében a nézeti modellben létrejön a {view_s1} és {view_s2} elem, valamint beállításra kerül a *servers* referencia {vm1, view_s1} és a {vm1, view_s2} elemek között.

2.6. Sirius

A *Sirius[10]* egy olyan *Eclipse* alapú technológia, melynek segítségével *EMF* modellek megjelenítésére és szerkesztésére is alkalmas felületet tudunk definiálni. Az eszköz egy jól és viszonylag egyszerűen használható eszköztárat biztosít, melynek segítségével könnyedén elkészíthetjük saját modelljeinkhez használt grafikus szerkesztő- illetve meg-

jelenítő felületünket. Alapvetően három opció közül választhatunk: lehetőségünk van diagram típusú, táblázatos valamint fa struktúrájú szerkesztő eszközök létrehozására. Dolgozatomban a továbbiakban a diagram típusúak bemutatásával foglalkozom.

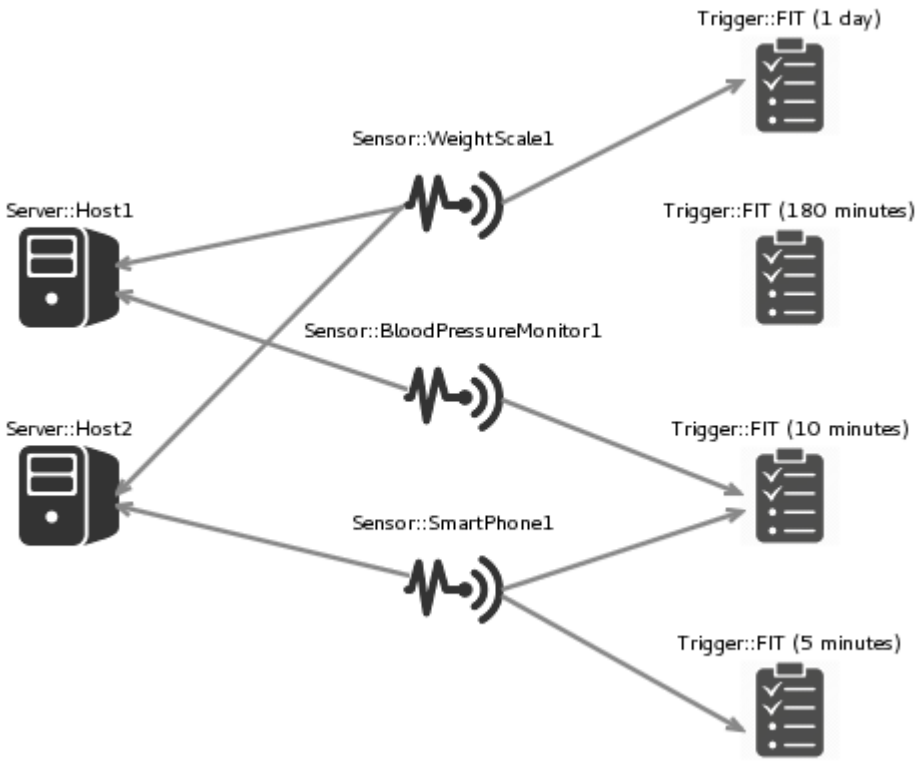
A *Sirius* egy leképzés alapú megközelítést használ diagram típusú felületek definiálásához. Egy felület elkészítése során a megjelenítendő modellben szereplő elemekhez típus szerint rendelhetjük hozzá a kívánt megjelenítési módot. Ez úgy valósítható meg, hogy meghatározunk a megjelenítendő modell metamodellje alapján egy leképzést az abban szereplő elemek, valamint az azokhoz tartozó grafikus reprezentációk között (ehhez a *Sirius* egy kényelmes felületet biztosít, ezt szemlélteti a(z) 2.13 ábra).



2.13. ábra: A Sirius által biztosított felület

A fenti ábra azt mutatja, hogy a 2.11 ábrán bemutatott nézeti metamodell alapján hogyan definiálható *Server* elemek esetében a megjelenítés módja. Látható, hogy szükséges megadni az elem típusát (*Domain Class* mező), mivel ezzel határozható meg, hogy milyen típusú elemekhez kívánjuk hozzárendelni a definiált megjelenítési módot. Szükség megadni továbbá a megjelenítendő elemeket kiválasztó kifejezést (*Semantic Candidates Expression*), mellyel tovább szűkíthetőek az adott típusba tartozó, diagramon megjelenő elemek halmaza. A *Sirius* támogat bizonyos nyelveket, melyeket használhatunk ezen kifejezés megadására, ilyen például az *Acceleo[11], OCL – Object Constraint Language[12]*, valamint *AQL – Acceleo Query Language[13]*. Az elmondottakon túl meg

kell végül határoznunk a megjelenítés módját (a példában ezt a *Workspace Image* elem írja le, mellyel egy tetszőleges képet rendelhetünk az egyes modell elemekhez). A korábban leírtaknak megfelelően látható, hogy ezzel a módszerrel meghatározhatjuk az egyes metamodellben szereplő elemek segítségével, hogy a nekik megfelelő példány modellbeli elemeknek milyen legyen a grafikus reprezentációja.



2.14. ábra: F.2 függelékben található nézeti modell grafikus reprezentációja

A 2.14 ábrán egy *Sirius*-ban elkészített felület látható, mely a dolgozatomban bemutatott esettanulmányhoz tartozó nézetet ábrázolja (a diagram definiálása során használt metamodellt a 2.11 ábrán ismertettem). Az ábra alapján látszik, hogy a metamodellben szereplő elemekhez (a gyökér elem kivételével) egy-egy ikont rendeltünk hozzá, míg az egyes elemek közti kapcsolatokat élek segítségével jelenítettük meg.

A leírtakból következik, hogy a *Sirius* eszköz csak *egy-egy leképzés definiálására képes* a modellben szereplő elemek, valamint a diagramon megjelenő elemek között.

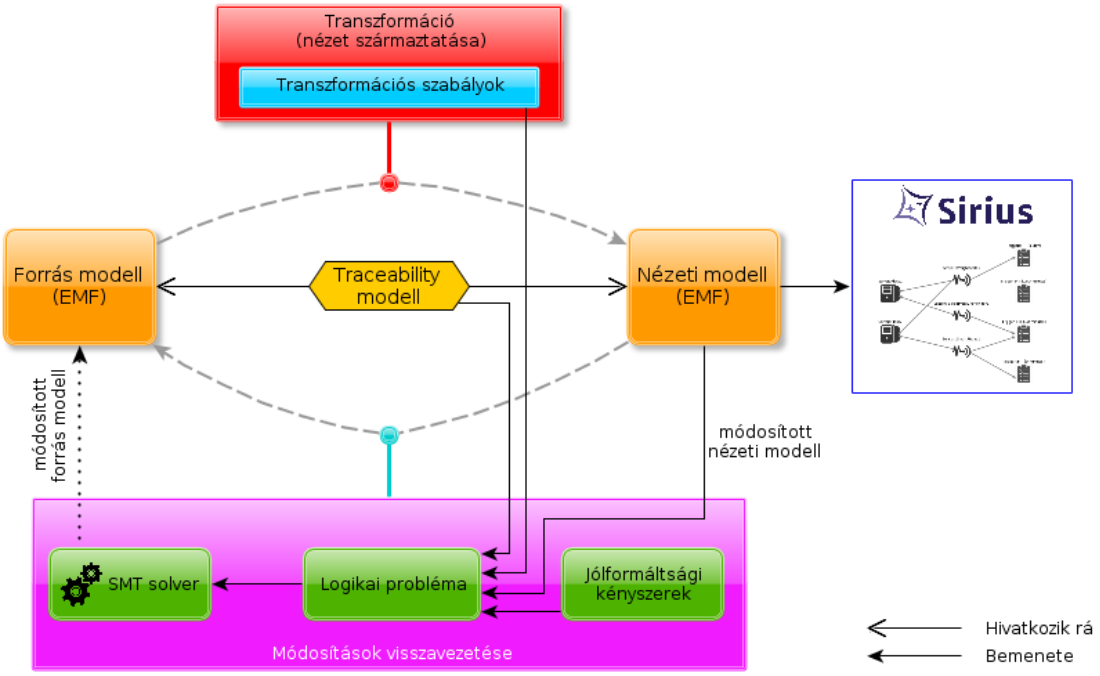
2.7. Összefoglaló

Ebben a fejezetben bemutattam a munkám során felhasznált technológiákat, valamint ismertettem a megértéséhez szükséges alapvető fogalmakat. A következőkben ráté-

rek a megvalósítással kapcsolatos részletekre: először ismertetem a rendszer főbb komponenseit (3. fejezet), majd ezt követően részletesen bemutatom az egyes komponensek működési elvét (4. fejezet).

3. Áttekintés

Ebben a fejezetben bemutatom az általam elkészített keretrendszer főbb komponenseit, valamint azok működésének fontosabb részleteit. A(z) 3.1 ábra a rendszer felépítését mutatja.



3.1. ábra: A keretrendszer felépítése

Munkám során olyan rendszer elkészítésén dolgoztam, mellyel lehetséges modellek közötti transzformációk definiálása és végrehajtása, ezzel lehetővé téve absztrakt nézetek definiálását és létrehozását *EMF* modellek fölött. A transzformáció olyan szabályok definiálásával lehetséges, melyeknek előfeltétele *IncQuery* minták segítségével adható meg, végrehajtott műveleteik pedig különböző, a rendszer által előre definiált lehetőségek közül kerülhetnek kiválasztásra. Az egyes szabályokhoz a következő, előre definiált művelet rendelhetők: elem létrehozása / törlése, referencia beállítása / törlése, attribútum beállítása / törlése. Nyilvánvaló, hogy *EMF* modellek esetén ezen műveletek felhasználásával tetszőleges struktúrájú nézeti (cél) modell származtatható, így a transzformációnak csak az előfeltételként alkalmazható gráfminták szabhatnak határt.

A nézeti modell származtatásán túl foglalkoztam azzal a problémával is, hogy hogyan lehetséges a nézeti modellen végzett módosítások „visszavezetése” a forrás modellre. A problémát alapvetően úgy közelítettem meg, hogy nem egy automatikus, két

irányú transzformáció definiálását tűztem ki célul, hanem a vissza irány (nézet módosításának visszavezetése) esetén csak a lehetséges forrásoldali módosítások meghatározását végzem el, melyek közül a felhasználó választhatja ki a neki megfelelőt / tetszőt. Ezzel a megközelítéssel a nézet származtatása során definiálható szabályok halmaza lényegesen nagyobb, mint automatikus visszavezetés esetén, mivel így nem kell a kölcsönösen egyértelmű megfeleltetést biztosítanunk a forrás valamint a származtatott cél modell között.

A nézeten végzett módosítások visszavezetése során a forrás modellen végzendő műveletek meghatározásához a feladatot logikai problémaként definiálom: arra a kérdésre keresem a választ, hogy a módosított nézeti modellhez létezik-e olyan forrás modell, melyre a megadott transzformációs szabályokat alkalmazva éppen a módosítás után kapott nézeti modell áll elő. Természetesen amennyiben létezik ilyen modell, a feladat részét képezi annak megtalálása is (amennyiben több ilyen modell is létezik, azok között a felhasználó választhat).

A logikai probléma felírásához szükség van arra, hogy kijelöljük a forrás modell azon részét, melynek módosításával a kívánt nézet elérhető (*változtatható rész*), valamint szükséges meghatározni a forrás modellnek azt a részét is, ami a megoldás keresése során nem változtatható meg (*fix rész*). Ezek meghatározásával a felírt probléma komplexitása csökkenthető, melynek köszönhetően a visszavezetés számítására fordított idő is rövidebb lesz. Mivel olyan lehetséges forrás modelleket keresünk, melyeknek transzformált képe éppen a módosított nézeti modell, így a megoldandó logikai problémához szükség van továbbá a transzformáció során alkalmazott szabályokra (a kapcsolódó mintákkal és illeszkedésekkel együtt) is. Végezetül további bemenetként a forrás modell struktúrájával kapcsolatos megkötésekre (metamodell), valamint jólformáltsági kényszerek (olyan egyéb kényszerek, melyek metamodell szinten nem voltak meghatározhatóak) megadására is szükség van, mivel így garantálható, hogy a következtető csak strukturálisan helyes megoldásokat talál meg. Az így definiált logikai probléma megoldása *SMT Solver* segítségével történik.

