



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Berkes Sándor

TULAJDONSÁG ALAPÚ ZÁROLÁS MEGVALÓSÍTÁSA EMF MODELLEK FELETT

KONZULENS

Debreceni Csaba

doktorandusz

BUDAPEST, 2015

Tartalomjegyzék

Összefoglaló	6
Abstract.....	7
1 Bevezetés	8
Dolgozat célja	9
Dolgozat struktúrája.....	9
2 Esettanulmány.....	10
Szélerőmű irányítási rendszer modellje	10
Zárolás megvalósítása szélturbina példánymodellel.....	10
3 Háttér technológiák	13
Verziókezelő Rendszerek	13
3.1.1 SVN	13
3.1.2 GIT.....	13
Eclipse.....	17
Eclipse Platform.....	17
3.1.3 Eclipse PDE	18
3.1.4 OSGi	19
3.1.5 EMF	19
3.1.6 IncQuery	22
3.1.7 EVM.....	23
3.1.8 Egyéb Eclipse által biztosított eszközök.....	24
Jersey	27
3.1.9 REST.....	27
3.1.10 JSON.....	28
4 Áttekintés	29
Kliens – Szerver felépítése.....	29
Kliens működése.....	30
5 Megvalósítás	32
Server	32
5.1.1 Projekthez zárfeltöltés.....	32
5.1.2 Adott projekthez tartozó zárok lekérdezése	33
5.1.3 Projekt egy zárjának letöltése	33

5.1.4 Funkciók összefoglalás	33
Kliens	33
5.1.5 Projekt leíró.....	33
5.1.6 Zárak definiálás.....	34
5.1.7 Zárak megvalósítása	35
5.1.8 Beállítások	39
5.1.9 Jelölök használata	39
5.1.10 Jersey használata.....	40
6 Kiértékelés	42
Memória igény	44
Zárak aktiválásához szükséges idő	45
Művelet végrehajtása zárolt modellen	46
Összegzés.....	47
7 Kapcsolódó munkák	48
Zárak megvalósítása általában	48
CDO	48
EMFStore.....	49
EMFStore és CDO összehasonlítása.....	49
7.1.1 Adattárolás	49
7.1.2 Kollaboráció.....	50
7.1.3 Jogok kezelése, zárolás	50
7.1.4 Konfliktusok megoldása	50
7.1.5 Skálázhatóság.....	51
7.1.6 Összefoglalás	51
MetaEdit.....	51
7.1.7 Modellszerkesztés	52
7.1.8 Kollaboratív munka támogatása	52
7.1.9 Integrálhatóság, más környezetekkel való kapcsolat	52
8 Összefoglalás.....	54
Továbbfejlesztési irányok	54
8.1.1 Nagyobb kifejező erővel rendelkező zárok.....	54
8.1.2 Zár sértés esetén visszajelzés	54
8.1.3 Intelligens zárelhelyezés	55
8.1.4 Hatékonyabb kommunikáció	55

8.1.5 Magasabb szintű integráció	55
Irodalomjegyzék.....	56

HALLGATÓI NYILATKOZAT

Alulírott Berkes Sándor, 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 hitelesített 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.

Kelt: Budapest, 2015. 05. 24.

.....
Berkes Sándor

Összefoglaló

Napjainkban a modellvezérelt fejlesztés (MDE) egyre népszerűbb a komplex rendszerek tervezése területén. A módszertant kiegészítve kollaborációs eszközökkel a mérnökök egy időben ugyanazon a modellen képesek dolgozni, így nagyban nő a produktivitásuk. A felhasznált eszközök funkciói ugyan már manapság is sokrétűek, azonban a folyamatosan növekedő igények az eszközök folyamatos bővítését és fejlesztését igénylik.

Hogy ezen eszközök megfelelő irányban fejlődjenek, a MONDO projekt a modell vezérlés elméleti alapjaival, skálázható modellezéssel és ezek egy nyílt-forrású felhő alapú rendszerbe való integrálhatóságával foglalkozik. A munkám itt kapcsolódik be.

