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

Modellelemek effektív jogosultságainak származtatása finomszemcsés hozzáférési szabályokból

Szakdolgozat

Készítette
Balogh Tímea

Konzulens
Debreceni Csaba

MTA-BME Lendület
Kiber-fizikai Rendszerek Kutatócsoport

2017. december 8.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Balogh Tímea*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2017. december 8.

Balogh Tímea
hallgató

Kivonat

Bizonyos informatikai rendszerek üzemeltetése esetén a velük szemben támasztott elsődleges követelmény, hogy ne veszélyeztessenek emberi életeket, ne okozzanak anyagi, természeti károkat. Ilyen úgynevezett biztonságkritikus rendszerek például a vasúti-, repülőgépirányítási berendezések, nukleáris erőművek.

Komplexitásuk miatt ezek tradicionális kód alapú fejlesztését egyre inkább felváltja a modellvezérelt megközelítés, amely során magasszintű modellekből kiindulva, azokat tovább finomítva a rendszer a legapróbb részletekig megtervezhető. A metodika előnyei többek között az automatikus kód-, teszteset- és dokumentáció-generálás, valamint, hogy a modellek verifikálásával már a fejlesztés korai szakaszában kiszűrhetők bizonyos hibák.

Ezeken a komplex rendszereken általában egy vagy akár több cég fejlesztő csapatai kollaboratív módon dolgoznak. Így felmerül a modellelemek biztonságának kérdése is, legyen szó olyan bizalmas adatról, létrejövő szellemi tulajdonról, amelyhez csak bizonyos pozíciókban lévő felhasználók férhetnek hozzá, vagy a rendszernek olyan kritikus részéről, amelyet csak megfelelő szaktudással rendelkező fejlesztők módosíthatnak.

A MONDO nemzetközi kutatási projektben készült kollaborációs keretrendszer modellszinten, finomszemcsés szabályok alapján végzi a hozzáférés-vezérlést. Ezekben a szabályokban gráflekérdezésekkel határozható meg, hogy a modellnek milyen típusú vagy pontosan mely elemeire milyen jogok vonatkozzanak különböző felhasználók tekintetében.

Munkám során szöveges szintaxist definiáltam a hozzáférési szabályok meghatározásához, majd implementáltam egy olyan algoritmust, amely képes ilyen szabályok EMF modellek feletti kiértékelésére, vagyis az effektív érvényre jutó hozzáférések kiszámítására. Az algoritmust a már említett MONDO projekt egyik esettanulmányaként használt szélturbina vezérlőről készült modellel teszteltem. Végül az elkészült nyelvtan és algoritmus integrálásra került a kollaborációs keretrendszerbe.

Abstract

Certain software systems have to meet the requirements of not to threaten human lives and not to cause financial or environmental damage. Examples for these so-called safety-critical systems can be controller devices of trains, airplanes or nuclear power stations.

Due to their complexity the traditional code-based development of these systems seems to be replaced by a model-driven approach. With this process starting from high-level models and through their refinement the system can be designed to the smallest details. The advantages of this method are for example the automatic generation of souce code, test cases, documentation and also that with model verification errors can be detected in the earlier phases of the development.

These complex systems are designed collaboratively by developer teams of one or more companies. This situation raises the question of security of model elements. They can be confidential or intellectual property of a company and therefore be accessible only by users in certain positions. Moreover, systems can have critical parts which should be modifiable only by developers with specialized knowledge.

A collaboration framework was built within the confines of the international research project of MONDO. It uses rule-based access control where read and write permissions can be defined in fine-grained rules for each user. The model elements which rules refer can be accessed through graph queries.

In my work firstly I defined a textual concrete syntax for writing access control rules. Then I implemented a method that can evaluate such rules over EMF models and calculate the ones which will be enforced. The algorithm was tested on one of MONDO project's case studies of a windturbine model. Finally, the access control language and the evaluation component were integrated into the collaboration framework.

1. fejezet

Bevezetés

A nagyméretű ipari szoftverek kód alapú fejlesztését egyre inkább felváltja a modellvezérelt megközelítés, melynek során magasszintű modellekből kiindulva, azokat tovább finomítva a rendszer a legapróbb részletekig megtervezhető. A modellekből automatikusan generálhatók például forráskódok, tesztesetek, dokumentációk; folyamatos verifikálásukkal pedig már a fejlesztés korai szakaszában kiszűrhetők bizonyos hibák.

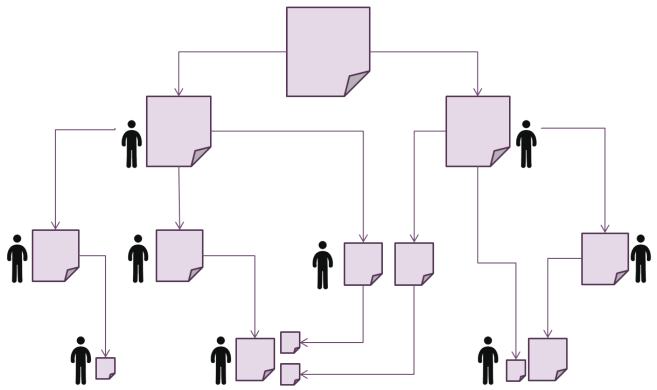
Egy ilyen komplex rendszer fejlesztése több ember együttes munkáját igényli. A tervezés modellalapú megközelítése azért előnyös, mert a magasszintű modellek akár különböző szakterületeken dolgozó fejlesztő csapatok számára is ugyanolyan módon értelmezhetők, ami elősegíti a hatékony, összehangolt munkavégzést.

Ezeket a nagy modelleket általában verziókezelt módon, egy közös tárhelyen (repository-ban) tárolják. A kollaboratív fejlesztés offline vagy online formában valósul meg. Előbbi esetben a felhasználók letöltik a közös tárhelyről a modell egy példányát, amit lokálisan, offline módosítanak. Ezt egy újabb kapcsolat létrehozása után küldik vissza a repository-ba, ami feldolgozás után a változtatásokat a közös modellen is végrehajtja. A többi felhasználó csak akkor értesül ezekről a változásokról, amikor egy újabb letöltéssel frissíti a nála lévő példányt. Így, ha közben ő is dolgozott rajta, akkor az összefésülendő verziók között adódhatnak konfliktusok. Ezzel szemben online kollaboráció során a tárhellyel fenntartott folyamatos kapcsolatból adódóan a közös modellen azonnal végrehajtódnak a felhasználók által eszközölt változások, így azok mindenki számára rögtön láthatóvá válnak.

Offline és online esetben is felmerül a modellelemek hozzáférés-szabályozásnak kérdése. Például amikor egy cég a munka egy bizonyos részét delegálja egy másik cégnek, elérhetővé teszi neki a modellt vagy annak egy részletét, amelyen dolgoznia kell. Ebben azonban lehetnek olyan bizalmas, a cég szellemi tulajdonának számító elemek, amikhez az adott alvállalkozó számára nem akar hozzáférést biztosítani. Az is előfordulhat, hogy egy modell olyan kritikus részekkel rendelkezik, amelyek fejlesztése speciális szaktudást igényel. Ideális esetben ezeket csak a hozzáértő felhasználóknak szabad módosítaniuk.

Modellek feletti hozzáférés-kezelésre létező gyakorlat, hogy a modelleket kisebb fragmensekre darabolják, fájlokba sorosítják, majd ezekhez a fájlokhoz határoznak meg olvasási,

írási jogosultságokat. Ha a rendszer további felhasználókkal bővül, és az újabb szabályok szerint a meglévő fragmensek még kisebb részeihez férhetnek hozzá, akkor azokat még további külön fájlokba kell tördelni. Ezt a jelenséget szemlélteti az 1.1 ábra, melynek hatására a modell elemek ezreire aprózódhat. A fájlszintű szabályozás hátránya, hogy a rendszert nehezen skálázhatóvá, rugalmatlanná teszi.



1.1. ábra. *A fájlszintű hozzáférés-szabályozás problémája*

Erre a problémára a hozzáférések modellszintű szabályozása nyújt megoldást. A MONDO [2] nemzetközi kutatási projektben készült kollaborációs keretrendszer finomszemcsés szabályok alapján végzi a hozzáférés-vezérlést. Ezekben a modell elemi részeire, objektumokra és azok attribútumaira, referenciáira külön-külön lehet hozzáférési jogokat meghatározni a különböző felhasználók tekintetében. A szabályok viszont úgy is megfogalmazhatók, hogy bizonyos tulajdonságú elemek halmazára vonatkozzanak, így egy milliós nagyságrendű modell esetén nem szükséges minden egyes elemre leírni a jogosultságokat. A finomszemcsézettségből fakadóan, mivel a szabályok apróbb részletei is konfigurálhatók, könnyen ellentmondásba kerülhetnek egymással. Leírhatunk például két olyan külön szabályt, amelyek közül az egyik engedélyezi egy elem olvashatóságát, a másik viszont nem. Ezek feloldásához szükséges egy olyan kiértékelő komponens, ami eredményként az effektív, valóban érvényre jutó hozzáférési szabályokat adja.

A fentiek alapján a szakdolgozat kidolgozása során kitűzött célok:

1. Hozzáférési szabályok megadásához alkalmas szöveges konkrét szintaxis tervezése,
    - amelyben a kívánt részletességgel választhatók ki azok a modellelemek, amelyekre vonatkoztatni akarjuk őket,
    - majd megadhatók hozzájuk az igény szerinti olvasási és/vagy írási jogok.
2. Kiértékelő komponens implementálása EMF modellek felett,
    - amely értelmezi a felhasználó által megadott szabályokat,
    - feloldja a közöttük fennálló ellentmondásokat,
    - majd a modell minden elemére eredményül adja a hozzá tartozó, valóban érvényesülő olvasási és írási jogosultságokat.

A szakdolgozat struktúrája

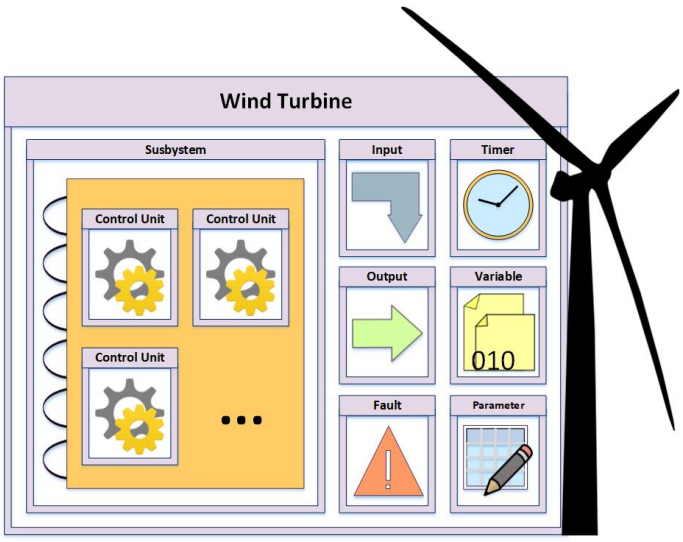
A továbbiakban a szakdolgozat második fejezetében bemutatom a MONDO projekt egy esettanulmányát. A harmadik fejezetben végigveszem a dolgozathoz kapcsolódó háttértechnológiákat és a hozzáférési szabályokhoz tartozó ismereteket. A negyedik fejezet áttekintést ad a megoldás megközelítéséről. Az ötödik fejezetben részletezem a feladatok megvalósítását, a hatodik pedig a megoldásomnak az esettanulmányon végzett mérési eredményeit tartalmazza. A hetedik fejezetben a szakdolgozat témájához kapcsolódó munkákról esik szó. A nyolcadikban pedig a teljesített feladatokat és a hozzájuk köthető további fejlesztési lehetőségeket foglalom össze.