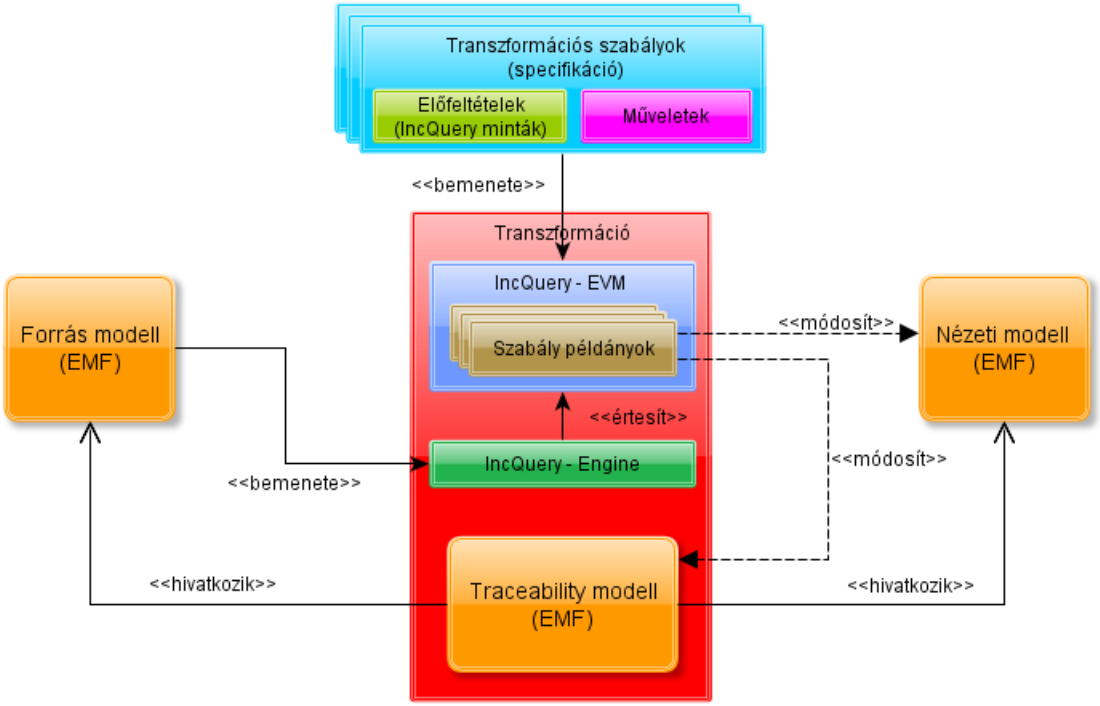
A fentieken túl foglalkoztam azzal is, hogy az elkészített rendszer integrálásra kerüljön az iparban széleskörűen használt *Sirius* editorba (2.6 fejezet). Mivel ez az eszköz alapvetően nem képes több-egy leképzésre így a célom az volt, hogy ezt a funkciót az elkészített keretrendszer segítségével tegyem elérhetővé.

4. Megvalósítás

Ebben a fejezetben részletesen bemutatom az elkészített rendszer működését. Először a nézeti modell származtatásával (*előre irány*) kapcsolatos részleteket, felmerült nehézségeket tárgyalom, majd ezt követően részletesen ismertetem a módosítások visszavezetésének (*vissza irány*) mikéntjét, a kapcsolódó problémákat valamint azok megoldásának lehetőségeit. Mind az előre- mind pedig a vissza iránnyal kapcsolatos fontosabb problémákat / nehézségeket példák segítségével szemléltetem.

4.1. Nézeti modell automatikus származtatása

A(z) 4.1 ábra a transzformáció működésének részletes bemutatására szolgál. Ahogyan azt már korábban ismertettem, a transzformációnak egy darab forrás modell a bemenete, melyből a szabályok végrehajtását követően a folyamat kimeneteként előáll a nézeti (cél) modell. Ehhez természetesen szükség van egy megfelelően definiált szabályhalmazra is, mely a leképzés pontos módját specifikálja. A transzformáció tervezése során kritérium volt, hogy olyan megoldást kell készítenem, mely inkrementális, vagyis a forrás modell változása esetén csak a szükséges műveleteket végzi el a nézeti modellen.



4.1. ábra: A transzformáció működése (előre irány)

4.1.1. Szabály alapú nézeti modell generálás

A fenti ábrát megvizsgálva látható, hogy az elkészített keretrendszer erősen támaszkodik az *IncQuery – EVM* által nyújtott szolgáltatásokra. Ahogyan arra már korábban kitértem (2.3.1.2 fejezet), az *IncQuery – EVM* egy olyan reaktív keretrendszer, mely képes különböző minták illeszkedésével kapcsolatos események (megjelenés, eltűnés, frissülés) hatására valamilyen akciót végrehajtani. Természetesen ezen viselkedés szabályokon keresztül definiálható, melyhez szükséges megadni a vizsgálandó mintát, a figyelt eseményeket valamint az elvégzendő akciókat.

A fentiekből következik, hogy az egyes szabályokban definiált műveletek végrehajtása minden egyes illeszkedés megjelenése, eltűnése, frissülése esetén meg fog történni, így olyan módon kell definiálnunk a szabályrendszert, hogy azok ilyen végrehajtás mellett is helyes modellt állítsanak elő. A szabályok alapvetően két csoportba oszthatóak:

1) a nézeti (cél) modellben szereplő elemhalmaz manipulációjáért felelős szabályok,
2) a nézeti (cél) modellben található strukturális tulajdonságok (attribútum, referencia) manipulációjáért felelős szabályok.

Fontos követelmény ezen csoportok elemeire vonatkozóan, hogy minden egyes olyan szabály, mely az 1) csoportban található hamarabb kell, hogy tüzeljen, mint bármelyik szabály a 2) csoportból. Erre azért van szükség, mert minden egyes 2)-beli szabály aktivációja esetén léteznie kell olyan elemnek a nézeten, melyre a definiált műveletet elvégezzük, egyébként annak hatása elveszik.

Ennek a problémának a kiküszöbölésére (megfelelően definiált minták esetén, lásd 4.1.2) az egyes csoportok alapján történő prioritásos végrehajtás egy jó megoldás lenne. Erre az *EVM* lehetőséget biztosít, mivel a definiált szabályokhoz prioritás rendelhető, így a végrehajtás sorrendje aktivált szabályok esetén ez által meghatározható.

4.1.2. Transzformációs szabályokra vonatkozó követelmények

A 2.5 fejezetben bemutatott általános esettől eltérően bizonyos elvárásoknak, kritériumoknak meg kell felelnie a transzformáció során alkalmazott szabályoknak, mely kritériumok első sorban az alkalmazható műveletekre vonatkoznak. Mivel minden egyes *EMF* modell kezelhető olyan speciális gráfként, mely gráf csomópontjai és élei típussal

jellemezhetőek, valamint csomópontjaihoz további tulajdonságok (attribútumok) rendelhetőek (*typed attributed graph[14]*), így elegendőnek bizonyult az elvégezhető műveleteket az alábbi lehetőségekre korlátozni:

a) elemek (adott típusú csomópontok) létrehozása

b) elemek (adott típusú csomópontok) törlése

c) referenciák (csomópontok közötti élek) beállítása

d) referenciák (csomópontok közötti élek) törlése

e) attribútumok (csomópontokhoz tartozó tulajdonságok) értékének beállítása

f) attribútumok (csomópontokhoz tartozó tulajdonságok) értékének törlése.

Látható, hogy ezen műveletek megfelelő végrehajtásával tetszőleges struktúrával rendelkező *EMF* modell létrehozható. Elmondható továbbá az is, hogy a fenti hat művelet éppen a 4.1.1 fejezetben meghatározott kategóriák további finomítása az alábbiak szerint: a) és b) műveletek az 1) csoportba tartoznak, míg c), d), e), f), műveletek pedig a 2) csoportba sorolhatóak.

Az a) szabály esetén meg kell határoznunk azt az elem-típust, amelyből egy újabb példányt kívánunk létrehozni. A szabály végrehajtását követően a létrehozott elem a nézeti (cél) modell gyökerében helyezkedik majd el, mivel itt még nem definiáljuk a tartalmazási hierarchiát. Az új elem attribútumainak értéke az alapértelmezettként megadott értékekre kerülnek beállításra.

A b) szabály esetén meg kell határoznunk azt az elemet, melyet a szabály tüzelésekor törölni kívánunk. Ennek meghatározásához olyan megoldásra van szükség, mely a szabály előfeltételeinek teljesülése esetén egyértelműen ki tud jelölni egy elemet a nézeten (amennyiben az még létezik).

A c) szabály esetén meg kell határoznunk a szabály tüzelésekor beállítandó referenciát, valamint a referencia forrását és célját adó elemeket. Elvárás továbbá, hogy amennyiben a szabály aktív, pontosan egy ilyen elempár létezzen a nézeti modellben.

A d) szabály esetén meg kell határoznunk a szabály tüzelésekor törlendő referenciát, valamint a referencia forrását és célját adó elemeket (mivel így tudjuk egyértelműen megadni, hogy melyik él törlendő a gráfból, amennyiben az még létezik).

Az e) szabály esetén definiálnunk kell a szabály végrehajtásakor beállítandó attribútumot, a beállítandó értéket, valamint azt a nézeti modellben szereplő elemet, amin a beállítást végezzük. Elvárás továbbá, hogy amennyiben a szabály aktív, létezzen a feltételeknek megfelelő elem.

Az f) szabály esetén meg kell határoznunk a módosítandó attribútumot, a módosításban érintett elemet valamint azt az értéket, melyet törölni fogunk (utóbbira azért van szükség, mert többes multiplicitás esetén nem lenne egyértelmű, hogy melyik értéket kell törölni a szabály tüzelésekor; amennyiben többször szerepel ugyan az az érték, annak csak egyik előfordulása törlendő).

Az eddig ismertetetteken túl van egy olyan speciális követelmény a 2) csoportba tartozó szabályok esetén, mely a használt minták definiálásával kapcsolatban támaszt néhány további kényszert. Mivel mind a c) mind pedig az e) típusú szabályok esetén elvárás, hogy a nézeti modellben léteznie kell a műveletben érintett elem(ek)nek a szabály aktivációjakor, így szükség van bizonyos megkötésekre a használt mintát illetően. Mindkét esetben az az elvárás, hogy a műveletben érintett elem(ek) létezését már a minta definíciójában követeljük meg. Erre mutat példát a(z) 4.2 ábrán látható minta (*ref_sensor_connectedTo*), mely az esettanulmányban szereplő nézeti modellben található *Sensor* illetve *Server* elemek közti *connectedTo* referencia beállításakor kerül használatra.

```
26
27 pattern ref_sensor_connectedTo(sensor : Sensor, host : Host) {
28     find element_sensor(sensor);
29     find element_server(host);
30
31     Sensor.measurementTypes.measurement(sensor, measurement);
32
33     EventCompletedTrigger.triggeredBy(ect, measurement);
34     ReportingEvent.trigger(re, ect);
35     ReportingEvent.address(re, host);
36 }
```

4.2. ábra: A *ref_sensor_connectedTo* minta

A mintából jól látszik, hogy az *element_sensor* valamint *element_server* (F.3 függelék) minták (melyek a nézeten található *Sensor* és *Server* elemek létrehozásakor használtak) meghívásával (*find* kulcsszó) garantáljuk azt, hogy ez a minta csak abban az esetben illeszkedik, ha biztosan van illeszkedése a hívott két másik mintának. További kényszer, hogy a *Sensor* elemhez kell tartozzon olyan „mérés” (*Measurement*), melynek eredménye az adott *Host*-ra kerül feltöltésre. Mivel a definiált kényszereknek köszönhetően

ez a minta csak akkor illeszkedik, ha a hívott minták is illeszkednek, a 4.1.1 fejezetben leírtak alapján ezen mintákkal definiált (a) típusú) szabályok minden esetben előbb tüzelnek, így garantált, hogy a nézeti modellben létezni fog az a *Sensor* illetve *Server* elem, melyek között a *connectedTo* referencia beállítható.

Minden szabály esetén követelmény még a *traceability* modell karbantartása az elvégzett műveletek után. Ennek a modellnek köszönhetően pontosan nyomon tudjuk követni azt, hogy milyen illeszkedések hatására milyen műveletek történtek a nézeti modellen. A következőkben bemutatom, hogy erre az információra nemhogy a visszafele irány, de már az előre irány megvalósítása során is szükség van. Az a) típusú szabály kivételével minden esetben tudnunk kell azt, hogy az elvégzendő műveletek mely nézeti modellbeli elemet érintenek. Ennek meghatározása az egyes esetekben az alábbiak szerint történik:

- b) típusú szabály abban az esetben kerül tüzelésre, amennyiben egy a) típusú szabályban használt minta *illeszkedése eltűnik.* Ebben az esetben a *traceability* modell alapján meghatározható, hogy az illeszkedéshez milyen elem tartozik a nézeten, így annak törlése elvégezhető. A korábbi példánál maradva az *element_server* minta illeszkedésekor egy *Server* elem jön létre a nézeten, majd az illeszkedés megszűnésekor a korábban létrehozott elem törlendő.
- c)-d)-e)-f) típusú szabályok esetén a definiált minta paraméterei között fel kell sorolni minden olyat, melyre az érintett nézeti elemek meghatározásakor szükség lehet (lásd 4.2 ábra: *sensor* és *host* paraméterek). Ezekből az információkból már kikövetkeztethetőek azon elem(ek), melyekre a szabály tüzeléséhez szükség van. A korábbi példa esetében a *host* paraméterben található elemből biztosan létrejött egy *Server* elem, valamint a *sensor* paraméterben található elemből pedig egy *Sensor* elem a nézeten, mely elemek a *traceability* modellből ezen információk segítségével megtalálhatóak. Ezt követően a kívánt művelet a nézeti modellben lévő elemeken elvégezhető.

