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

Dávid István

Testreszabható modelltranszformációk

Konzulens
Gönczy László

Budapest, 2009

Absztrakt

Az informatikai rendszerek modellvezérelt fejlesztése napjainkban egyre gyakrabban használt paradigmává válik. Ennek oka az, hogy a modellezés során viszonylag magas absztrakciós szinten lehet megadni a tervezett rendszer jellemzőit (pl. biztonsági követelmények), ami megkönnyíti a rendszerek korai analízisét, emellett alapul szolgálhat automatizált kódgeneráláshoz. A modellvezérelt fejlesztés hátránya lehet azonban, hogy gyakran mosódnak össze az informatikai, illetve a domain-specifikus tudást igénylő szerepkörök, hiszen a modellezési folyamat a két terület határán helyezkedik el. További problémát okozhat a modellből forráskódot, vagy egyéb konfigurációs leírókat generáló transzformációk fejlesztésének munkaigénye és hordozhatósága – gyakori eset, hogy a transzformációk nem általánosíthatók megfelelő szinten, így ez a lépés sok munkával járó és redundáns transzformációkat eredményező folyamattá válhat.

TDK dolgozatom során egy olyan koncepció kidolgozását és VIATRA keretrendszerbeli implementációval történő megvalósítását tűztem ki célul, amely a fenti két problémára megoldást nyújt. A TDK során megalkotok egy egyszerűen használható, szakterület-specifikus modellezési nyelvet, melyet mérnöki modellekhez csatolva részben modellhelyességi követelményeket adhatunk meg, részben a transzformáció működését szabályozhatjuk oly módon, hogy a modellvezérelt fejlesztési folyamat lépésétől és az adott szakterülettől függően dolgozza fel a forrásmodell elemeit. Így lehetővé válik a transzformációs váz megírásának és a specifikációs információ megadásának szétválasztása, a keletkező transzformáció pedig hordozható lesz, alkalmazható különböző célplatformokra és a fejlesztési folyamat különböző lépéseiben (helyességellenőrzés, teljesítményelemzés, kód/konfigurációgenerálás).

A módszer technológiai megvalósításához a szolgáltatásorientált rendszerek (SOA) területét választottuk, a rendszerek nem-funkcionális tulajdonságainak leírására koncentrálva. Ezen a dinamikusan fejlődő területen számos, gyorsan változó szabvány és implementációs platform érhető el, így megfelelő hátteret biztosít az eredmények bemutatására. Munkám során egy elérhető transzformációs lánc egyszerűsítésén és általánosításán keresztül mutatom be a módszer előnyeit és alkalmazom a transzformációt különböző webszolgáltatás szabványoknak megfelelő konfigurációk analízisére és előállítására.

Tartalom

1. Bevezetés

A szolgáltatásorientált architektúra (SOA, *Service-Oriented Architecture*) alapelveit követő rendszerek napjainkra a nagy és komplex alkalmazások alappilléreivé váltak viszonylag fiatal koruk ellenére. E gyors fejlődés nyomán létrejövő hiányosságok azonban alapjaiban határozzák meg ezen rendszerek főbb kutatási-fejlesztési irányait.

A hiányosságok a biztonság, megbízhatóság, teljesítőképesség, stb. kapcsán merülnek fel – ezen jellemzőket közös néven a szolgáltatásorientált rendszerek **nem funkcionális paramétereinek** nevezzük (NFP*, non-functional properties*)[1][2], amelyek a rendszernek általában nem a struktúrájából, nem a funkcionalitásából adódó jellemzők - innen a megnevezés.

Ezen jellemzők definícióit a rendszerek konfigurációs fájlok formájában tartalmazzák, amelyek létrehozhatók manuálisan, vagy választhatunk egy sokkal hatékonyabb módszert is.

A modellvezérelt fejlesztés (MDD*, Model-Driven Development*) sok olyan tulajdonságot hordoz magában, amik a SOA rendszerek nem funkcionális paramétereit már modellezési szinten megfoghatóvá, tervezhetővé teszik. Ilyen módon a rendszerek kritikus pontjai már korai tervezési fázisban elemezhetőkké válnak, analízisük megfelelő eszközökkel megvalósítható. A modellvezérelt fejlesztés során az implementáció nem manuális módon áll elő, hanem automatizált leképezések, úgynevezett **modelltranszformáció**k segítségével jön létre. Az implementáció jelen dolgozat esetén az Apache Axis2[21][22][23][35] platformra létrehozott konfigurációs állományokat jelenti.

Korábbi munkáim során[3][4] tanulmányoztam a modellalkotási és transzformáció-fejlesztési folyamatokat. A munkák eredményei a 2009-es *SENSORIA Summer School* előadás- és labor-szekciójában is szerepeltek, mint a fejlesztési módszerek bemutatására alkalmas mintapéldák[4]. A jelenlegi eredmények egy része (modellek teljesség- és helyességellenőrzése a modelltranszformációk szintjén) ott már bemutatásra került egy előzetes verzióban.

Ezen munkálatok során szerzett tapasztalatok mutattak rá, hogy amennyit a MDD által a kód automatikus előállításával nyerünk, azt szinte elveszítjük a modelltranszformációk implementálásánál, mert ezen a transzformációk előállítása legalább annyira erőforrás-igényes, mintha az egész rendszer kódját kézzel írnánk meg.

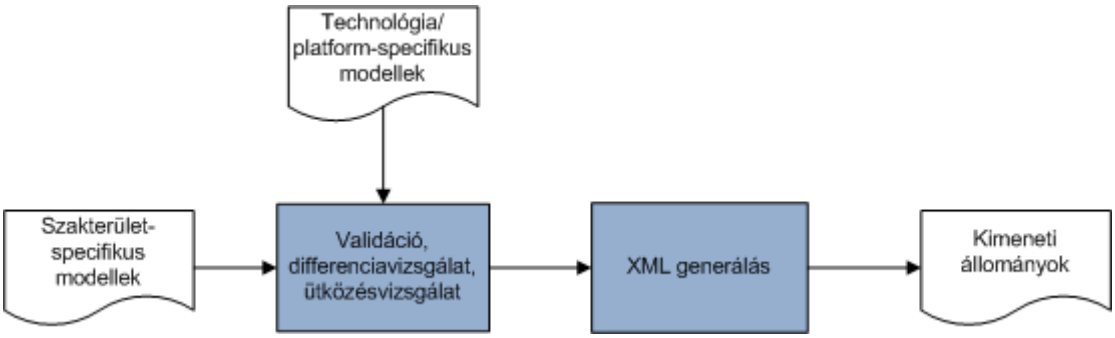
1.1. Problémafelvetés

A szolgáltatásorientált rendszerek nem funkcionális paramétereinek tervezhetővé tétele kritikus a rendszer későbbi implementációjának szempontjából. Hiszen ha egy rendszert nem tervezünk eleve pl. biztonságosra, akkor később nagyon nehéz és költséges feladat lesz biztonságossá tenni.

A nem funkcionális jellemzőket leíró konfigurációs állományok fejlesztését célszerűen a modellvezérelt fejlesztés eszközeivel végezzük. Ezek az eszközök a modelltranszformációk, amik azonban nem elég rugalmasak ahhoz, hogy költséghatékony megoldások legyenek.

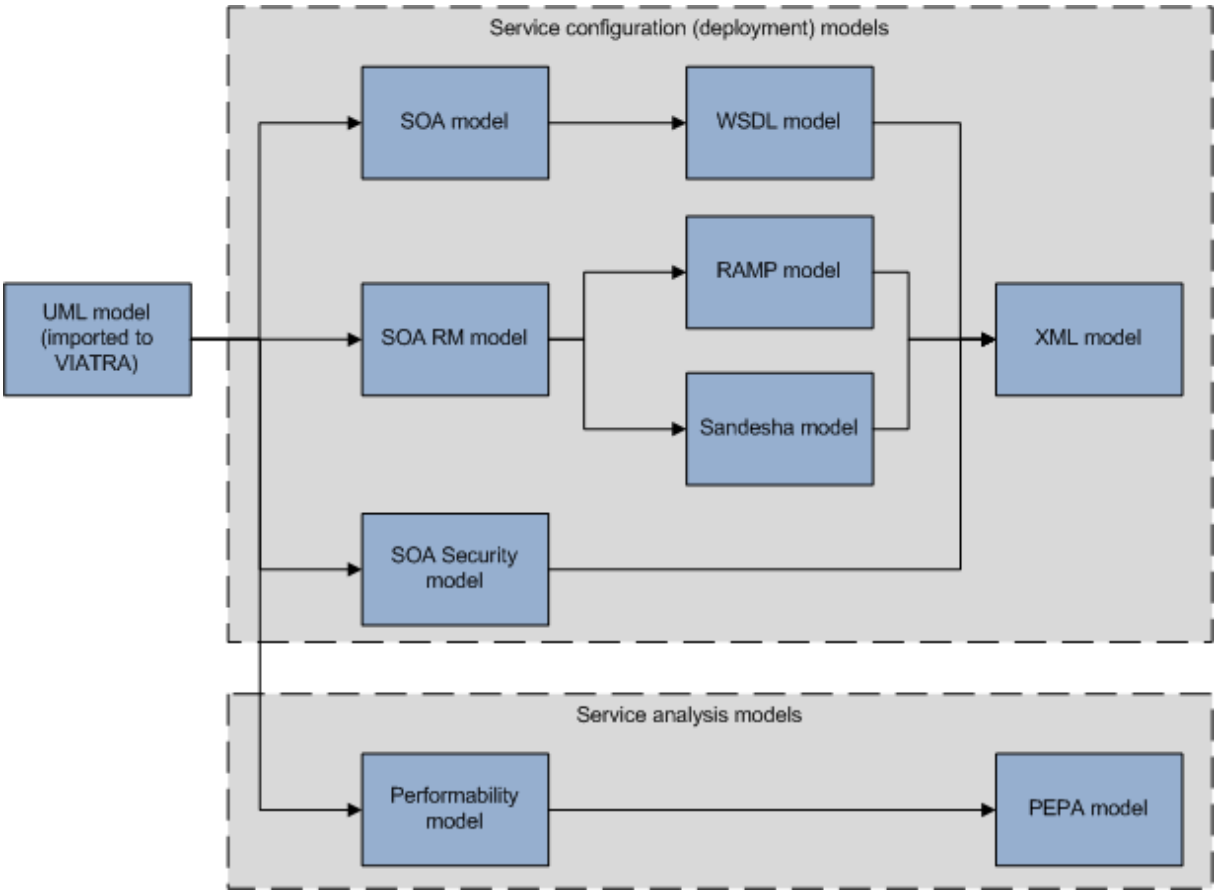
Az 1. ábrán egy olyan modelltranszformációs folyamat látható, ahol a bemeneti állományok validáció, valamint differencia- és ütközésvizsgálat után a kimeneten generált XML-t adnak. Ennél a felépítésnél a transzformációk általános műveleteket végeznek el még akkor is, ha egy adott típusú elemet kell generálniuk. Ennek ára azonban az, hogy a modellterek megnőnek és az a fajta komplexitás, ami

eddig a transzformációkat jellemezte, most a modelltérben fog megjelenni. Ez azonban egy hatékonyabb megoldás, hiszen modellezési szakemberből több van, mint a modelltranszformációkat összeállítani képes szakemberekből – ezzel tehát a munka nagyobbik részét áthelyezzük a modellezőkre.



1. ábra - Szeparált modellek modelltranszformációs feldolgozási folyamata.

Az [1] irodalom többek között a SOA konfigurációs állományok modellvezérelt fejlesztésével is foglalkozik. A publikációban összeállított transzformáció folyamata látható a 2. ábrán.



2. ábra – Egy modelltranszformációs folyamat.[1]

Látható, hogy az *XML model* kimenethez, ami a konfigurációs állomány, több olyan transzformáción is keresztül kell vinni a modellt, amik speciálisan egy feladatra készültek el. Az ilyen transzformációs blokkok karbantartása, változtatása, továbbfejlesztése igen nehéz és költséges folyamat.

Ezen folyamat (költség)hatékonnyá tehető, ha a transzformációkat rugalmassá tesszük. Erre két módszer adódik: egyrészt magukat a transzformációkat alakítjuk át az újrahasznosíthatóságot az előtérbe véve (8.1.fejezet), vagy a bemeneti modellt alakítjuk át úgy, hogy az viszonylag magasan absztrahált transzformációkkal legyen transzformálható. E módon az előzőekben látott transzformációs folyamat egyes blokkjai helyett a modell tartalmaz majd olyan megkötéseket, amiknek segítségével a transzformáció végbemehet. A kellően magas szintű transzformációk érdekében az eddigi egy darab bemeneti modellt több részre választjuk szét (*model separation*).

Az 1. ábrán látható a folyamatban a bemeneti adatok a *Szakterület-specifikus modell* és a *Technológia-specifikus modell*. A két modellt a transzformációs folyamat összefésüli (*merge*) és a validáció után kimeneti kóddá képzi le.

A 2. ábrán megfigyelhető a *Service configuration models*, tehát a telepíthető/konfigurációs modellek mellett a *Service analysis models*, azaz a szolgáltatások analízisét lehetővé tevő modellek. A szolgáltatásorientált rendszerek egyik fő kutatási területe napjainkban épp azok teljesítőképességének (*performability*) vizsgálata.[5][6]

1.2. Célkitűzés

Dolgozatomban célként tűzöm ki a **modelltranszformációk testreszabhatóságának modell-szeparáción** alapuló megvalósítását. Így a cél egy olyan **modellezési nyelv** és egy olyan **módszer kidolgozása**, amely segítségével a modellalkotás során úgy különülnek el egymástól a modellek, hogy azok minél inkább elősegítsék a modelltranszformációk általánosíthatóságát.

A modelleket ezért két nagy csoportra osztjuk: technológia- és szakterület-specifikus modellekre, amelyek közül a technológia-specifikus modellek egy szakterületen belül nagyon ritkán változó, állandó modellek, míg a szakterület-specifikus modellek alkalmazásról-alkalmazásra változhatnak.

A két modellcsoport megalkotása egymástól független folyamat, így lehetőség nyílik arra, hogy azokat két különböző ismeretekkel rendelkező tervező – egy technológia-specifikus tudással rendelkező **integrációs szakértő**, és egy **szakterület**-specifikus tudással rendelkező **specialista** (aki azonban minimális informatikai tudással is rendelkezik) – állítsa össze. Célként tűzöm ki ezért a modellek **célterületei közti átfedés minimalizálását**, azaz hogy a technológia-specifikus modell összeállításához minél kevesebb szakterület-specifikus tudásra legyen szükség, valamint a szakterület-specifikus modellhez egyáltalán ne legyen szükség platform-specifikus ismeretekre.

Technológiai oldalról cél egy olyan **UML-profil létrehozása**, amely az UML4SOA[7] profillal együtt segíti a modellalkotási folyamatot.

További cél a szeparált **modellek ellenőrzéseinek** és **összefésülésének megvalósítása** a transzformációk szintjén, azaz a **modellszintézis** folyamatának definiálása és kialakítása.

Végezetül célként tűzöm ki az **elméleti** technikák **prezentálását** egy rövid fejlesztési feladat végigvezetésével a modellalkotástól kezdve a implementációk prototípusainak létrehozásáig bezárólag.

1.3. Eredmények

A dolgozat során a következő eredmények előállítása a cél:

- **egy modellalkotási módszer, amely támogatja a modelltér technológia-szakterület irányú szeparációját**
  Ezen belül meg kell alkotnom a módszerhez egy konkrét modellezési nyelvet, annak szintaktikai, szemantikai alapelemeit – ezek, mivel grafikus modellezésről van szó, metamodellekkel lesznek megadva. A nyelvet egy UML Profillal kell támogatni, amely a modellezési munkákat segíti.
  A teljességi- és modellhelyességi elvek definiálása után meg kell határozni az analizációs és validációs elveket a transzformációk szintjén.
- **kódgenerálási folyamatok (transzformációk) előállítása**

A fenti eredmények után szükséges a módszer a módszer kipróbálása egy adott esettanulmány alapján. Itt a modellezési folyamatot alaposan megvizsgálva és elvégezve, a modelltranszformációk gy-egy prototípus készítve tesztelem le a módszer hatékonyságát.

Fontos megjegyezni, hogy az alábbiakban bemutatott módszerek nem csupán a szolgáltatásorientált rendszerek modellvezérelt fejlesztése esetén működnek, hanem a platform-függetlenség okán más típusú rendszerekre, alkalmazásokra, valamint más aspektusú modelltranszformációkra is.

A módszer használható beágyazott rendszerek, önálló desktop-alkalmazások, stb. fejlesztéséhez. A módszer ugyan hatékonyabb, de mégis analóg a PEPA-teljesítményelemzési modelleket[8] előállító és teljesítményt analizáló transzformációkkal, amelyek korábbi tanszéki munkák eredményeiként álltak elő[5][6].

A dolgozat felépítése

- A dolgozat során egy rövid áttekintést adok a technológiai háttérről, rámutatva azokra a pontokra, amik a dolgozat során feldolgozásra kerültek. (2. fejezet)
- A 3. fejezetben az esettanulmányok kerülnek bemutatásra, amelyek közül az egyiket fel is használjuk a módszerek teszteléséhez (6. fejezet)
- A 4. és 5. fejezetek a fejlesztés általános menetét, valamint a modellezési módszereket mutatják be.
- A 7. fejezet a kapcsolódó módszereket és publikációkat tekinti át, amelyek már rendelkezésre álltak a dolgozat elkészítése előtt.
- A 8. fejezet az esetleges továbbhaladási irányokat méri fel.
- Végül a 9. fejezet összegzi a dolgozat során elvégzett munkát, eredményeket.

2. Technológiai háttér

2.1. A modellvezérelt fejlesztés (MDD) és a modellvezérelt architektúra (MDA)

A modellvezérelt fejlesztés (MDD – *Model-Driven Development*) a modellvezérelt architektúrával együtt (MDA – *Model-Driven Architecture*) egyike napjaink legdinamikusabban fejlődő rendszerfejlesztési paradigmáinak. Különbséget kell tenni a két fogalom között: az MDD egy, az MDA köré épülő fejlesztési irányzat, amihez ma már szakterület-specifikus eszközökkel rendelkezünk.

Az MDA-t az *Object Management Group* (OMG) szabványosította 2001-ben. A specifikációk a hivatalos honlapon érhetők el[9].

A továbbiakban a két fogalom közül kiemelten az MDD-vel, azaz modellvezérelt fejlesztéssel foglalkozunk.

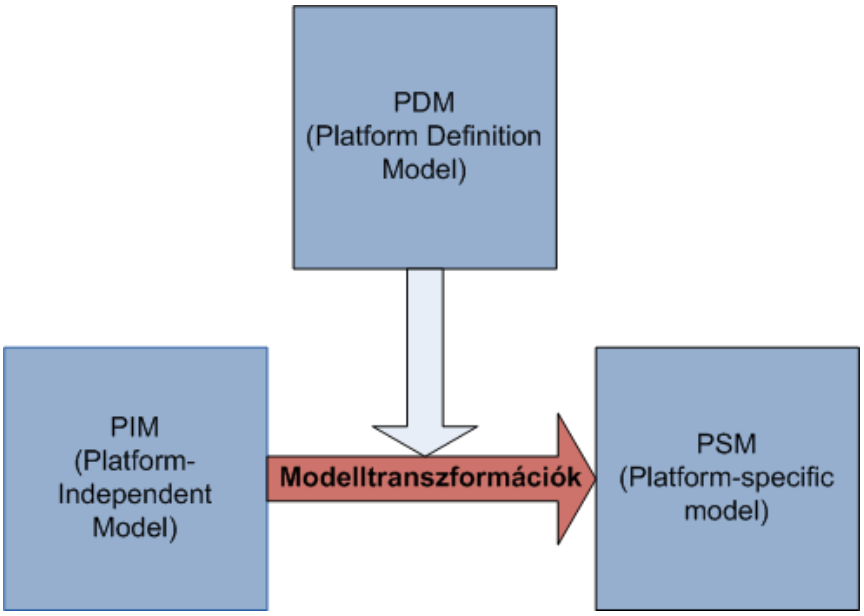
A modellvezérelt fejlesztés, bár kétség kívül sok előnyös oldallal rendelkezik, az ipari szektorokban történő elterjedése két komolyabb akadályokba ütközik. Egyrészt az fejlesztést segítő eszközök, alkalmazások még nem igazán elterjedtek, másrészt pedig nincs olyan kipróbált és bejártatott fejlesztési módszertan, ami célzottan az MDD-t segíti, és egy nagyobb IT-vállalat számára számottevő kockázati tényező nélkül adaptálható lenne.
Ezen problémák leküzdésére nagy számú kutatási-fejlesztési projekt indult, melyeknek hatására az MDD egyre gazdagabb és használhatóbb terület lett. Nem kell túl messzire menni a példáért: jelen dolgozatban is találkozhatunk a BME-MIT Hibatűrő Rendszerek Csoportjának VIATRA2 keretrendszerével[10], amelyet mi a modell-kód transzformációk implementálására és futtatására használunk. Hasonló célú és szintén az Eclipse platformra épülő keretrendszerek az OAW[11] (*openArchitectureWare*) és az ATL[12] (*ATLAS Transformation Language*).

A modellvezérelt fejlesztés alapkoncepciója

A modellvezérelt fejlesztés fontos motivációja a modell és az architektúra különválasztása. Ezzel több előnyös tulajdonságát alakítjuk ki a rendszernek.

A modellt, illetve az architektúrát implementáló technológiák és technikák közti könnyű átjárhatóság a módszer egyik legjelentősebb vívmánya. A szoftverrendszereknél maradva: például ha egy, már létrehozott rendszer architektúráját Java alapúról .NET-re szeretnénk cserélni, akkor optimális esetben a modell bármiféle módosítása nélkül megtehetjük ezt. Igaz ez a másik oldalra is, tehát a modellező technológiát nyugodtan cserélhetjük az architektúra megbontása nélkül, feltéve, hogy az új technológia is ugyanúgy tesz eleget a szabványoknak, azaz ugyanazt az interfészt biztosítja a transzformációk felé, mint a lecserélendő.