2. fejezet

Esettanulmány

Nagy, összetett ipari rendszerek tervezésében széles körben elterjedt módszer a modellvezérelt szoftverfejlesztés. Az ehhez jelenleg rendelkezésre álló modellezőeszközök gyakran ütköznek skálázhatósági korlátokba. A MONDO EU FP7 kutatási projekt [6] célja ezen kihívások megoldása olyan technológiák, algoritmusok, eszközök kifejlesztésével, amelyek a mostaninál nagyobb hatékonyságot, rugalmasságot biztosítanak a rendszermodellezés terén. A projekt nemzetközi ipari résztvevői közül egy szélturbina-vezérlő egységeket összefogó rendszer esettanulmányát vizsgáltam.

A rendszer leegyszerűsített struktúráját a 3.1 ábra szemlélteti. A szélturbina vezérlőegységei a modellen belül alrendszerekbe csoportosulnak, ezeken kívül pedig még bemenetek, kimenetek, hibadetektorok, időzítők, változók és paraméterek is vannak a rendszerben.



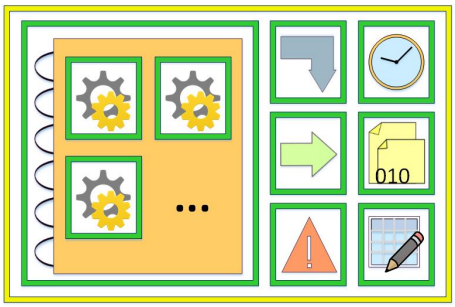
2.1. ábra. *A szélturbina-vezérlő rendszer felépítése*

A modell fejlesztése kollaboratív módon zajlik. Különböző beosztású és szaktudású mérnökök dolgoznak rajta, akik számára a modellnek egy-egy olyan egyedi nézetét kell biztosítani, amelyben csak a számukra szükséges elemek találhatók. Továbbá azt is szabályozni kell, hogy csak azokon a részeken végezhessenek módosításokat, amelyekhez értenek,

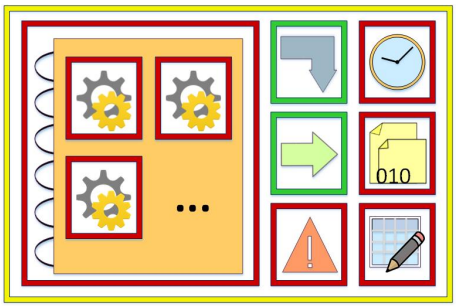
vagy pozíciójukból adódóan joguk van hozzá. Jelen esetben három különböző feladatkörrel rendelkező felhasználótípust definiáltam: Principal Engineer, I/O Manager és Subsystem Manager, akikre az érvényesíteni kívánt hozzáférési szabályok a következők:

- A fő szerkezeti egységet senki nem módosíthatja.
- **Principal Engineer:** adminisztrátor jogokkal rendelkezik. A fő szerkezeti egységen kívül mindent láthat és módosíthat.
- **I/O Manager:** a be- és kimenetek felelőse, ezeket olvashatja és írhatja, de a modell többi része számára rejtett.
- **Subsystem Manager:** A Principal Engineer-hez képest annyival van kevesebb joga, hogy a be- és kimeneteket csak láthatja, de nem módosíthatja.

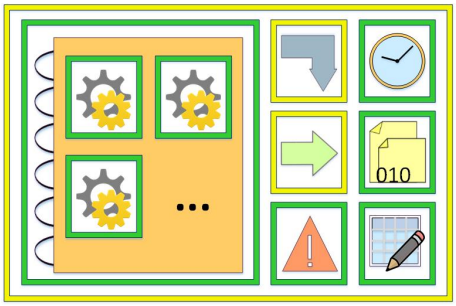
Ezeket a szabályokat szemlélteti a 2.2 ábra, amelyen zöld keret jelöli azokat a modellelemeket, amelyekhez az adott felhasználó olvasási és írási jogosultsággal is rendelkezik, sárga azokat, amelyeket csak láthat, de nem módosíthat, valamit piros azokat, amelyekhez egyáltalán nem férhet hozzá.



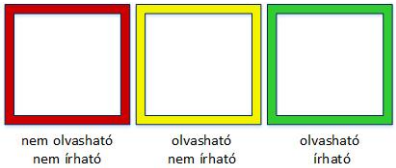
(a) *Principal Engineer*



(b) *I/O Manager*



(c) *Subsystem Manager*



(d) *Jelmagyarázat*

2.2. ábra. *A felhasználókra vonatkozó hozzáférési szabályok*

A feladat első lépésben ezeknek a szabályoknak egy erre a célra kifejlesztett, egyszerű nyelven való megfogalmazása. A közöttük adódó konfliktusok feloldására pedig szükséges egy olyan komponens implementálása, ami képes ezeket szabályokat értelmezni, ha vannak, akkor a közöttük lévő ellentmondásokat feloldani, majd futásának eredményeképp kiválogatni közülük a ténylegesen érvényre jutó, effektív jogosultságokat.

3. fejezet

Háttértechnológiák, ismeretek

3.1. Modellezés

3.1.1. Modellvezérelt szoftverfejlesztés

Az MDE (Model-Driven Engineering) [1] egy olyan szoftvertervezési módszer, amelynek célja a rendszer magasszintű modellekkel való leírása és az automatizálás növelése a fejlesztés során. A magas absztrakciós szint a nagyméretű, akár több millió kódsort tartalmazó, összetett szoftvereket átláthatóvá, könnyen kezelhetővé teszi, ugyanakkor a modellt tovább finomítva a rendszer a legapróbb részletekig megtervezhető. A modellből többek között futtatható forráskód, tesztesetek vagy dokumentáció is automatikusan generálható. Ez és a magasszintű modellek tervezés közbeni verifikációja csökkenti az implementálás során előfordulható emberi hibák mértékét, ami különösen előnyös például biztonságkritikus rendszerek fejlesztése esetén.

A magas absztrakciós szintű modellek előnye, hogy különböző iparágak - informatikához és programozáshoz nem feltétlenül értő - szakértői számára is érthetővé tehetők. Azt a folyamatot, amikor a modellt egy adott szakterület (domain) szerint tervezünk, szakterület-specifikus modellezésnek nevezzük. Ilyen domain-specifikus modell az esettanulmányban említett szélturbina-vezérlők modellje is, amit a hozzá definiált felhasználók ugyanolyan módon tudnak értelmezni.

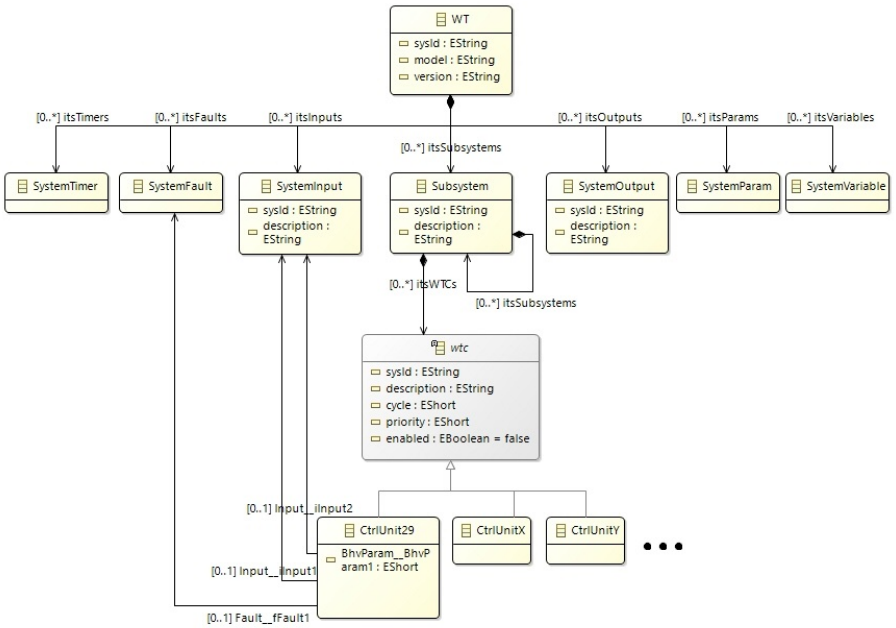
3.1.2. Eclipse Modeling Framework

Az Eclipse Modeling Framework (EMF) [8] egy modellező és kódgeneráló keretrendszer domain-specifikus modellek fejlesztéséhez. Megkülönbözteti a metamodellt a tényleges modelltől, előbbi a modell struktúráját írja le, utóbbi a metamodell egy konkrét példánya.

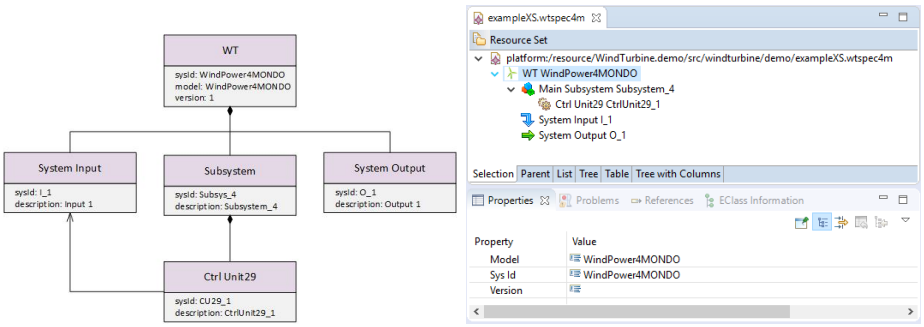
Az úgynevezett ecore fájlban tárolt metamodell egy UML osztálydiagramhoz hasonló módon épül fel. Ennek elemeit az EMF Ecore nevű (meta)metamodellje biztosítja. Az általunk elkészíthető metamodell osztályokat reprezentáló EClass-okat tartalmaz, amelyek

tulajdonságait EAttribute-ok írják le, a közöttük lévő kapcsolatokat pedig EReference-ek jelzik. Ezek a referenciák lehetnek egyszerű asszociációk vagy tartalmazások is.

Az esettanulmány szélturbina modelljét a 3.1 ábra mutatja. A modell gyökéreleme maga a szélturbina (WT), ami további egymásba ágyazható alrendszerekben (Subsystem) tárolja a vezérlőegységek (CtrlUnit) bővíthető halmazát az őket azonosító, leíró attribútumokkal, valamint a modell többi elemére hivatkozó referenciákkal. Ezek az egységek a megegyező attribútumaikat tároló közös ősből származnak le (wtc). A gyökérelem az alrendszereken kívül tartalmaz még bemenetet, kimenetet, időzítőt, hibadetektort, paramétert és változót (Input, Output, Timer, Fault, Param, Variable).



3.1. ábra. *Szélturbina-vezérlők metamodellje*



3.2. ábra. *Szélturbina-vezérlők példánymodellje*

Az ecore metamodellből újrafelhasználható Java kódok generálhatók, többek között egy olyan fa struktúrájú szerkesztő plugin, amellyel a modell különböző konkrét példányait lehet létrehozni. A szélturbina-vezérlők metamodelljének egy egyszerű példányát mutatja a 3.2 ábra, melynek jobb oldalán látható az EMF alapértelmezett tree editorja. A gyökér-

elem egy vezérlőegységet tartalmaz egy alrendszer alá rendezve, valamint egy-egy bemenetet és kimenetet, amelyek közül a vezérlő az előbbire tartalmaz referenciát.