További elvárás a szabályokat illetően, hogy azokat úgy kell definiálni, hogy az általuk a nézeti modellen végzett műveletek nem írhatják felül egyetlen másik szabály hatását sem (*vagyis az egyes szabályok hatása független kell legyen*). Ez a követelmény (a korábbiak mellett) a transzformáció egyértelműségének biztosítását szolgálja.

A fent leírtakból látszik, hogy az ilyen módon definiált szabályok segítségével már garantáltan helyes lesz a származtatott modell. A következő fejezetben ábrák segítségével szemléltetem a korábbi példában bemutatott transzformációkat.

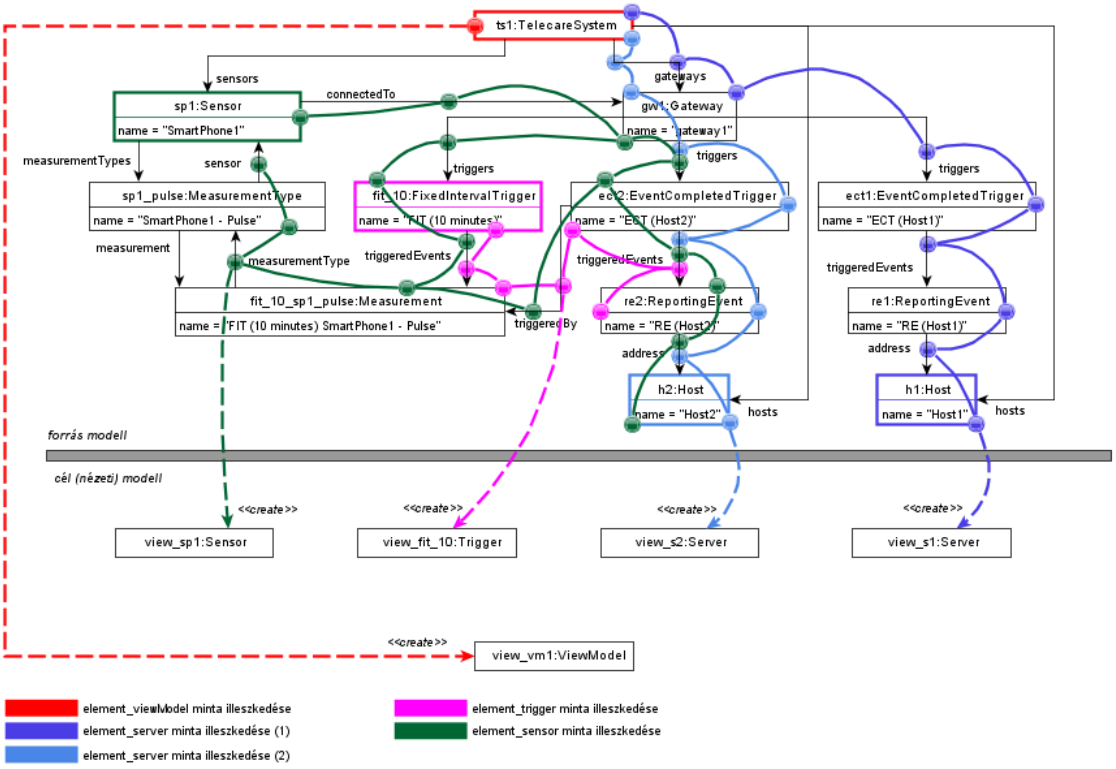
4.1.3. Működés szemléltetése

A következőkben bemutatom az esettanulmányban használt transzformáció működését néhány példán keresztül (nem térek ki minden egyes szabály részletes ismertetésére, csak a folyamat megértéséhez szükségesekre). Az alábbi ábrák az esettanulmányban megismert távfelügyeleti rendszert leíró modell (F.1 függelék) egy részét, valamint a hozzá tartozó nézet (F.2 függelék) egy megfelelő darabját ábrázolják. A folytonos színes vonallal összekötött megjelölt elemek illetve referenciák egy-egy minta illeszkedését mutatják (színes keret jelöli a *publikus* elemeket, míg az illeszkedéshez tartozó többi elem *rejtett*, lásd 2.3.1.1 fejezet), valamint az illeszkedés hatására végzett műveleteket szaggatott vonallal jelöltem (ez tekinthető a *traceability* modellben tárolt információk reprezentációjának is).

A(z) 4.3 ábra azt szemlélteti, hogy az egyes a) típusú szabályok esetén milyen illeszkedések találhatóak a forrás modellben, valamint azok hatására milyen elemek jönnek létre a nézeti (cél) modellben (szaggatott vonal). Az ábrán az alábbi szabályok végrehajtása látható:

- *ViewModel* elem létrehozása: a szabály aktivációjához az *element_viewModel* minta (F.3 függelék) illeszkedésének megjelenése szükséges a forrás modellben. A példa esetében a {ts1} elemre illeszkedik a minta, így a nézeti modellben létrejön a {view_vm1} elem.
- *Server* elem létrehozása: a szabály aktivációjához az *element_server* minta (F.3 függelék) illeszkedésének megjelenése szükséges a forrás modellben. A példában a {h1} valamint {h2} elemekre is illeszkedni fog a minta, így a nézeti modellben létrejön a {view_s1} és {view_s2} elem.
- *Trigger* elem létrehozása: a szabály aktivációjához az *element_trigger* minta (F.3 függelék) illeszkedésének megjelenése szükséges a forrás modellben. A példában a {fit_10} elemre illeszkedik a minta, így a nézeti modellben létrejön a {view_fit_10} elem.

- *Sensor* elem létrehozása: a szabály aktivációjához az *element_sensor* minta (F.3 függelék) illeszkedésének megjelenése szükséges a forrás modellben. A példában az {sp1} elemre illeszkedik a minta, így a nézeti modellben létrejön a {view_sp1} elem.



4.3. ábra: Transzformáció működésének szemléltetése (elemek létrehozása a nézeti modellben)

A(z) 4.4 ábrán néhány c) típusú szabály működése került bemutatásra. Látható, hogy az egyes illeszkedések hatására egy-egy meghatározott referencia kerül beállításra. Az 4.1.2 fejezetben leírtaknak megfelelően minden ilyen típusú szabályhoz tartozó minta esetén a helyes működéshez meg kell hívnunk a megfelelő kapcsolódó mintákat (habár az illeszkedéshez ezek is szükségesek, az átláthatóság érdekében nem kerültek újból ábrázolásra; a színes kerettel kiemelt elemekre tekintsünk úgy, hogy azok teljesítik a kapcsolódó mintákban meghatározott elvárásokat). Az ábrán az alábbi szabályok végrehajtása látható:

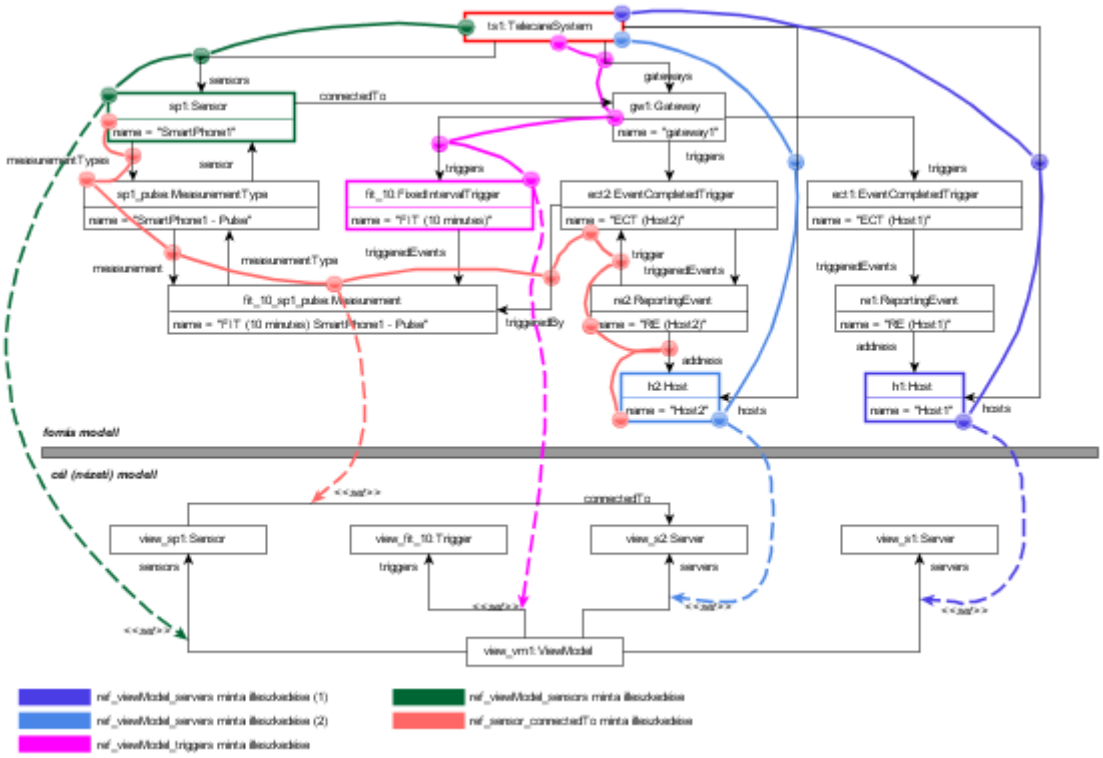
- *servers* referencia beállítása*:* a *ref_viewModel_servers* (F.4 függelék) minta minden illeszkedésének megjelenése esetén a *telecareSystem* paraméterben található elemből (a megfelelő a) típusú szabály hatására) létrehozott (nézeti modellbeli) elem, valamint a *host* paraméterében található elemből korábban létrehozott elem

között a *servers* referencia beállításra kerül. A példában ez a {ts1, h1}, {ts1, h2} illeszkedésekhez a {view_vm1, view_s1} valamint {view_vm1, view_s2} elemek közti *servers* referencia beállítását jelenti.

- *triggers* referencia beállítása: a *ref_viewModel_triggers* (F.4 függelék) minta minden illeszkedésének megjelenése esetén a *telecareSystem* paraméterben található elemből (a megfelelő a) típusú szabály hatására) létrehozott (nézeti modellbeli) elem, valamint a *trigger* paraméterében található elemből korábban létrehozott elem között a *triggers* referencia beállításra kerül. A példában ez a {ts1, fit_10} illeszkedéshez a {view_vm1, view_fit_10} elemek közti *triggers* referencia beállítását jelenti.

- *sensors* referencia beállítása: a *ref_viewModel_sensors* (F.4 függelék) minta minden illeszkedésének megjelenése esetén a *telecareSystem* paraméterben található elemből (a megfelelő a) típusú szabály hatására) létrehozott (nézeti modellbeli) elem, valamint a *sensor* paraméterében található elemből korábban létrehozott elem között a *sensors* referencia beállításra kerül. A példában ez a {ts1, sp1} illeszkedéshez a {view_vm1, view_sp1} elemek közti *sensors* referencia beállítását jelenti.

- *connectedTo* referencia beállítása: a *ref_sensor_connectedTo* (F.4 függelék) minta minden illeszkedésének megjelenése esetén a *sensor* paraméterben található elemből (a megfelelő a) típusú szabály hatására) létrehozott (nézeti modellbeli) elem, valamint a *host* paraméterében található elemből korábban létrehozott elem között a *connectedTo* referencia beállításra kerül. A példában ez az {sp1, h2} illeszkedéshez a {view_sp1, view_s2} elemek közti *connectedTo* referencia beállítását jelenti.



4.4. ábra: Transzformáció működésének szemléltetése (referenciák beállítása a nézeti modellben)

A bemutatott működéssel kapcsolatban ki kell emelnem, hogy természetesen a(z) 4.3 ábrán ismertetett műveleteknek a(z) 4.4 ábrán bemutatottak előtt kell befejeződniük, mivel máskülönben a transzformáció nem működne. A fenti példában nem tértem ki azokra az esetekre, amikor egy már létező illeszkedés (például {ts1, sp1}, {sp1, h2}, stb.) megszűnik. Ezekben az esetekben a *traceability* modellben tárolt információk alapján egyértelműen meghatározható, hogy mely elemek érintettek a nézeten, így a korábban végzett műveletek "visszavonhatóak" (például referencia törlése; elem törlése; stb.).

Végezetül pedig meg kell említsem, hogy az e) és f) típusú (attribútumokkal kapcsolatos) szabályok a bemutatott c) és d) típusú szabályokkal teljesen analóg módon működnek, így ezek ismertetésére külön nem térek ki (ilyen szabályokhoz tartozó minták a(z) F.5 függelékben találhatóak).

4.1.4. Összefoglalás

Ebben a fejezetben bemutattam a megvalósított eseményvezérelt, inkrementális modelltranszformációs keretrendszert, mellyel lehetséges tetszőleges absztrakt nézetek definiálása gráfminta alapú szabályok segítségével. Kitértem a transzformáció során

használható szabályokra valamint mintákra vonatkozó megkötésekre, elvárásokra, valamint egy komplex példán keresztül szemléltettem a leképzés működését. Munkám során nagyban támaszkodtam konzulensem publikációjában foglaltakra [16].