Az MDD alapgondolata, hogy egy adott szoftverrendszert egy adott platformra magas szinten automatizált **transzformációkkal hozzunk létre** közvetlenül a modellből. Ezt mutatja be a 3. ábra.



3. ábra – A modell-transzformáció folyamata. A platform-független modellből a platformdefiníciós modell hozzáadásával platform-specifikus modellt generálunk.

A 3. ábra megadja a modellvezérelt fejlesztés alapvető elemeit:

1. **PIM** (*Platform-independent model* – Platform-független modell)
   A tervező létrehozza a modellt egy arra alkalmas nyelven (pl. UML). Ez a modell még független a célplatformtól, tehát semmiféle információt, adatot nem tartalmaz az implementációval kapcsolatban.
2. **PSM** (*Platform-specific model* – Platform-specifikus modell)
   A platform-specifikus modell a PIM-ből képződik, viszont a platformra jellemző tulajdonságokat már magában hordozza. Ilyen speciális (bár alapjában véve egészen triviális) tulajdonság lehet például, hogy egy Apache technológiákat használó SOA rendszer minden biztonsági jellemzőjének kötelezően nevet kell adni, vagy hogy a .NET platformon milyen változótípusokat használhatunk.
3. **PDM** (*Platform-definition model* – Platform-definíciós modell)
   A célplatform kiválasztása után rendelkezésünkre áll a platformot (szemantikát, szintaktikát) leíró modell. Itt találhatók meg azok az adatok, amelyek segítségével egy-egy PIM-beli elem a PSM-be transzformálható. Ezeket a transzformációkat nevezzük *mapping*nek. Egy jellemző PDM-információ lehet például, hogy azok az UML elemek, amelyek egy osztályt reprezentálnak a PIM-ben, a transzformációk során a jól megszokott *class(){…}* alakúvá mappelődjenek a PSM-ben.

A *mapping* egy olyan transzformáció (vagy transzformáció-halmaz), amelynek két bemeneti paramétere a platform-független modell és a platformdefiníciós modell, kimenete pedig a platform-specifikus modell.

Látható, hogy a PIM és a PSM egymástól teljesen független, és a PSM csak a **mapping hatására képes változni**. Fontos megjegyezni, hogy bár a folyamat szemmel láthatóan forward engineeringre van optimalizálva, de az OMG ADM (*Architecturedriven Modernization*) projektje a reverse engineering-et érintő megoldásokkal is foglalkozik.[13]

A modellvezérelt fejlesztés előnyei és hátrányai

Egy fejlesztési módszer megítélésben a legfontosabb kérdés az, milyen előnyei és hátrányai vannak, valamint hogy ezek fényében van-e létjogosultsága, mint önálló fejlesztési irányzatnak.

A modellvezérelt fejlesztés több előnyös tulajdonsággal is rendelkezik, melyek közül elsőként **a modell és a kód szinkronban tartás**át szokták említeni. Az MDD ugyanis az implementációt elsődlegesen közvetlenül a modellből nyeri mégpedig modelltranszformációk révén. Ezek a magasan automatizált műveletek kiszűrik az összes olyan hibaforrást, amelyek a modelltől szemantikájában eltérő, vagy szintaktikailag inkorrekt implementációt, kódot eredményezhetnek. Ezek a hibaforrások a kézi módszerek (kódírás, „kódolás”) során nagy valószínűséggel vannak jelen. Szinte nincs olyan fázisa a fejlesztésnek, amikor egy-egy kódírási szakasz hibátlanul zajlana – tekintsünk akár egy napi, akár egy havi inkrementumot a rendszer implementációjában.

További fontos előnye a modellvezérelt fejlesztésnek a **magas absztrakciós szint**en történő modellezés lehetőségének megteremtése. Ennek a tulajdonságnak köszönhetően a rendszeranalitikai szimulációkat már korai fázisban használhatjuk a hibás vagy gyenge pontok, felderítésére, a rendszer korlátainak meghatározására, valamint hála az UML rugalmasságának, képesek vagyunk megfelelő módon modellezni a rendszerek kritikus pontjait is.

A fenti két tulajdonság mellett az MDD magában hordozza a következő előnyös jellemzőket is: lehetővé teszi az architektúrát implementáló **technológiák közti könnyű átjárhatóság**ot, a modellnek a tiszta kódhoz viszonyított átláthatósága okán **gyors tanulási görbét biztosít** az újonnan bekapcsolódó szakemberek számára, technikáinál fogva kikényszeríti a sok módszertan által nem biztosítható **dokumentáltság**ot, stb.

Mindazonáltal a modellvezérelt fejlesztés hátrányokat, kényelmetlenségeket is magáénak tudhat.

Jellemző, hogy bár a modellvezérelt fejlesztés modelltranszformációi által időt és ilyen módon költségeket takarítunk meg, a transzformációk implementálása viszont erőforrás-igényes munka, ugyanis a transzformációk általában magukban hordoznak olyan megkötéseket, amelyek nyomán nem túl rugalmasak, nem újrahasznosíthatóak.

2.2. A szolgáltatásorientált architektúra (SOA)

A szolgáltatásorientált architektúra (SOA – *Service Oriented Architecture*) fiatal kora ellenére széles körben elterjedt technológiává nőtte ki magát. Ezt támasztja alá az is, hogy a legnagyobb multinacionális IT-cégek, mint az IBM, Oracle, Microsoft, mind rendelkeznek a piacon kapható megoldásokkal a SOA rendszereket illetően. A szolgáltatásorientált megoldások napjainkra a szoftverrendszerek alappillérei lettek, köszönhetően a nagy méretű és komplex alkalmazások kiépítését szolgáló nézőpontjának.

A SOA alapkoncepciója

A szolgáltatásorientált architektúrák kialakításának motivációja a nagy vállalatok felől érkező igény volt, amelyek jelentős költségű IT-rendszereket üzemeltettek. A vállalati infrastruktúrák ugyanis tipikusan heterogének az alrendszerek és alkalmazások tekintetében, így azok kommunikációja és összekapcsolása nem triviális. A SOA erre a problémára keres megoldást úgy, hogy a feladat megvalósítása során megtartsa platformfüggetlen jellegét.

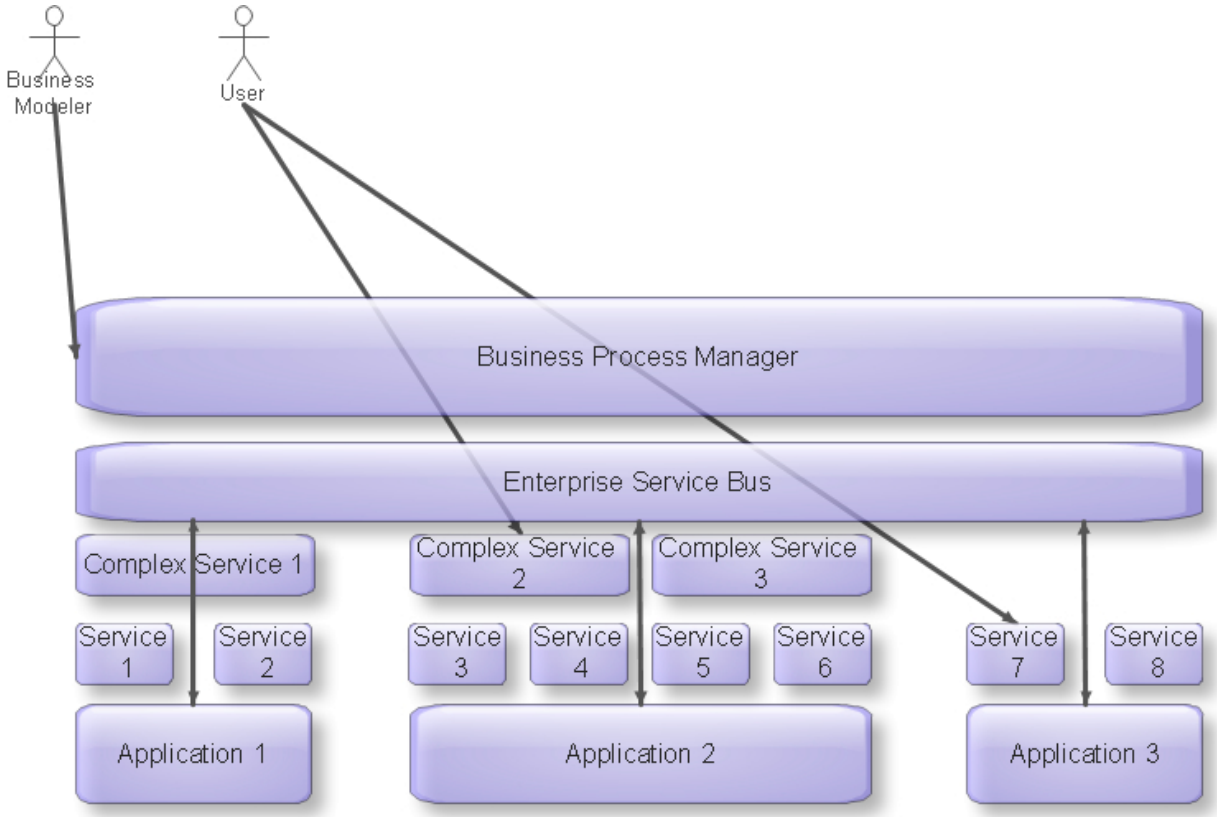
A SOA a már meglévő rendszerek funkcióit felhasználva **laza csatolás** segítségével épít új architektúrát. Ez az új architektúra kellően magas absztrakciós szinten szemlélve egy koherens egészt alkot, amely ugyanakkor jellemzően fizikailag is elkülönülő, egymástól teljesen különböző jellegű rendszereket integrálhat. Az integrált rendszerek platformfüggő tulajdonságai eltűnnek és az elemi funkciók egységesen hívhatók lesznek a távoli metódusok által.

A nagyvállalati környezeti jelleg miatt a SOA-nak meg kell találnia azt a kommunikációs csatornát, amely minden nagyvállalatnál egyformán rendelkezésre áll. A válasz kézenfekvőnek tűnik: az ilyen szigorúan szabályozott IT-környezetek egyetlen közös pontja nagy valószínűséggel a **http protokoll** használata. Ezért a SOA rendszerek kommunikációjának alapját képző webservice-ek kommunikációs csatornája maga a web.

Az egyes rendszerek szolgáltatásának leírására, publikálására az XML tűnt a legalkalmasabb megoldásnak, mint önleíró és platformtól függetleníthető, könnyűsúlyú (*lightweigth*) platform. A szolgáltatásorientált architektúrák az XML-nek egy szakterület-specifikus változatát – a WSDL-t (*WebService Description Language*) – használják. A WSDL definiálja többek között, hogy a rendszernek milyen funkciói érhetők el a szolgáltatási rétegen (*webservice layer*), azok milyen típusú adatokat várnak, illetve adnak vissza, valamint definiálja az adatok kardinalitását is, az adatkötéseket, stb.

Ez a fajta leírás egyáltalán nem függ a rendszer platformjától, hiszen legfeljebb annyi olvasható ki a WSDL-ből, hogy a bemeneti paraméter egy adott metódusnál egy egész szám – azt már nem adja meg a WSDL, hogy ezt a számot hogyan (hány biten, milyen struktúrában) ábrázolja, kezeli és tárolja a célrendszer. Ennek kezeléséért maga a rendszer a felelős.

Egy általános szolgáltatásorientált architektúrát követő rendszer felépítését mutatja a 4. ábra.



4. ábra - SOA rendszer és komponensei (a [3]-as irodalomból átvéve).

Ebben a konkrét példában az architektúra felépítéséhez 3 rendszert használunk (Application1, Application2, Application3), amelyek egymástól fizikailag is elkülönülhetnek, valamint nem követeljük meg tőlük a platform-azonosságot.

Az alkalmazás-réteg felett helyezkedik el az úgynevezett szolgáltatás-réteg (*webservice layer*). Ezt az ábrán az *Service*-ek és *Complex Service*-ek által határolt réteg jelenti.

Webservice-nek, vagy magyarul webszolgáltatásoknak nevezzük azokat a szolgáltatásokat, amelyeket az egyes rendszerek nyújtanak a szabványos XML interfészen (WSDL) keresztül. Mivel ez az interfész a http protokollon helyezkedik el, a szolgáltatások a **Web** előtaggal bővülnek, hangsúlyozandó azoknak a háttérrendszertől való elkülönülését. A komplex webservice-ek több webservice-ből épülnek fel. Létezésüket az indokolja, hogy bár egy webservice képes leírni egy rendszer funkcióit, mégis előfordulhat, hogy önmagában egy-egy webservice nem képes a külvilág felé azt a funkcionalitást nyújtani, amire szükség van.

A SOA rendszerek jellemzően nagyvállalati része az ESB (*Enterprise Service Bus*). Feladata a nagyvállalati kommunikációnak egy olyan közeget adni, ami a belső alkalmazások számára elérhető és használható, tipikusan *middleware* technológiákkal implementálva.

Eggyel magasabb szint az üzleti folyamatok vezérlő rétege. E helyen kerül definiálásra, hogy az egyes folyamatok az egyes szolgáltatásokat milyen módon veszik igénybe. A modellezés itt már nem feltétlenül IT-szakembert igényel – tipikusan az adott domain tervezője az, aki a folyamatokat összeállítja az alkalmas tervezőeszközzel (pl. *IBM WebSphere Business Modeler*) és implementációs nyelvvel (pl. *BPEL*).

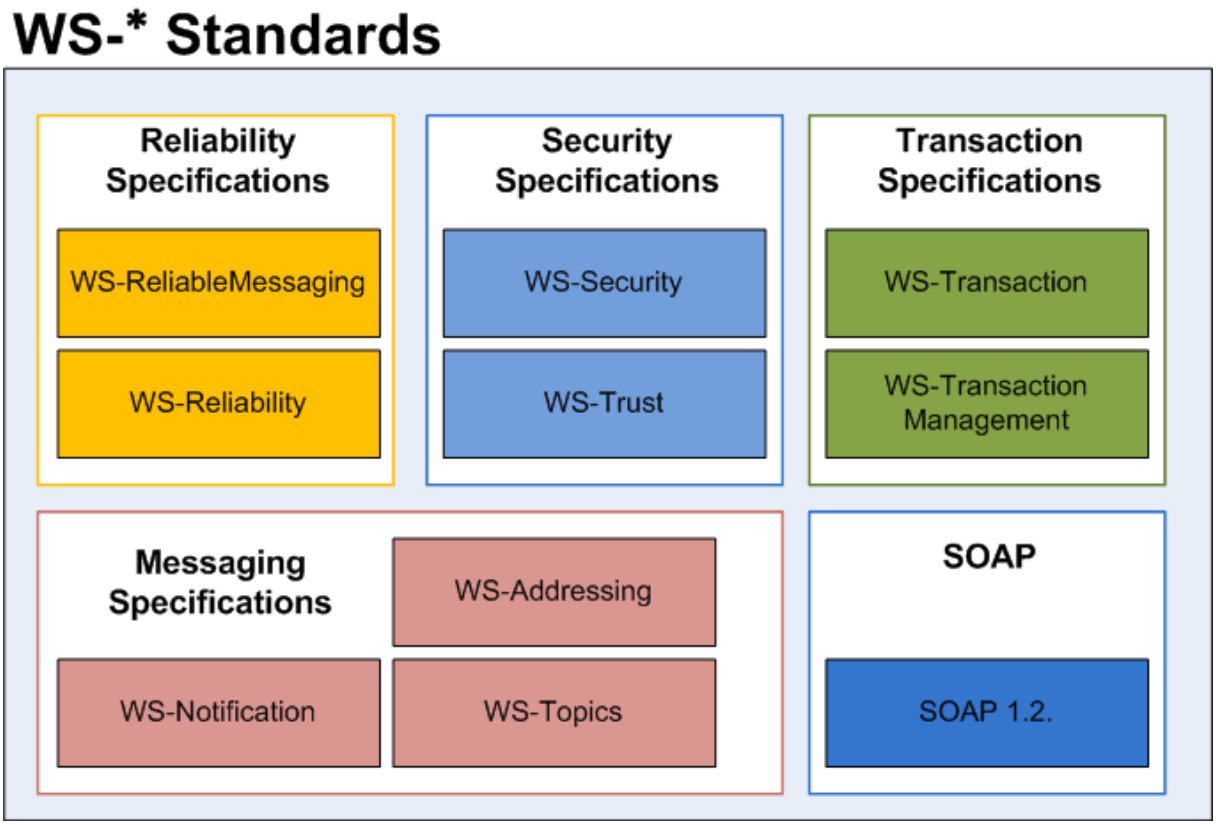
A SOA rendszerek problémái

A SOA rendszereket sok kritikai éri, amelyek általában a webes kommunikációs közeget, az XML platformból adódó hatékonyság-csökkenést, a platformtól való függetlenség gyenge pontjait, vagy épp a biztonságtechnikai hiányosságokat érintik. [14][15][16]

Tény, hogy a szolgáltatásorientált architektúrát gyors felfutása miatt a kutatás és a fejlesztés csak lassabban tudta követni, így napjainkban az egyik legégetőbb probléma a SOA-alapú rendszerek kapcsán azok úgynevezett **nem funkcionális** követelményeinek (megbízhatóság, biztonság, teljesítmény, stb) vizsgálata, modellezése, azok bevezetése a tervezésbe.

Ezeket a paramétereket a WS-* szabványoknak nevezett szabványgyűjtemény írja le. (A név a SOA rendszerek alapegységére, a webservice-re utal, mert a szabályok erre az elemre fogalmaznak meg alapelveket.) Mivel nem adható meg konkrét implementáció ezen szabványok kezelésére, így szinte minden gyártó saját megoldással állt elő mára a szabványokat illetően, hivatalosnak mégis az OASIS (*Organization for the Advancement of Structured Information Standards)* által lefektetett szabványokat tekinthetjük.

A WS-* szabványok egy része látható az 5. ábrán, ám ez csak egy kis része az egésznek. Ez a nagy és redundánsnak tűnő szabvány-csoport gyakori célpontja a SOA-t érő kritikáknak.[17] Az 1. függelékekben részletes ábra írja le a *WS-* stack*et, a specifikációk a hivatalos oldalon érhetők el.[18]



5. ábra - A WS-* szabványok egy része.

A fontosabb WS-* területek és az azokat leíró szabványok, amelyek érdekesek lehetnek a dolgozat szempontjából, a következők:

- **WS-Security**
  Biztonsági jellemzőket definiáló szabvány. Apache Axis2[21][22][23][35] platformon vett megvalósítása a Rampart[3][18][19]. Kapcsolódó szabványok: WS-Trust, WS-Federation
- **WS-ReliableMessaging**
  Biztonságos üzenetküldést definiáló szabvány. Apache Axis2 platformon az Apache Sandesha dedikáltan foglalkozik vele[3][18][20]. Hasonló célú szabvány még a WS-Reliability.
- **WS-Transaction** és **WS-Perfomance**
  Tranzakciókezeléssel és teljesítménnyel kapcsolatos szabványok. Közülük csak a WS-Performance-ot használjuk majd.

Látható, hogy még ez a nagyon tömör kivonat is sokféle irányelvet adó szabványt tartalmaz egy-egy területen belül is. A dolgozat példáiban, prezentációiban a WS-ReliableMessaging, WS-Security és a WS-Performance szabványokat használjuk majd.

NFP-jellegű konfigurációs állományok

A nem funkcionális jellemzőkkel kapcsolatos direktívákat a rendszerek egy arra alkalmas XML állományban tárolják. Ez a konfigurációs állomány megadja, hogy az előzőekben ismertetett WS-* szabványok közül melyiket kell használni, azokat milyen paramétereikkel és milyen értékkel kell beállítani, hogy a nem funkcionális kritériumoknak eleget tegyünk.

Egy Apache Sandesha konfigurációs állomány például a következőképpen nézhet ki[3]:

```
<?xml version="1.0" ?>
<service name="BalanceValidationService">
  <operations />
    <wsp:Policy wsu:Id="BalanceValidationServiceRMPolicy"
      xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis
        200401-wss-wssecurity-utility-1.0.xsd"
      xmlns:wsrm="http://ws.apache.org/sandesha2/policy">
      <wsp:ExactlyOne>
        <wsp:All>
          <wsrm:filterDuplicates>true</wsrm:filterDuplicates>
          <wsrm:needsAck>true</wsrm:needsAck>
          <wsrm:maxNumberOfRetrans>2</wsrm:maxNumberOfRetrans>
          <wsrm:retransInterval>20000</wsrm:retransInterval>
          <wsrm:timeout>10</wsrm:timeout>
        </wsp:All>
      </wsp:ExactlyOne>
    </wsp:Policy>
  </service>
```

A fenti leíró egy megbízható üzenetküldést (*Reliable Messaging*) definiáló leíró, amely meghatározza a üzenetküldési paramétereket és azok értékeit.