3.1.3. Modellezési nyelvek

A modellek leírásához szakterület-specifikus modellezési nyelveket használunk. Ezeknek részei az absztrakt szintaxis és a konkrét szintaxisok. Előbbi azt határozza meg, hogy a nyelvnek milyen típusú elemei vannak és ezek milyen kapcsolatban állnak egymással, vagyis ez maga a metamodell. Ehhez több konkrét szintaxis is megadható (ilyen például az EMF által generált tree editor is), amik szöveges vagy grafikus megjelenítést biztosítanak a példánymodellhez. Ezektől válik az adott szakterület hozzáértői számára olvashatóvá és szerkeszthetővé.

3.1.4. Xtext

Az Xtext keretrendszer [10] segítségével szöveges konkrét szintaxis készíthető. Ehhez egy nyelvtant kell megadni, az ehhez tartozó absztrakt szintaxis egy EMF modell lesz, ezt a metamodellt le tudja generálni a keretrendszer, és hozzá egy egyszerű editort is nyújt, amiben az adott nyelvtannak megfelelően lehet fejleszteni. Ez a folyamat fordított irányban is történhet, már létező ecore modellhez is generálható nyelvtan. Egyéb szolgáltatásai például a forráskódszínezés (syntax highlighting), hibajelzés (validation) és automatikus kiegészítés (content assist), ezek Java-ban való tovább finomítására is van lehetőség.

3.1.5. VIATRA Query

A VIATRA Query [9] egy deklaratív lekérdezési nyelvvel rendelkező modelltranszformációs eszköz. A nyelv segítségével a lekérdezéshez gráfmintákat fogalmazhatunk meg a metamodell osztályaival, attribútumaival, referenciáival, a rendszer pedig azokat a modellelemeket adja vissza, amelyek illeszkednek a megadott mintára. A 3.1 egyszerű gráfminta például a szélturbina-modell összes SystemInput típusú objektumát adja vissza.

```
1 pattern objectInput(input : SystemInput) {
2     SystemInput(input);
3 }
```

3.1. Forráskód. *Egy egyszerű gráfminta*

3.1.6. Belső konzisztencia

Offline kollaboráció során amikor a felhasználó lekéri a modellt a szerverről, akkor annak egy olyan reprezentációját tölti le, amiben a hozzáférési szabályok által engedélyezett modellelemek találhatók. Ahhoz, hogy ezzel a részmodellel lokálisan dolgozni tudjon (például végigiteráljon rajta vagy sorosítsa), ennek egy teljes, érvényes modellnek kell lennie. Ez úgy valósulhat meg, ha a központi modellhez hasonlóan megfelel bizonyos kényszereknek, amik fenntartják a modell belső konzisztenciáját:

K1 Objektum létezése:
Attribútumok és referenciák létezése azt feltételezi, hogy van mögöttük egy megfelelő típusú objektum, ami tartalmazza őket.

K2 Tartalmazási hierarchia:
Ha egy objektum nem gyökérelem, akkor tartalmazottnak kell lennie egy gyökérelem által akár közvetlenül, akár közvetetten objektumok láncolatán keresztül.

K3 Ellentétes referenciák:
A kétirányú referenciatípusok párban példányosíthatók.

K4 Számossági kényszerek:
Egy objektum adott típusú referenciáinak száma meg kell hogy feleljen a típushoz definiált számosságnak.

Mivel a modell lekérésekor történő modelltranszformáció során kerül sor a hozzáférés-szabályozásra is, ezért a belső konzisztencia megtartása azon is múlik, hogy milyen hozzáférési szabályok érvényesülnek a modellen.

3.1.7. Asset

Modellszintű hozzáférési szabályokkal a modell logikai egységeire határozhatunk meg olvasási és írási jogosultságokat. Ezeket a modellelemeket [3] alapján asseteknek nevezzük, és EMF modellek esetén a következőképpen csoportosítjuk őket:

- **Objektum asset:** egy modellelem és az ő típusa, vagyis egy EObject és egy EClass párosa határozza meg. Az esettanulmányban pl. obj(I_1, System Input).
- **Attribútum asset:** az őt tartalmazó EObject illetve az attribútum nevének (EAttribute) és értékének hármasa alkotja, pl. attr(I_1, description, input 1).
- **Referencia asset:** a kezdő- és végpontjában lévő modellelemek, valamint a referencia típusa, vagyis két EObject és egy EReference hármasa azonosítja mind az asszociációt és a tartalmazást is, pl. ref(CU29_1, Input_iInput1, I_1).

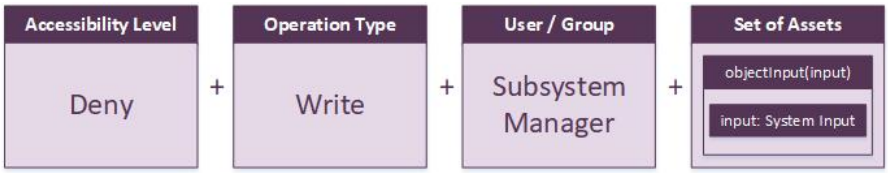
3.1.8. Modell obfuszkáció

Az obfuszkáció lényege, hogy egy kívülállók elől elrejteni kívánt információt eltorzítunk olyan módon, hogy az minél nehezebben értelmezhető legyen, ugyanakkor visszafejtés után az eredeti adatot kapjuk vissza. Vagyis például két megegyező információ obfuszkált értéke teljesen eltér egymástól, de egyedi azonosítójukat megőrizve visszafejtés után is azonosak lesznek. Modellek hozzáférés-kezelésénél akkor használunk obfuszkációt, ha egy assetnek csak a létezésére akarunk utalni anélkül, hogy annak tulajdonságait láthatóvá tennénk. Ha objektumról van szó, akkor az azonosító attribútumait is obfuszkáljuk, a többi jellemzője pedig rejtve marad.

3.2. Hozzáférés-szabályozás

3.2.1. Hozzáférési szabály

Egy hozzáférési szabálynak [3] alapján négy alapvető eleme van. Az egyik a hozzáférés szintje, vagyis hogy engedélyezünk, tiltunk, vagy obfuszkálunk. A második az operáció típusa, ami olvasás vagy írás lehet. Azt is meg kell határozni, hogy melyik felhasználóra vagy azoknak mely csoportjára vonatkozik a szabály, valamint ki kell választanunk asseteknek azt a halmazát, amelyekhez szabályozzuk a hozzáférést. A 3.3 ábra azt a szabályt szemlélteti, amelyben a Subsystem Manager felhasználó számára tiltjuk a System Input típusú modellelemek írását.



3.3. ábra. *Hozzáférési szabály felépítése*

Az, hogy egy szabály egyszerre több assetre is értelmezhető, gráflekérdezések segítségével érhető el. Különböző gráfmintákban fogalmazzuk meg, hogy pontosan milyen tulajdonságú elemekre van szükségünk, a rendszer pedig visszaadja a mintára illeszkedő találatokat. A szabályban egy ilyen gráfmintára hivatkozunk, és akár még több szűrő feltétel megadásával kiválasztjuk a találatok közül azokat az asseteket, amelyekre szabályozni akarjuk a hozzáférést. A fenti szabályban az objectInput(input) mintára hivatkozunk, amely a modell összes System Input típusú objektumát adja vissza.

Gráflekérdezéssel ugyan nem kell egyesével definiálni a hozzáférést minden egyes assetre, viszont egy kellően nagy modellnél még assetek csoportjaira is túl sok szabályt kellene megfogalmazni. Ezért is jó megoldás, ha a szabályokat egy nagyobb egységbe, úgynevezett policyba zárjuk, amire megszabhatunk default jogosultságokat. Ezeket használhatjuk, ha valamelyik modellelemre nem találunk megadott hozzáférést.

3.2.2. Szabályok közötti konfliktusok

A hozzáférési szabályok finomszemcsézettsége miatt előfordulhatnak közöttük az ésszerűség ellen szóló vagy a modell belső konzisztenciáját megbontó konfliktusok. Ezeket [4] szerint három típusba soroljuk az alapján, hogy melyik assetre és operációra vonatkoznak.

- **I. típusú konfliktus:** Ugyanarra az assetre éa ugyanarra az operációra vonatkozó szabályok ellentmondása, például egy adott assetre az egyik szabály engedélyezi az olvasást, a másik pedig tiltja azt.
- **II. típusú konfliktus:** Ugyanarra az assetre de különböző operációkra vonatkozó szabályok esetén abban az esetben fordul elő, amikor az egyik szabály engedélyezi az

asset módosítását, a másik viszont tiltja a láthatóságát. Az ésszerűség úgy diktálná, hogy ha egy asset írható, akkor legyen olvasható is, vagy ha nem olvasható, akkor írni se lehessen.

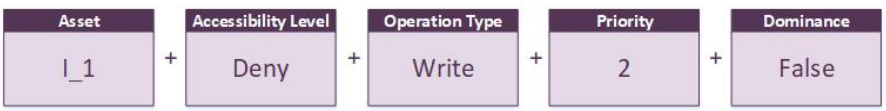
- **III. típusú konfliktus:** Különböző assetekre és különböző operációkra vonatkozó szabályok a modell belső konzisztencia kényszerei miatt kerülhetnek konfliktusba egymással. Ha például egy attribútum olvasható, de az őt tartalmazó objektum nem, az a 3.1.6 részben említett K1 konzisztencia szabálynak mond ellent, hiszen a felhasználó, akire a hozzáférés vonatkozik, csak az attribútumot fogja látni a modellben, mintha nem létezne mögötte objektum. Az ilyen és ehhez hasonló konfliktusok feloldására az olvasási és írási függőségek bevezetése a megoldás. A fenti esetben, ha az attribútumra vonatkozó hozzáférés teljesül, akkor az őt tartalmazó objektumra is ki kell kényszeríteni az olvashatóságot.

3.2.3. Effektív jogosultságok

A megadott hozzáférések tehát csak nominális szabályok, a valóban érvényre jutó, effektív jogosultságok ezektől különbözőek lehetnek. A kiértékelés során a modell belső konzisztenciájának megtartása érdekében fel kell oldani a felmerülő konfliktusokat. Ehhez a feloldáshoz a szabályokból az általa meghatározott összes assetre először úgynevezett judgementeket származtatunk, amelyek a 3.4 ábrán látható módon épülnek fel.

A felhasználó és a default beállítás által közvetetten megadott judgementekből a 3.2.2 részben említett olvasási és írási függőségek alapján újabb judgementeket határozunk meg. Így kapunk egy olyan jogosultsághalmazt, amiben a modell minden egyes assetjére megtalálhatók az érvényesíteni kívánt jogok. Ekkor jutunk el oda, hogy már csak azt kell vizsgálnunk, hogy egy adott asset esetén a különböző operációkra milyen engedélyt adunk. Vagyis ezzel a módszerrel minden konfliktust I. típusúra egyszerűsítünk, ami már könnyen feloldható.

Ehhez az I. típusú konfliktusfeloldáshoz a judgementek az eddig említett asseten, hozzáférési szinten és operációtípuson kívül prioritást és dominanciát jelző flaget is tartalmaznak. Az utóbbi két jellemző alapján döntjük el, hogy melyik judgement az erősebb, és ezáltal melyik fog érvényre jutni a kettő közül. Az elsődleges szempont a prioritás, ezt a felhasználó hozzáférési szabályonként állíthatja be. Ha egy prioritási osztályban vannak, akkor a dominancia dönthet közöttük. A szabályokat összefogó policyhoz lehet engedélyező vagy tiltó tulajdonságot beállítani, amik közül az előbbi az engedélyező szabályokat teszi dominánssá a tiltókkal szemben, az utóbbi pedig fordítva.