4.2. Nézeti modellen végzett műveletek visszavezetése

Ebben a fejezetben bemutatom azt a megközelítést, melyet alkalmazva bizonyos megkötések mellett képesek vagyunk a nézeti modellen elvégzett változtatásokat visszavezetni a forrás modellre. A következőkben ismertetem az alapvető fogalmakat, a feladatban rejlő nehézségeket, valamint bemutatom az általam használt megközelítés működését.

4.2.1. Problémafelvetés

A probléma precíz megfogalmazása érdekében először be kell vezessek néhány alapvető fogalmat.

Definíció** (Nézeti modell módosítása)**: a nézeti modellen végzett bármilyen változtatás (pl.: elem létrehozása / törlése, referencia beállítása / törlése, attribútum értékének beállítása / törlése).

Definíció** (Nézeten végzett módosítás visszavezetése; visszavezetés)**: a nézeten végzett módosítások leképezése a forrás modellen végrehajtandó módosításokra.

Definíció** (Helyes visszavezetés)**: egy visszavezetés abban az esetben tekinthető helyesnek, ha a meghatározott műveleteket végrehajtva a forrás modellen egy olyan helyes (metamodellnek és jólformáltsági kényszereknek megfelelő) modellt kapunk, melyre alkalmazva a transzformációs szabályokat éppen a módosított nézeti modell áll elő.

Definíció** (Hibás visszavezetés)**: minden olyan visszavezetés hibás, ami nem helyes.

A megismert fogalmakat felhasználva a feladat úgy fogalmazható meg, hogy *a cél a nézeti modellen végzett tetszőleges módosításhoz meghatározni a helyes visszavezetéseket (amennyiben azok léteznek)*. A következőkben bemutatom a problémával kapcsolatos nehézségeket néhány egyszerű példa segítségével.

Nyilvánvaló, hogy amennyiben a forrás- és a nézeti modell között kölcsönösen egyértelmű megfeleltetést definiálunk a transzformációs szabályok által, úgy a nézeten végzett módosítások visszavezetése is triviális feladat. Ezt az esetet szemlélteti a(z) 4.5

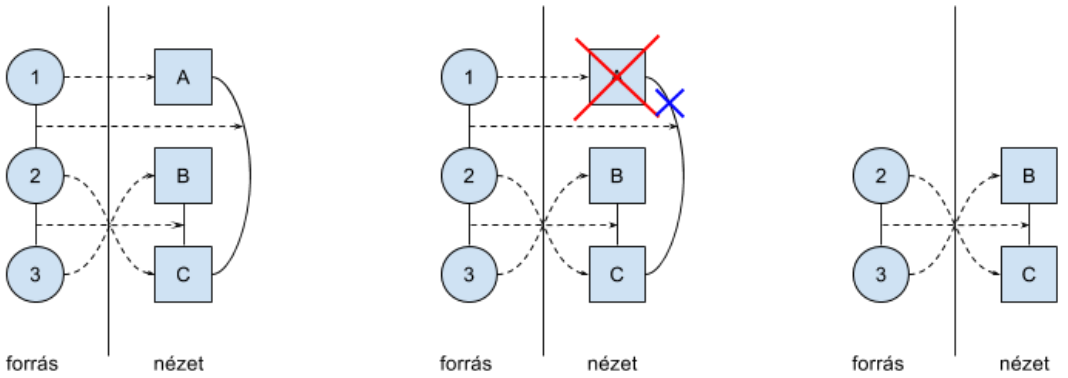
ábra. Látható, hogy a nézeten kiválasztott elem törlése maga után vonja a belőle kiinduló élek törlését is a nézetről, és az így előálló modellhez kell megkeresnünk azt a forrás modellt, mely ezzel konzisztens. Ehhez pedig éppen a törölt elemhez illetve élhez tartozó forrás modellbeli elemek törlése szükséges. A bemutatott esettel kapcsolatban néhány újabb fogalmat vezethetünk be.

***Definíció** (Egyértelmű nézet):* amennyiben egy módosításhoz pontosan egy visszavezetés tartozik, úgy a nézetet *egyértelműnek* hívjuk.

***Definíció** (Nem egyértelmű nézet):* amennyiben a nézeten végzett módosításhoz több, helyes visszavezetés is létezik, úgy *a nézet nem egyértelmű*, de *konzisztens.*

***Definíció** (Konzisztens nézet):* amennyiben a nézethez létezik olyan helyes (metamodellnek és egyéb jólformáltsági kényszereknek megfelelő) forrás modell, melyen elvégezve a transzformációt a kívánt nézet áll elő, akkor a *nézetet konzisztensnek* hívjuk.

***Definíció** (Inkonzisztens nézet):* minden olyan nézet *inkonzisztens,* ami nem *konzisztens.*

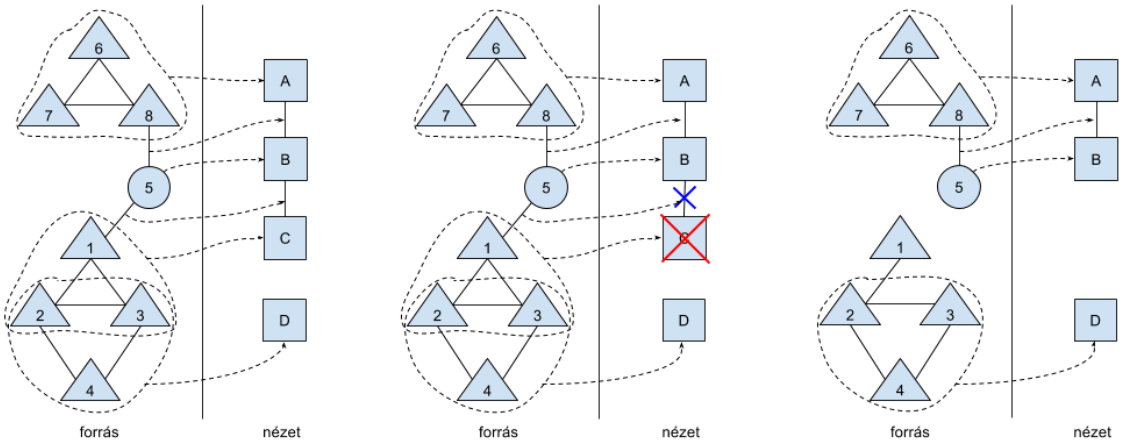


4.5. ábra: Nézet módosítása kölcsönösen egyértelmű leképzés esetén (*egyértelmű nézet*)

A(z) *4.6* ábrán látható eset a *nem egyértelmű* nézetekre mutat egy példát. A transzformációs szabályok legyenek a következők:

- minden forrás oldali *körrel jelölt elemhez* tartozzon egy *négyzettel jelölt elem* a nézeten,
- minden forrás oldali *háromszöggel jelölt elemből* álló *háromcsúcsú teljes gráfhoz* tartozzon egy *négyzettel jelölt elem* a nézeten,
- amennyiben két illeszkedés között van él a forrás modellben és az illeszkedések nem metszik egymást, akkor legyen él a hozzájuk tartozó nézeti elemek között is.

Látható, hogy a nézeten a *C* elem törlésével automatikusan törlődik a {B, C} él is, melyre igaz, hogy a leképezés miatt egyértelműen meghatározható, hogy ehhez mely él törlése szükséges a forrás oldalon. A *C* elem törlése miatt viszont a *nézet nem egyértelmű*, mivel az elemhez tartozó illeszkedés (vagyis a *háromszöggel jelölt elemekből* álló *háromcsúcsú teljes gráf*) megszüntetésére több lehetőség is kínálkozik. Fontos felhívnom a figyelmet viszont arra az esetre, hogy sem a *2-es*, sem a *3-as*, sem pedig a köztük húzódó {2, 3} él törlése nem eredményez helyes megoldást, mivel ekkor a *D* elemhez tartozó illeszkedés is sérülne (ez által pedig a *D* elem is törlésre kerülne a nézetről, ami nem megengedett). Ez alapján látható, hogy *konzisztens* nézet esetén a lehetséges *visszavezetések* meghatározása igen komplex feladat lehet.

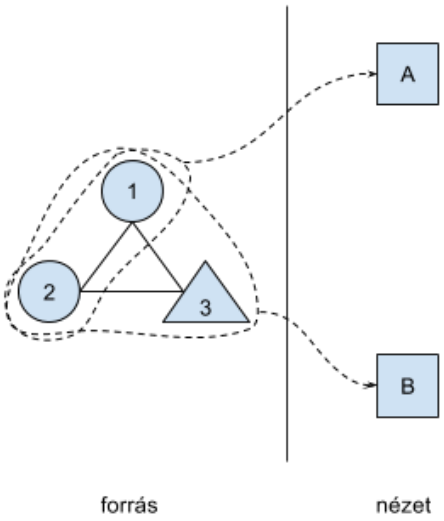


***4.6.* ábra: Nézet módosítása nem kölcsönösen egyértelmű leképzés esetén (nem egyértelmű nézet)**

A(z) 4.7 ábra az *inkonzisztens nézetek* szemléltetésére mutat egy példát. A transzformációs szabályok legyenek a következők:

- minden olyan esetben, ahol a forrás modellben két *körrel jelölt elem* között van él hozzunk létre egy *négyzettel jelölt elemet* a nézeti modellben,
- minden olyan esetben, ahol a forrás modellben két *körrel* valamint egy *háromszöggel* jelölt elemek *háromcsúcsú teljes gráfot* alkotnak, hozzunk létre egy *négyzettel jelölt elemet* a nézeti modellben,
- amennyiben két illeszkedés között van él a forrás modellben és az illeszkedések nem metszik egymást, akkor legyen él a hozzájuk tartozó nézeti elemek között is.

Látható, hogy a nézeti modellből nem törölhető úgy az *A* elem, hogy a *B* megmaradjon, mivel *A* törlésekor *inkonzisztens nézet* állna elő. Ettől függetlenül *B* elem törölhető úgy, hogy *A* megmaradjon például a *3-as* elem eltávolításával a forrás modellből.



4.7. ábra: Nézet módosítása nem kölcsönösen egyértelmű leképzés esetén (inkonzisztens nézet)

A fentieken túl természetesen igen összetett problémát jelenthet elemek létrehozása, illetve referenciák vagy attribútumok beállítása a nézeti modellen, mivel mindhárom esetben figyelembe kell venni, hogy a forrás modellen végzett módosítások hatására akár több új illeszkedés is megjelenhet, melyek adott esetben nem kívánatosak, továbbá az elvégzett módosításokhoz meg kell találni az összes helyes visszavezetést (amennyiben azok léteznek). Ezen okokból kifolyólag tehát elmondható, hogy az ismertetett problémák minden lehetséges, nézeti modellt érintő módosítás esetén előfordulhatnak.

A bemutatott esetekből ugyanakkor már az is nagyon jól látszik, hogy a nézeti modell változtatása során előálló problémák megoldása már igen egyszerű modellek esetén sem mindig triviális feladat. A következő fejezetben bemutatok egy olyan általánosan is jól használható megoldást, melynek segítségével bizonyos korlátok mellett a fent ismertetett problémák megoldhatóak.

4.2.2. Módosítások visszavezetése logikai következtetők segítségével

Ebben a fejezetben bemutatom az ismertetett problémák esetén is jól használható megoldás működésével kapcsolatos fontosabb tudnivalókat kitérve a főbb komponensek szerepeire, feladataira.

A(z) 3.1 ábrán bemutatottak alapján a nézeten végzett módosítások visszavezetésének alapvető gondolata, hogy a nézet módosításának hatására előálló feladatot (4.2.1 fejezet) logikai problémaként írjuk fel, és annak keressük a lehetséges megoldásait különböző logikai következtetők segítségével (*Z3*, *Alloy*). A felírt probléma a következő:

keressük azokat a (metamodellnek és további jólformáltsági kényszereknek megfelelő) forrás modelleket, melyekre igaz, hogy azokon végrehajtva a definiált transzformációkat pontosan a módosításoknak megfelelő nézeti modellt kapjuk. Mivel a lehetséges megoldásokból több is létezhet, így a feladat megoldása nem mindig egyértelmű, ezért ilyen esetben a felhasználó felelőssége a lehetőségek közül a megfelelő kiválasztása.

A logikai következtetők számára definiálni kell a visszavezetés során előálló problémát. A probléma felírásához szükséges

a) a forrás modell azon része, melynek módosítása nem megengedett a megoldás keresése során (úgynevezett *fix rész*),
b) a forrás modell azon része, melynek módosítása megengedett a megoldás keresése során (úgynevezett *változtatható rész*),
c) a *változtatható részben* található illeszkedések megadása a mintával együtt (a módosítást követően is pontosan ez az illeszkedés halmaz kell létezzen a *változtatható részben*),
d) felső korlát a logikai következtető számára,
e) a forrás modellhez tartozó metamodell megadása,
f) a forrás modellhez tartozó jólformáltsági kényszerek megadása (amennyiben léteznek).