3. Esettanulmányok

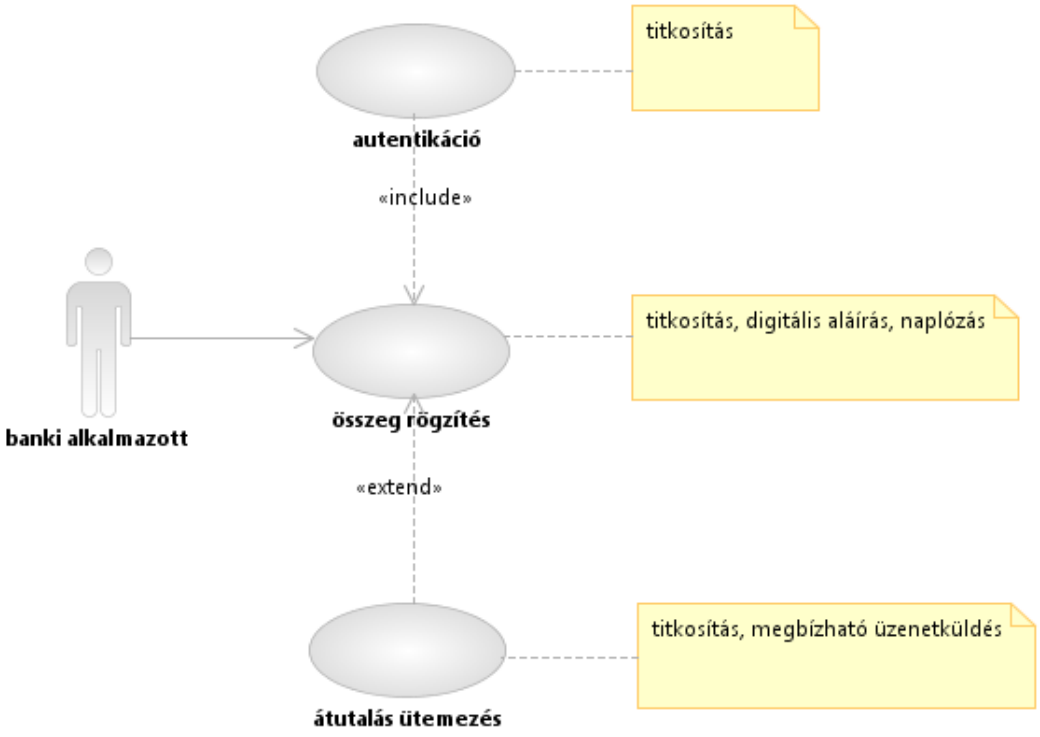
Az esettanulmányok során olyan helyzetek bemutatása volt a célom, amelyek tapasztalatom szerint jellemzően olyan igényeket vetnek fel, amik a szolgáltatásorientált rendszerek nem funkcionális paramétereivel vannak összefüggésben.

3.1. Nemzetközi pénzügyi átutalás esettanulmány (International credit transfer case-study)

Az esettanulmány egy olyan rendszert mutat be, amely egy nemzetközi banki átutalásokat lebonyolító folyamatot támogat. A folyamat első lépéseként egy banki alkalmazott berögzíti az átutalásra szánt összeget és beütemezi azt egy adott időpontra. Ehhez egy autentikációs folyamaton kell átesnie, majd titkosított csatornán keresztül viheti fel az adatokat az adatbázisba. Miután végzett egy-egy tétel felvételével, digitális aláírását és a módosítás időpontját rögzíti a rendszer a tételhez. Az átutalások szintén csak titkosított csatornán keresztül továbbíthatóak, mindemellett megköveteljük a tranzakciók használatát.

A rendszer csak az adatbázis műveleteket naplózza, a service szinten a naplózás nem jelenik meg, mert az adatok titkosításához a bank egy saját algoritmust (*ALG123*) használ, ami azonban az ilyen folyamatoknál a naplózást kizárja.

A rendszer monitorozásához és a hibák előrejelzéséhez alapesetben szükséges a teljesítmény mérése, így a háttérben mindig fut egy vonatkozó alkalmazás. Ezek az alkalmazások azonban annyira lassítják az átutalások kapcsán történő rendszerszintű adatrögzítéseket, hogy a teljesítménymérést szintén kizárja ez a folyamat.



6. ábra – Nemzetközi pénzügyi átutalás esettanulmány.

3.2. Vállalati HR-rendszer esettanulmány (Corporate HR-system case-study)

Az előzőekben bemutatott bank rendelkezik egy belső ügyeket intéző alkalmazással is, amely célzottan HR-rendszer, azaz a bank munkatársainak nyilvántartása, az ledolgozott munkaórák naplózása, riportok generálása és egy speciális belső kommunikációs platform nyújtása a feladata.

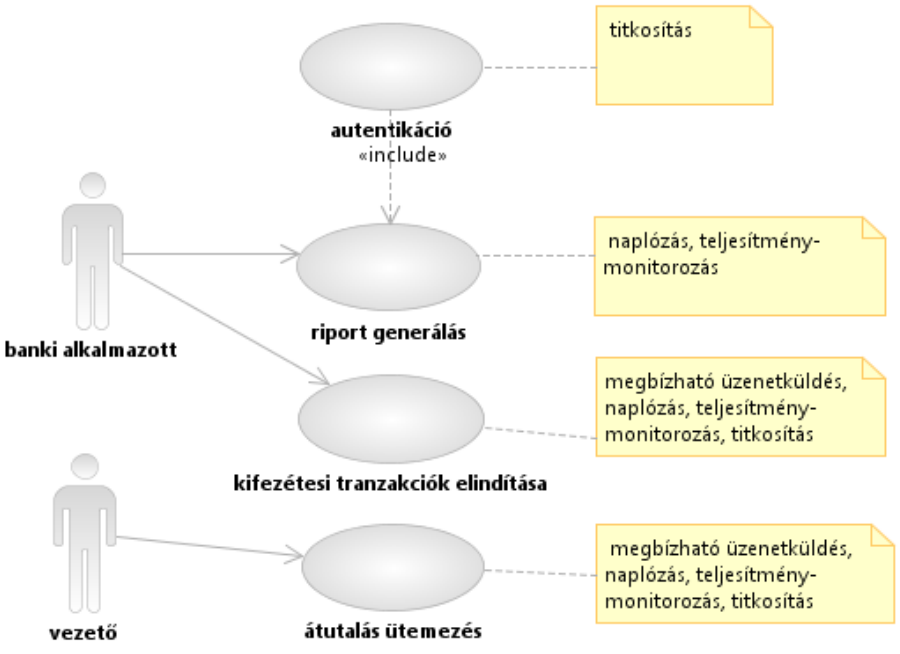
Az esettanulmány bemutatja a hó végi elszámolásokhoz szükséges riportok generálásának menetét.

Első lépésben a számlázási osztály egyik alkalmazottjának autentikálnia kell magát, azaz felhasználónevével és jelszavával be kell jelentkeznie a rendszerbe. Ezután a rendszer engedi hozzáférni a riportokat összeállító felületre. Mivel a rendszer belső használatú és a külső hálózattól fizikailag is el van különítve, ezen a kommunikációs szálon egyelőre nem követeljük meg a titkosított csatornán történő adatközlést.

Második lépésben az alkalmazott kiválasztja az előzőleg összeállított riport-sablont, amely az előző havi munkaórák számát tartalmazza, és alapot képez a kifizetésekhez. Miután a riport összeállt, az alkalmazott jóváhagyja annak legenerálását, ami elindítja a minden egyes alkalmazott munkabérének automatikus átutalását kezelő folyamatot. Ilyenkor a számlázási osztály vezetőjének jóvá kell hagynia a tranzakciót és a kifizetések rögzítésre kerülnek, ezekről egy e-mail értesíti az alkalmazottakat, valamint a riport archiválódik a központi adattárba.

Az előző esettanulmányban már említett rendszermonitorozó alkalmazás működése által okozott teljesítményvesztés ebben a folyamatban nem kritikus, így megengedhető annak használata. Szintén engedélyezve van, sőt kötelező jelleggel használandó a naplózást végrehajtó szolgáltatás is, amelynek minden információt tárolnia kell a rendszer folyamatainak változásával kapcsolatban.

A kifizetések rögzítése miatt már megköveteljük a titkosított csatornán történő adatközlést, amihez a bank egy másik belső algoritmust, a *CRP987*-et használja. Ez az algoritmus nem zárja ki a naplózást.



7. ábra – Vállalati HR-rendszer esettanulmány.

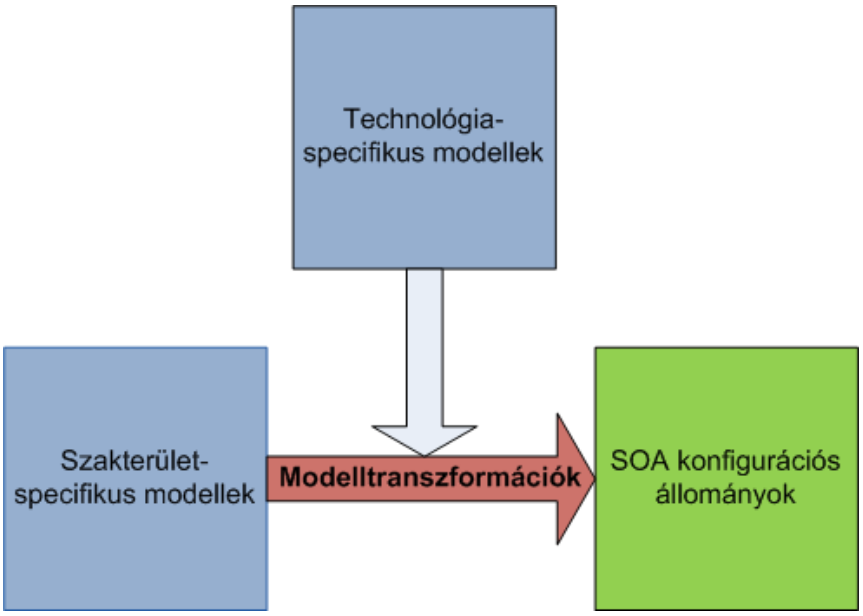
4. A fejlesztés folyamata

Egy átlagos ipari rendszer- vagy alkalmazásfejlesztési projekt első lépésben mindig a megrendelő igényeit méri fel a kivitelező. Ezután fejlesztési módszertantól függően előbb tervezési majd fejlesztési egységek követik egymást. Ezen szakaszok végigviteléhez a kivitelezőnek jól meghatározott képességű szakembereket kell biztosítania (projekt menedzser, tervező, fejlesztők, stb.).

A megrendelői oldal is delegálhat szakembereket a projekthez, és célszerűen meg is teszi ezt, jellemzően a szakterület-specifikus problémák kezelésére. Egy megrendelő oldali szakember szakterület-specifikus tudással nagy terhet vehet le a kivitelező válláról, hiszen így a kivitelező megspórolhatja a szakterület felmérésének, illetve mély megismerésének költségeit.

Ebben az esetben tehát a modellezési feladatok egy részét a megrendelő oldal is elláthatja. Ehhez arra van szükség, hogy a rendszer modelljét alapvetően két nagy részre tudjuk bontani: a szakterület-specifikus tudást igénylő modellre, amelyet a domain szakembere állít össze, valamint a platform-specifikus tudást igénylő modellre, amelynek összeállítása a technológiai szakember feladata.

A fejlesztés során a megalkotott modelleket szintén általunk írt modelltranszformációkkal képezzük le a kívánt formára. A transzformációk kimenete egy-egy XML leíró lesz, amelyek az alkalmazásszerver konfigurációs állományai.



8. ábra - A fejlesztés be- és kimenetei.

4.1. A fejlesztés résztvevői

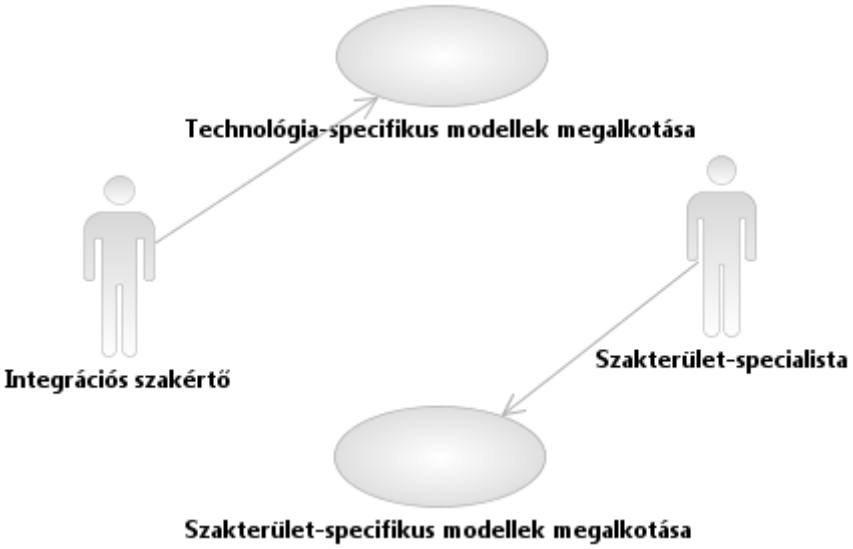
Fontos meghatározni az egyes szereplők tevékenységi és ismereti körét. Két fontos szereplőt definiálunk:

Szakterület-specialista

Ismeretei nagyrészt a saját szakterületére terjednek ki, azonban birtokában van olyan tudásnak is, amely alapján meg tudja határozni egy informatikai rendszer által támasztott követelményeit, de a platform-specifikus jellemzőkkel nincs tisztában és nem is foglalkozik velük.

Integrációs szakértő

Nem lát bele mélyen a szakterület által támasztott igényekbe és nem is foglalkozik velük. Az ő ismeretei a célplatformra terjednek ki: tisztában van azzal, hogy a platform milyen típusú szolgáltatásokat képes nyújtani, hogy az egyes szolgáltatások egymással milyen viszonyban vannak (egymás jelenlétét megkövetelik, avagy épp ellenkezőleg, kizárják egymást), ismeri a platform szintaktikáját, szemantikáját.



9. ábra - A modellek és az őket összeállító szakértők.

4.2. Modellezési jellemzők, UML4SOA

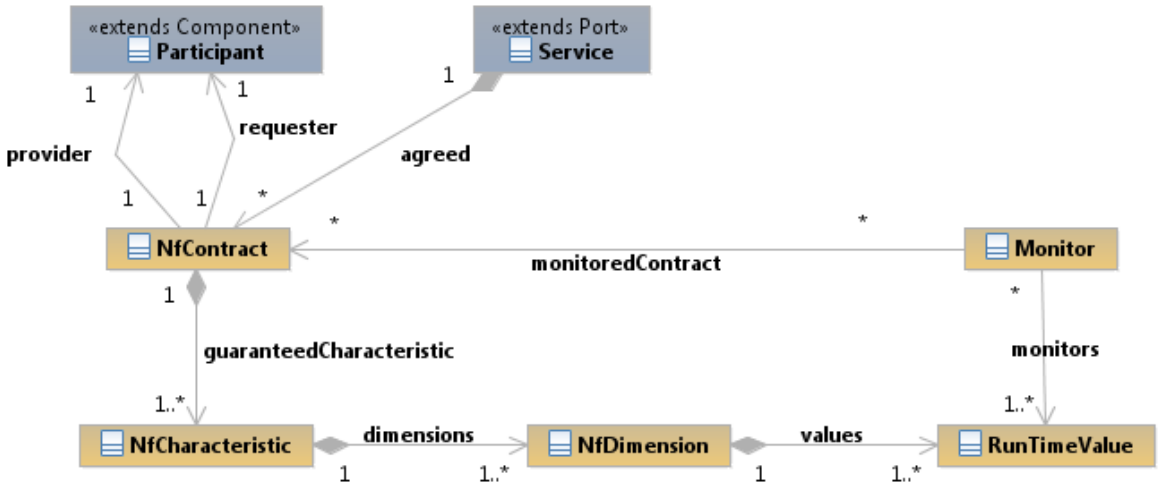
A modellezés során célunk a szolgáltatásorientált architektúrák nem funkcionális (NF) jellemzőinek modellezése. Erre mi az UML szakterület-specifikus kiterjesztését, az UML4SOA nevű profilt használtuk[7]. Természetesen használhattunk volna más modellezési nyelvet, platformot, ami megfelelően képes ábrázolni a számunkra fontos célterületet. Pl. a SoaML[24] egy ilyen alkalmas választás lett volna.

Az UML4SOA profilban vannak definiálva azok a fogalmak, megkötések, amelyekre szükségünk van a SOA rendszerek NF-paramétereinek modellezéséhez.[1],[2],[3],[4]

A modellezési alapok is megtalálhatók a hivatkozott irodalmakban, azonban az alapvető koncepcionális tudnivalókra pár mondat erejéig itt is kitérünk.

NFP modellek

Az NFP modellek a SOA rendszerek nem funkcionális paramétereit tartalmazzák. A SENSORIA projekt BME-MIT tanszéken zajlott munkálatai során az alábbi, 10. ábrán látható metamodell került definiálásra[1] a szolgáltatásorientált architektúrák leírására.

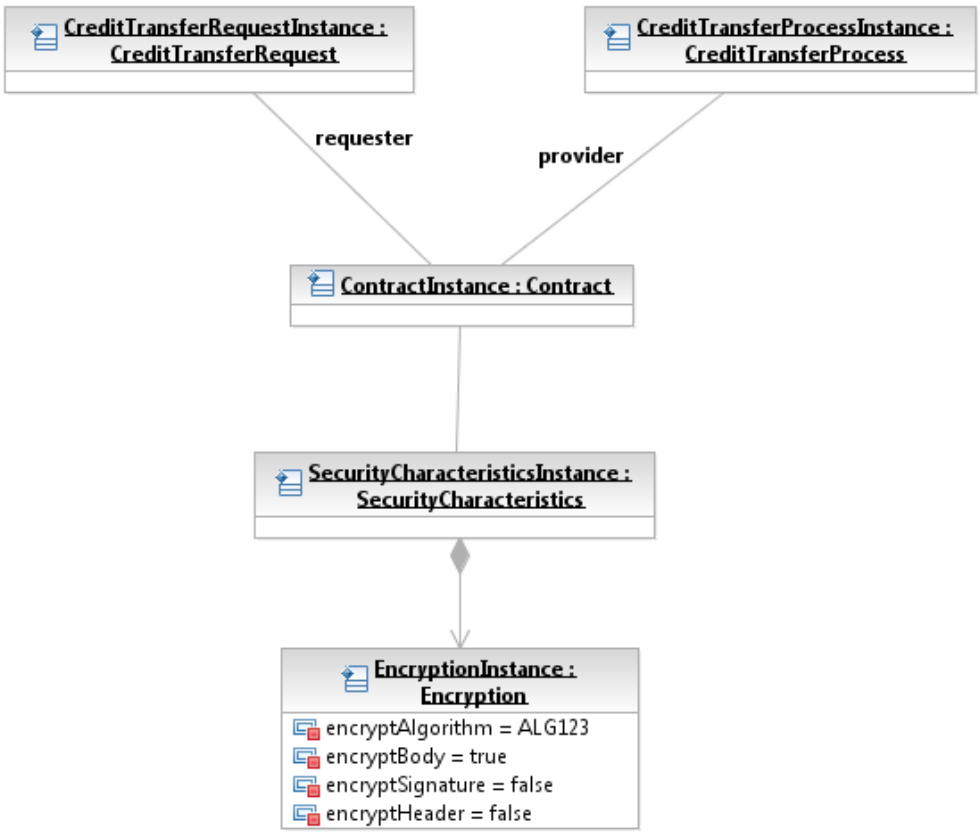


10. ábra - NF-paraméterek leírására alkalmas modellek metamodellje.[1]

A modell központi eleme az *NfContract*, amely egy (SLA-jellegű) szerződés a kommunikáció két résztvevője (*Participant*) között. A két résztvevő szerepe *requester* és *provider* lehet, azaz a szolgáltatást igénybevevő- és az azt szolgáltató fél. A szerződés egy szolgáltatáshoz (*Service*) kapcsolódik, hiszen azt írja le.

A modell felépítésének hierarchiájában lefelé haladva találjuk meg a karakterisztikákat (*NfCharacteristic*), dimenziókat (*NfDimension*) és futásidejű értékeket (*RunTimeValue*), amelyeket egy arra alkalmas egység figyel (*Monitor*).

Példaként vegyünk egy rendszer biztonsági jellemzőit leíró modellt. Ebben az esetben a karakterisztika a *Security*, annak egy lehetséges dimenziója a titkosítás jellege (*Encryption*), amit ábrán látható értékekkel töltünk fel.



11. ábra - SLA-jellegű szerződés a biztonsági nem funkcionális paraméterrel és annak titkosítás dimenziójával.

4.3. Apache Axis2

Az Apache, mint az open-source mozgalom egyik fontos szereplője, ingyenes megoldásokat kínál az egyes SOA szabványok implementálásra. A dolgozat során az Apache Axis2[21][22][23][35] platform alább leírt elemeit fogjuk használni.

Ezek a modulok jelentik majd a modelltranszformációk kimeneteit. Egy-egy ilyen modultól azt várjuk el, hogy adjanak pontos definíciót a konfigurációs elemek, a service-leírók és egyéb kód-jellegű források szemantikájához és szintaktikájához, ezért egy-egy példa is feltüntetésre kerül az alábbi implementációkhoz.

Rampart

Az Apache Rampart[19] az Apache Axis2 platform biztonsági (*Security*) szabványokat implementáló része. Egy tipikus Rampart leíró a következőképpen néz ki:

```
<sp:Encryption>
  <wsp:Policy>
    <wsp:encryptBody/>
    <wsp:encryptAlgorithm>
      <wsp:Policy>
        <sp:Default/>
      </wsp:Policy>
    </wsp:encryptAlgorithm>
  </wsp:Policy>
</sp:Encryption>
```

A fenti XML azt írja le, hogy a titkosítás (*Encryption*) nem-funkcionális *dimenzió* milyen paraméterekkel rendelkezik. Jelen esetben azt kötjük meg, hogy a titkosítás rendelkezzen egy *body*-val, és hogy az algoritmus *Default* legyen.

Sandesha2

Az Apache Sandesha[20] a Ramparthoz hasonló Apache WS-kiegészítés, amely viszont a WS-ReliableMessaging szabályokat és implementációkat definiálja, azaz a megbízhatósági (*Reliability*) jellemzőket.