3.4. ábra. *Judgement felépítése*

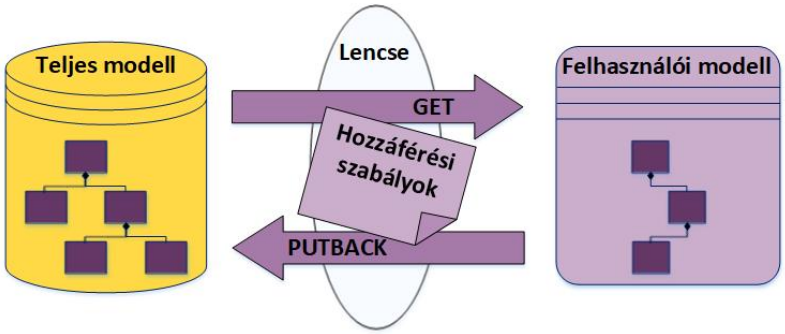
4. fejezet

Áttekintés

4.1. Kétirányú modelltranszformáció

Modellek kollaboratív fejlesztésekor a teljes modell egy közös tárhelyen érhető el, a munkában résztvevők ezen végeznek különböző módosításokat. Hozzáférés-szabályozással nem csak azt lehet megszabni, hogy ki milyen változtatásokat hajthat végre a modellen, hanem azt is, hogy egyáltalán mely modellrészekről lehet tudomása. Olvasási hozzáférési szabályok alapján minden felhasználó számára a teljes modellnek egy olyan - szintén teljes és konzisztens modellnek tűnő - nézetét kell biztosítani, amiben csak olyan elemek szerepelnek, amelyeknek a láthatósága engedélyezett.

Az egyedi nézetek modelltranszformáció segítségével alakulnak ki. Mivel a többi fejlesztő úgy értesülhet az ezen végzett változásokról, ha azok a teljes modellen is megjelennek, ezért a műveletet visszafele is végre kell hajtani. Ez a kétirányú modelltranszformációs típus az úgynevezett lencse [5], amelynek a működését a 4.1 ábra szemlélteti.

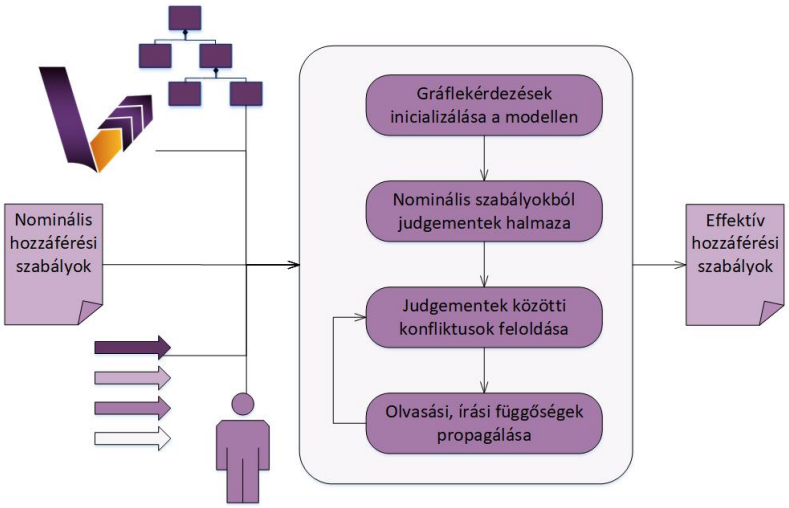


4.1. ábra. *Kétirányú modelltranszformáció*

A kétféle transzformáció a GET és a PUTBACK. Az előbbi az adott felhasználóra definiált olvashatósági szabályoknak megfelelően alakítja a modellt. Az utóbbi pedig visszatöltési szándék esetén először ellenőrzi, hogy az adott modellelem módosítására van-e engedély a szabályok gyűjteményében. Ha igen, végrehajtja a transzformációt, és ezzel a többiek számára is elérhető változtatásokat a teljes modellben.

4.2. Hozzáférési szabályok kiértékelése

A modelltranszformációt végző lencse a már végleges, a modell belső konzisztenciájának megtartását elősegítő hozzáférési szabályok alapján működik. A kívülről megadott, nominális szabályokból a 4.2 ábrán látható mechanizmus kalkulálja ki ezeket az effektív szabályokat.



4.2. ábra. *Effektív hozzáférési szabályok számítása*

A megvalósított algoritmus egy nulladik lépésben a VIATRA lekérdezőnyelvén megfogalmazott gráfmintákat illeszti a szintén bemenetként kapott modellre. Így később a találatokból már csak ki kell választani a szükséges elemeket.

A kívülről megadható hozzáférési szabályokat az általam definiált Xtext nyelvtannal lehet leírni. Megfogalmazható, hogy melyik felhasználónak, pontosan mely assetekre engedélyezzük vagy tiltjuk az olvasás/írás műveletét. A szabályokhoz prioritás is rendelhető, az őket összefogó policyhoz pedig olyan tulajdonság, ami meghatározza, hogy az engedélyező vagy a tiltó szabályok közül melyek az erősebbek. Ezekkel egyfajta fontossági sorrend állítható fel a szabályok között. Azokra az assetekre, amelyeket nem akarunk külön szabályba foglalni, megadható egy globális default hozzáférés.

Ezeket a nominális szabályokat judgementekre bontva - vagyis minden asset minden műveletére megfogalmazva - az algoritmus összeállítja a hozzáférések kezdeti halmazát. Ezek közül pedig ciklikusan kiválasztja a legerősebbet, aminek mindenképp érvényesülnie kell, majd a halmaz maradék elemei közül kiszórja azokat, amelyek ezzel ellentmondásban állnak. A folyamat bemeneteként megadható írási és olvasási függőségi szabályokat figyelembe véve az algoritmus felveszi a listába az érvényesülő judgement által eredményezett újabb judgementeket. A legdominánsabb szabály kiválasztása, a konfliktusfeloldás és függőség-propagálás ciklusa addig tart, amíg a listában lévő összes judgement feldolgozásra nem kerül. Az eredmény, a folyamat kimenete pedig az adott felhasználóra vonatkozó, ténylegesen érvényre jutó, effektív hozzáférési szabályok, amelyeket alkalmazva a modell teljes és konzisztens marad.

5. fejezet

Megvalósítás

A szakdolgozat kidolgozása során kitűzött célok egyike egy olyan szöveges konkrét szintaxis készítése, amely lehetővé teszi EMF modellek feletti hozzáférési szabályok definiálását. A másik pedig egy olyan komponens megvalósítása, ami ilyen módon megadott szabályhalmazt alapul véve, illetve a modell belső konzisztenciájának megtartása érdekében a közöttük lévő konfliktusok feloldásával képes meghatározni az effektív hozzáférési jogosultságokat a modell minden egyes elemére.

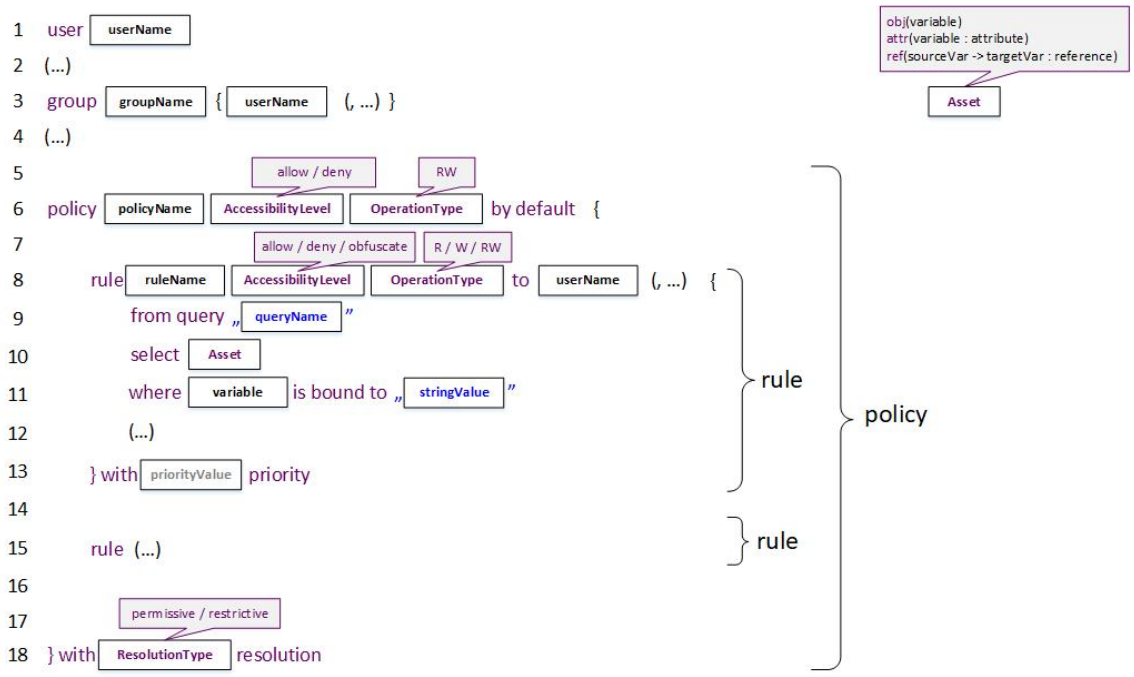
5.1. Hozzáférés-szabályozási nyelv

5.1.1. Nyelvtan

Az első feladat megoldását, vagyis a szöveges konkrét szintaxis készítését egy Xtext nyelvtan definiálásával kezdtem. A keretrendszer generálta le a hozzá tartozó metamodellt és szöveges editort. Az 5.1 ábra azt mutatja, hogy ebben a szerkesztőben milyen módon lehet a specifikált nyelvtan alapján hozzáférési szabályokat megadni. A forráskódszínezés beállítása szerint a rögzített kulcsszavak és az enumerációkból választható értékek lila színnel jelennek meg, az egyedi azonosításra szolgáló nevek, számértékek és a gráflekérdezések paraméterei feketével, az idézőjelek között megadott stringek pedig kékkel. Az ábra továbbá bekeretezve tartalmazza a felhasználó által megválasztható paramétereket.

A rendszer felhasználóit illetve a belőlük összeállítható csoportokat a szabályok felírása előtt kell megadni. A későbbiekben mindegyik az alkalmas kulcsszó (user/group) után írt egyedi névvel azonosítható. A csoport elnevezése után kapcsos zárójelek között adandó meg az azt alkotó felhasználó(k) listája.

A szabályokat egyetlen policy foglalja magába, amelyhez a policy kulcsszó után a nevét, majd a modell összes elemére vonatkozó default jogosultságot lehet megadni. A második egy hozzáférési szintből és egy operációtípusból áll, ezek az AccessibilityLevel és OperationType enumerációkból választhatók ki. Előbbi engedélyezést (allow), tiltást (deny) vagy obfuszkálást (obfuscate) tesz lehetővé, utóbbi pedig alapértelmezetten vonatkozhat olvasásra (R), írásra (W) vagy mindkettőre (RW). (Itt a default jogosultságok megadásánál



5.1. ábra. *Szöveges konkrét szintaxis*

csak a RW engedélyezett, mert mindegyik operációtípusra meg kell határozni.) A policyt bezáró zárójel után a ResolutionType enumeráció kétféle eleme közül kell kiválasztani, hogy melyik határozza meg az egyenként megadott szabályok dominánsságát. A permissive tulajdonságot választva az engedélyező szabályok lesznek erősebbek, restrictive esetén pedig a tiltók.