A következtető feladata tehát a definiált probléma megoldása az összes lehetséges megoldás megtalálásával, amennyiben az létezik. Az egyes megoldások a forrás modell *változtatható részének* módosításával keresendők úgy, hogy a módosítás során előálló modellek a különböző kényszereknek megfeleljenek. Ebből kifolyólag minél nagyobb a *változtatható rész* a forrás modellben, annál több lehetőséget kell megvizsgálnia a következtetőnek (habár nem egy *brute force* típusú megközelítésről van szó, a lehetséges kombinációk halmaza így is jelentősen befolyásolja a keresés sebességét illetve a probléma megoldhatóságát).

Amennyiben a megadott információk alapján a definiált korlát mellett létezik megoldása a problémának, azt a következtető biztosan megtalálja (ez alól kivétel, ha túlságosan bonyolult a probléma, mivel ilyenkor belátható időn belül nem áll elő a megoldás). Ha a következtető megoldás nélkül áll le, akkor vagy a megadott kényszerek ellentmondásosak (vagyis a nézetünk *inkonzisztens*), vagy a megadott korlát mellett nem létezik megoldás.

A visszavezetés logikai problémaként való megfogalmazásának köszönhetően olyan módon keressük a feladat megoldását, mely általánosan is jól használható tetszőleges forrás- és nézeti modell esetén. Ugyanakkor az is igaz, hogy ennek a megközelítésnek is vannak korlátai a transzformáció során használt gráfmintákra valamint a visszavezetés során vizsgált elemhalmaz méretére vonatkozóan. A következőkben ezek ismertetésével foglalkozom.

4.2.3. Támogatott minták

A logikai megoldókra építkező megközelítés habár sok esetben megoldást kínál a helyes visszavezetések megtalálására, ennek a módszernek is vannak bizonyos korlátai, melyet figyelembe kell vennünk a használata során. A módszerrel kapcsolatos korlátok alapvetően a transzformáció során használható kényszerekre, gráfmintákra vonatkoznak. Az alábbiakban bemutatom az *IncQuery* keretrendszer által biztosított gráfminta leíró nyelv (*IQPL*) esetén használható különböző kényszer-típusokat, valamint kitérek azokra is, melyek használata nem megengedett a helyes visszavezetések megtalálása érdekében.

A nézeten végzett módosítások visszavezetése során alapvető cél, hogy a logikai következtető számára minél inkább behatároljuk a forrás modellen azt a területet, melynek módosításával a kívánt nézet előállítható. A visszavezetés során vizsgált terület növekedésével ugyanis exponenciálisan nő a következtető által bejárandó állapottér, melynek következtében előfordulhat, hogy a következtető megakad, így sosem kapunk megoldást a megfogalmazott problémára (tehát még az sem derül ki, hogy egyáltalán létezike megoldás vagy sem). A támogatott *IQPL* kényszer-típusok köre alapvetően a szerint került meghatározásra, hogy a segítségükkel megfogalmazott minták illeszkedései a visszavezetések keresése során mennyire jól behatárolhatóak a forrás modellben.

A logikai következtetők alkalmazása esetén támogatott *IQPL* kényszer-típusok az alábbi listában kerültek felsorolásra (az egyes kényszerek a(z) 2.3.1.1 fejezetben kerültek bemutatásra):

- *típuskényszer,*
- *referenciákra vonatkozó útvonalkényszer,*
- *attribútumokra vonatkozó útvonalkényszer,*
- *egyenlőségi kényszer,*
- *diszjunkció (több, egymással vagy kapcsolatban álló törzs definiálása egy mintához)*

- *mintahivatkozás (amennyiben a hivatkozott minta is csak a felsoroltaknak megfelelő kényszer-típusokat tartalmazza).*

Általánosan igaz, hogy az ismertetett támogatott kényszer-típusokkal olyan minták definiálhatóak, melyek illeszkedéseinek mérete meghatározható a minta definíciója alapján, így ezáltal a definícióban szereplő kényszerek egyértelmű felső korlátot adnak a minta illeszkedésének méretére. Elmondható ugyanakkor az is, hogy amennyiben a transzformáció során csak olyan mintákat használunk, melyek a fenti kényszer-típusokon kívül mást nem tartalmaznak, akkor ezen minták esetében a visszavezetés során elegendő csak a minta illeszkedésében megtalálható elemek vizsgálata (ezáltal jelentősen szűkítve a következtető által bejárandó állapotteret).

A logikai következtetők alkalmazása esetén nem támogatott *IQPL* kényszer típusok az alábbi listában kerültek felsorolásra:

- *negatív mintahivatkozás (**neg find**),*
- *tranzitív lezárt (+),*
- *illeszkedés számosság (**count find**),*
- ***check*** típusú kényszerek,
- ***eval*** típusú kényszerek.

A nem támogatott kényszer-típusok két csoportba oszthatóak. Vannak olyanok, melyek hatása a modellben túlságosan nagy, így a helyes visszavezetések keresése során definiált probléma olyan nagyra nő, hogy azt a következtető már nem képes belátható időn belül megoldani (ebbe a kategóriába sorolható a *negatív mintahivatkozás,* a *tranzitív lezárt* és az *illeszkedés számosság* is). A másik csoportba tartozó kényszerek azért nem támogatottak, mivel azokat nem lehet felírni logikai problémaként (ide tartoznak az *illeszkedés számosság* bizonyos esetei, a *check* illetve *eval* típusú kényszerek, valamint a *tranzitív lezárt* bizonyos esetei).

Összefoglalva tehát azt mondhatjuk, hogy olyan mintákat támogatunk a transzformáció során, melyekre a minta definíciója alapján egyértelműen meghatározható, hogy a nézeten végzett változtatások milyen forrás oldali elemeket érinthetnek és az érintett elemek halmazának mérete felülről becsülhető a minta definíciója alapján.

4.2.3.1. Visszavezetés során érintett elemhalmaz meghatározása

Ebben a fejezetben bemutatom, hogy bizonyos minták esetén hogyan határozható meg a visszavezetés során módosítható elemhalmaz a forrás modellben. Először tekintsük a(z) 2.6 ábrán bemutatott *element_server* mintát. Látható, hogy a mintában csak olyan kényszer-típusokat használtam, melyek támogatottak a visszavezetés során. Általánosan elmondható, hogy az egyes *típus-, útvonal- és egyenlőségi kényszerek* esetén már a minta definíciójából meghatározható, hogy pontosan hány elem lesz az illeszkedésben, mivel

- ***Type** (variable)* típusú kényszer maximum egy elemre illeszkedhet,
- ***Type.referenceName** (var1, var2)* típusú kényszer további kettő vagy nulla elemre illeszkedhet,
- ***Type.ref1.ref2…refN** (var1, var2)* típusú kényszer további *N+1* vagy nulla elemre illeszkedhet (a láncolt referenciák miatt),
- ***Type.attributeName** (var1, var2)* típusú kényszer további kettő vagy nulla elemre illeszkedhet,
- míg az *egyenlőségi kényszer* már nem hoz be újabb elemeket az illeszkedésbe.

A leírtak alapján látható, hogy a megadott szabályok segítségével precíz felső becslést tudunk adni a minták illeszkedésének elemszámára. Elmondható ugyanakkor az is, hogy a minta által kijelölt illeszkedő részgráfon túl biztosan nem létezik olyan elem / referencia / tulajdonság a forrás modellben, mely az illeszkedést befolyásolná, ez által kijelenthető, hogy **egy nézeten végzett művelet visszavezetése során a következtető által vizsgált terület (*változtatható rész*) az adott illeszkedésben részt vevő elemekre korlátozódhat**. A(z) 4.3 ábrán bemutatott példa – többek között – az *element_server* mintához tartozó illeszkedést is szemlélteti. Jól látszik, hogy amennyiben a nézeti modellből törölni szeretnénk a {view_sp1} elemet, ahhoz a hozzá tartozó illeszkedést kellene megszüntetni, ami kizárólag az illeszkedésben megjelenő elemek és referenciák módosításával (akár törlésével) érhető el.

A következőkben azt szemléltetném, hogy miért okoz problémát a nem támogatott kényszer-típusok esetén egy nézeten végzett módosítás visszavezetése. A(z) 2.8 ábra egy olyan mintát ábrázol, melynek akkor van illeszkedése, ha a forrás modellre legalább háromszor illeszkedik az *element_sensor* minta (F.3 függelék). Az előző példától eltérően a visszavezetés során egy ilyen minta esetén már nem határozható meg egy olyan szűk elemhalmaz, melynek módosításával a kívánt nézeti modell elérhető, mivel figyelembe

kell venni az *element_sensor* minta összes illeszkedését, mely *a modell módosításával (pl. újabb elemek hozzáadásával) változhat*. Fontos különbség az előző példához képest, hogy ebben az esetben lehetséges a modellt úgy módosítani, hogy az az illeszkedést ne befolyásolja, ugyanakkor a visszavezetés során bejárandó állapotteret növelje. ***Mivel a bejárandó állapottér a minta definíciójától függetlenül is nőhet (a modell módosításával), így ezen típusba tartozó minták nem támogatottak annak érdekében, hogy a következtető jól kezelhető problémákkal tudjon dolgozni.***

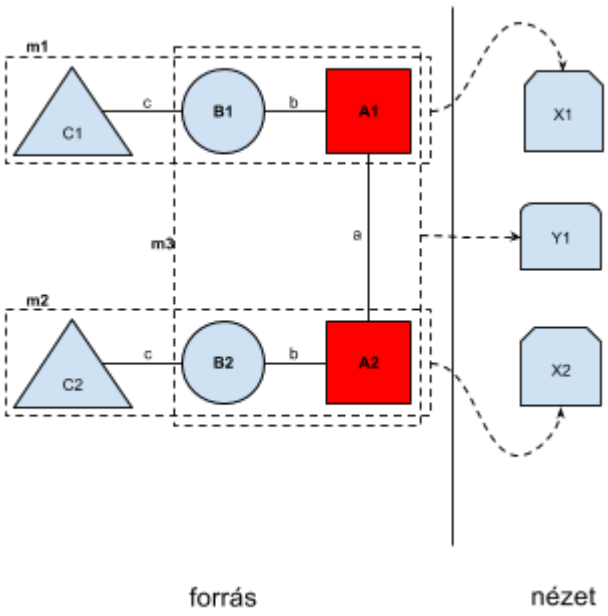
Megjegyzés: a korábbi példa esetén is természetesen lehetséges olyan nagyméretű mintát definiálni, melynek visszavezetése már túlságosan összetett problémát eredményezne, ugyanakkor azzal az előfeltételezéssel éltünk, hogy a felhasználás során ilyen komplex minta nem kerül megfogalmazásra (ehhez az szükséges, hogy a minta illeszkedése legalább 30 elemet tartalmazzon).

A következőkben a *rejtett változók (*lásd 2.3.1.1*)* okozta nehézségeket mutatom be a visszavezetés során. Legyen *p1* és *p2* minta a következők szerint adott:

```
pattern p1 (a : A) {
	A.b.c(a, _);
}

pattern p2 (a1 : A, a2 : A) {
	A.a(a1, a2);
	A.b(a1, _);
	A.b(a2, _);
}
```

Látható, hogy mind a *p1* mind pedig a *p2* minta paraméterei között csak az illeszkedésben szereplő *A* típusú elemek található meg. A *p1* minta esetében elmondható, hogy a *b* és *c* referenciákon keresztül az illeszkedésekben lesz két olyan elem, melyek a paraméterek között nem jelennek meg (vagyis az illeszkedés szempontjából *rejtettek*), míg a *p2* mintára az igaz, hogy a *b* referencián keresztül ugyancsak lesz egy-egy rejtett elem az illeszkedésében. Egy lehetséges problémát szemléltet a(z) 4.8 ábra. Adott a *p1* minta egy-egy illeszkedése (m1={A1, B1, C1}, m2={A2, B2, C2}), valamint a *p2* minta egy illeszkedése (m3={A1, A2, B1, B2}). Az *IncQuery* keretrendszerben az egyes illeszkedések esetén csak a publikus elemek érhetőek el. Az m*1* illeszkedés esetén ez {A1}, *m2* illeszkedés esetén {A2} elem lesz, míg az *m3* illeszkedés esetén pedig {A1, A2} elemek lesznek elérhetők (ezek az illeszkedések *publikus elemei*). Amennyiben a visszavezetés során csak a *publikus elemek* körében keressük a megoldásokat, úgy ***előfordulhat, hogy nem találunk helyes visszavezetést, vagy nem minden helyes visszavezetést találunk meg***.



4.8. ábra: Rejtett változók a visszavezetés során

A(z) 4.8 ábrán bemutatott eset azt szemlélteti, hogy az *X1* elem törlésekor amennyiben csak az *m1* illeszkedés publikus részeit vesszük figyelembe a visszavezetés során, úgy nem fogunk helyes megoldást találni (miközben az létezik). Ennek az az oka, hogy pusztán az *A1* elem módosításával (vagy törlésével) habár megszüntethető az *m1* illeszkedés, az biztosan maga után vonja *m3* illeszkedés megszűnését is (vagyis *Y1* elem is törlődik a nézetről). A fenti probléma megoldása *C1*, *m1* illeszkedéshez tartozó *rejtett elem* törlése lenne a forrás modellből, mivel így csak az *m1* illeszkedés szűnne meg, melynek eredménye *X1* elem törlése lenne a nézeti modellről. A probléma úgy oldható meg, hogy a használt minták esetén gondoskodni kell az azokban található ***rejtett változók paraméterlistára történő kivezetéséről***, így a mintákhoz tartozó illeszkedésekben már nem lesznek *rejtett elemek,* ez által biztosítható a visszavezetés helyes működése.