A Sandesha leíró XML-ek tehát a WS-ReliableMessaging-gel kapcsolatos nem funkcionális paramétereket írják le:

```
<wsrm:maxNumberOfRetrans>2</wsrm:maxNumberOfRetrans>
<wsrm:retransInterval>20000</wsrm:retransInterval>
```

A fenti leírás azt adja meg, hogy az elveszett üzeneteket legfeljebb kétszer, egyenként 20.000 ms-os intervallumban küldjük el.

Kiegészítő modulok

A fenti két modulon kívül definiálhatunk saját modulokat is, amit mi meg is teszünk a naplózás kapcsán. A naplózást egy *Logging* jellemzővel fogjuk megadni a modellben, ami az alábbi minta alapján fog implementálódni.

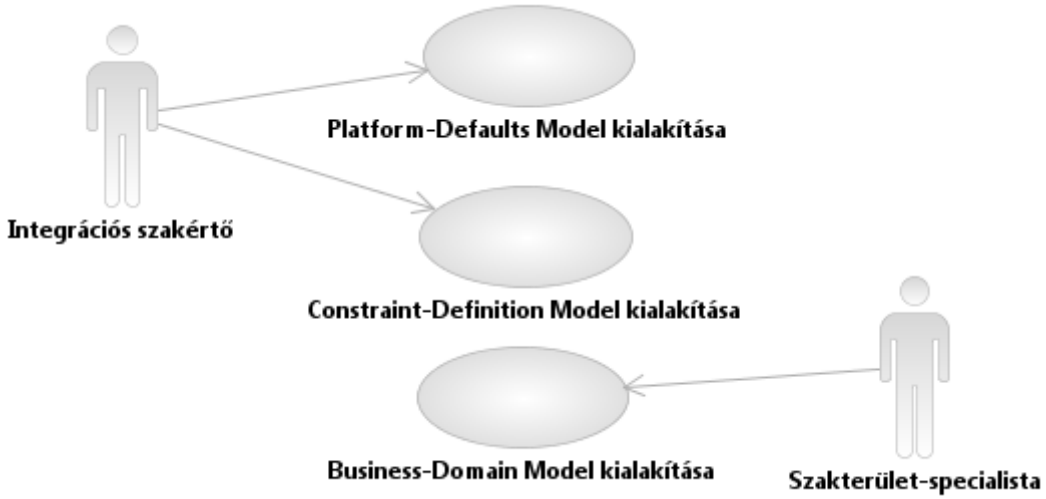
A leíró azt adja meg, hogy a naplózás során *Verbose* jelleggel rögzítsünk minden (*All*) eseményt, mégpedig egy adatbázisba (*database*), amit a megadott módon tudunk elérni a *log* felhasználóval, a *password* jelszóval a *logdb.myserver.com:1434* URI alatt.

```
<wsp:ExactlyOne>
  <wsp:All>
    <wslog:messageLevel>
      <wsp:Policy>
        <sp:Verbose>
      </wsp:Policy>
    </wslog:messageLevel>
    <wslog:triggers>
      <wsp:Policy>
        <sp:All>
      </wsp:Policy>
    </wslog:triggers>
    <wslog:provider>
      <wsp:Policy>
        <sp:database>
      </wsp:Policy>
    </wslog:provider>
    <wslog:connectionString>
      <wsp:Policy>
        „log/password@logdb.myserver.com:1434”
      </wsp:Policy>
    </wslog:connectionString>
  </wsp:All>
</wsp:ExactlyOne>
```

5. Modellezés

A modellezés során a modelleket két nagyobb részre osztjuk: szakterület-specifikus és technológia-specifikus részre. Előbbit értelemszerűen a szakterület-specialista, utóbbit az integrációs szakértő készíti el. Célunk, hogy a két területet a lehető legjobban elválasszuk és a két szakértőnek ne kelljen egymás munkája által befolyásolva dolgozni.

A 12. ábra nagy vonalakban szemlélteti a megoldás koncepcióját.



12. ábra - Rendszermodellek és tervezőik.

Mivel a szakterület-specialista részéről nem feltételezünk komoly és mélyreható ismereteket a technológiával és a platformmal kapcsolatban, ésszerű elvárás lehet, hogy a platform tulajdonságait tegyük transzparenssé számára. Az általa előállított modell csak azt tartalmazza, hogy milyen az ideális rendszer konfigurációja az ő nézőpontjából szerint. (Pl. legyen-e benne naplózás, és ha igen, akkor milyen jellegű, stb.) Ezt a modellt nevezzük *Business-Domain Model*nek (**BDM**).

A másik oldalról viszont a BDM-ből kihagyott platform-megkötéseket kell megadni. Az integrációs szakértőtől elvárható, hogy magas szinten ismerje a platform tulajdonságait. Azaz például tudja megadni, hogy egy biztonsági NF-paraméternek milyen tulajdonságaira van szükség egy szóban forgó jellegű rendszernél, valamint hogy ezek a tulajdonságok általában milyen alapértelmezett értékekkel inicializálhatók. Erre akkor lehet szükség, ha például a szakterület-specialista nem foglalkozik a modell minden elemével, azaz a modell részleges vázát készíti csak el. (Ami egyébként jellemző esetnek tekinthető.) Ezt a modellt nevezzük *Platform-Defaults Model*nek (**PDM**).

Az egyes modellbeli elemek azonban nem kezelhetők egymástól elkülönítve, hiszen jellemzően egymást kiegészítő és egymás használatát igénylő egységek fordulnak elő egy-egy komplexebb rendszermodellben. Például – az esettanulmánynál maradva – megkövetelhetjük a rendszerrel szemben, hogy a digitális aláírást titkosított csatornán keresztül küldje csak át. Itt máris két különböző NF-jellemzőt kapcsoltunk össze: a *WS-ReliableMessaging* által tartalmazott digitális aláírást és a *WS-Security* által tartalmazott titkosítást.

Meg kell tudni tehát adni azt, hogy a modell elemei közt milyen összeköttetések állnak fenn. A feladatot megfordítva: meg kell tudnunk adni, hogy mely modellbeli elemek **zárják ki egymást**. Ezt definiálja a *Constraint-Definition Model* (**CDM**).

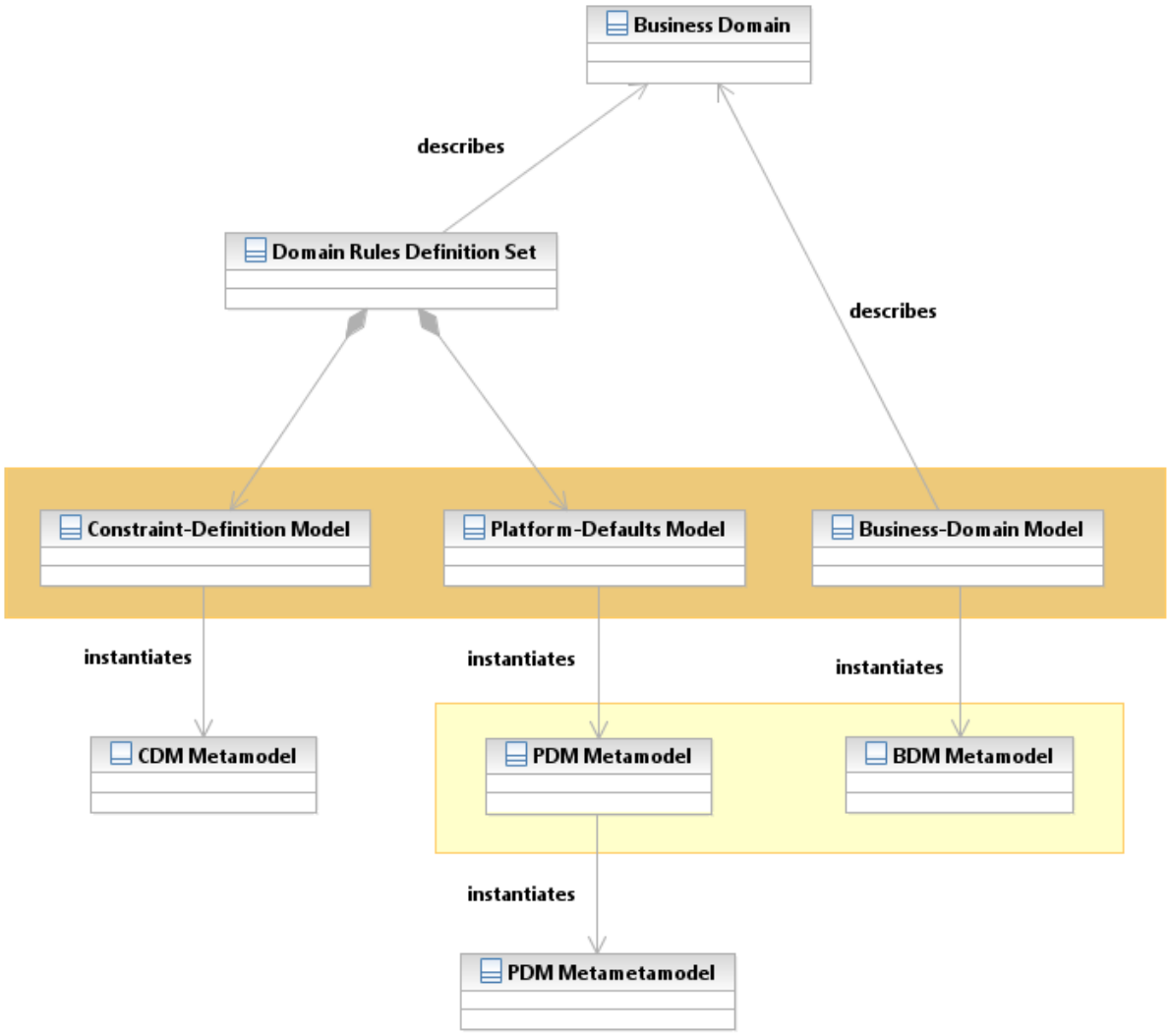
A *Platform-Defaults Model* a *Contstraint-Definition Model*-lel kiegészülve alkotja a *Domain Rules Definition Set*et (**DRDS**). A DRDS önmagában nem jelenik meg a modellben, csupán egy logikai összefogása a PDM-nek és az CDM-nek.

A PDM és az CDM is egy-egy szakterület-specifikus modell olyan értelemben, hogy függenek az adott terület jellegétől. (Hiszen a szakterület jellegétől függően adhatjuk meg például a PDM-ben, hogy mely jellemzőkre lehet szüksége a szakterület-specialistának.) Ezen a modellek tulajdonságait azonban magasabb szinten metamodellekkel írhatjuk le, amelyek már részben, vagy teljesen elszakadnak a területtől. Hogy ez a függetlenség teljes, avagy részleges-e, az absztrakciós szint határozza meg, amelyen a metamodellt létrehozzuk.

Technikai jellemzők

Platform-Defaults Model és *Business-Domain Model* esetén technikai oldalról tudható, hogy a modell példányok **objektum-diagramok** formájában implementálandók, a metamodellek azonban **osztály-diagram**ként. A *Constraint-Definition Model* esetében nem képzünk objektum-példányt, mivel arra nincs szükség. Az objektum-modell nyújtotta különbség – hogy az egyes attribútumok értéket kaphatnak a modellben – nem szempont, hiszen a CDM egy példányában a *RunTimeValue* típusú osztályokhoz *value* típusú osztályok kapcsolódnak, amik magukban hordozzák ezt a plusz információt.

A 13. ábrán látható a modellek teljes hierarchiája. Sötétebb színnel az **erősen domain-függő** modellek, míg világosabb színnel a **gyengén domain-függő** modellek láthatóak. Erős domain-függésnek nevezzük azt, amikor a modell a szakterülettől elválaszthatatlan – pl. mert annak egy konkrét leírója. A gyenge domain-függés esetén a modell a szakterülettől elválasztható, mert bár a szakterület határozta meg, hogy milyen elemek jelenjenek meg a modellben, a modell logikailag nem tartalmaz semmiféle megkötést a szakterületre vonatkozólag.



13. ábra - A modellek hierarchiája. Sötétebbel jelölve az erősen domain-függő modellek (CDM, PDM, BDM), világosabb színnel a gyengén domain-függő modell (PDM metamodell és BDM metamodell).

5.1. A *SeparatedModeling* UML profil

Profilok használatával hatékonyan bővíthetjük ki az UML platformot saját igényeink szerint. A munkálatok során megalkottam a *Business-Domain Model* és a *Constraint-Definition Model* összeállításához használható profilt, a *SeparatedModeling*-et.

A profil CDM-specifikus része a következő elemeket tartalmazza:

- *excludes* asszociáció – a CDM modellbeli kizárást valósítja meg
- *requires* asszociáció – két elem közti, egymásra utaló igényt valósít meg a CDM-ben
- *value* osztály – a *runtimeValue* típusú osztályokhoz tartozó érték

A profil tartalmaz továbbá egy olyan halmazt is, amely a szakterület-specialistának segít a modellek felépítésében. Ennek összeállításánál figyelembe vettem a SENSORIA Summer School utáni visszacsatolásokat is. Jellemző volt ugyanis, hogy modellalkotás során a nem gyakorlott felhasználók nehezen tudták adoptálni modelljeikben az UML4SOA profil sztereotípiáit. Azonban még fontosabb indok volt a profil létrehozása mellett, hogy az UML4SOA nem tartalmaz konkrét karakterisztika- és

dimenzió-sztereotípiákat, csak egy *nfCharacteristics* és egy *nfDimension* sztereotípiát. Ezekkel a sztereotípiákkal csak azt tudjuk megadni, hogy az elem a modell melyik szintjén van – nincs lehetőség tehát pl. a karakterisztika jellegének pontos azonosítására. Amíg egy, a platformot ismerő tervezőt feltételezünk (aki a PDM-et összeállítja), ezzel nincs is probléma, mert célszerűen tudja elnevezni modellbeli elemeit, feltüntetve az elem jellegét és a vonatkozó NF tulajdonságot – pl. *SecurityCharacteristics*. A gondok akkor jönnek elő, amikor a platformot nem ismerő szakember modelljéből szeretnénk transzformálni, mert nála nem feltételezünk ilyen szintű felkészültséget, ezért az ő munkáját meg kell támogatni olyan profilelemekkel, amik definiálják mind a modellbeli elem jellegét, mind a vonatkozó NF tulajdonságot. Így például a fenti példánál maradva a PDM-beli *SecurityCharacteristics* elemmel analóg BDM-beli elem sztereotípiája nem csak egy *nfCharacteristics* lesz, hanem egy *SecurityCharacteristics* is. Ezzel elkerüljük az esetleges elnevezésekből történő problémákat.

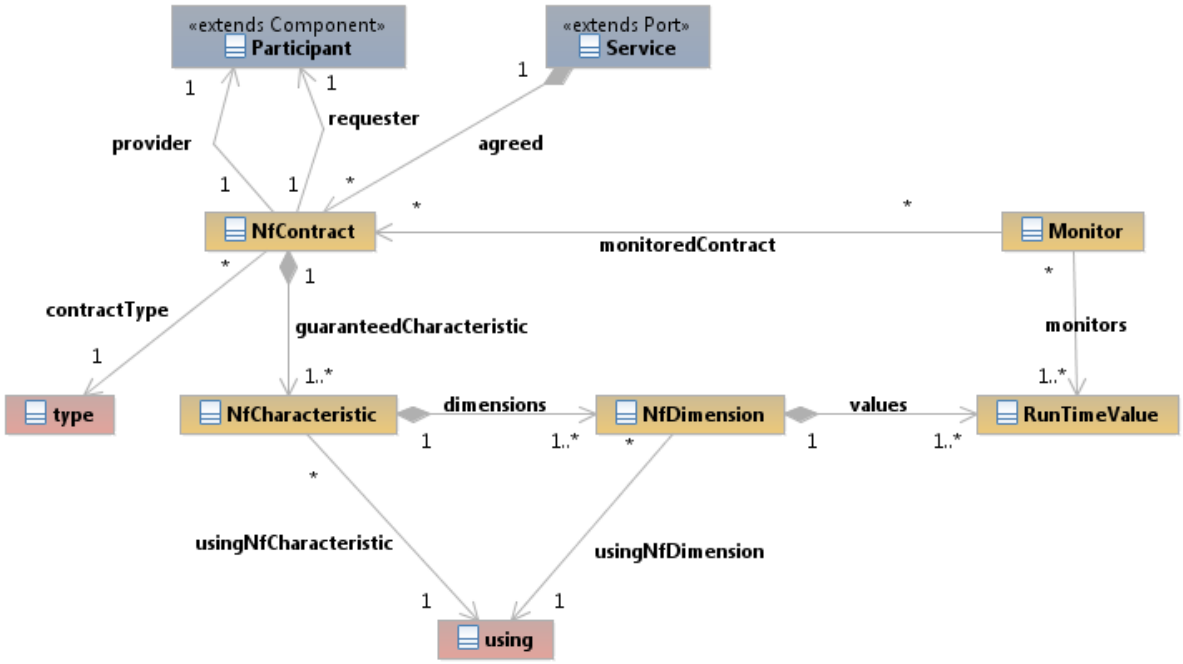
A profil ezen része tehát tartalmazza az összes szóba jöhető karakterisztikát és dimenziót.

5.2. Platform-Defaults Model (PDM)

Platform-Defaults Metametamodel

A 4. fejezetben (10. ábra) bemutatott NFP-modell jól használható SOA rendszerek NF-paramétereinek modellezésére, ezért ezt vehetjük alapul a jelenleg vizsgált problémák megoldásához. Számunkra mégsem tekinthető teljes megoldásnak, hiszen nekünk az NF-paraméterek modellezésén túlmutató céljaink vannak a *Platform-Defaults Modellel*, a PDM segítségével szeretnénk ugyanis megadni, hogy egy-egy karakterisztika, vagy dimenzió részt vesz-e a rendszerben, valamint hogy egyes elemek milyen alapértelmezett értékkel rendelkeznek a rendszeren belül.

Ennek megfelelően a modellt két elemmel kell bővítenünk.



14. ábra – PDM metametamodell. A modellszétválasztást támogató kiegészítések pirossal láthatók.

A PDM metametamodell a már ismert tulajdonságokon kívül definiál a szerződéshez egy **típust**, ami a domainre jellemző, de a modell jellegének azonosításán túl nem használjuk másra.

A karakterisztikák és a dimenziók bővültek egy tulajdonsággal (**using**). Ez a jellemző adja meg, hogy az adott karakterisztika, vagy dimenzió a modellben megtalálható-e. Ennek a tulajdonságnak a modellben majd a következő értékkészletből kell kikerülnie: {*required, enabled, disabled*}.

Required – Karakterisztika esetén a karakterisztikának kötelezően szerepelnie kell a modellben. Dimenzió esetén csak akkor vizsgáljuk, ha a dimenziót tartalmazó karakterisztika benne van a modellben – ekkor azonban a dimenzió is benne kell legyen a modellben.

Enabled – Karakterisztika esetén a karakterisztika szerepelhet a modellben, ha a szabály-definíciós modell (CDM) ezt nem tiltja. Dimenzió esetén csak akkor vizsgáljuk, ha a dimenziót tartalmazó karakterisztika benne van a modellben – ekkor a dimenzió is benne lehet a modellben, ha a CDM nem tiltja.

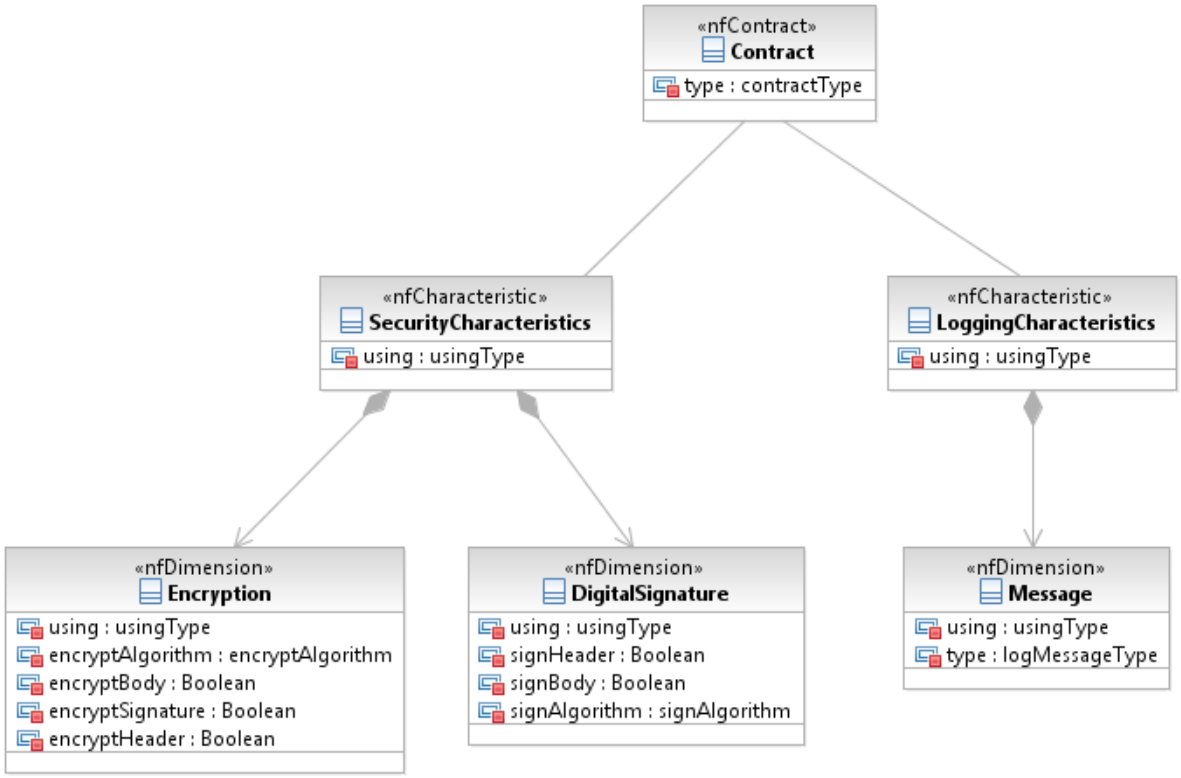
Disabled – Az adott karakterisztika vagy dimenzió nem lehet benne a modellben.