A policyn belül tetszőleges számú hozzáférési szabály leírható, ezek is a rule kulcsszó után írt nevükkel különböztethetők meg. Ugyanebbe a sorba írjuk az adni kívánt jogosultság hozzáférési szintjét és operációtípusát, valamint a to kulcsszó után a felhasználó(ka)t, aki(k)re a szabály vonatkozik. A szabály törzsében fogalmazzuk meg, hogy pontosan mely modellelemekhez adja azt a bizonyos jogosultságot. Ehhez első lépésben a from query után meg kell adni string formájában annak a gráfmintának a nevét, amelynek az illeszkedési eredményéből szeretnénk kiválasztani a megfelelő asseteket. Ez a kiválasztás a select kulcsszóval történik, ezután a különböző típusú assetek a következőképpen adhatók meg:

- **Objektum asset: obj(variable)**
  A kívánt objektumtípus kiválasztásához a zárójelek közé kell írni a gráfminta paraméter változói közül a megfelelőt.
- **Attribútum asset: attr(variable : attribute)**
  Az assetet a gráflekérdezés eredményeként kapott találat megfelelő objektum paramétere és egy alkalmas attribútuma határozza meg.
- **Referencia asset: ref (sourceVar -> targetVar : reference)**
  A referencia két végpontja, valamint maga a referencia neve definiálja.

A szabály maradék soraiban a select-tel kiválasztott assetek listája szűrhető tovább úgy, hogy a gráfminta megfelelő típusú paraméter változóinak string értékét kötjük meg. Ez is erősíti a szabályok finomszemcsézettségét, hiszen így a legapróbb részletekig tudjuk specifikálni, hogy mely modellelemekre akarjuk meghatározni a hozzáférési jogosultságot.

5.1.2. Extra funkciók

A szintaxis olyan plusz funkciókkal rendelkezik, amelyek kényelmesebbé teszik a fejlesztést a szabályokat megadó felhasználók számára. Az egyik ilyen az automatikus formázás, a megfelelő billentyűkombinációt lenyomva a szöveg az 5.1 kódon látható módon tagolódik.

Egy másik ilyen funkció az úgynevezett scope provider, ami szintén a megfelelő gyorsbillentyűre reagálva előhozza az adott helyre írható elemek alternatíváit. A szabályok fejlécének végén a lehetséges felhasználókat, csoportokat, a szabály törzsében a from query kulcsszavak után a létező gráfmintákat ajánlja fel. Ha megadtuk ezt a gráfmintát, akkor a következő sorokban, az asset kiválasztásánál és a szűrőfeltétel fogalmazásánál már annak a paramétereiből lehet választani. Ha attribútum vagy referencia assetről van szó, akkor pedig a leírt paraméter típusának megfelelő attribútumokat/referenciákat képes megkeresni.

A szabályok logikai helyességének kikényszerítésében pedig egy úgynevezett validator komponens segít. Mivel az obfuszkációt csak az olvasás műveletére értelmezzük, méghozzá csak objektumok és attribútumok esetén, ezért a validáció egyrészt ellenőrzi, hogy szerepele operációtípus a leírt obfuscate kulcsszó után, valamint hogy assetnek referenciát adtunk-e meg. Ezen kívül azt is validálja, hogy a policy fejlécében meg lett-e adva mindkét operációtípus (RW) a default jogosultságok meghatározásához. Ezekben az esetekben a problémát világosan kifejező hibaüzenettel figyelmezteti a felhasználót.

5.1.3. Esettanulmány

Az esettanulmányban felvetett hozzáférés-jogosultsági igények kielégítésére az 5.1 kódon látható szabályokat határoztam meg a fent bemutatott nyelven. Az általuk használt gráfmintákat az 5.2 mutatja. A háromféle munkakör kifejezésére a PrincipalEngineer, IOManager és SubsystemManager felhasználókat vettem fel a rendszerbe. Mivel közülük a PrincipalEnginner-nek és a SubsystemManager-nek is több objektum elérhetőségét engedélyezzük mint tiltjuk, ezért default jogosultságnak azt adtam meg, hogy minden asset írható és olvasható. Így a tiltások kevesebb szabályban is összefoglalhatók. Az esettanulmányban szereplő követelmények alapján a következő szabályokat határoztam meg:

- restrictRoot: Az "objectRoot" lekérdezés a modell gyökérelemét adja vissza, erre vonjuk meg az írási jogot az összes felhasználótól.
- restrictNotIO: Az "objectNotIO" a modell összes olyan objektumát visszaadja, ami nem SystemInput vagy SystemOutput típusú. Ezeknek a láthatóságát tiltjuk meg az IOManager felhasználónak.

- restrictIO: A Subsystem Manager számára tiltja a SystemInput és SystemOutput típusú objektumok módosítását.

A szabályokhoz prioritást is rendeltem, a policyhoz pedig restrictive tulajdonság tartozik, vagyis a tiltó szabályok számítanak erősebbnek. Ezeknek a szabályokat kiértékelő algoritmus működésében lesz jelentős szerepük.

```
user PrincipalEngineer
user IOManager
user SubsystemManager

policy DemoPolicy allow RW by default {

  rule restrictRoot deny W to PrincipalEngineer , IOManager , SubsystemManager {
    from query "objectRoot"
    select obj(root)
  } with 1 priority

  rule restrictNotIO deny R to IOManager {
    from query "objectNotIO"
    select obj(object)
  } with 1 priority

  rule restrictIO deny W to SubsystemManager {
    from query "objectIO"
    select obj(object)
  } with 2 priority

} with restrictive resolution
```

5.1. Forráskód. *Az esettanulmányhoz definiált hozzáférési szabályok*

```
pattern objectRoot(root : WT) {
 WT(root);
}

pattern objectIO(object) {
 SystemInput(object);
} or {
 SystemOutput(object);
}

pattern objectNotIO(object) {
 find objectAllObjectsWithoutRoot(object);
 neg find objectIO(object);
}
```

5.2. Forráskód. *Az esettanulmányhoz definiált gráflekérdezések*

5.2. Szabályokat kiértékelő komponens

A 3.2.2 alfejezetben említett különböző assetekre és különböző operációkra vonatkozó szabályok közötti III. típusú konfliktusok feloldására az olvasási és írási függőségek bevezetése nyújt megoldást.

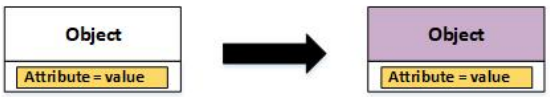
5.2.1. Alapvető függőségek

A [4] cikk alapján a megvalósításomban az alábbi konfliktusok feloldására vezettem be olvasási (R) és írási (W) függőségeket. Ezeket egy-egy ábra is szemlélteti a leírások után, amelyekhez az 5.2 ábrán lévő színkódolás tartozik. Sárga jelöli az olvasható modellelemeket, piros a nem olvashatókat, zöld az írhatókat, kék a nem írhatókat és lila az obfuszkáltakat.

5.2. ábra. *Jelmagyarázat*

F1 Objektum olvasható →← Attribútum nem olvasható

A K1 konzisztencia szabály alapján az attribútumoknak egy-egy létező objektumhoz kell tartozniuk, vagyis ha egy attribútum olvasható, akkor a hozzá tartozó objektumnak is meg kell jelennie (legalább obfuszkáltan) a modellnézetben.

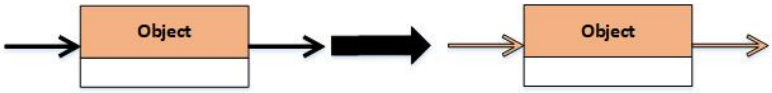


F2 Referencia olvasható →← Kezdő-/végpontbeli objektum nem olvasható

a Az F1 függőséghez hasonlóan a K1 konzisztencia szabály kielégítésére ha egy referencia látható, akkor mivel tartoznia kell két objektumhoz, a kezdő- és végpontjában lévő objektumoknak is láthatónak kell lenniük legalább obfuszkálva.

b Ezt kiegészítendő, ha a forrás vagy a cél objektum nem olvasható, akkor a közöttük lévő referencia se legyen az.

(a)

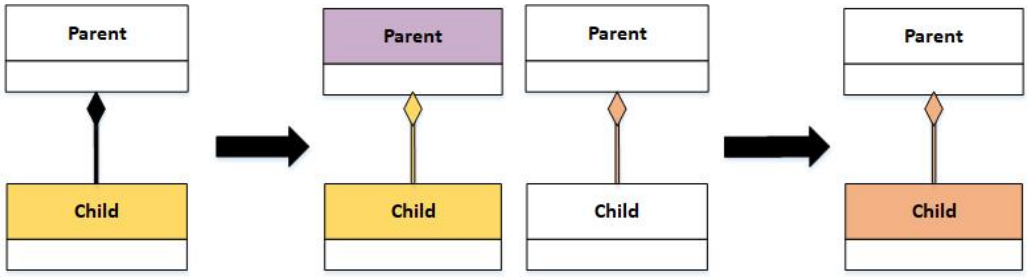


(b)

F3 Objektum olvasható →← Szülő objektum nem olvasható

a A K2 konzisztencia szabály értelmében ha egy objektum nem gyökérelem, akkor egy másik által tartalmazottnak kell lennie. Azért, hogy a megjelenített modellben létezzen szülője, az őt tartalmazó objektumot - ha más szabály esetleg nem engedi a láthatóságát - legalább obfuszkálni kell, a közöttük lévő referenciát pedig olvashatóvá kell tenni. Többszörös propagálás után ennek eredményeképp az egész tartalmazási hierarchia megjelenik egészen a gyökérelemig.

b Ezt a konfliktust a másik oldalról feloldva, ha egy tartalmazási referencia nem látható, akkor a tartalmazott objektum se legyen az.



(a) **(b)**

F4 Objektum olvasható →← ID attribútum nem olvasható

a Mivel objektumok között az egyedi azonosítójukkal tehetünk különbséget, így ha egy objektum látható, akkor az őt azonosító ID attribútum(ok)nak is olvashatónak kell lennie.

b Hasonlóan a másik irányba, ha valamelyik ID attribútum rejtve van, akkor maga az objektum se látszódhat.

(a)

(b)

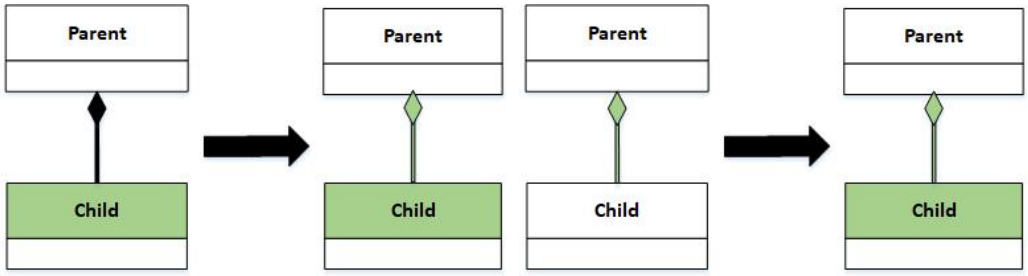
F5 Obfuszkáció

Ha egy objektum vagy attribútum elemre nincs meghatározva olvashatósági engedély, csak azért szerepelnek a modellben, mert az olvasási függőségek kikényszerítik azt, akkor obfuszkáltan jelennek meg. Egy ilyen objektumnak az azonosító attribútumait is obfuszkálni kell, a többi attribútum pedig rejtve marad a modellben.