A helyes visszavezetés megtalálása esetén gondot okozhatnak a *mintahivatkozások* is, mivel a hivatkozott minták is tartalmazhatnak további rejtett változókat. Amennyiben ezek esetén is ***megköveteljük a rejtett változók kivezetését a minta paraméterlistájára***, úgy ez a probléma is megoldható.

Végezetül pedig meg kell említenem a diszjunkciót tartalmazó mintákat is, mivel ezek esetén is bizonyos kritériumokat kell szabnunk a helyes visszavezetés megtalálása érdekében. Ebben az esetben is igaz, hogy ha ***a mintához tartozó egyetlen törzsben sem létezik rejtett változó, akkor a logikai következtető minden helyes visszavezetést megtalál***.

4.2.4. Logikai probléma felírása

Ebben a fejezetben a logikai következtető számára definiált probléma felírásához használt bemenetek (lásd 4.2.2) szükségességének egy részletesebb magyarázata következik.

Ahogy az már korábban említésre került, a következtetők alapvetően úgy dolgoznak, hogy olyan modelleket próbálnak előállítani, melyek megfelelnek a megfogalmazott követelményeknek. Elmondható ugyanakkor az is, hogy amennyiben egy nézet módosítása során előálló probléma megoldása a feladat, úgy *olyan helyes visszavezetések megtalálása a cél, mely minél kisebb forrás oldali módosítással éri el a kívánt eredményt* (vagyis azt a helyes forrás modellt, melynek transzformált képe éppen a módosított nézet). Ennek biztosításához az szükséges, hogy a forrás modellen ki tudjuk jelölni azt a legkisebb *változtatható részt*, melynek módosításával a feladat megoldható. A modell többi része (az úgynevezett *fix rész*) változatlan marad, ezzel biztosítva, hogy a visszavezetés a lehető legkevesebb módosítást tartalmazza.

További előnye az előbbi megközelítésnek az is, hogy a következtető így hamarabb képes megoldani a problémát, mivel kisebb a bejárandó állapottér. Ennek ellenére szükséges megadni a következtető számára egy felső korlátot is, melynek célja, hogy amennyiben a bejárandó állapottér végtelen nagyra nőhetne, annak egy felső korlátot adjon a létrehozható elemek számának szabályozásával (ugyanis ha véges sok elem hozható létre a forrás modellben a megoldás keresése során, akkor előbb utóbb ezek összes kombinációja kipróbálásra kerül).

Fontos megemlíteni továbbá azt is, hogy a probléma definiálása során minden esetben szükséges megadni a forrás modell struktúráját definiáló metamodellt, valamint – amennyiben azok léteznek – egyéb jólformáltsági kényszereket. Erre azért van szükség, mert előfordulhat olyan eset, amikor a következtető által talált megoldás egy olyan modell, melynek a transzformált képe megfelel a módosított nézetnek, ugyanakkor nem felel meg a metamodell illetve a jólformáltsági kényszerek által szabott strukturális kritériumoknak.

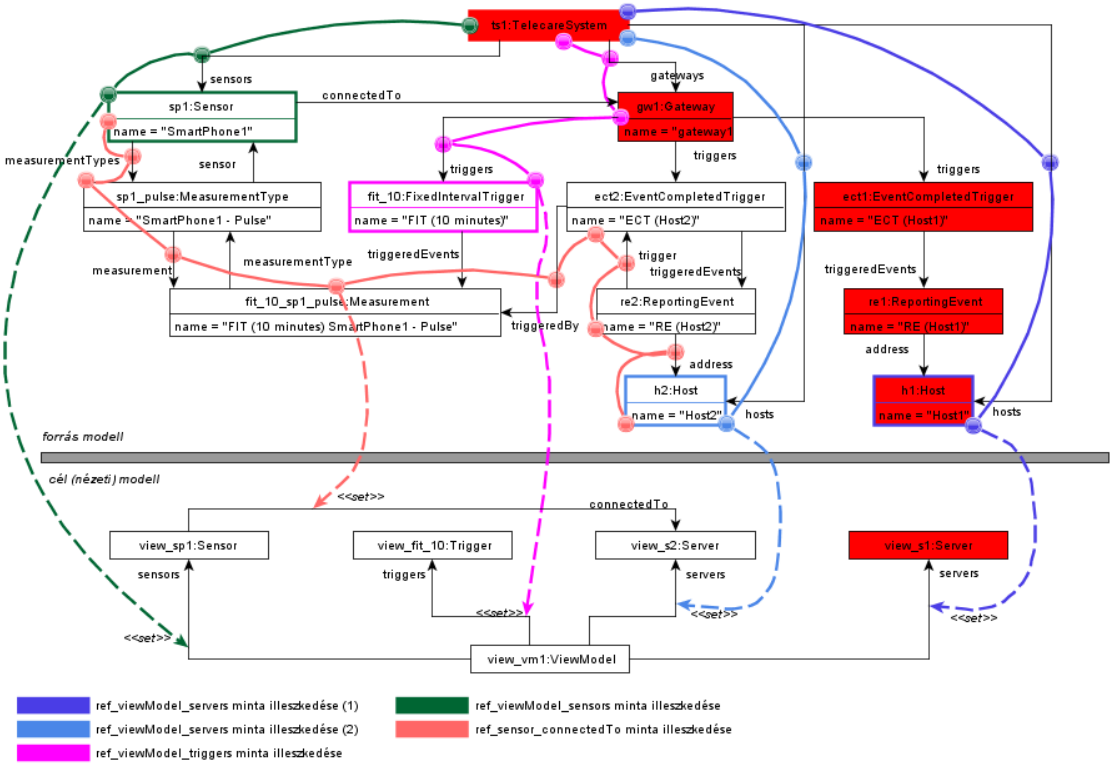
Végezetül pedig szükséges megadni természetesen azon illeszkedéseket (és a hozzájuk tartozó mintákat), melyek megléte szükséges a forrás modell *változtatható részében*. Az illeszkedések megadása azért szükséges, mert a forrás modellen végzett változtatások esetén figyelembe kell venni, hogy csak azok a megoldások helyesek, melyekben

megtalálhatóak a megadott illeszkedések, mivel ezek hiányában nem a megfelelő nézeti modellt kapnánk (a *fix részre* vonatkozó illeszkedések megadása természetesen nem szükséges, mivel ez a rész nem változhat a megoldás keresése során).

Amennyiben a logikai problémát a fentieknek megfelelően definiáljuk, úgy egészen biztos, hogy ha a tételbizonyító megáll, és talál megoldást az helyes lesz, amennyiben megáll, és nem talál megoldást, a definiált probléma ellentmondásos (vagyis a *nézet inkonzisztens*) vagy a megadott korlát mellett nem létezik megoldás.

4.2.5. Működés szemléltetése

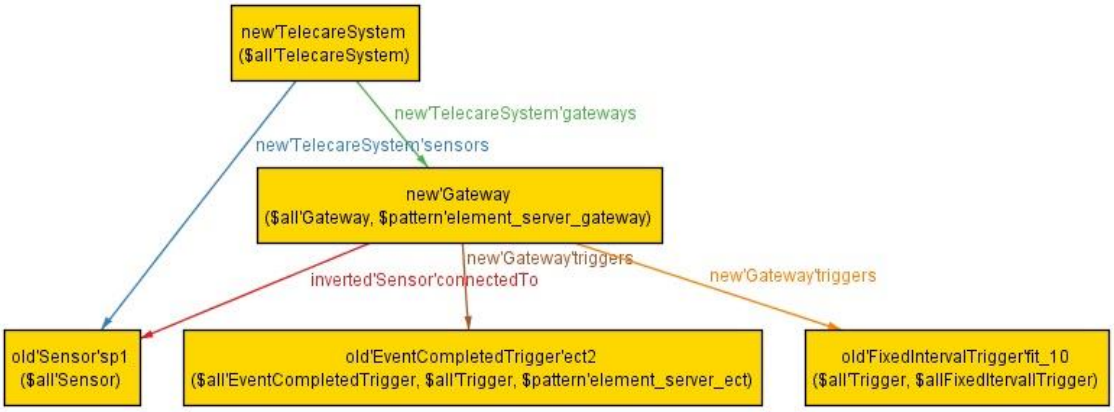
A visszavezetés logikai számítására mutat példát a(z) 4.9 ábra. Az ábrán a korábbiakban bemutatott kiindulási modell és nézeti modell látható, melyen a felhasználó a nézeti modellt kívánja szerkeszteni. Először egy törlés műveletet, majd egy beszúrás műveletet mutatunk be.



4.9. ábra: Visszavezetés által érintett modellrészlet (piros)

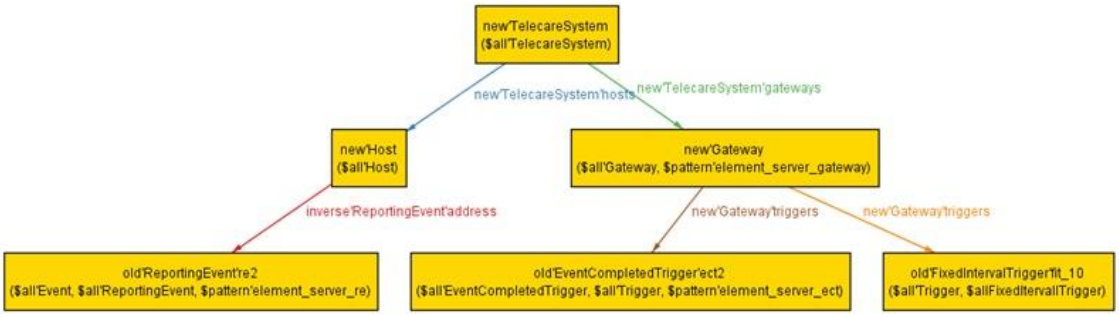
I. ***Server törlése:*** a felhasználó a nézeti modellből törli a {view_s1} elemet. Ennek hatására a pirossal jelölt rész kerül kijelölésre a visszavezetés számítása során, amelyen a logikai következtető dolgozik. A következtető

kételemű *scope*-ra ad először megoldást, melyet az alábbi ábrán szemléltetünk. A „*new”* kezdetű elemek kerülnek a pirosak helyére, míg az „*old*”-al jelölt elemek változatlanok maradnak. Ezek szerint az eredeti modellen az {ect1}, {re1} és a {h1} elemek törlésével kaphatjuk meg legegyszerűbb módon a kívánt változtatást.



4.10. ábra: A következtető által megtalált egyik helyes megoldás (törlésre)

II. ***Server beszúrása:*** miután a *Server* elemet töröltük, egy újabb *Server* elem beszúrását kezdeményezi a felhasználó az absztrakt nézeten. Ilyenkor a következtető megkeresi hogyan lehet a legkevesebb számú új elem beszúrásával megvalósítani a kívánt nézetet. Ilyenkor egy háromelemű változtatást javasol, ahol a *TelecareSystem* és *Gateway* mellett csupán csak egy újabb *Host*-elem létrehozását javasolja, melyet közvetlenül az {re2} elemhez csatol. A javasolt modellt az alábbi ábra szemlélteti. **Fontos megjegyezni, hogy az ismételt beszúrásra más, nézeti modelljében viszont megegyező megoldást javasol.**



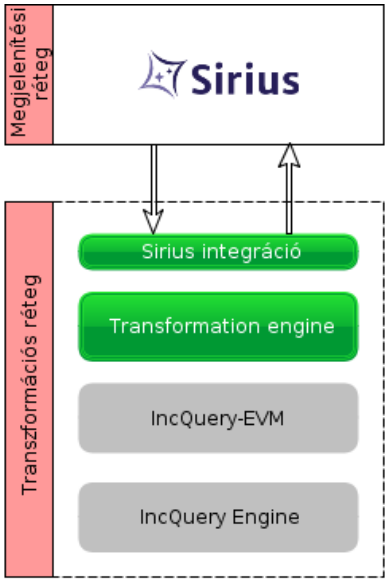
4.11. ábra: A következtető által megtalált egyik helyes megoldás (létrehozásra)

4.2.6. Összefoglalás

Ebben a fejezetben bemutattam, hogyan használhatók logikai következtető eszközök a származtatott nézeti modelleken végzett módosítások visszavezetése során. Dolgozatomban ismertettem azon *IQPL* kényszer-típusok halmazát, melyek támogatottak a visszavezetés számításakor, valamint kitértem a visszavezetés során módosítható forrás oldali elemhalmaz meghatározásának módszerére, továbbá az ott felmerülő nehézségekre. Logikai következtetők modellvezérelt fejlesztés során történő használatáról konzulensem kapcsolódó publikációjában[15] lehet bővebben olvasni.