Erre a megközelítésre azért van szükség, mert az UML-beli megkötésekre általánosan elterjedt nyelv, az OCL használatával nehézkes azokat az összetett globális függőségeket leírni, mint amilyenekre nekünk szükségünk van. (Pl. a biztonsági és a tranzakciós jellemzők szinteken átnyúló egymással alkotott kapcsolata.) Lásd még a kapcsolódó publikációkról szóló szakaszban (7.2. fejezet).

Platform-Defaults Metamodel

Ezen a metaszinten már megadjuk a konkrét karakterisztikákat, dimenziókat, a futásidejű változókat pedig tipizáljuk, ahogy a *using* és a *type* paramétereket is.

A metamodell feladata, hogy belőle lehessen elkészíteni a PDM-et. A modellhez tartozó diagram még mindig class-jellegű, hiszen a modellelemek nem példányosítottak egyelőre.



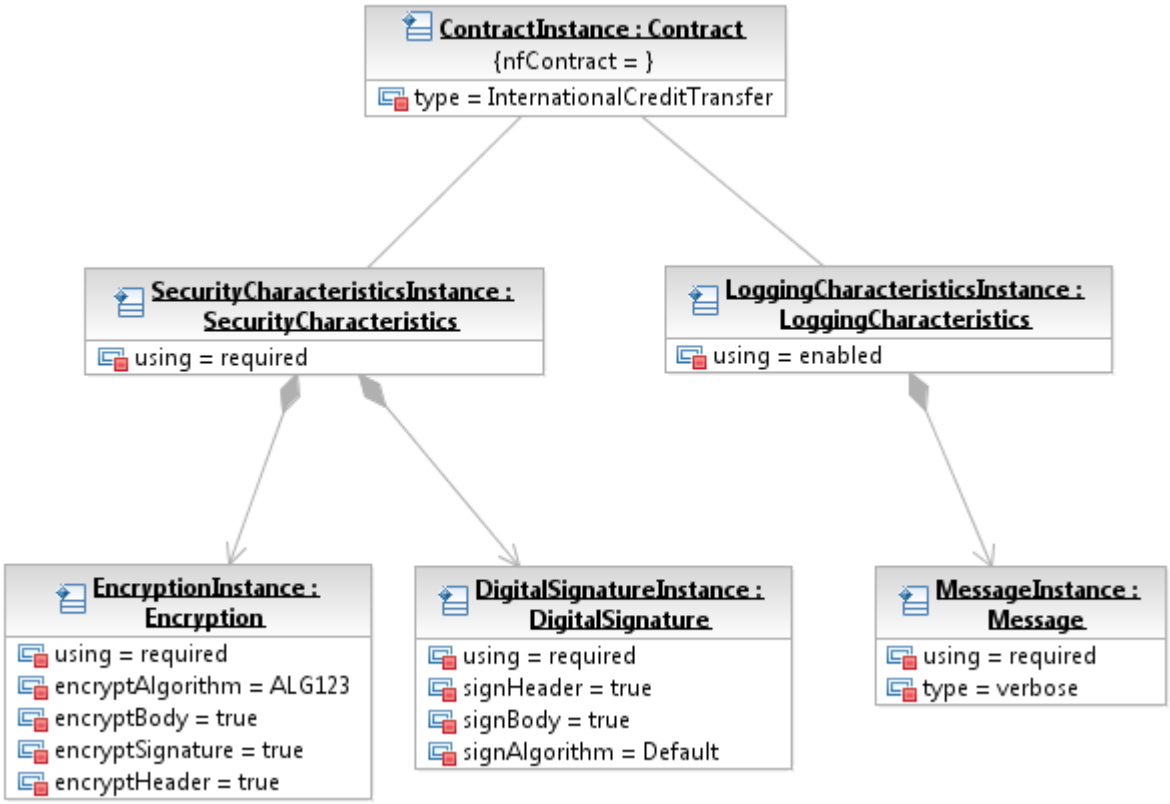
15. ábra - PDM metamodell.

Platform-Defaults Model (PDM) példány

Utolsó lépésben a Platform-Defaults Metamodelben megadottakkal összhangban példányosítjuk a modellelemeket értéket adva a futásidejű változóknak, a *using* és *type* paramétereknek.

Az így kialakult modell diagramja egy objektum-diagram, amelyben a metamodell által definiált attribútumok – az UML szabályaival összhangban – értéket kapnak.

Ezek az értékek a rendszer platformon értelmezett alapértékei (*platform defaults*). Használatáról a BDM-ről és a modellszintézisről szóló szakaszban is szó lesz.



16. ábra - PDM példány.

5.3. Business-domain model (BDM)

A *Business-Domain Model*, ahogy azt neve is mutatja, a szakterület-specialista által összeállított modell.

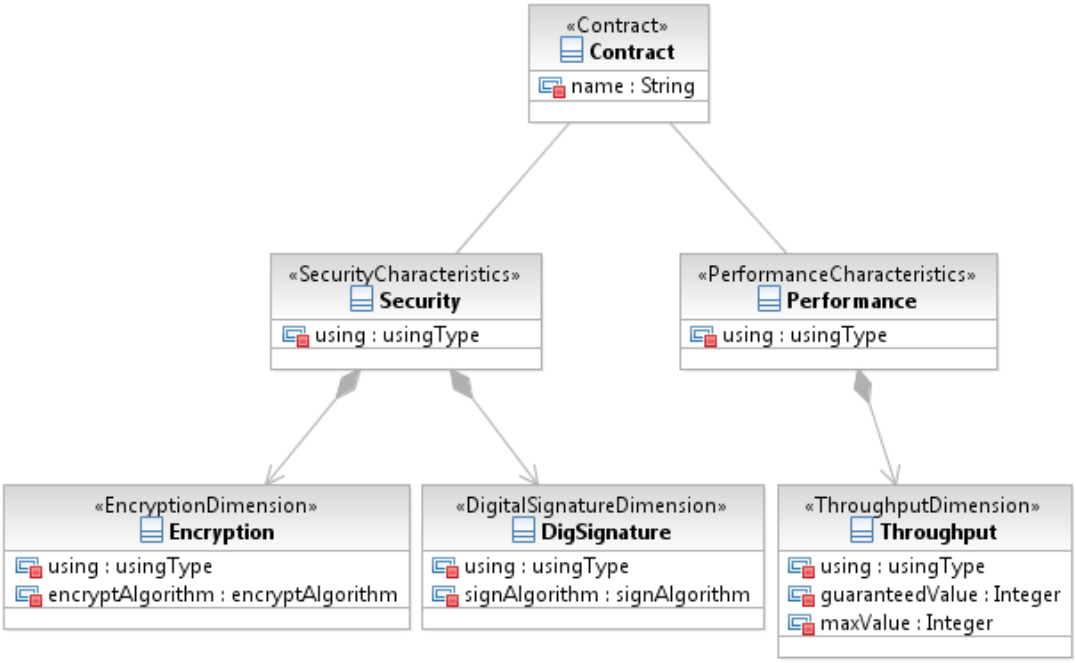
A BDM-mel kapcsolatban a következők az elvárásaink:

- **legyen képes pontosan definiálni a domain igényeit**
  A domain igényei alatt a rendszerrel szemben fennálló igényeket értjük. Egy ilyen kívánalom lehet például, hogy a rendszer használjon titkosítást az érzékeny adatok átvitelére, vagy hogy a rendszer naplózza az eseményeket.
- **lehetőleg ne tartalmazzon platform-specifikus részeket**
  A szakterület-specialistától nem várhatunk el magas fokú ismereteket a platformmal kapcsolatban, így a modelltől megköveteljük, hogy ezeket az adatokat tegye transzparenssé a szakterület-specialista számára.
- **amennyire csak lehet, legyen támogatva automatikus eljárásokkal (automatikus modellkitöltések, validáció, stb.)**
  A BDM, ahogy minden manuális munkafolyamat végeredményeként előálló modell vagy kód, magában hordozhatja a szintaktikai, szemantikai hibák lehetőségét. A szemantikai és szintaktikai helyesség és teljességvizsgálatoknak automatizáltnak kell lenniük, hiszen a BDM-et összeállító domain-szakember oldaláról nem feltételezünk nagyfokú platform-specifikus ismereteket.

Business-Domain Metamodel

A BDM metamodell összeállítása során a szakterület-specialista elsődleges feladata, hogy megszabja, milyen NF-jellemzőket szeretne a modelljében ábrázolni. Ehhez egy *Contract* típusú osztály köré kell szerveznie a *SeparatedModeling* profil segítségével a számára érdekes elemeket és azok tulajdonságait. A *usingType*, *encryptAlgorithm*, stb. *Enum* típusú struktúrákat a PDM-et összeállító integrációs szakértő szolgáltatja.

A legfontosabb különbség a PDM metamodellel szemben, hogy itt a szerződésnek nem típusa, hanem neve van. Ez egy lazább jellemző, mert nem egy modellbeli típusra hivatkozik, hanem egy egyszerű *String* típus.



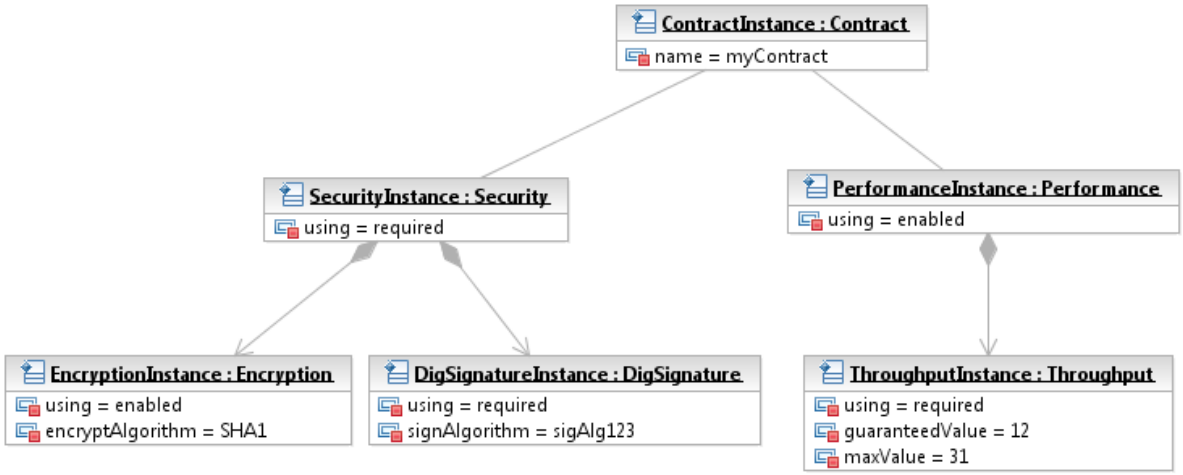
17. ábra - BDM metamodell.

Business-Domain Model (BDM) példány

A *Business-Domain Model* példány tehát az őt leíró metamodell alapján készül el. A modell teljessége, valamint szintaktikai és szemantikai korrektsége végett szükséges a metamodell alapján történő validáció. Erről a modellszintézisről szóló szakaszban olvashatunk.

A BDM elemeinek fontos attribútuma a *using*, ami azt írja le, hogy a szakterület-specialista szerint mennyire fontos az egyes elemek megjelenése a transzformációk eredményét jelentő implementációban. Értékei ugyanazok lehetnek, mint a PDM esetén: *required, enabled* és *disabled*.

Az ábrán látható, hogy egyes dimenzióknak vannak hiányzó attribútumai. Ezeket a későbbiekben a PDM alapértelmezett értékeivel tölti majd ki a transzformációs folyamat – itt nyernek tehát értelmet a Platform-Defaults Model ezen adatai.



18. ábra - BDM példány.

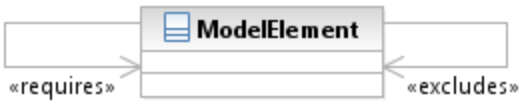
5.4. Constraint-Definition Model (CDM)

Bár az eddig definiált modellek segítségével jól leírhatók az egyes nem-funkcionális tulajdonság felé támasztott követelmények – azaz, hogy szerepeljen-e egy adott karakterisztika, annak milyen dimenziói legyen a modellben, stb. – azt mégsem tudják leírni ezek a modellek, hogy az egyes karakterisztikák, dimenziók, *run-time value*-k egymással milyen kapcsolatban állnak, azaz hogy az előző elemek egymás jelenlétét megkövetelik-e, avagy épp ellenkezőleg: kizárják egymást. Ezt a fajta információt írja le az *Constraint-Definition Model*.

Constraint-Definition Metamodell

A CDM metamodellje nagyon egyszerű, hiszen ennél a modellnél két elem között fennálló kétféle lehetséges kapcsolatot kell ábrázolni.

A 19. ábrán látható, hogy egy modellbeli elem (*ModelElement*) egy másik modellbeli elemmel kétféle kapcsolatban állhat, ha áll valamilyenben. A kapcsolat kizárási, vagy kapcsolódási lehet, és irányított – azaz a forrás elem (ahonnan a „nyíl indul") kizárhatja, avagy épp ellenkezőleg, megkövetelheti a másik modell elem jelenlétét.



19. ábra - CDM metamodell.

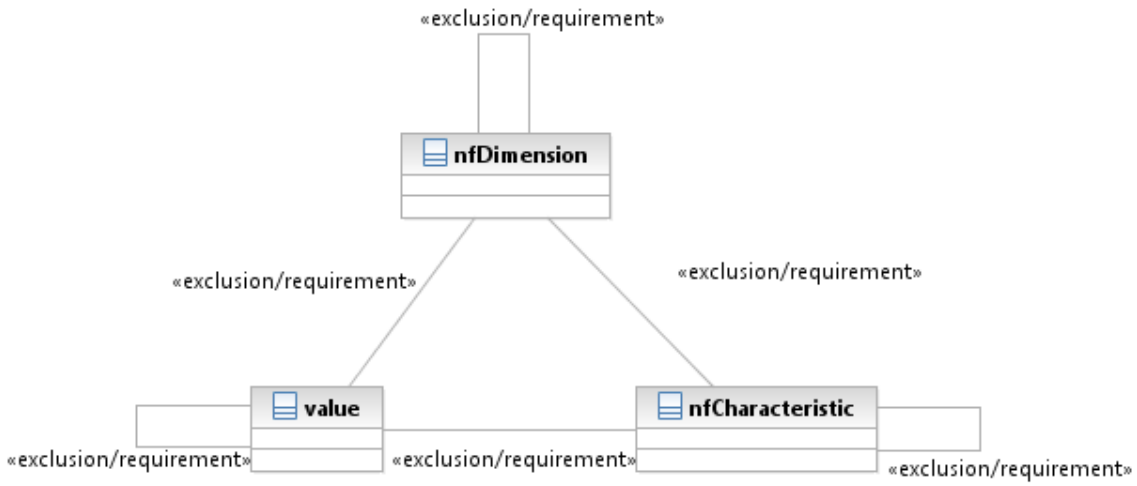
Constraint-Definition Model példány

A CDM létrehozása során felmerülő probléma lehet, hogy bonyolultabb összefüggéseket nehéz egy modellben kezelni, mert a nagy számú összefüggések átláthatatlanná teszik a grafikus nézetet. Ezért minden egyes karakterisztikára létrehozható egy CDM, amelyben megadható, hogy az adott

karakterisztika és az alatta lévő elemek (dimenzió, dimenzió attribútuma, annak értéke) milyen kizárási kapcsolatban vannak egy (vagy több) tetszőlegesen kiválasztott modellbeli elemekkel.

Vegyük észre, hogy az CDM segítségével nem csak az adható meg, hogy egy-egy karakterisztika egymással kizárási viszonyban van-e. Képesek vagyunk megadni az egész hierarchián átívelő kizárási kapcsolatokat is, mint pl. amikor egy dimenzió attribútumának futásbeli értéke kizárja egy másik karakterisztika jelenlétét.

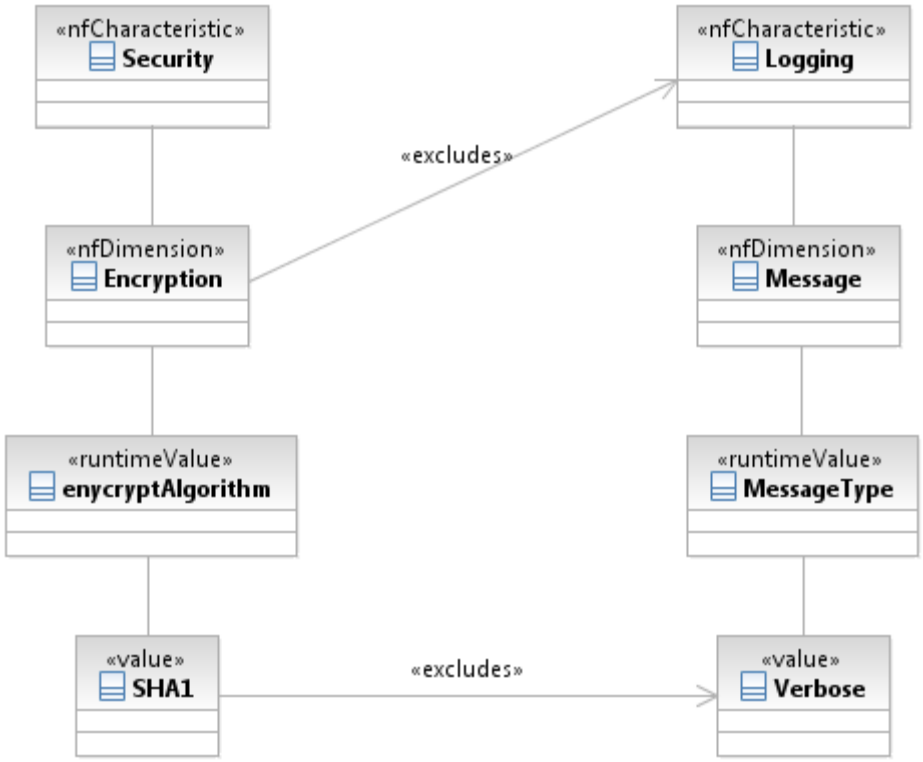
Kapcsolatokat tehát a 20. ábra írja le:



20. ábra - Kizárási/követelményi párok.

A 21. ábrán 2 különböző típusú kizárási kényszer látható:

- az *Encryption* dimenzió kizárja a *Logging* karakterisztikát
- az *SHA1 t*ípusú kódolás kizárja a *Verbose t*ípusú naplózási üzenetet



21. ábra - CDM példány.

A *Constraint-Definition Model* példánynál meg kell jegyezni, hogy bár példányról beszélünk, mégsem objektum-modellben adjuk meg az elemeket. Lásd: 5. fejezet, Technikai jellemzők.

Kényszerek definiálása OCL segítségével

Az UML-beli kényszerek leírására általánosan használt nyelvvel, az OCL-lel való összehasonlítást lásd az *Egyéb módszerek a modellvezérelt fejlesztés hatékonyságának kapcsán* c. fejezetben (7.2. szakasz).

5.5. Értékelés

A fejezet elején leírt modellbővítéssel elérhető vált számunkra, hogy az egyes modellelemekhez megadható legyen azok fontossági szerepe – a *Platform-Defaults Model*ben a platform szempontjából, míg a *Business-Domain Model* esetén a szakterület szempontjából.