F6 Objektum írható →← Tartalmazási referencia nem írható

a Egy objektum eltávolításához nem elég, ha ő maga írható, az is szükséges, hogy a rá mutató tartalmazási referencia is az legyen, mert ebben az esetben azt is törölni kell a modellből, hiszen önmagában már nem értelmezhető.

b Ezzel összefüggésben, ha egy tartalmazási referencia módosítható, akkor a gyerek objektumok abban az esetben törölhetők vagy áthelyezhetők, ha ők is módosíthatók. Azért, hogy ez mindig teljesüljön, az algoritmus ilyen esetben is propagálja az írhatóságot.

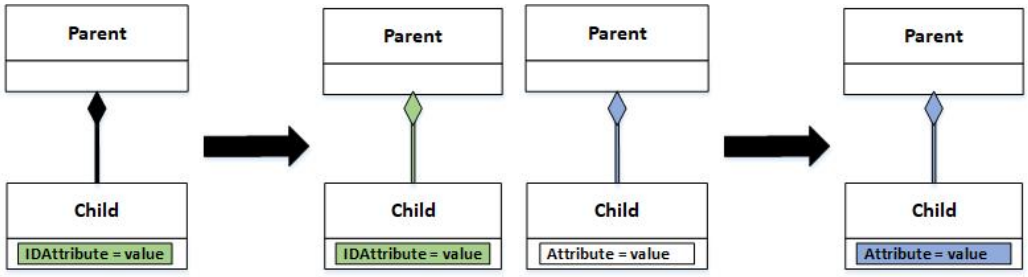


(a) **(b)**

F7 ID attribútum írható →← Tartalmazási referencia nem írható

a Az előző konfliktushelyzetet továbbgondolva akkor is törlésre kerül egy objektum, ha valamelyik ID attribútuma módosul. Ilyenkor eltávolítás után egy új objektum kerül a helyére, ami már az új beállított értékkel rendelkezik. Tehát ilyen attribútumok írhatósága esetén erre a helyzetre felkészülve az objektum tartalmazási referenciájának is írhatónak kell lennie.

a Fordított helyzetben ha a referencia nem módosítható, akkor nem csak az objektum, hanem az azt azonosító attribútumok sem módosíthatók.



(a) **(b)**

Az írási és olvasási függőségek bevezetésével nem csak a III-as, hanem a II-es típusba tartozó konfliktusok is feloldhatók. Ezek az ugyanarra az assetre és különböző operációkra vonatkozó szabályok között két helyzetben állnak fent, ezeket oldják fel az alábbi függőségek:

F8 Asset írható $\rightarrow\leftarrow$ Asset nem olvasható

a Egy bármilyen asset írhatóságának akkor van értelme, ha olvasható is, hiszen hiába van joga a felhasználónak azt módosítani, ha a számára megjelenő modellnézetben nincs benne az adott asset. Tehát az írhatósággal minden asset esetében együtt kell hogy járjon az olvashatóság is.

b Fordított irányban, ha egy asset nem látható, akkor értelemszerűen módosítható sem lehet.

F9 Asset obfuszkált $\rightarrow\leftarrow$ Asset írható

Egy asset obfuszkáltsága esetén kérdéses, hogy változtatás után mi történik vele. Ekkor a felhasználók csak obfuszkált értékeket változtathatnak, aminek következtében a visszaállítás után nem feltétlenül a kívánt eredmény érhető el. Az én megoldásomban minden obfuszkált assetre tiltva van az írhatóság.

5.2.2. Olvasási és írási függőségek konfigurációja

Az implementációban egy *IConsequence* interfészt a *propagate*() metódusával megvalósítva a felhasználó hozzáadhatja az általa definiálni kívánt függőségeket a komponenshez. Így a modell értelmességét és belső konzisztenciáját támogató függőségeken kívül akár még több megkötéssel is konfigurálhatja az effektív hozzáféréseket számító algoritmust.

5.2.3. Algoritmus működése

Az effektív hozzáférési szabályokat kiválasztó függvény a [4] cikkben tárgyalt megoldásból kiindulva az alábbi algoritmus szerint működik.

Algoritmus 1 Effektív jogosultságok számolása

```
function GetEffectivePermissions(model, strongCons, weakCons, explRules, user)
    permissionList ← getInitialPermissions(model, explRules, user)
    processed ← ∅
    while permissionList is not empty do
        j ← chooseDominant(permissionList)
        processed ← processed ∪ {j}
        permissionList ← permissionList \ {j}
        for all j' ∈ permissionList do
            if conflict(j, j') then ▷ I. típusú konfliktus esetén igazzal tér vissza
                permissionList ← permissionList \ {j'}
            end if
        end for
        for all strongCons do
            consequences ← propagate(j)
            permissionList ← permissionList ∪ consequences
        end for
        for all weakCons do
            consequences ← propagate(j)
            permissionList ← permissionList ∪ consequences
        end for
    end while
end function
```

1. **Kiindulási judgementek listája:**

   (a) **Explicit szabályokból:** A kívülről megadott explicit hozzáférési szabályok *permissionList*-hez való hozzáadásához végigiterál a kérdéses felhasználóra vonatkozó szabályokon. Minden lépésben létrehozza a gráflekérdezésből megkapott, majd a szabályban megfogalmazott feltételekkel szűkített assethalmaz elemeit tartalmazó judgementeket. Ezek tárolják, hogy melyik assetre, milyen operációtípus esetén, milyen hozzáférést szeretnénk, milyen prioritással, és hogy ez a szabály a policy engedélyező vagy tiltó tulajdonsága alapján domináns-e vagy sem. A *permissionList*-nek a folyamat többi részének tekintetében is fontos tulajdonsága, hogy az ugyanarra az assetre azonos jogosultságot adó judgementeket egyenlőnek veszi, és közülük csak a nagyobb prioritásút tartja meg.

   (b) **Default szabályokból:** A policyban meghatározott default jogosultságok hozzáadásához bejárja a modell minden egyes objektumát, annak attribútumait, referenciáit, és az összes assetre, mindkét operációtípusra létrehozza az adott hozzáférési szintű judgementet a lehető legkisebb prioritással. Így a default judgementekből összesen kétszer annyi lesz mint a modellelemek száma, hiszen mindegyik olvashatóságáról és írhatóságáról is lesz egy-egy kijelentés.

2. **Effektív eredmények listája:** A *permissionList*-en kívül egy másik judgementek tárolására szolgáló lista is létrejön, a *processed*, amelybe futása során az algoritmus a már érvényre jutó judgementeket helyezi.

A judgement-listák inicializálása után megkezdődik az effektív jogosultságok számítása. A program egy while ciklusban hajtja végre azt az iterációt, melynek során az eredmény a permissionList kiürülése közben a *processed* listába kerül, és amelynek pontos lépései a következők:

3. **Legdominánsabb judgement kiválasztása:** Ez gyakorlatilag a *permissionList* első elemének kivételét jelenti, ugyanis a lista eszerint rendezett. Elsősorban a nagyobb prioritású elemeket veszi előre, az azonosak közül pedig az engedélyező vagy tiltó tulajdonságukból fakadóan dominánsak sorolódnak a prioritási osztály elejére. Egy elem listabeli helye hozzáadáskor alakul ki, ekkor a program egy mapből olvassa ki, hogy hol van az első ugyanilyen fontosságú judgement, az elé szúrja be, majd ezek alapján frissíti a mapet.

4. **Kiválasztott judgement áthelyezése** *permissionList*-**ből** *processed*-**be:** Mivel ennél a judgementnél nincs fontosabb a listában, ezért ő mindenképp érvényre fog jutni, úgyhogy a nominális szabályokat tartalmazó listából átkerül az effektív szabályok közé.

5. **Konfliktusfeloldás:** Judgementek bevezetésével könnyen detektálhatóvá váltak a szabályok közötti I. típusú konfliktusok. A *permissionList*-en végigiterálva olyan elemeket keresünk, amelyek ugyanarra az assetre és ugyanarra az operációtípusra, viszont eltérő hozzáférési szintre vonatkoznak, mint a kiválasztott judgement. Ezeket egyszerűen kitöröljük a listából, hiszen már úgysem érvényesülhetnek.

Az 5.2.1 fejezetrészben felsorolt függőségeket az algoritmus "erős" konzekvenciaként kezeli, vagyis ha egy judgement érvényre jut, akkor a belőle ezek alapján származtatott judgementek megöröklik a prioritásukat. Ennek köszönhetően mivel a következő körökben ők lesznek a legdominánsabbak, szintén érvényre fognak jutni. Ezeken kívül a program "gyenge" konzekvenciákat is figyelembe vesz. Ezek értelmében az objektumok attribútumai és referenciái megöröklik az objektumhoz tartozó hozzáférési jogot, tehát például ha az olvasható, akkor az attribútumok, referenciák is azok lesznek. Ezeket a judgementeket a defaultnál nagyobb, de az erős konzekvenciáknál kisebb prioritással adja hozzá az algoritmus az érvényesíteni kívánt szabályok halmazához, tehát ez utóbbi felülírhatja őket.

6. **Erős konzekvenciák propagálása:** A külön erős konzekvenciákat tároló *strongCons* listán iterálunk végig, és minden *IConsequence*-t megvalósító osztály *propagate*() függvényének eredményét, az új judgementeket hozzáadjuk a *permissionList*-hez.

7. **Gyenge konzekvenciák propagálása:** Az előző ponthoz hasonlóan, csak itt a *weakCons* listán iterálunk végig.

A különböző függőségek propagálása után újraindul a while cikluson belüli folyamat, és egészen addig tart, amíg a *permissionList* ki nem ürül. Eközben a $getEffectivePermissions()$ függvény visszatérési értéke, a *processed* lista feltelítődik az effektív hozzáférési jogokkal.

5.2.4. Algoritmus értékelése

Az algoritmus terminál. A while ciklus addig fut, amíg a nominális judgementeket tartalmazó *permissionList* ki nem ürül. Ez akkor lehetséges, ha a hozzáadható judgementek száma véges. A modellelemek számának végessége miatt a lista inicializálásakor hozzáadott explicit és default judgementek száma is az lesz. A ciklus futása alatt pedig a függőségek propagálása miatt bővül a lista, de mivel a kódban az is szabályozva van, hogy olyan judgementet ne lehessen hozzáadni a *permissionList*-hez, ami már szerepel a *processed*-ben, ezért ezek száma is véges.

Utóbbi gondolatra egy példa az F2 és F3 függőségek kapcsolata. Ha érvényesül egy tartalmazási referencia láthatósága, akkor a gyerek objektumnak is láthatónak kell lennie. Ha ez utóbbi is érvényesül, akkor a propagálásnál azt a judgementet adnánk a listához, hogy a rá mutató tartalmazási referencia is olvasható. Ha hozzáadnánk, akkor ennek a kiválasztásánál újraindulna ez a kör, ezt nem engedi az algoritmus.

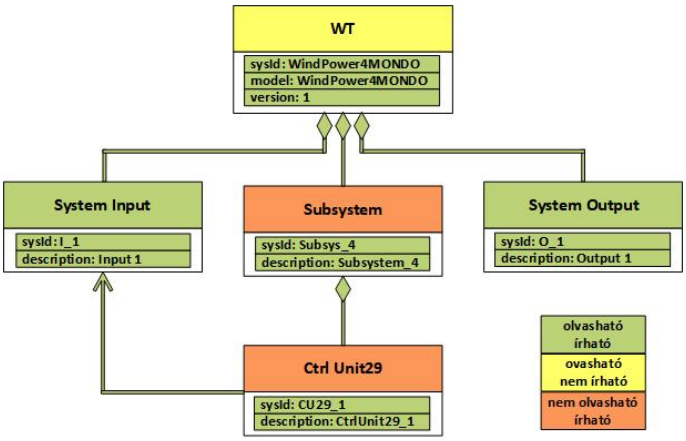
Az algoritmus kétszer annyi judgementet ad vissza mint az assetek száma. A kétszeres szorzót úgy kapjuk, hogy minden assethez egy darab olvasási és egy írási jogosultságot határozunk meg, vagyis az szükséges, hogy a *processed* listába minden asset mindkét operációtípusához csak egy judgement kerülhessen át a *permissionList*-ből.