4.3. Integráció

A tervezés során nagy hangsúlyt fektettem arra, hogy olyan eszközt készítsek, mely könnyen integrálható más, *EMF*-alapú technológiákba. Erre a feladatra viszont nem létezik olyan általános megoldás, mely minden esetben garantálná a helyes működést, így az elkészített transzformációs keretrendszer integrálásához a különböző eszközökhöz specifikus integrációs modul szükséges.



4.12. ábra: A transzformációs keretrendszer integrációja

Munkám során megvalósítottam az elkészített transzformációs keretrendszer integrációját a széles körben használt *Sirius* eszközbe, melyet a(z) 4.12 ábra szemléltet. Látható, hogy a keretrendszer a korábban leírtak szerint támaszkodik az *EMF-IncQuery* eszköz által biztosított szolgáltatásokra, míg az integrációért felelős modul a keretrendszer és a *Sirius* eszköz között teremti meg a kapcsolatot.

Mivel a *Sirius* alapvetően nem képes absztrakt nézetek definiálására (*több-egy* leképzésre a forrás- és nézeti modell között), így a célom az volt az integrációval, hogy ezt a funkciót az általam fejlesztett keretrendszerre támaszkodva elérhetővé tegyem az eszközben. Megközelítésemmel a *több-egy* leképzés olyan módon valósítható meg, hogy definiálunk egy transzformációt, mely előállítja a kívánt absztrakt nézeti modellt, majd ezen modellhez létrehozunk egy *Sirius*-os felületet (például egy diagram típusút, lásd 2.6 fejezet). Az integrációért felelős komponens az alábbiakat biztosítja:

- a felület megnyitásakor elvégzi a transzformációt és gondoskodik a származtatott nézeti modell megjelenítéséről,
- a forrás modell változtatása esetén automatikusan frissíti a nézeti modellt, és ezzel együtt a megjelenített diagramot is.

További lényeges, kapcsolódó fejlesztés volt az *IncQuery* által definiált nyelv (*IQPL – IncQuery Pattern Language*, lásd 2.3.1.1) integrálása a *Sirius* eszközbe, melynek köszönhetően a(z) 2.6 fejezetben bemutatott nyelvek (*AQL*, *OCL*, stb.) mellett már annak használatára is van lehetőség az egyes felületek definiálása során.

5. Mérés

5.1. Transzformáció sebességének mérése

Az elkészített rendszer teljesítményét a bemutatott *CONCERTO* esettanulmányhoz tartozó modellek segítségével mértem ki. A mérés során azt vizsgáltam, hogy a transzformáció bemenetének növekedése hogyan befolyásolja a transzformáció végrehajtásához szükséges időt, valamint megmértem a különböző méretű modellek esetén a forrás oldalon végzett módosítások nézeti modellre történő leképzésének idejét is.

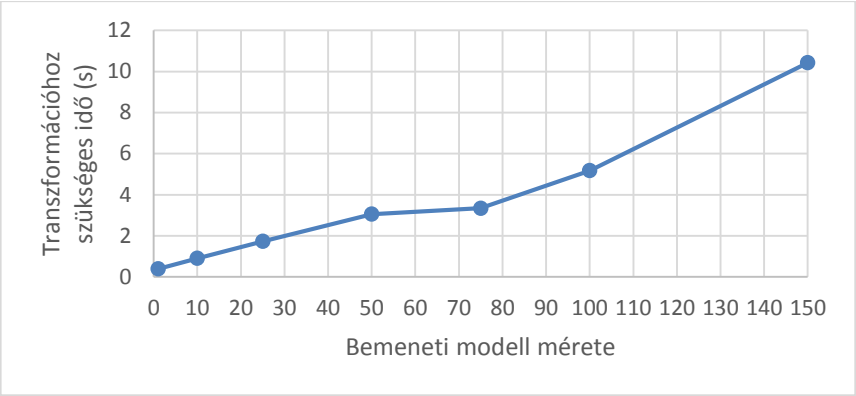
A mérést hét, különböző méretű modellen végeztem el, melyeket a(z) F.1 függelékben bemutatott modell sokszorozásával kaptam. Minden mérést ötször futtattam le, majd végül a kapott értékek mediánját vettem. A használt modellek a(z) 5.1 táblázatban összefoglalt tulajdonságokkal rendelkeznek. A táblázat alapján látható, hogy az egyes mérések során *33...4801* illeszkedés volt a forrás modellen, ami egyben azt is jelenti, hogy éppen ennyi művelet történt a nézeti modellen a transzformáció során.

Modell	Bemeneti modell elem-száma	Bemeneti modell éleinek száma	Kimeneti modell elem-száma	Kimeneti modell éleinek száma	Kimeneti modell attribútumainak száma	Minta illeszkedések
1x	38	89	9	16	8	33
10x	371	890	81	160	80	321
25x	926	2225	201	400	200	801
50x	1851	4450	401	800	400	1601
75x	2776	6675	601	1200	600	2401
100x	3701	8900	801	1600	800	3201
150x	5551	13350	1201	2400	1200	4801

5.1. táblázat: Méréshez tartozó bemeneti és kimeneti modellek főbb tulajdonságai

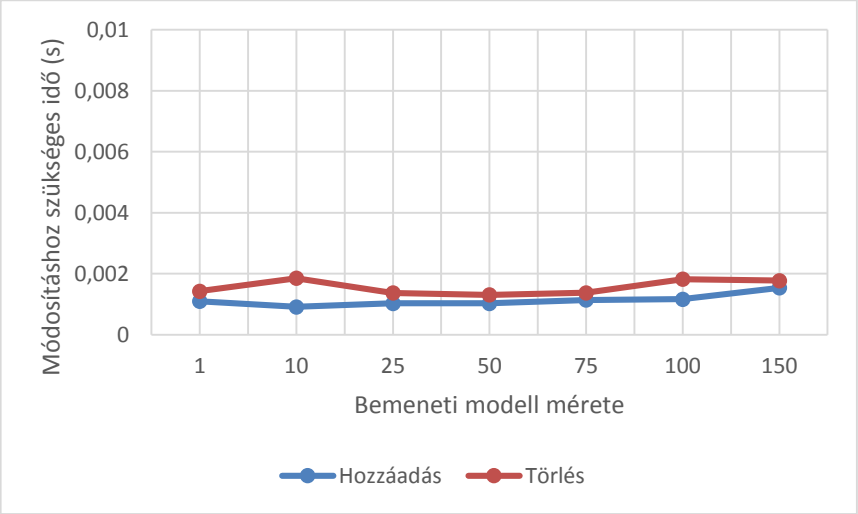
A mérések elvégzésére egy *Intel Core i7-M620 2.67GHz* processzorral és *4GB RAM*-mal rendelkező gépet használtam. Az eszközön *Debian 8 (Jessie)* operációs rendszer futott, a mérésekhez az *Eclipse 4.4 (Luna)* kiadását használtam.

A(z) 5.2 ábra a transzformáció futási idejét (függőleges tengely) szemlélteti a táblázatban bemutatott, különböző méretű (vízszintes tengely) bemeneti modellek esetén. Jól látható, hogy a bemeneti modell növekedésével közel lineárisan növekszik a végrehajtáshoz szükséges idő. Fontos még kiemelnem, hogy a mérési idő magában foglalja a bemeneti modellek betöltését valamint az *IncQuery* rendszer (2.3.1 fejezet) inicializálását is.



5.2. ábra: Transzformáció futási ideje különböző méretű modellekre (kezdeti transzformáció esetén)

A(z) 5.3 ábra azt mutatja, hogy az egyes modellek esetén végrehajtott módosítások hatása milyen gyorsan jelent meg a nézeti modellen. A módosítások során egy olyan elemcsoport beszúrására került sor a forrás modellbe, melynek hatására minden esetben megjelent egy-egy új illeszkedés, így a nézeten létrejött egy-egy új (*Server*) elem. A beszúrást követően az elemek törlésre kerültek a forrás modellből, melynek következtében a nézetről is törlődött a korábban létrehozott elem.

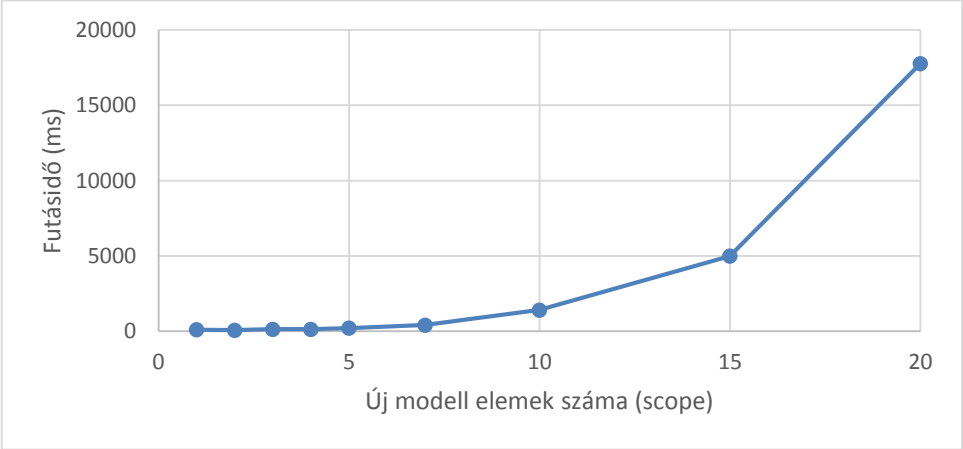


5.3. ábra: Transzformáció futási ideje különböző méretű modellekre (elemek hozzáadása és törlése esetén)

A mérés eredményéből az látható, hogy a modell méretétől függetlenül a transzformációs keretrendszer közel egységes (ezredmásodpercekben mérhető) idő alatt képes elvégezni a forrás modell módosításának hatására végrehajtandó műveleteket (elem létrehozása illetve törlése) a nézeti modellen. Ez természetesen a transzformáció inkrementális megvalósításának tudható be, minek köszönhetően egy-egy forrás oldalon végzett módosítás nem igényli a teljes transzformáció újbóli végrehajtását, csak azokét, melyek a módosításban érintettek.

5.2. Visszavezetés mérése

A mérés során a(z) 4.2.5 fejezetben bemutatott példa futásidejét mértük ki különböző méretű *scope*-okra, hogy kimérjük a mögöttes következtető skálázódását. A futásidők az alábbi ábrán láthatóak. A futásidő gyorsan növekszik, ami bizonyítja, hogy fontos kijelölni minél szűkebben a változtatható részeket, hiszen ezzel arányosan exponenciálisan nő a várható futásidő.



5.4. ábra: Visszavezetés futásideje különböző méretű *scope*-okra

Annak megmérése, hogy a nem változó részeknek a futásidőre kifejtett hatása milyen skálázódást mutat, a jövőbeni munka részét képezi, bár egyelőre optimista eredményeket mutat.

6. Kapcsolódó munkák

Jelenlegi state-of-the-art modellek szinkronizációjára két fő irányvonalat definiál az alapján, hogy a szinkronizáció csak a cél modellt képes frissíteni, vagy a cél modell változása esetén a forrás modellt is képes karban tartani. Ez alapján beszélhetünk egyirányú (csak cél modell frissítése), illetve kétirányú (cél- és forrás modell frissítése) transzformációról.

6.1.1. Egyirányú szinkronizáció

Egyirányú szinkronizáció esetében a forrás modellen végrehajtott módosítások hatására automatikusan frissítésre kerül a cél modell.

Query Based Resource (QBR)

A *QBR[16]* keretrendszer egy olyan gráfminta alapú modell transzformációt megvalósító eszköz, mely saját megközelítésemhez hasonlóan az *EMF-IncQuery* keretrendszerre épít. Az általam készített megoldás tervezésekor megvizsgáltam a *QBR* eszközt, és próbáltam azt úgy továbbfejleszteni, hogy végül egy általánosabb, jobban testre szabható eszközt kapjak.

A *QBR* rendszer esetében elmondható, hogy az egy szűkebb szabálykészlettel dolgozik, melyet saját megvalósításomban került kiterjesztésre. Az általa definiált *traceability* modellt is továbbfejlesztettem úgy, hogy képes legyen a kétirányú szinkronizáció támogatására.

Virtual EMF

A *Virtual EMF[17]* szintén egyirányú szinkronizációra képes, mely saját *EMF EObject* (*ECore* példány modellbeli elemek közös őse) és *Resource* (példány modelleket tartalmazó entitás) implementációval rendelkezik, mely *proxy*-ként viselkedik a cél modell irányából és transzparensen értékeli ki az aktuális értékeket.

Ellentétben az általam definiált megközelítéssel, nem módosítások hatására reagál a cél modell frissítése, hanem lekérdezés vezérelten működik. A forrás- és cél modellek közötti szinkronizációhoz *traceability* modellt nem használ.

Változásvezérelt modell transzformációk

Egyirányú szinkronizáció megvalósítására alkalmazhatóak változásvezérelt modell transzformációk[18]. Ez egy jóval komplexebb és általánosabb megoldást ad modellek szinkronizációjára. A transzformációk definiálására egy teljes transzformációs nyelvet használ, így nincsenek nyelvi korlátai a szinkronizációs szabályok meghatározására. Alapötlete, hogy a változások által triggerelt transzformációkat aszinkron módon hajtja végre. Ezeket a transzformációkat egy *Change History* modellben tárolja, és sorrendben egymás után dolgozza fel.