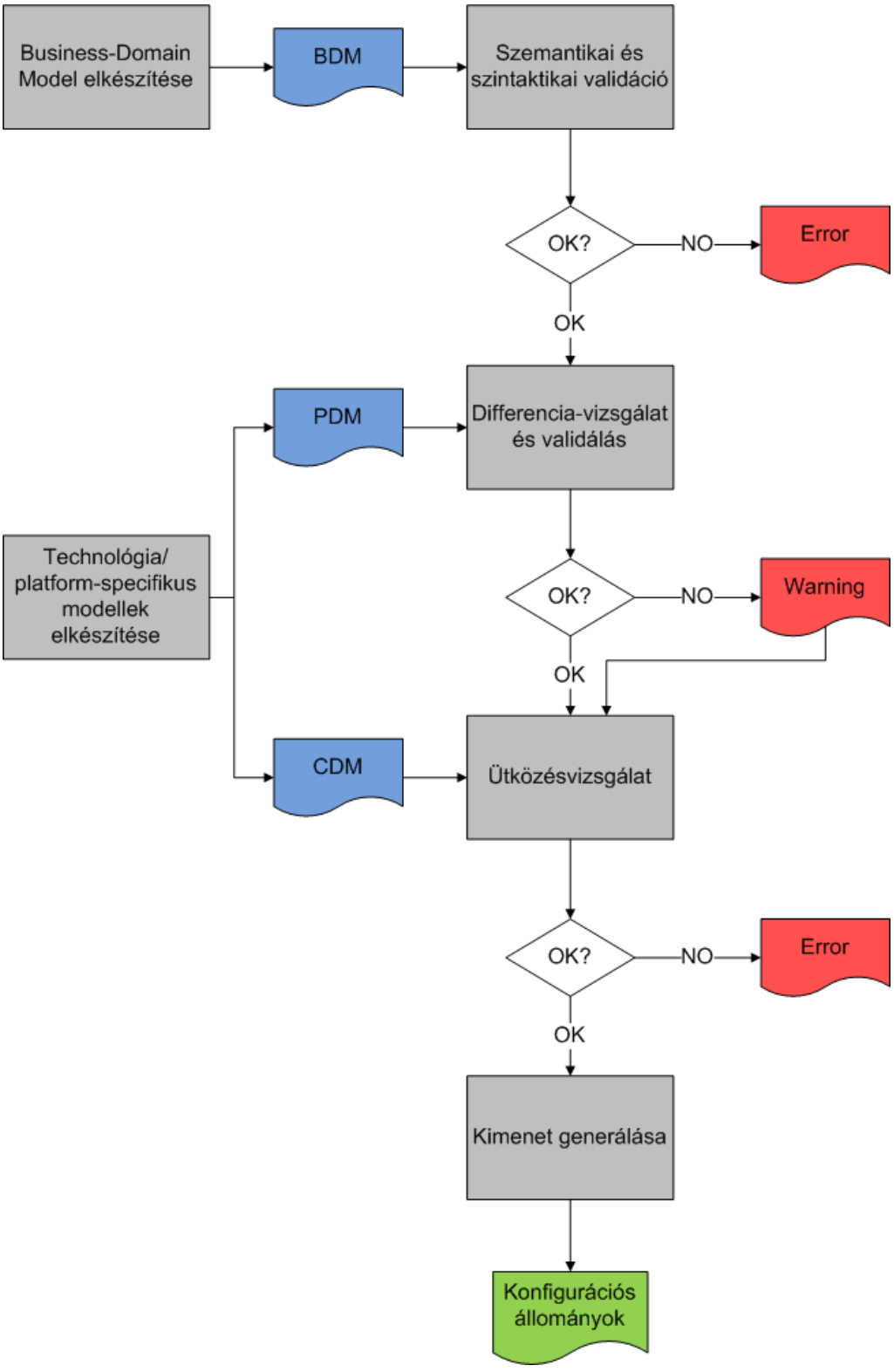
Sikerült feloldani a megkötést, amit az okozott, hogy a szakterület szakértője nem ismeri a platform sajátosságait. Így automatikusan megoldható a részleges BDM-ek kiegészítése az alapértelmezett platform-leírók nyomán. A modellezés így nagy mértékben egyszerűsödött.

A CDM hasznosan egészíti ki ezt a két modellt, hiszen lehetővé teszi a nagy méretű függéshalmazok egyszerű grafikus ábrázolását.

5.6. Modellszintézis

A modellek elkészülte után azok szintézise következik. A szintézis egy jól meghatározott folyamatot jelent, amelynek során a modellekből végül a transzformációk segítségével implementáció keletkezik.

A 22. ábra mutatja a modellszintézis pontos menetét.



22. ábra - A modellszintézis folyamata.

Első lépésben a szakterület-specialista által összeállított modell (BDM) szemantikai- és szintaktikai validációja történik meg. Ennek során az alapvető UML, valamint metamodellbeli megkötéseket ellenőrizzük (nevek egyedisége, attribútumok értékei), stb. Ha az ellenőrzés során hibát találunk, hibaüzenetet adunk vissza és a folyamat megszakad, hiszen nincs értelme azt tovább folytatni egy alapjaiban hibás modellel.

A második lépésben a *Platform-Defaults Model* szerinti validáció történik meg. Ennek során először megvizsgáljuk a modell teljességét (megvan-e minden kötelező elem, nincs-e tiltott elem, stb.), majd feloldjuk a modellek közti differenciákat és a PDM alapján kitöltjük a modell hiányzó részeit.

A harmadik lépésben az ütközések és hiányosságok megtalálása végett a *Constraints-Definition Modellel* vetjük össze a szakterület-specialista által megalkotott modellt. Ha itt mindent rendben találunk, le tudjuk generálni a kimenetet, a konfigurációs állományt, állományokat.

Szemantikai és szintaktikai validáció

Ezen transzformációk során a *Business-Domain Model*t vizsgáljuk pár előre rögzített szabály alapján. A szabályok jellemzően a metamodell és az UML adta megkötésekből származnak:

- Minden modellbeli elemnek rendelkeznie kell névvel, ami a modellen belül egyedi (*unique*).
- Az objektum-modellek sajátosságai okán minden attribútumnak rendelkeznie kell értékkel.
- Az egyes modellbeli szintek össze kell legyenek kötve. Tehát pl. mindenképp futnia kell egy élnek a *Security* karakterisztika és az *Encryption* dimenzió között.

Modell-differenciák kezelése

A szakterület-specialista és az integrációs szakértő modellje között szinte biztosan találunk a nyilvánvaló strukturális eltéréseken is túlmutató különbségeket. A következő modell-differenciákat kell értelmezni és a transzformációk során kezelni őket:

- a PDM tartalmaz olyan elemet, amelyet a BDM nem definiál
- a BDM tartalmaz olyan elemet, amelyet a PDM nem definiál

Az első esetben használható ki igazán a PDM által tartalmazott adathalmaz, amely a platform alapértelmezett beállításait tárolja. Ennek alapján ugyanis kitölthetőek a nem teljes modellek is. Ha például egy *required* típusú karakterisztika nem szerepel a szakterület-specifikus modellben, akkor a differenciát úgy oldjuk fel, hogy a PDM alapértelmezett értékeivel inicializáljuk a kimeneti modellt, majd ezek az értékek jelennek meg a konfigurációs állományokban.

A második eset kezelése nem ennyire triviális. Fontos lenne ugyanis, hogy semmilyen információ ne vesszen el a szakterület-specifikus modellből, fontos azonban az is, hogy a kimeneti eredmények a PDM szerint helyesek legyenek. Nem szeretnénk például olyan karakterisztikákat látni a konfigurációs állományokban, amik nem is léteznek.

A nem létező, vagy nem használt karakterisztikák azonban csak esztétikai okokból lehetnek zavaróak egy konfigurációs állománynál, hiszen az XML fájlban legfeljebb nem fogjuk kihasználni ezeket a bejegyzéseket. Így megengedhetjük, hogy problémát ne a transzformációs folyamat leállításával, hanem egy figyelmeztető üzenet visszaadásával folytassuk és továbblépjünk a folyamatban.

A using direktívák kezelése

A validálás során fontos feladat a *using* direktívák kezelése. Megeshet ugyanis, hogy egyes modellbeli elemek a *Business-Domain Model*ben és a *Platform-Defaults Model*ben különböző *using* értékkel szerepelnek. Valójában ez egy természetes velejárója a modellek kettéválasztásának és a platform-specifikus tulajdonságok transzparenssé tételének.

Ha egy BDM-beli és PDM-beli elem *using* attribútuma nem egyezik meg egymással, az alábbi táblázat alapján járunk el, ami a validálás és a transzformációk után keletkező értékeket a tartalmazza.

BDM érték	PDM érték	Végeredmény (Az elem a modellben jelen lesz?)
enabled	disabled	-
enabled	required	+
disabled	enabled	-
disabled	required	*Mapping error.*
required	enabled	+
required	disabled	*Mapping error.*

A fenti táblázatot ki kell egészíteni a dimenziók esetében. Itt ugyanis fontos megvizsgálni, hogy a dimenziót tartalmazó karakterisztika egyáltalán benne van-e a modellben.

A lehetséges eseteket az alábbi táblázat foglalja össze.

karakterisztika érték	dimenzió érték	Végeredmény (A dimenzió a modellben jelen lesz?)
enabled	enabled	+
enabled	disabled	-
enabled	required	+
disabled	enabled	-
disabled	disabled	-
disabled	required	-
required	enabled	+
required	disabled	-
required	required	+

Az előző táblázat alapján ha egy karakterisztika nem szerepel a modellben, akkor egyik dimenziója sem fog szerepelni. Ha pedig egy karakterisztika szerepel a modellben, akkor a dimenzión múlik, hogy részt vesz-e a modell felépítésében.

Ütközésvizsgálat

A validáció és differenciák feloldása után a modellt ütközésvizsgálatnak vetjük alá. Itt a CDM alapján vizsgáljuk meg a modell helyességét. Ha a modellben itt bármilyen probléma akad, nem engedhetjük tovább a transzformáció futását, hiszen a CDM-től való eltérés hiányos vagy hibás konfigurációs állományokhoz vezetne. Ezért egy error-típusú üzenettel szakítjuk meg a folyamatot.

Ha a CDM szerinti validáció során minden rendben zajlott, akkor a transzformációk lefutnak és legenerálják a végeredményt.

6. Egy fejlesztési eset prezentálása

Ebben a szakaszban az előzőekben már ismertetett Nemzetközi pénzügyi átutalás esettanulmány alapján építjük fel a modelleket, majd az implementált transzformációkkal legeneráljuk a konfigurációs állományokat.

Az esettanulmánynak csak egy részét fogjuk lemodellezni és a fejlesztésben felhasználni – ennek ok az, hogy a modellek méretük okán nehezen ábrázolhatók a dolgozat által nyújtott felületen, így azok egyszerűsítésére van szükség.

Nemzetközi pénzügyi átutalás

Az esettanulmány egy olyan rendszert mutat be, amely egy nemzetközi banki átutalásokat lebonyolító folyamatot támogat. A folyamat első lépéseként egy banki alkalmazott berögzíti az átutalásra szánt összeget és beütemezi azt egy adott időpontra. Ehhez egy autentikációs folyamaton kell átesnie, majd titkosított csatornán keresztül viheti fel az adatokat az adatbázisba. Miután végzett egy-egy tétel felvételével, digitális aláírását és a módosítás időpontját rögzíti a rendszer a tételhez. Az átutalások szintén csak titkosított csatornán keresztül továbbíthatóak, mindemellett megköveteljük a tranzakciók használatát.

A rendszer csak az adatbázis műveleteket naplózza, a service szinten a naplózás nem jelenik meg, mert az adatok titkosításához a bank egy saját algoritmust (*ALG123*) használ, ami azonban az ilyen folyamatoknál a naplózást kizárja.

A rendszer monitorozásához és a hibák előrejelzéséhez alapesetben szükséges a teljesítmény mérése, így a háttérben mindig fut egy vonatkozó alkalmazás. Ezek az alkalmazások azonban annyira lassítják az átutalások kapcsán történő rendszerszintű adatrögzítéseket, hogy a teljesítménymérést szintén kizárja ez a folyamat.

Az esttanulmány felhasznált elemei

Az esettanulmányból a következő elemeket emeljük ki a fejlesztéshez.

Figyelembe vesszük, hogy az átutalást berögzítő alkalmazottnak autentikálnia kell magát és az adatokat titkosított csatornán viheti fel. Ezért az SLA-hoz mindenképpen felveszünk egy *Security* karakterisztikát, amelyhez egy *Encryption*, egy *Authentication* dimenziót kapcsolunk. A tételek rögzítéskor a digitális aláírás bekerül az adatbázisba, ezért a *DigitalSignature* dimenzió is megjelenik majd a modellben, csakúgy mint a *Timestamp* az időbélyegek okán.

A naplózás (*Logging*) megjelenik a technológiai modellben, de a szakterület-specifikusban nem fog, vagy ha igen, akkor tiltva lesz.

A teljesítmény mérését a *Performace* karakterisztika, valamint annak *Throughput* és *ResponseTime* dimenziója írja le, mivel általában ezekre az értékekre vagyunk kíváncsiak teljesítménymérésnél.

A nagyobb megbízhatóság érdekében *ReliableMessaging* karakterisztikát is vezetünk be, mert nem szeretnénk, hogy elvesszenek adatok a távoli kommunikáció során.

6.1. Modellalkotási eljárások

A modellek sztereotípiáit az UML4SOA és a *SeparatedModeling* profilok szolgáltatják, az ábrák RSA 7.5.[25][26][27] segítségével készültek.

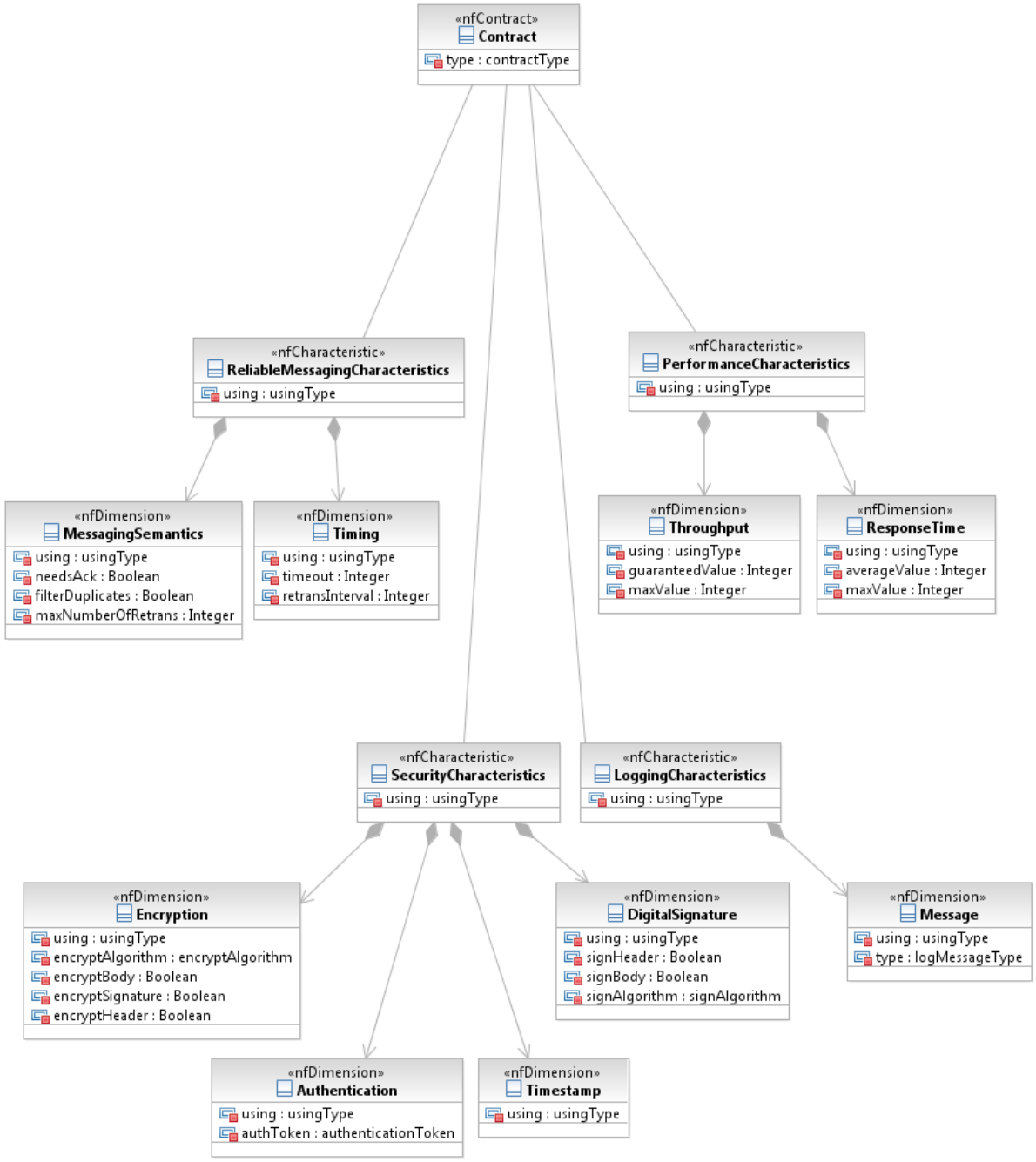
A Platform-Defaults Metamodel összeállítása

A *Platform-Defaults Modelt* leíró metamodell egy osztálydiagram. Központi eleme a *Contract* nevű osztály, amely az SLA-t jelképezi a modellben. Sztereotípiája *nfContract*. A *Contract*ról csak annyit tartunk nyilván, hogy milyen típusú. Fontos ezt megadni, mert később a már elkészült modelleket újra fel lehet használni, ehhez azonban tudni kell, milyen típusú modellről van szó. Pár példa a modelltípusokra (a bankrendszernél maradva): *InternationalCreditTransfer*, *PublicTransfer*, *CorporateHR*, stb.

A *Contract* alatti karakterisztikák – *ReliableMessagingCharacteristics, PerformanceCharacteristics, LoggingCharacteristics* és *SecurityCharacteristics* – mind el vannak látva a már említett *using* paraméterrel. Metamodellről lévén szó, értékeket nem adunk meg, csupán típusokat definiálunk.

A karakterisztikák alatti dimenziók tartalmazzák mind a *using* attribútumot, mind azokat az attribútumokat, amelyeket az előzőekben az esettanulmány értelmezésénél kiemeltünk.

A *ContractType, usingType, logMessageType, encryptAlgorithm, authenticationToken, signAlgorithm* mind a modellben definiált *Enum* típusú struktúrák.



23. ábra - PDM metamodell.

A Platform-Defaults Model összeállítása

A PDM metamodellje alapján nincs nehéz dolgunk, hiszen csak példányosítani kell a metamodellt. (Ráadásul ma már szinte minden tervezőeszköz nyújt segítséget ehhez a lépéshez.)

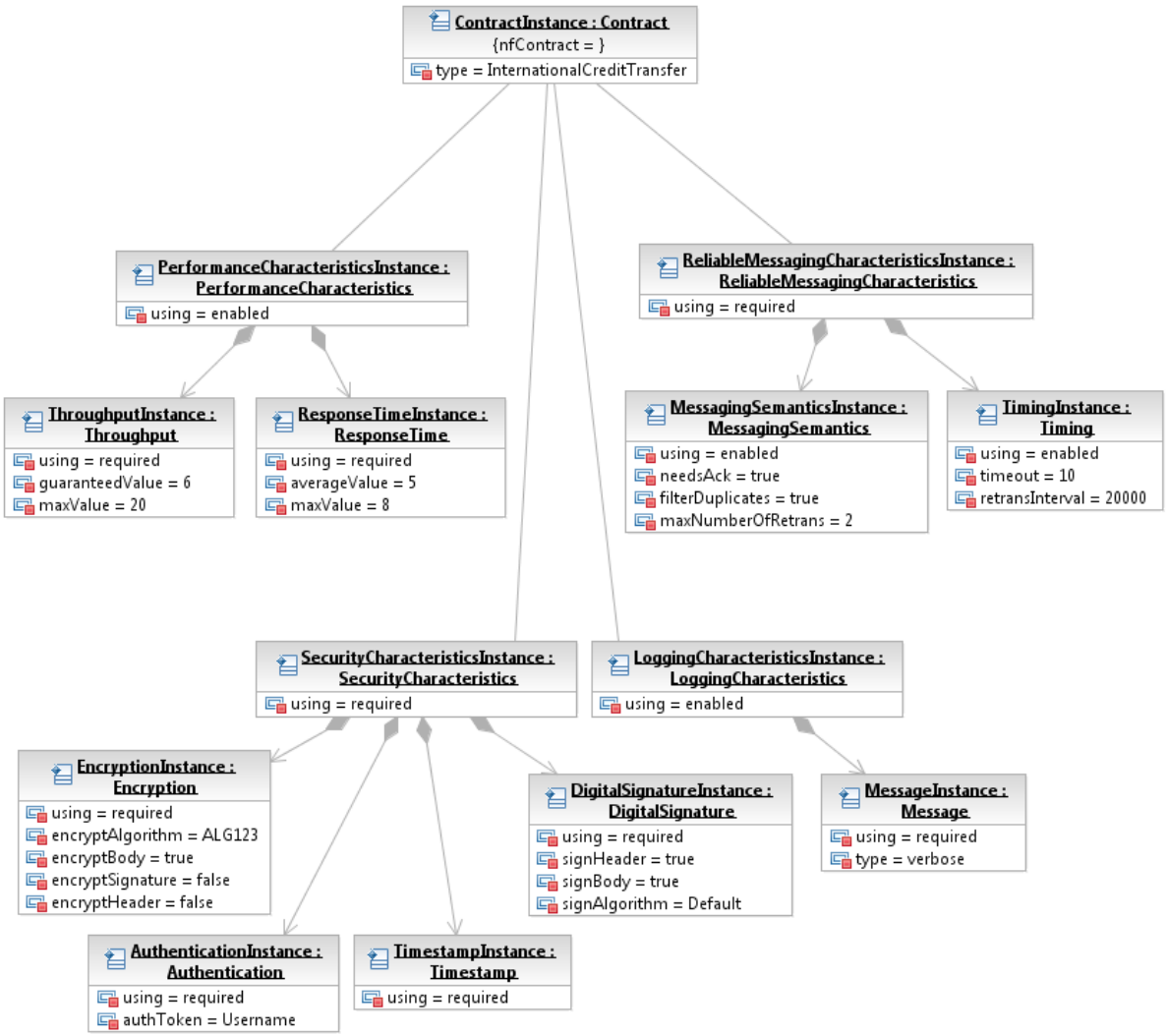
A PDM-ben megadott értékek az integrációs szakértő véleményét tükrözik a rendszerrel kapcsolatban. Természetesen ő nem lát bele nagyon mélyen, ezért csak így, nagy vonalakban tudja megadni, hogy mire milyen érték lehet jó és alapértelmezett.

A *Contract(Instance)* típusa *InternationalCreditTransfer* lett. Ez egész pontosan jellemzi a modellt.

Az esettanulmánnyal összhangban megköveteljük a *Security* karakterisztika jelenlétét a rendszerben, ahogy összes dimenziójáét is, ezért ezek *using* attribútuma required értéket kap, csakúgy, mint a *ReliableMessaging* karakterisztika és annak minden dimenziója.

A *Logging* és *Performance* karakterisztikák használatát nem követeljük meg, azonban előállhatnak olyan esetek, amikor mégis szeretnénk használni, ezért *enabled* értékkel inicializáljuk a *using* attribútumot.