A judgement kiválasztásának pillanatában a listában nem szerepel ugyanerre az operációra és hozzáférési szintre vonatkozó másik judgement, mert a listához adáskor ezt figyelembe veszi a program és az ilyenek közül csak a magasabb prioritásút teszi bele. Ugyanerre az operációra és különböző hozzáférési szintre vonatkozó judgement azonban lehet benne, de a konfliktusfeloldás részeként törlődnek.

Felmerülhet a kérdés, hogy miután már a *processed* listában van az elem, egy másik judgement függőségeinek propagálásakor nem kerülhet-e be egy ugyanolyan vagy éppen az ellenkezőjét kifejező judgement a *permissionList*-be. Az ugyanolyan esetre az előző bizonyításban is szereplő módszer a megoldás, vagyis arra figyel a program, hogy *processed*-ben lévő judgement-et még egyszer ne adjon hozzá a listához. Ellentétes hozzáférési szintű judgement pedig az általam definiált függőségek bizonyítható logikussága miatt nem lehetséges.

5.2.5. Esettanulmány

A megoldás esettanulmányon való bemutatásához a 2 fejezetben meghatározott hozzáférés-szabályozási követelmények alapján az 5.1 kódon látható explicit és default szabályok fogalmazhatók meg. Ezek alapján a 3.2 ábrán lévő példánymodellen az I/O Manager felhasználóra az 5.8 ábrán látható hozzáféréseket szeretnénk biztosítani a különböző modellelemekhez. A gyökérelemet csak láthatja, de nem módosíthatja, a SystemInput és SystemOutput objektumokat olvashatja és írhatja is, a Subsystem és Ctrl Unit29 objektumokhoz pedig egyáltalán nem férhet hozzá. Mivel az objektumok attribútumaira és referenciáira nem adtunk meg szabályt, ezért ezekre a default beállítást értelmeznénk, ami mindkét műveletet engedélyezi.



5.8. ábra. *Az I/O Manager számára adni kívánt hozzáférések*

1. *permissionList* inicializálása:

   (a) Az explicit szabályok közül a "restrictRoot" a gyökérelemre tiltja meg az írhatóságot 1-es prioritással, a policy tiltó tulajdonságával, így az ebből származtatott judgement:
   $j(WT, deny, W, 1, true)$
   A másik rá vonatkozó szabály, a "restrictNotIO" nem engedi a gyökérelemen, bemeneteken és kimeneteken kívüli objektumokhoz való hozzáférést, ami ebben az esetben a következő judgementeket eredményezi:
   $j(Subsystem, deny, R, 1, true)$
   $j(CtrlUnit29, deny, R, 1, true)$

   (b) A default szabályok a modell minden assetjére olvashatóságot és írhatóságot biztosítanak a lehető legkisebb, -1 prioritással, így a két judgement amit mind a 21 modellelemhez rendel, és a *permissionList*-hez az ad az algoritmus:
   $j(asset, allow, R, -1, false)$
   $j(asset, allow, W, -1, false)$

2. *processed* lista inicializálása, amibe az effektív eredmény kerül majd

Ezek után az érvényesíthető judgementek *permissionList*-ből *processed*-be kerülésének iterációnkénti lépései az alábbiak:

3. Mivel a judgementek dominánssága szerint rendezett *permissionList*-be egymás elé szúrjuk be az ilyen szempontból azonos elemeket, ezért az első tagja:
   $j(CtrlUnit29, deny, R, 1, true)$

4. Mivel ez mindenképp érvényre jut, hozzáadódik a *processed* listához, feldolgozás után pedig törlődik a *permissionList*-ből.

5. Azért, hogy a *processed*-be ne kerülhessen olyan judgement, ami az épp kiválasztottal és hozzáadottal konfliktusba kerülhet, az algoritmus még a *permissionList*-ből kitörli az ugyanarra az assetre, ugyanarra az operációtípusra, viszont más hozzáférési szintre vonatkozó judgement-eket. Ebben az esetben egy konfliktushelyzet adódik:
   $j(CtrlUnit29, allow, R, -1, false)$
   Ez a defaultként hozzáadott judgement a már érvényre jutottal szemben engedélyezné a Ctrl Unit29 objektum olvashatóságát, ezért törlésre kerül a *permissionList*-ből.

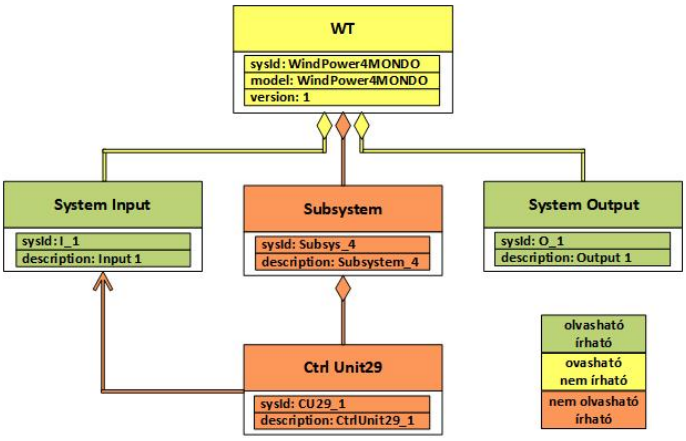
6. Az érvényesülő judgementre a következő erős konzekvenciák vonatkoznak:

   - F8 értelmében ha egy asset nem olvasható, akkor írható sem lehet, ezért a következő judgement amit a listához kell adni:
     $j(CtrlUnit29, deny, W, 1, true)$
   - F2 szerint ha egy objektum nem olvasható, akkor a be- és kimenő referenciák se legyenek azok:
     $j(Subsystem \rightarrow CtrlUnit29, deny, R, 1, true)$
     $j(CtrlUnit29 \rightarrow SystemInput, deny, R, 1, true)$

7. A gyenge konzekvenciák miatt pedig az objektum attribútumaira és referenciáira ruházzuk tovább az olvashatatlansági jogot a defaultnál nagyobb, de a többinél kisebb 0 prioritással. Mivel az egy szóba jövő referenciára már az előző pontban megadtunk egy erősebb judgementet, ezért arra már ezt felesleges lenne a listához adni. Az attribútumokra pedig ezek vonatkoznak:
   $j(CtrlUnit29.sysID, deny, R, 0, true)$
   $j(CtrlUnit29.description, deny, R, 0, true)$

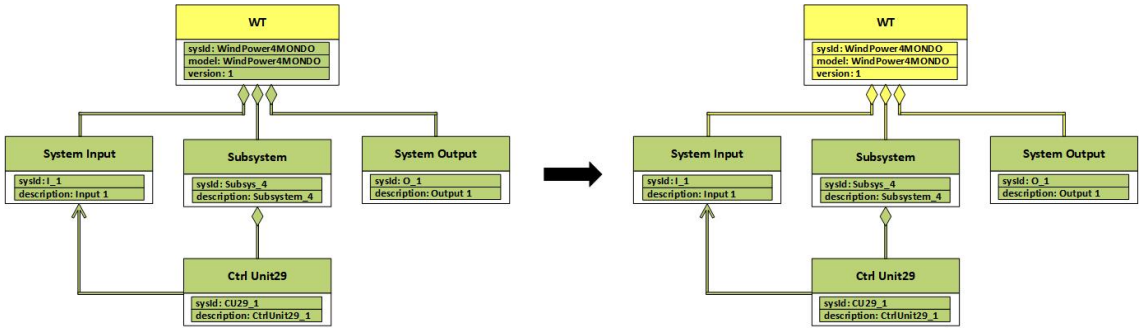
A függőségek propagálása után a while ciklus következő iterációja jön, ahol a *permissionList*-hez utoljára hozzáadott legdominánsabb judgementet választja ki az algoritmus és teszi át a *processed* listába, ez pedig a $j(CtrlUnit29 \rightarrow SystemInput, deny, R, 1, true)$. Konfliktusa szintén az assethez rendelt default hozzáféréssel lesz, a definiált függőségek közül csak az F8 vonatkozik rá, ezért a listához azt a judgementet is hozzá kell adni, ami az asset írhatóságát tiltja.

A cikluson ilyen módon addig iterálunk végig, amíg a *permissionList* összes elemét fel nem dolgoztuk akár a *processed* listába való áthelyezéssel, akár törléssel. A folyamat végére az I/O Manager számára a rá vonatkozó effektív hozzáférési szabályok alapján az 5.9 ábrán látható konzisztens modellnézet érvényesül.

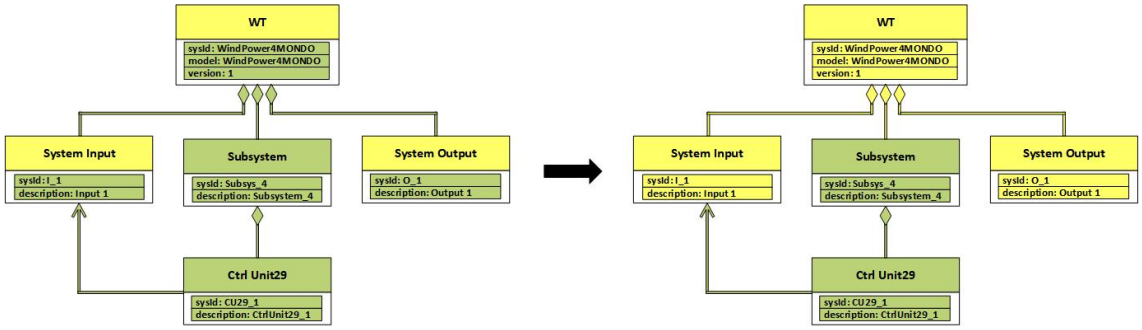


5.9. ábra. *Az I/O Manager számára érvényesülő effektív hozzáférések*

Az esettanulmányban szereplő másik két felhasználóra futtatva az algoritmust a **??** ábrán látható eredmények születtek. Bal oldalon szerepelnek a szabályok által igényelt hozzáférések, jobb oldalon pedig az algoritmus futási eredményeként kapott effektív jogosultságok.