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

6.1.2. Kétirányú szinkronizáció

Kétirányú szinkronizáció esetén a forrás modell változásai megjelennek a cél modellen, valamint a cél modellen történő változások is szinkronizálódnak a forrás modellel.

Query / View / Transformation (QVT)

A *Query/View/Transformation (QVT)* egy *OMG* által meghatározott deklaratív nyelv, mellyel egyirányú és kétirányú transzformációt is definiálhatunk. Transzformációk definiálása során explicit meg kell adni mindkét irány származtatásához szükséges szabályokat. Ezeket a szabályokat meghívhatjuk konzisztencia ellenőrzés miatt *check only* módban, majd az *enforce* módon történő végrehajtás esetén kerülnek véglegesítésre.

Az általam tervezett megközelítés esetén nincsenek explicit definiált vissza-irányú transzformációs szabályok, ehelyett logikai következtetők segítségével történik a módosítások visszavezetése.

Triple Graph Grammar (TGG)

A *Triple Graph Grammar (TGG)* esetén, hasonlóan a *QVT*-hez mindkét irányban definiálni kell a műveleteket, azonban itt a műveletek gráfmintákként realizálódnak. Előnye, hogy képes meglévő nézeti modelleket is összekapcsolni forrásmodellekkel, majd ezután ezeket inkrementálisan szinkronizálni.

A *TGG* is definiál *traceability* modellt, mely a kapcsolatot teremti meg a forrás- és cél modellbeli elemek között. Ezt a modellt azonban explicit definiálni kell a

szinkronizációs szabályok meghatározásával. Ezzel szemben az én megoldásom egy általános *traceability* modellt használ.

6.1.3. További technológiák

Elérhetőek olyan további, modellezést támogató eszközök, melyek segítségével jólformáltsági kényszereknek megfelelő modellek generálása végezhető el (akár egy félkész modellből kiindulva is). Ezen eszközök logikai következtetők segítségével próbálják a feladatot megoldani. Ezt a funkciót támogatja a *Formula[21]*, a *Clafer[22]* valamint az *Alloy[23]* is.

7. Összefoglalás

Munkám során elkészítettem egy olyan keretrendszert, melynek segítségével lehetséges (i) absztrakt nézeti modellek gráfminta alapú származtatása, valamint (ii) a nézeten végzett módosítások, forrás oldali visszavezetése. Ezen túlmenően a megvalósított rendszert integráltam is az iparban széleskörűen használt *Sirius* eszközbe.

A munkám során a tanszék által fejlesztett EMF-IncQuery eszközre építettem és ennek keretében az alábbi elméleti és gyakorlati eredményeket értem el:

- **Elméleti**
  - Meghatároztam azon modell transzformációs szabályok halmazát, amely esetében lehetséges SMT alapú forrás oldali modellek származtatását.
  - Meghatároztam azon kényszer-típusok halmazát, melyek támogatottak a nézeten végzett módosítások visszavezetése során;
  - Kidolgoztam egy eljárást, amellyel meghatározhatóak a visszavezetés során módosítandó forrás oldali elemhalmazok.
- **Gyakorlati**
  - Kidolgoztam egy olyan megoldást, mely az *IncQuery – EVM* rendszerre építve valósít meg inkrementális model transzformációt (építve konzulensem korábbi munkájára);
  - Integráltam a megoldásomat a Sirius keretrendszerbe a nézeti modellek megjelenítésére és manipulálhatóságára.
  - Méréseket végeztem az újonnan megvalósított előre irányú megközelítsem teljesítményének vizsgálatára egy iparilag releváns modellen.
  - Mérésekkel igazoltam a visszafele irányú leképezés alkalmazhatóságát egyszerű esetekben.

A munkám során egy könnyen bővíthető keretrendszert valósítottam meg, amelyet a jövőben az alábbi irányokban szeretnék kiterjeszteni:

- Megvalósítani egy automatikus leképezést a visszafele irányú módosítások számításhoz szükséges logikai kifejezések generálására.
- Kicserélni az SMT megoldót a Microsoft által megvalósított *Z3* megoldójára, amely hatékonyabban tudja kezelni a modellben nem változó részek ábrázolását.

Irodalomjegyzék

[1] Concerto honlap: http://www.concerto-project.org/

[2] ECore metamodell: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html

[3] EMF-IncQuery honlap: https://www.eclipse.org/incquery/

[4] Event-driven Virtual Machine dokumentáció: https://wiki.eclipse.org/EMFIncQuery/DeveloperDocumentation/EventDrivenVM

[5] Event-driven Virtual Machine áttekintő ábra: https://wiki.eclipse.org/File:EMFIncQuery_EventDrivenVM_overview.png

[6] EMF tranzakciós csomag honlap: https://projects.eclipse.org/projects/modeling.emf.transaction

[7] Alloy honlap: http://alloy.mit.edu/alloy/

[8] Z3 projekt: https://github.com/Z3Prover/z3

[9] Előadás jegyzet: http://inf.mit.bme.hu/edu/courses/mdsd/materials - Model and Graph Transformations

[10] Sirius honlap: https://eclipse.org/sirius/

[11] Acceleo honlap: https://eclipse.org/acceleo/

[12] OCL honlap: http://www.omg.org/spec/OCL/

[13] AQL honlap: https://www.eclipse.org/acceleo/documentation/aql.html

[14] Ehrig, Hartmut, Ulrike Prange, and Gabriele Taentzer. "Fundamental theory for typed attributed graph transformation." Graph transformations. Springer Berlin Heidelberg, 2004. 161-177. Debreceni Cs., „Automatikus absztrakció a modellvezérelt fejlesztésben”, TDK 2013, http://tdk.bme.hu/VIK/SW1/Automatikus-absztrakcio-a-modellvezerelt

[15] Semeráth, O., Barta, Á., Szatmári, Z., Horváth, Á., and Varró, D., "Formal Validation of Domain-Specific Languages with Derived Features and Well-Formedness Constraints", International Journal on Software and Systems Modeling, In press, 2015.

[16] Debreceni, Csaba, et al. "Query-driven incremental synchronization of view models." Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling. ACM, 2014.

[17] Brunelière, Hugo, and Grégoire Dupé. "Virtual EMF-transparent composition, weaving and linking of models." EclipseCon Europe 2011. 2011.

[18] Ráth, István, Gergely Varró, and Dániel Varró. "Change-driven model transformations." Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, 2009. 342-356.

[19] QVT honlap: http://www.omg.org/spec/QVT/

[20] Giese, Holger, and Robert Wagner. "Incremental model synchronization with triple graph grammars." Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, 2006. 543-557.

[21] Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Reasoning about metamodeling with formal specifications and automatic proofs. In: Proc. of the 14th Int. Conf. on MODELS, LNCS, vol. 6981, pp. 653–667 (2011)

[22] Bak, K., Czarnecki, K., Wasowski, A.: Feature and metamodels in clafer: Mixed, specialized, and coupled. In: 3rd International Conference on Software Language Engineering. Eindhoven, The Netherlands (2010). DOI 10.1007/978-3-642-19440-5 7

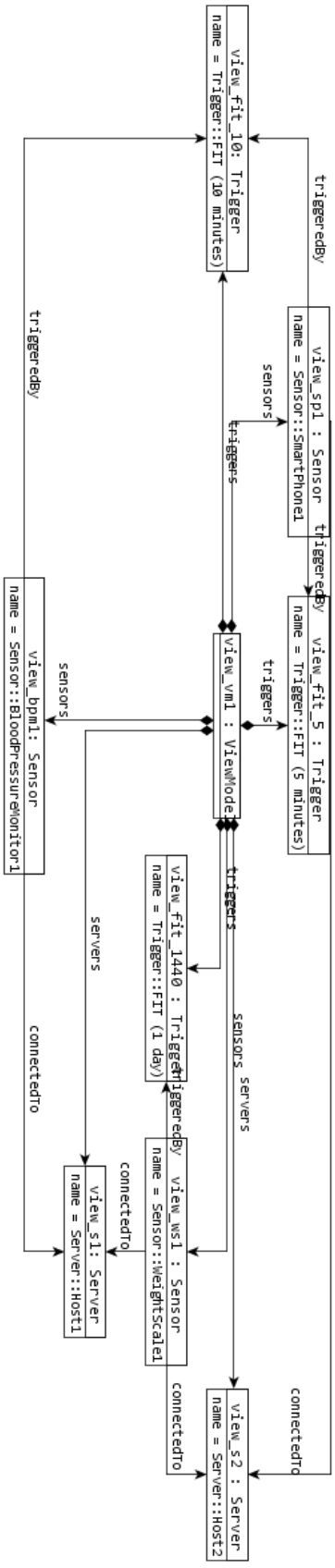
[23] Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into use. In: TOOLS'11 - Objects, Models, Components and Patterns, LNCS, vol. 6705, pp. 290–306 (2011)

Függelék

F.1. Telecare rendszer minta modell (forrás modell)

Mivel a hivatkozott modell túlságosan nagyméretű, így azt az alábbi címen tettem elérhetővé a jobb olvashatóság érdekében: https://inf.mit.bme.hu/tdk15_lengyela

F.2. Telecare rendszer minta modell (nézeti modell)



F.3. Transzformáció során használt minták (elemek létrehozása / törlése)

```
pattern element_viewModel(telecareSystem : TelecareSystem) {
	TelecareSystem(telecareSystem);
}


pattern element_server(host : Host) {
	TelecareSystem.gateways(_, gateway);

	Gateway.triggers(gateway, ect);

	EventCompletedTrigger(ect);
	EventCompletedTrigger.triggeredEvents(ect, re);
	ReportingEvent.address(re, host);
}


pattern element_trigger(trigger : FixedIntervalTrigger) {
	FixedIntervalTrigger.triggeredEvents(trigger, measurement);
	Measurement(measurement);

	find eventCompletedTriggerWithMeasurement(_, measurement);
}


pattern element_sensor(sensor : Sensor) {
	Sensor.connectedTo(sensor, gateway);

	Gateway.triggers(gateway, ect);
	Gateway.triggers(gateway, fit);

	EventCompletedTrigger.triggeredEvents(ect, re);
	ReportingEvent.address(re, _);

	FixedIntervalTrigger.triggeredEvents(fit, measurement);
	EventCompletedTrigger.triggeredBy(ect, measurement);
	Measurement.measurementType.sensor(measurement, sensor);
}


private pattern eventCompletedTriggerWithMeasurement(ect :
EventCompletedTrigger, m : Measurement) {
	EventCompletedTrigger.triggeredBy(ect, m);
	EventCompletedTrigger.triggeredEvents(ect, re);

	ReportingEvent(re);
}
```

F.4. Transzformáció során használt minták (referenciák beállítása / törlése)

```
pattern ref_viewModel_servers(telecareSystem : TelecareSystem, host :
Host) {
	find element_viewModel(telecareSystem);
	find element_server(host);

	TelecareSystem.hosts(telecareSystem, host);
}


pattern ref_viewModel_triggers(telecareSystem : TelecareSystem, trigger
: FixedIntervalTrigger) {
	find element_viewModel(telecareSystem);
	find element_trigger(trigger);

	TelecareSystem.gateways.triggers(telecareSystem, trigger);
}


pattern ref_viewModel_sensors(telecareSystem : TelecareSystem, sensor :
Sensor) {
	find element_viewModel(telecareSystem);
	find element_sensor(sensor);

	TelecareSystem.sensors(telecareSystem, sensor);
}


pattern ref_sensor_connectedTo(sensor : Sensor, host : Host) {
	find element_sensor(sensor);
	find element_server(host);


	Sensor.measurementTypes.measurement(sensor, measurement);
	EventCompletedTrigger.triggeredBy(ect, measurement);
	ReportingEvent.trigger(re, ect);
	ReportingEvent.address(re, host);
}


pattern ref_sensor_triggeredBy(sensor : Sensor, trigger :
FixedIntervalTrigger) {
	find element_sensor(sensor);
	find element_trigger(trigger);

	FixedIntervalTrigger.triggeredEvents(trigger,measurementEvent);
	Measurement.measurementType.sensor(measurementEvent, sensor);
}
```

F.5. Transzformáció során használt minták (attribútumok beállítása / törlése)

```
pattern attr_server_name(host : Host, value) {
	find element_server(host);

	Host.name(host, name);

	value == eval ("Server::" + name);
}


pattern attr_trigger_name(trigger : FixedIntervalTrigger, value) {
	find element_trigger(trigger);

	Trigger.name(trigger, name);

	value == eval ("Trigger::" + name);
}


pattern attr_sensor_name(sensor : Sensor, value) {
	find element_sensor(sensor);

	Sensor.name(sensor, name);

	value == eval ("Sensor::" + name);
}
```

F.6. Kényszerek

Csak a FixedIntervalTrigger tartalmazhat Measurement elemet.

Csak az EventCompletedTrigger tartalmazhat ReportingEvent elemet.