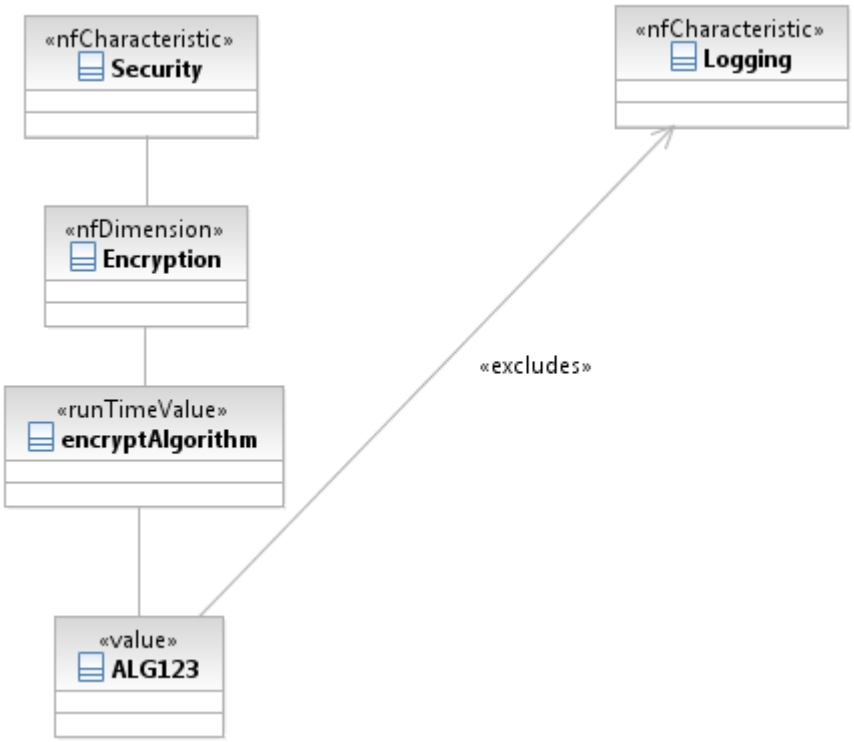
Ezen karakterisztikák dimenziói mind *required* értékkel szerepelnek a *using* attribútum vonatkozásában, ami azt jelenti, hogy ha ezek a karakterisztikák használatban vannak, a kapcsolódó dimenziókat is kötelező használni, egyéb esetben nem jelennek meg a modellben.



24. ábra - PDM példány.

A Constraint-Definition Model összeállítása

A *Constraint-Definition Model* megtervezésénél csak egy dolgot kell figyelembe vennünk: a bank által használt *ALG123* titkosítási algoritmus kizárja a *Logging* karakterisztika használatát.

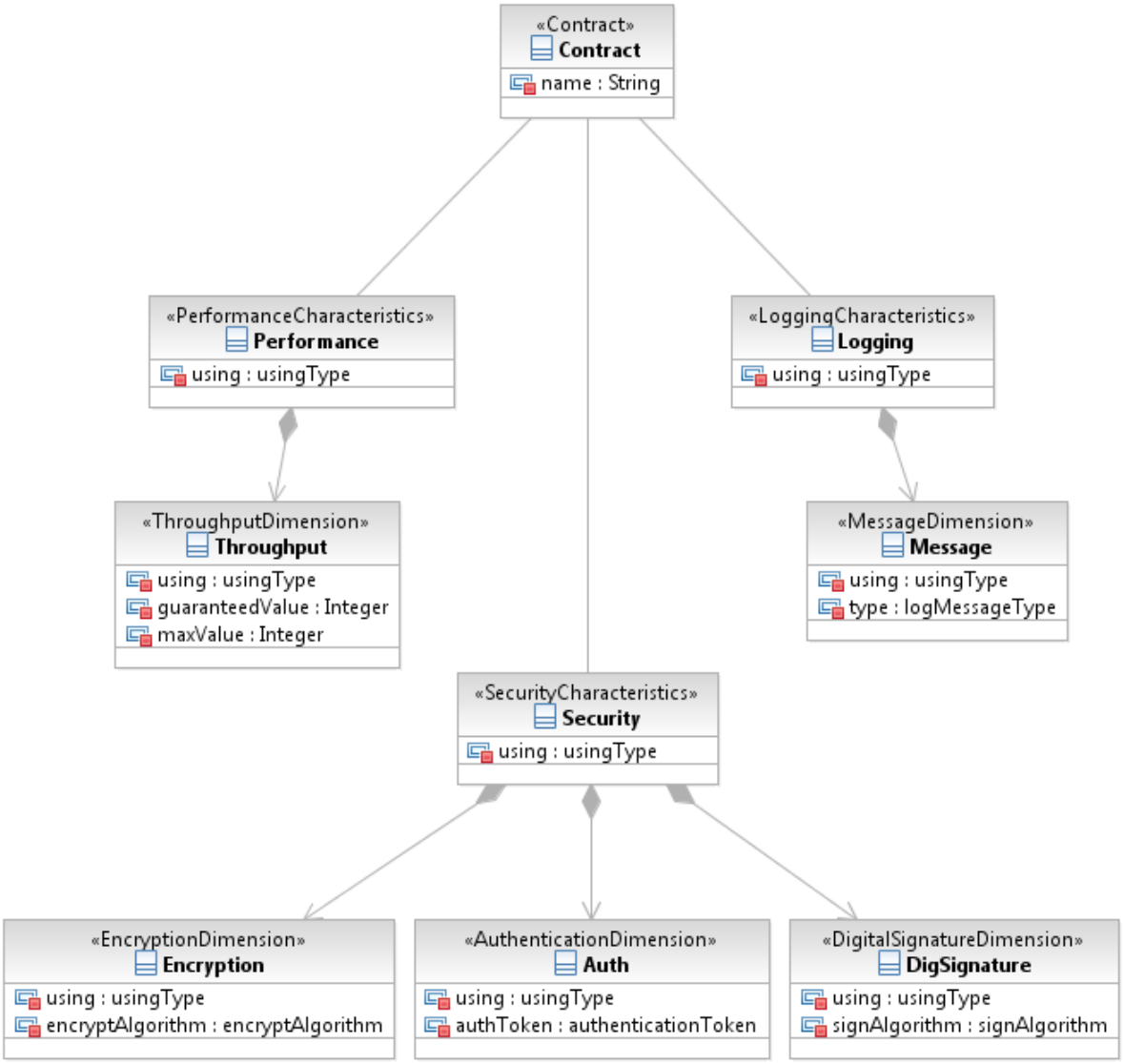


25. ábra - CDM példány.

A Business-Domain Metamodel összeállítása

A BDM metamodellnél csupán azt adjuk meg, hogy a modellben *Performance*, *Logging* és *Security* karakterisztikáink lesznek, valamint ezeknek *Throughput*, *Message*, illetve *Encryption*, *Authentication* és *DigitalSignature* dimenzióik jelennek meg az ábrán látható attribútumokkal.

A modellen látható, hogy a szakterület-specialistának nem kellett pontos neveket (*Auth*, *DigSignature*) adnia, valamint nem kellett minden attribútumot és karakterisztikát felvennie a modellben.

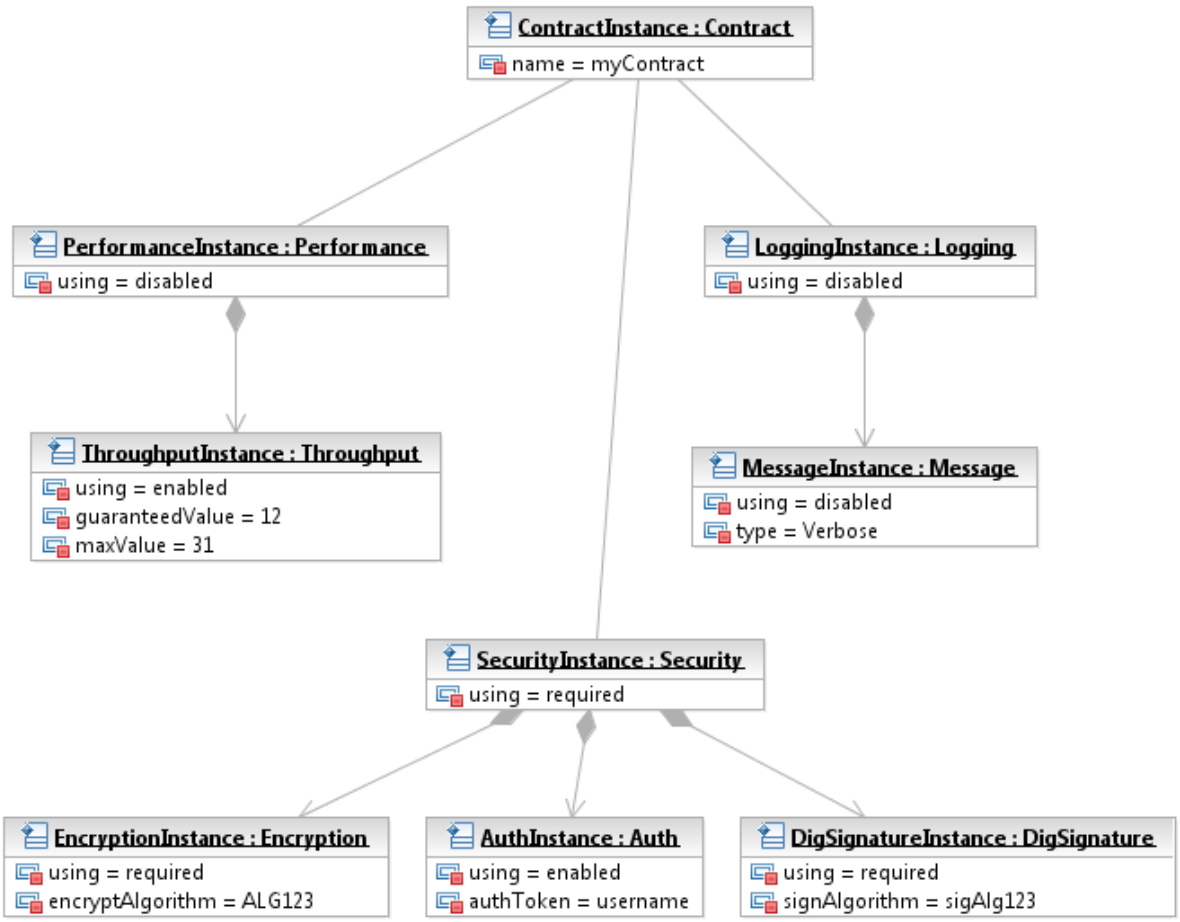


26. ábra - BDM metamodell.

A Business-Domain Model összeállítása

A BDM összeállítása a metamodell alapján nagyon egyszerűen történik – csupán arra kell figyelnie a szakterület-specialistának, hogy minden attribútum értéket kapjon.

A 27. ábrán látszik, amint a szakterület-specialista tervező a *Logging* és *Performance* karakterisztikák *using* attribútumait *disabled* értére állította – így ezek a kimeneten nem jelennek meg.



27. ábra - BDM példány.

6.2. Transzformációs eljárások

Az esettanulmány végigvitele során a transzformációk prototípusait készítem el. A cél a demonstráció, nem a teljes funkcionalitás.

A transzformációk a VIATRA2 keretrendszerben kerülnek implementálásra. (A keretrendszer kapcsán lásd a 10.2. fejezetet.)

Jelen dolgozat keretei nem elégségesek a transzformációk pontos ismertetéséhez, ezért csak részleteket idézek belőlük.

Szemantikai és szintaktikai validáció

A szemantikai és szintaktikai validációt végző egység egy már kipróbált transzformáció. Előzetes verziója a 2009-es *SENsoria SUmmer School* labor-foglalkozásaira készült el.

A 22. ábra alapján ennek bemenete a *Business-Domain Model*. A transzformáció során az 5.6. szakaszban kifejtett kérdések alapján döntjük el, hogy a modell szemantikailag, szintaktikailag korrekt-e.

A következő kódrészletben azt ellenőrizzük, hogy létezik-e minden *Contract* példányhoz legalább egy *Characteristic* példány, valamint hogy ezek neve egyedi-e (*unique*) az adott modellben.

```
forall NFContractClass below ModelIn with find
  nfContractInstancePattern(NFContractClass) do seq {
    choose CharacteristicClass below ModelIn with find
      nfContractWithNfCharacteristicPattern(NFContractClass,
        CharacteristicClass) do seq {
          update Char2ContractNum = Char2ContractNum + 1;
      }
    forall CharacteristicClass1, CharacteristicClass2 below ModelIn with
      find
        nfCharacteristicUniqeNamePattern(CharacteristicClass1,
          CharacteristicClass2) do seq {
            update CharUniqueName = CharUniqueName + 1;
        }
  }
```

A kódrészlethez egyik tartozó minta (a *Contract*ot definiálja):

```
@incremental pattern nfContractInstancePattern(NFContractClass) = {
  uml2.metamodel.Stereotype(NFContractStereotype);
  check(name(NFContractStereotype)=="Contract");
  uml2.metamodel.Class(NFContractClass);
  uml2.metamodel.Element.appliedStereotype(AS1, NFContractClass,
    NFContractStereotype);
  uml2.metamodel.InstanceSpecification(NFContracInstance);
  uml2.metamodel.InstanceSpecification.classifier(C1, NFContracInstance,
    NFContractClass);
}
```

És végül a kimenet:

```
if (CharNum !=0) seq {
  if (CharUniqueName==0)
    println("NF-Characteristic names.........OK!");
  else seq {
    println("NF-Characteristic names.........ERROR!");
    println("\tERROR MESSAGE: Names of some NF-Characteristics related to
      the same NF-Contract are duplicated!");
    fail;
  }
}
```

PDM-alapú differenciavizsgálat

A *Platform-Defaults Model*-alapú differenciavizsgálat során a PDM-ben lévő elemeken végighaladva megpróbálunk keresni azoknak egy *Business-Domain Model*-beli párt. Ha létezik ilyen, akkor a BDM-beli elem által leírt tulajdonságokat készítjük be a kimeneti fájlba, ha nincs ilyen pár, akkor a PDM-beli alapértékek kerülnek a kimenetre a későbbiekben.

Ezután a BDM elemein haladunk végig, és megvizsgáljuk, van-e olyan elem, amihez nincs PDM-beli pár. Ha van ilyen, akkor *warning* üzenetet adunk az elemre hivatkozva, de az elemet bekészítjük a kimenetre.

CDM-alapú ütközésvizsgálat

A *Constraint-Definition Model*-szerinti ütközésvizsgálat során végigmegyünk a CDM-en *excludes* és *requires* sztereotípiájú asszociációt keresve. Egy-egy ilyen találtnál megvizsgáljuk a résztvevő feleket, akik az asszociáció két végén helyezkednek el, majd miután meghatároztuk a CDM-beli megkötés irányát, hasonló mintát keresünk a BDM és PDM által közösen alkotott modellben.

Ha találunk olyan elemet, amely megsért egy CDM-beli szabályt (*antipattern*, *constraint-violation*), akkor *error*t adunk vissza az elemre hivatkozva.

6.3. Eredmény

Az eredményben a BDM-ből csak a Security karakterisztika jelenik meg, illetve annak dimenziói. Viszont a PDM-ben definiáltuk a ReliableMessaging karakterisztikát, ami szintén meg fog jelenni.

A kimeneti XML dokumentum:

```
<?xml version='1.0'?>
<contract name="myContract">
  <wsp:Policy wsu:Id="myContractSecurityPolicy"
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
     wssecurity-utility-1.0.xsd"
     xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:Authentication
           xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <wsp:Policy>
            <wsp:authToken>
              <wsp:Policy>
                <sp:Username/>
              </wsp:Policy>
            </wsp:authToken>
          </wsp:Policy>
        </sp:Authentication>
        <sp:Encryption
           xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <wsp:Policy>
            <wsp:encryptBody/>
            <wsp:encryptAlgorithm>
              <wsp:Policy>
                "ALG123"
              </wsp:Policy>
            </wsp:encryptAlgorithm>
          </wsp:Policy>
        </sp:Encryption>
        <sp:DigitalSignature
           xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <wsp:Policy>
            <wsp:signHeader/>
            <wsp:signBody/>
            <wsp:signAlgorithm>
              <wsp:Policy>
```

```
                "sigAlg123"
              </wsp:Policy>
            </wsp:signAlgorithm>
          </wsp:Policy>
        </sp:DigitalSignature>
        <sp:Timestamp
           xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <wsp:Policy>
            <wsp:useTimestamp/>
          </wsp:Policy>
        </sp:Timestamp>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  <wsp:Policy wsu:Id="myContractRMPolicy"
        xmlns:wsp=http://schemas.xmlsoap.org/ws/2004/09/policy
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
           wssecurity-utility-1.0.xsd"
        xmlns:wsrm="http://ws.apache.org/sandesha2/policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <wsrm:filterDuplicates>true</wsrm:filterDuplicates>
        <wsrm:needsAck>true</wsrm:needsAck>
        <wsrm:maxNumberOfRetrans>2</wsrm:maxNumberOfRetrans>
        <wsrm:retransInterval>20000</wsrm:retransInterval>
        <wsrm:timeout>10</wsrm:timeout>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
</contract>
```

Látható, hogy a kimeneti állományban a modell szerinti elemek és értékek jelentek meg.

7. Kapcsolódó technikák és publikációk

7.1. Grafikus programozás és on-the-fly futtatókörnyezetek

Fontos észrevenni az összefüggést: a modellvezérelt fejlesztés folyamata analóg a nem-modellvezérelt módszertanok kódírás-fordítás folyamatával.

Utóbbiaknál ugyanis, bár manuálisan implementálunk, „kézzel írunk kódot”, az eredmény még mindig nem az, amit a futtatókörnyezet használ – csupán egy modellje annak. Igaz, hogy alacsony szinten absztrahált és nem klasszikus értelemben vett modell. A futtatókörnyezet számára azonban teljesen mindegy az, hogy a gépi kódot a fordító egy imperatív programozási nyelvből, avagy egy rendszermodellből nyeri, ha mindkettő platform megvalósítja a fordító által definiált interfészeket.

A *LabView* megjelenése és elismerése bizonyította, hogy a „grafikus programozásnak” helye van a modern fejlesztési technikák között.

Már ma is rendelkezésre állnak olyan technológiák, amelyek bemeneti oldalukon nem kódot, hanem egy modellt várnak, amit on-the-fly transzformációk segítségével bájtkóddá alakíthatnak és futtathatnak.

Executable UML (xUML)[28]

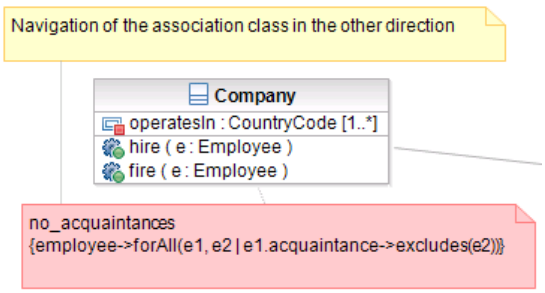
Az *xUML* a rendszer modellezését UML profilokkal kiegészítve oldja meg, azaz elvileg nincs szükség más programnyelv bevonására. A modellek magas szinten absztraháltak, de tesztelhetők és lefordíthatók alacsonyabb szintű programnyelvekre. Az MDA-koncepciót a platformfüggetlen modellek és a platformfüggetlen modelleket platform-specifikus modellekké leképző transzformációk implementációjával támogatja.

Action Semantics[29]

Az *Action Semantics* egy platform-független objektum-orientált megközelítést használó módszer UML modellbeli végrehajtható akciók és a metódusok viselkedésének leírására. Lehetővé teszi magas szinten automatizált és optimalizált kódok generálását UML CASE eszközökhöz.

7.2. OCL (Object-Constraint Language)

Az OCL egy, az IBM által kifejlesztett deklaratív nyelv, ami UML modellek szabályainak leírására képes. A Constraint-Definition Modelnél említésre került ez a technika, mint esetleges alternatíva a modellelemek viselkedésének leírására, azonban az OCL a modell számunkra fontos aspektusait csak bonyolult és nagy méretű adathalmazokkal képes megadni.



28. ábra - Egy egyszerű OCL leíró.[36]

A 28. ábra egy egyszerű OCL leírást ad meg. Látható, hogy egy kétváltozós deklaratív kifejezés sem olvasható olyan egyszerűen, mintha grafikusan ábrázolnánk azt, főleg ha több kizárás/összetartozás párost definiálunk. Akkor sem egyszerű a dolgunk, ha esetleg nem két tagból álló kapcsolatokat kell leírni, hanem ennél több modellelemre kell definiálni szabályokat.

Egy nagyobb modell szabályainak leírása tehát OCL segítségével azért nehézkes, és azért megfelelőbb számunkra a grafikus ábrázolás, mert: OCL esetén az adatok áttekinthetetlenül halmozódnak, valamint a bonyolultabb megkötések definiálása megköveteli a deklaratív programozás alapelveinek ismereteit és egy új programozási nyelv elsajátítását.

A *Constraint-Definition Model*-ről szóló fejezetben láttuk, hogy az ott ismertetett módszer egyszerűbben kezeli ezeket az aspektusokat a modellben.

7.3. Kapcsolódó publikációk

A szolgáltatás-orientált rendszerek nem-funkcionális jellemzőinek vizsgálatával, modellezésével és transzformációk útján történő konfigurációs elemek létrehozásával foglalkozik az [1] irodalom, ami a jelen dolgozat elsődleges forrását képezte. Ebben a publikációban bemutatásra kerül a SOA rendszerek NF-jellemzőit leíró modellek egy olyan metamodellje, amely később több helyen is alapul szolgált a további munkálatokhoz[3][4], valamint az UML4SOA UML profil is, amelyet a SENSORIA projekt keretein belül épp ezen aspektusok modellezésére fejlesztettek ki.