(a) *Principal Engineer*



(b) *Subsystem Manager*

6. fejezet

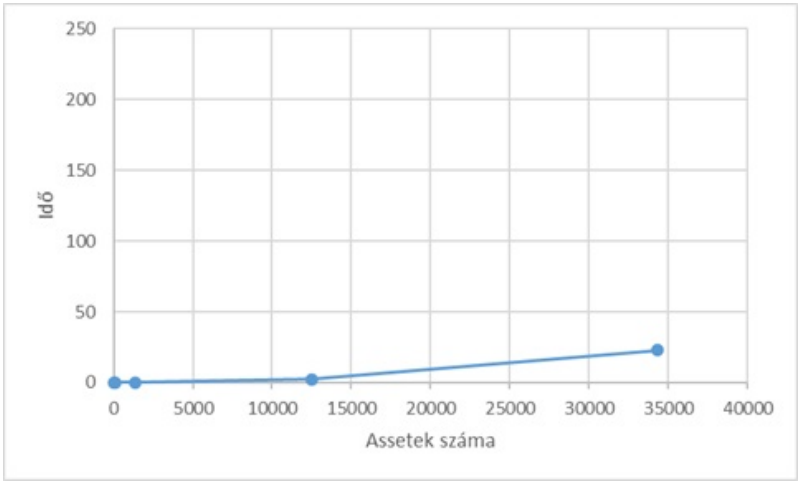
Kiértékelés

Az elkészült szabálykiértékelő komponenst futási teljesítménye szempontjából értékeltem. A teszteléshez 5 különböző méretű példánymodellt készítettem annak vizsgálatára, hogy a modellelemek számának növekedésével milyen arányban nő az algoritmus számítási ideje, majd ezeket mind a három típusú felhasználóra lefuttattam. A tesztesetek bemenetei a következők voltak:

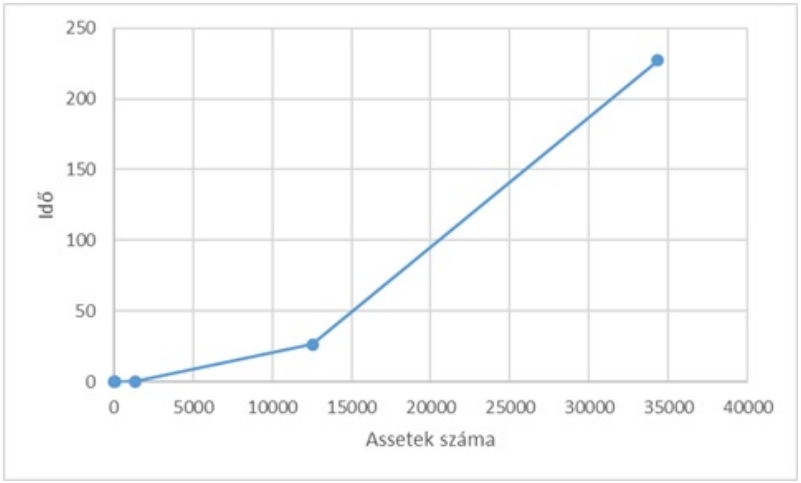
- az esettanulmány követelményei alapján definiált explicit hozzáférési szabályok
- a szabályok által használt gráfminták
- a korábban tárgyalt írási és olvasási függőségek
- példánymodell
- felhasználó

A méréshez Windows10(x64) operációs rendszerű, egy Intel Core i3-as, 1,80 GH-es processzorral és 8 GB memóriával rendelkező gépet használtam, Eclipse Neon.3 keretrendszerrel. A teszteseteket egyenként 10-szer lefuttatva, majd a mérési eredményeket átlagolva a 6.1, a 6.2 és a 6.3 ábrákon látható eredményt kaptam. A grafikonok vízszintes tengelye a modellelemek számát mutatja, függőlegesen pedig a futási idők olvashatók le róla.

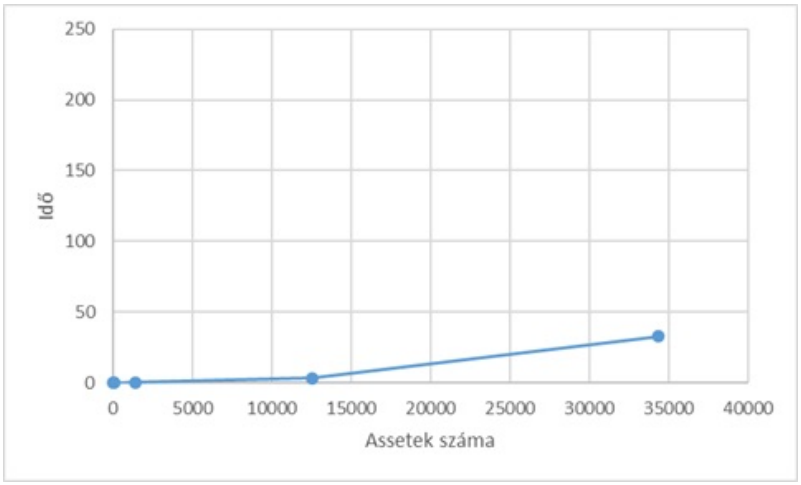
A vártaknak megfelelően a modell méretének növekedésével egyenes arányosságban emelkedik a futási idő, hiszen így több szabályt is kell kiértékelni. Megfigyelhető még, hogy a másik két felhasználóhoz képest az I/O Manager-hez tartozó effektív jogosultságokat nagyságrendekkel több ideig tart kiszámolni. Ennek szintén a nagyobb méretű judgementhalmazkezelés az oka, mert erre a felhasználóra olyan szabályok lettek explicit módon megadva, amelyek függőségei miatt még több jogosultság propagálódik.



6.1. ábra. *Principal Engineer*



6.2. ábra. *I/O Manager*



6.3. ábra. *Subsystem Manager*

7. fejezet

Kapcsolódó munkák

7.1. Fájl alapú hozzáférés-szabályozás

A dolgozat bevezetésében is említett fájl alapú hozzáférés-szabályozásban a fájlrendszerek megkövetelik a fájloktól, mappáktól, hogy valamilyen hozzáférési jog explicit legyen hozzájuk definiálva. Ez történhet például az objektumhoz csatolt úgynevezett Access Control List (ACL) formájában, aminek egy-egy bejegyzése tartalmazza, hogy mely felhasználók vagy rendszerfolyamatok férhetnek hozzá az objektumhoz, illetve, hogy milyen műveleteket hajthatnak végre rajta. Például az (Alice: read,write; Bob: read) tartalmú ACL olvasási és írási jogot biztosít az Alice nevű felhasználónak, valamint csak olvasási jogot Bob-nak. A listában lévő szabályok közötti ellentmondásokat a sorrendjük alapján meghatározott prioritásuk oldja fel, a II-es és III-a típusú konfliktusok esetén pedig a tiltó szabályok részesülnek előnyben.

Modellvezérelt fejlesztés során ez a hozzáférés-szabályozási módszer is alkalmazható, bár a szakdolgozatban bemutatott megoldás ezzel szemben több előnnyel is rendelkezik. Többek között biztosítja a szabályok finomszemcsézettségét, akár egyesével is meg lehet határozni a különböző elemekre vonatkozó jogosultságokat, nem csak az egy fájlban lévő elemek csoportjára. Ezen kívül a szabályok implicit módon megfogalmazhatók, azok alkalmazásakor figyelembe veszi a modell belső konzisztenciáját és rugalmasabb megoldást nyújt konfliktusok kezelésére.

7.2. XML dokumentumok hozzáférés-szabályozása

XML dokumentumokhoz finomszemcsés hozzáférés-szabályozást többek között az eXtensible Access Control Markup Language (XACML) [7] is nyújt, amellyel XML-ben lehet felhasználók objektumokra vonatkozó hozzáférési szabályait megfogalmazni. Ez a típusú dokumentum modellekhez hasonlítható, ugyanis csomópontok attribútumokkal, és ezeken belül tartalmazható csomópontok találhatók benne. Az egymásnak ellentmondó szabályok feloldásához az XACML több egymással is kombinálható algoritmust nyújt, a szakdolgozatban is megvalósított logika alapján képes kiválasztani a dominánsabb szabályokat.

8. fejezet

Összefoglalás

A nagyméretű ipari szoftverek tervezése egy vagy akár több céghez tartozó fejlesztő csapatok együttes munkáját igényli. A modellvezérelt szoftverfejlesztési paradigma előnye az automatikus kód-, teszteset-, dokumentációgeneráláson kívül, hogy az átlátható, magas absztrakciós szintű modellek különféle szakterületekhez értő mérnökök számára is ugyanolyan módon értelmezhetők, ezzel összehangolva a közös munkavégzést.

Ilyen komplex modellek és ennyi különböző szerepkörű, érdekeltségű kollaborátor esetén elengedhetetlen a modellek hozzáférés-szabályozása. Egyrészt egy adott domain szakemberének is megkönnyíti a fejlesztést, ha csak a számára releváns részeit látja a modellnek, másrészt lehetnek benne olyan bizalmas vagy kritikus elemek is, amelyekhez ideális esetben kizárólagos joggal rendelkeznek a hozzáértők.

A modellszintű hozzáférés-szabályozás lényege, hogy finomszemcsés szabályokban fogalmazhatjuk meg, pontosan mely modellelemekre milyen írási és olvasási jogot szeretnénk biztosítani. Ezeket a szabályokat azonban nem tudja közvetlenül felhasználni az a modelltranszformáció, amely az adott felhasználó számára elkészíti az általa olvasható elemekből álló modellnézetet, mert a szabályok finomhangolási lehetőségei miatt adódhatnak közöttük olyan konfliktusok, amelyek veszélyeztetik a modell transzformáció után is megtartandó belső konzisztenciáját.

A szakdolgozat célja egy ilyen modellszintű hozzáférés-szabályozás helyes működésének támogatása volt, amelyhez az alábbi részfeladatokat valósítottam meg:

1. Készítettem egy hozzáférési szabályok leírására alkalmas szöveges konkrét szintaxist, amely az alábbi jellemzőkkel rendelkezik:
    - Gráflekérdezéssel a szabályban tetszőleges modellelemek halmazára fogalmazható meg a jogosultság.
    - Finomszemcsézett, mert a gráflekérdezés akár egyetlen assetet is adhat eredményként, assethalmazra pedig további szűrőfeltételek is megadhatók.
    - A szabályok között fontossági sorrend állítható fel a hozzájuk rendelt prioritás és a policy-hez rendelt engedélyező vagy tiltó tulajdonság segítségével.

- A policy-hoz alacsony prioritású default hozzáférési jog is definiálható, amely azokra az assetekre érvényesül, amire más szabály semmit nem mond.

2. Megvalósítottam a szabályokat EMF modellek felett kiértékelő algoritmust,
   - amely értelmezi a felhasználó által megadott szabályokat,
   - és a modell belső konzisztenciáját a judgement-ek közötti függőségek propagálásával megtartva,
   - a közöttük emiatt esetlegesen kialakuló konfliktusokat feloldva
   - a modell minden elemére eredményül adja a hozzá tartozó, valóban érvényesülő olvasási és írási jogosultságokat.

Az elvégzett feladat egy további fejlesztési iránya lehet a rendszer inkrementalitásának bevezetése, hogy a szabályokon végzett kisebb változtatás után a most megvalósított számítás újrafuttatása nélkül frissülhessen az effektív hozzáférési jogosultságok halmaza.

Irodalomjegyzék

[1] Suzana Andova, Mark G. J. van den Brand, Luc J. P. Engelen, and Tom Verhoeff. *MDE Basics with a DSL Focus*, pages 21–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[2] Alessandra Bagnato, Etienne Brosse, Andrey Sadovykh, Pedro Maló, Salvador Trujillo, Xabier Mendialdua, and Xabier De Carlos. Flexible and scalable modelling in the MONDO project: Industrial case studies. In *XM@ MoDELS*, pages 42–51, 2014.

[3] Gábor Bergmann, Csaba Debreceni, István Ráth, and Dániel Varró. Query-based access control for secure collaborative modeling using bidirectional transformations. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, pages 351–361, 2016.

[4] Csaba Debreceni, Gábor Bergmann, István Ráth, and Dániel Varró. Deriving effective permissions for modeling artifacts from fine-grained access control rules. In *Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016) co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), St. Malo, France, October 4, 2016.*, pages 17–26, 2016.

[5] Zinovy Diskin. *Algebraic Models for Bidirectional Model Synchronization*, pages 21–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[6] MONDO. The mondo project.

[7] Oasis. Xacml.

[8] Eclipse Project. Eclipse modeling framework.

[9] Eclipse Project. Viatra.

[10] Eclipse Project. Xtext.