Bár jelen dolgozat nem tér ki rá, a szolgáltatás-orientált rendszerek vizsgálatának nagyon gyakori szempontja a teljesítőképesség (*performability*) területe. Az ilyen jellegű módszereket tekinti át a [6] irodalom, amely az UML4SOA profil segítségével létrehozott modelleket a PEPA (*Performance Evaluation Process Algebra*) teljesítményelemző eszközzel vizsgálja.

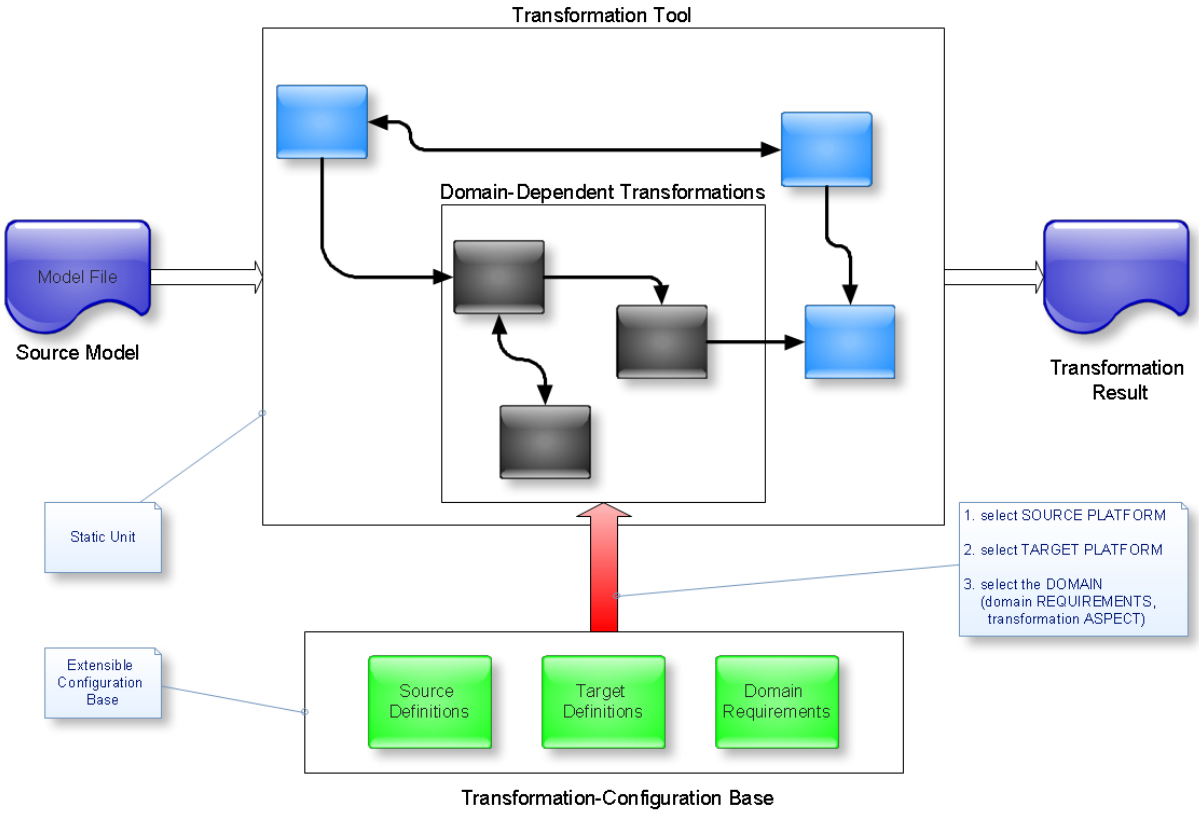
Az [5] irodalomban a szerzők bemutatják az SLA-specifikáció alapú teljesítőképesség-analízis (*performability analysis*) módszerét megbízható üzenetküldést (*reliable messaging*) használó middleware-ek kommunikációjának segítségével, szembe állítva a technikát klasszikus workflow-alapú teljesítmény-analízissel. Az modellezéshez, valamint az analízishez ez esetben is az UML4SOA, illetve a PEPA eszközök kerültek felhasználásra.

8. Továbblépési lehetőségek

8.1. Automatikus mintakitöltés

A dolgozat korai szakaszában felmerült az ötlet, hogy a transzformációk fejlesztésének költséghatékonyságán javítsunk az *automatikus mintakitöltés* módszerével. A módszer alapötlete szerint tehát nem a modellek költséghatékony tervezésére, hanem a transzformációk ilyen jellegű fejlesztésére helyezzük a hangsúlyt.

A cél az egyes transzformációk „rugalmassá” tétele olyan módon, hogy azokat később más típusú modellek *mapping*jéhez is fel lehessen használni. Vannak ugyanis olyan közös részei egy-egy transzformációnak, amelyeket kiemelve a transzformációs láncból, majd általánosítva azokat, felhasználhatók lennének más típusú transzformációknál is.



29. ábra

A 29. ábra szemlélteti a mintakitöltés alapú megoldás menetét.

A fejlesztő kiválasztja a bemeneti modell platformját. Célunk lehet támogatni pl. UML-től különböző nyelveket, az UML különböző profiljait, de az is elképzelhető, hogy nem a SOA-rendszerekhez megszokott modellel van dolgunk, de szeretnénk azt is transzformálhatóvá tenni. Ezután a célplatformot adja meg a fejlesztő, amire a modellt transzformálni szeretné. Ez lehet akár különféle WS-platformok egyike, analízis eszközök platformja, stb.

Az utolsó lépésben pedig a domain-jellemzőket adhatjuk meg. Itt a domain követelményeire kell elsősorban gondolni, azonban lehetőség van megadni egy transzformációs aspektust is, ami meghatározza, hogy a kimeneti oldalon a rendszer milyen jellemzői jelennek meg (pl. biztonság, vagy teljesítőképesség).

A módszer három könyvtárat használ, amelyek közül kiválasztható a három lépéshez az érték. Ezek a könyvtárak (*Source Definitions, Target Definitions, Domain Requirements*) az *Extensible Configuration Base*-ben (XCB) találhatók meg. Az XCB egy könnyen bővíthető, egyszerű modellekből álló gyűjtőhely.

A cél egy olyan transzformáció létrehozása volt, amely a SOA rendszerek NF modelljeiből egyrészt képes az [1] irodalomban látható módon xml-alapú konfigurációs állományokat létrehozni, másrészt ugyanezen modellekből PEPA-kompatibilis[5] implementációkat is képes előállítani.

A módszernek több különböző VIATRA2 keretrendszerbeli lehetséges támogatását sikerült azonosítani, így azt gondolom, a dolgozat elején vázolt problémákat erről az oldalról is megközelíthetjük, és valószínűleg ebből az irányból is hatékony módszereket tudnánk kialakítani, ennek eldöntésére azonban a dolgozat keretei nem elégségesek.

A feketedoboz-megközelítés okén hasznos megközelítésnek tűnik a **software factory**. A módszer lényege, hogy rendszer factory módszerrel gyártja le azokat a komponenseket, amikre a fejlesztőnek szüksége van. A komponensek jellegének meghatározása természetesen nem csak runtime értékmegadással történhet, hanem konfigurációs modell alapján is.

Egy-egy fekete dobozról csak annyit tud a fejlesztő, hogy pl. a mögötte álló transzformáció validációt végez a bemeneti modelleken. Természetesen a validáció aspektusa, kiértékelése, érzékenysége, stb. függ a célplatformtól és a transzformáció jellegétől. Itt jön a képbe a software factory. A fő transzformációs szál egy-egy fekete dobozhoz érve meg tudja határozni, hogy milyen jellegű validációra van szükség az adott ponton, így on-the-fly építi meg a komponenst – a fejlesztőnek egy sort sem kell írnia hozzá, mert az egész a már előzőleg létrehozott libraryk-ból, az XCB-ben lévő információkból épül fel.

A komponensek összekapcsolásáról az **aspektusorientált programozás**ból ismert *model weaver*, azaz „modell szövő” eljárás gondoskodik[30].

9. Eredmények, értékelés

Dolgozatom célja az volt, hogy a korábbi munkákban[1][2][3] már ismertetett modelltranszformációk alapján testreszabható, költséghatékonyan fejleszthető és karbantartható transzformációkat fejlesszek ki.

Ennek érdekében a dolgozat során felállítottam egy jól meghatározott szemantikai és szintaktikai szabályokat követő **grafikus nyelvet**, amely az UML alapelemeinek, valamint az UML4SOA profil elemeinek használatára épít. A nyelvet kiegészítettem egy **UML profillal**, a *SeparatedModeling* nevűvel, amely a **modellek szétválasztását** támogatja.

A grafikus nyelvet metamodellekkel, modellekkel és a már említett profillal definiáltam.

Módszert adtam a modellek kialakítására, aminek nyomán a rendszerek modellezése **szétválaszthatóvá** válik technológia- és szakterület-specifikus részekre. Így a feladatkörök is jobban elválaszthatók egymástól és az egyes területek szakembereinek felelősségi köre egymástól távol tarható, az átfedések minimalizálhatók. A módszernek köszönhetően sikerült a modelleket olyan szinten szétválasztani, hogy egymásra egyáltalán ne legyenek hatással, így az egyes modellek cseréje megoldható a többi módosítása nélkül, ilyen módon lehetőség nyílik az üzleti igények módosítására a platform érintése nélkül, illetve a platform cseréje az üzleti réteg változtatása nélkül.

Így biztosítottam a **transzformációk testreszabhatóságát** és költséghatékony fejlesztését, karbantartását.

Konkrét **esettanulmányon keresztül mutattam be** a módszerek használhatóságát, amelynek során elvégeztem a technológia- valamint a szakterület-specifikus modellek felállítását és modelltranszformációk prototípusainak implementálását.

A dolgozat során megalkotott nyelv *Platform-Defaults Model* része egy olyan modellt ad meg, amelynek segítségével lehetővé válik a nem-funkcionális modellezés **technológiai jellemzőit** megadni anélkül, hogy a szakterület, a *business-domain* kívánalmait mélyrehatóbban ismernénk.

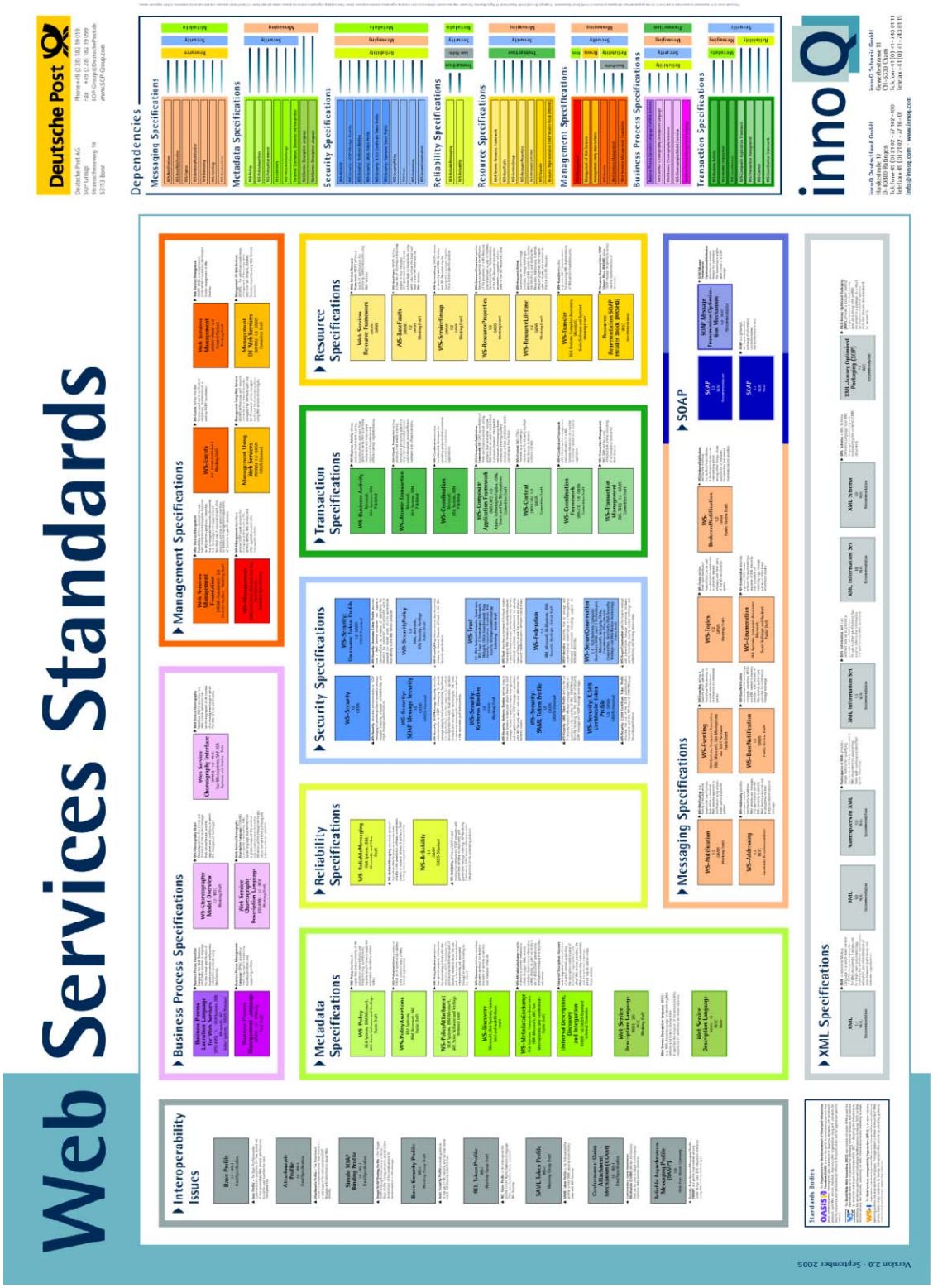
A *Constraint-Definition Model* egy olyan technikát definiál, amely grafikusan képes ábrázolni a modellben jelen lévő **globális megkötéseket** a modellelemek egymással alkotott viszonyának aspektusában.

A szakterület-specifikus igényeket leíró *Business-Domain Modell* **transzparenssé teszi** számunkra **a platform technológiai jellemzőit**, részleteit, így a modell megalkotása közben nem kell figyelembe venni azokat.

Előállítottam a **modelltranszformációk prototípusait**, amelyek segítségével a modellszintézis megvalósítható.

10. Függelék

10.1. WS-* szabványok[31]



http://www.commentcamarche.net/faq/images/hB3X94QbtyKVECuE.png

10.2. A dolgozat során használt eszközök

RSA 7.5.

Az RSA 7.5 (*Rational Software Architect*) az IBM Eclipse alapú modellező eszköze. A [3] irodalom kitér a 7.5-ös verziónak a VIATRA keretrendszerrel való inkompatibilitási problémáira is, amelyek idő közben azonban megoldódtak.

További refenrenciák: [25],[26],[27].

http://www-01.ibm.com/software/awdtools/architect/swarchitect/

VIATRA2

A VIATRA2 (*VIsual Automated model TRAnsformations*) a BME-MIT Hibatűrő Rendszerek Csoportja (FTSRG) csoportja által 1998 óta fejlesztett és karban tartott többfunkciós eszköz. Számunkra a modelltranszformációk előállításához használt fejlesztői környezetet szolgáltatja.

Referenciák: [32],[33],[34].

http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html

11. Rövidítések, szakkifejezések jegyzéke

BDM – *Business-Domain Model*. A szakterület-specialista által összeállított modell a szakterület fontosabb követelményeivel.

CDM – *Constraint-Definition Model*. Megadja, ha egyes elemek csak egymással, illetve egymás nélkül tudnak szerepelni a modellben.

DRDS – *Domain Rules Descriptor Set*. A *Constraint-Definition Model*t és a *Platform-Defaults Model*t fogja össze egy helyen.

ESB – Enterprise Service Bus. A SOA architektúrák azon eleme, amelyen nagyon gyors az adatközlés, és amelyet jellemzően csak nagyvállalatoknál találunk meg.

OASIS – Oragnization fot the Advancement of Structured Information Standards. Szabványügyi szervezet.

OCL – Object-Constraint Language. Az UML modellek szabályait leírni képes deklaratív nyelv, amelyet az IBM fejlesztett ki.

MDA – *Model-Driven Architecture*

MDD – *Model-Driven Development*

NFP, NF-Paraméter – nem funkcionális (Non-Functional) paraméter

SLA - *Service Level Agreement*. A szolgáltató és az igénylő közti szerződés, amely a szolgáltatással kapcsolatban tartalmazza a legfontosabb NF-paraméterek.

SOA – *Service-Oriented Architecture*. Szolgáltatásorientált Architektúra.

SoaML – SOA Modeling Language. Az UML egy kiterjesztése a SOA modellezésének céljából.

UML – Unified Modeling Language.

UML4SOA – UML profil a SOA rendszerek modellezése céljából.

PDM – *Platform-Defaults Model*. Definiálja a modell alapvető felépítését és megadja az NF-paraméterek alapértelmezett értékeit – mindezt a technológiai oldalról megközelítve.

RSA – *Rational Software Architect*. Az IBM modellező alkalmazása.

VIATRA – *VIsual Automated model TRAnsformations*. A BME-MIT FTSRG által létrehozott és karbantartott MDD-vonatkozású alkalmazás. Modelltranszformációk létrehozására alkalmas.

WS – *Webservice*

WS-* - A webservice-ekhez kapcsolódó szabványok összefoglaló neve.

WSDL – *WebService Description Language*. A webservice „külső” felületét írja le a mögötte lévő alkalmazások funkcióinak publikálásával.

12. Irodalomjegyzék

[1] L. Gönczy, D. Varró – *Design and Deployment of Service Oriented Applications with Non-Functional Requirements*, (2008)
[2] S. Gilmore, L. Gönczy, N. Koch, P. Mayer, M. Tribastone, D. Varró – *Non-Funtional Properties in the Model-Driven Development of Service-Oriented Systems*, (2008)
[3] Dávid István – *Szolgáltatásorientált rendszerek modell alapú fejlesztése* (2009)
[4] L. Gönczy, D. Varró – *Non-Functional Properties in Model-Driven Development of Service Oriented Systems*, SENSORIA Summer School 2009,
http://sensus09.mit.bme.hu/download/sensus2009_nfp_talk.pdf 16.-17. oldal, (2009)
[5] L. Gönczy, Zs. Déri, D. Varró - *Model Transformations for Performability Analysis of Service Configurations*
[6] Zs. Déry - *Model-driven development and analysis of service-oriented architectures*, BME TDK, (2007)
[7] http://www.mdd4soa.eu/
[8] http://www.dcs.ed.ac.uk/pepa
[9] http://www.omg.org/mda/specs.htm.
[10] http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html
[11] http://www.openarchitectureware.org/
[12] http://www.eclipse.org/m2m/atl/
[13] http://adm.omg.org/
[14] Joe McKendrick - *Bray: SOA too complex; 'just vendor BS'*, http://blogs.zdnet.com/service-oriented/?p=597, 2006
[15] Joe McKendrick - *SOA services still too constrained by applications they represent*,
http://blogs.zdnet.com/service-oriented/?p=2306, 2009
[16] JP Morgenthal - *The Reason Enterprise Architects Should Study Economy*,
http://www.jpmorgenthal.com/morgenthal/?p=144, 2009
[17] Tim Bray – The Loyal WS-Opposition,
http://www.tbray.org/ongoing/When/200x/2004/09/18/WS-Oppo, (2004)
[18] http://www.oasis-open.org/specs/
[19] http://ws.apache.org/rampart/
[20] http://ws.apache.org/sandesha
[21] N. Mihindukulasooriya – *Web Services Security with Apache Rampart* – Part 1 (Transport Level Security), http://wso2.org/library/3190, (2008)
[22] N. Mihindukulasooriya – *Understanding WS – Security Policy Language*,
http://wso2.org/library/3132,(2008)
[23] Ch. Jayalath – *Apache Sandeshsa2 Configuration Framework*, http://wso2.org/library/204, (2008)
[24] http://www.omgwiki.org/SoaML/doku.php
[25] D. Misic - *Authoring UML profiles using Rational Software Architect and Rational Software Modeler*, http://www.ibm.com/developerworks/rational/library/05/0906_dusko/ , (2005)
[26] D. Misic - *Authoring UML Profiles: Part 2. Using Rational Software Architect, Rational Systems Developer, and Rational Software Modeler to manage change in UML Profiles*,
http://www.ibm.com/developerworks/rational/library/08/0429_misic2/index.html, (2008)
[27] *IBM RSA documentation*,
http://publib.boulder.ibm.com/infocenter/rsasehlp/v7r5m0/index.jsp
[28] http://www.executableumlbook.com/
[29] D. Varró, A. Pataricza - *UML Action Semantics for Model Transformation Systems*, (2003)
[30] P. Domokos, I. Majzik – *Design and Analysis of Fault Tolerant Architectures by Model Weaving*

[31] innoq.com – Web Services Standard sas of Q1 2007, 2007
[32] OptXware Research and Development LLC – *The VIATRA-I Model Transformation Framework – Pattern Language Specification*, (2006)
[33] OptXware Research and Development LLC – *The VIATRA-I Model Transformation Framework – User's Guide*, (2007)
[34] OptXware Research and Development LLC – *The VIATRA-I String Manipulation Function Library*, (2006)
[35] http://ws.apache.org/axis2/
[36] http://www.omg.org/technology/documents/formal/ocl.htm

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani konzulensemnek, **Gönczy László**nak, amiért vállalta TDK dolgozatom végigvezetését, pótolhatatlan segítséget és útbaigazításokat nyújtva az elmúlt hónapok során.

Továbbá köszönetet mondok **Varró Dániel**nek szakmai tanácsaiért, **Ráth István**nak és **Bergmann Gábor**nak a VIATRA2 keretrendszerben időszakosan felmerülő problémák orvoslásáért és a keretrendszerrel kapcsolatos segítségért.