A feladatom egy kollaborációt támogató, EMF modelleken tulajdonság alapú zárolást megvalósító eszköz létrehozása volt. A munkám során megvizsgáltam több már meglévő alternatívát, megvizsgáltam a verziókezelő rendszereket és az Eclipse eszközeit is. Ezek után az Eclipse IncQuery eszköze segítségével hoztam létre egy szakterület és szerkesztő független modell zárolási eszközt.

Eredményül egy nagy memória igényű, de hatékony, gyors eszközt kaptam. Az IncQuery használata által egy könnyen értelmezhető, deklaratív zármintákat kaptam. Ezen minták változói fix értékeket vehetnek fel, vagy szabad paraméterként funkcionálhatnak. A zárok könnyen megoszthatóak egy központi szerver segítségével, majd ezek alkalmazhatóak a kliensek oldalain. Így a megoldás egy jó kollaborációs alapréteget nyújt a modellek fejlesztése során.

Abstract

Nowadays the model driver engineering (MDE) is getting more and more popular in the field of complex system development. Expanding the methodology with the tools of collaboration the engineers can work on the same model at the same time, as for that their productivity increase greatly. Features of the applied tools are diverse in these day too, but the continuously growing requirements needs continuous expand and evolution of the tools.

As for making these tools progress on the right way, the MONDO project is working on fundamentals of control theory, scalable modelling and integrating these into an open source cloud base system. My work starts here.

My task is to create a tool that implements property based lock feature over EMF models besides supporting collaboration. Through my work experiences, I explored some already existing option, I checked version control system and the tools of Eclipse. After these with the help of Eclipse IncQuery I have created a domain and editor independent model locking tool.

As a result, I have got a high memory demanding, but efficient, fast tool. As using IncQuery, I have got an easily interpretable, declarative lock pattern. The variables of the patterns, can be fixed to values, or they can be non-determined. The locks can be easily shared with the help of a central server, then they can be applied on the clients' side. By this way, this solution gives a good base collaboration layer for model development.

1 Bevezetés

Napjainkban a komplex rendszerek tervezése és üzemeltetése igen nagy kihívást jelent a fejlesztők számára. Erre adhat megoldást a Model Driven Engineering[1] (MDE), a modell alapú rendszertervezés. Az MDE módszere által a fejlesztés során modellek jelennek meg a szokványos program kódok mellett. A modellek az adatkezelési struktúrák mellett a szoftver modellezésére is használhatóak, így a fejlesztés átláthatósága, hatékonysága nagymértékben nőhet. A magas szintű ábrázolásmód miatt, sok esetben lehetőség van a modellből való kódgenerálásra, ezáltal a modellen végzett transzformációk egyből megjelenhetnek az alkalmazás kódjában is.

A megközelítés lehetőségei miatt széles körben kezdték el felhasználni az MDE módszertanát. Azonban ahogy egyre komplexebb rendszerek jelentek meg, és ezzel egyre nagyobb kihívások is jelentek meg, a jelenlegi fejlesztőeszközök mind teljesítményben mind hatékonyságban kezdtek kevésbé válni. Épp ezért szükség van ezen eszközök további fejlesztésére, a témában való kutatásokra, hogy MDE módszertana megfelelően tudjon megbirkózni a folyamatosan növekvő igényekkel. A MONDO projekt célja pedig pont ez, megpróbál új alapokat, megközelítéseket adni a módszertannak további kihasználása érdekében. A MONDO[2] fő céljai a következő eszközök, funkciók megteremtése:

- Nagy modellek hatékony létrehozása szakterület specifikus nyelvek segítségével
- A létrehozott modelleken kollaboratív munka lehetősége
- Fejlett modell szerkesztők, absztrakt ábrázolási formák, ezáltal széles funkcionalitású transzformációk, keresési lehetőségek
- Hatékony modell tárolási rendszerek létrehozása

Az Eclipse Modelling Framework egy nagy testreszabhatósággal, gazdag funkcionalitással rendelkező modellező eszköz. Kollaboratív munka támogatására több eszközzel is rendelkezik, azonban ezen megoldások zárolási funkciói szegényesek, egy komolyabb, több felhasználós környezetben nem elegendőek.

A feladat részeként irodalomkutatást végeztem a jelenleg népszerű kollaborációs technológiák terén, itt a GIT verziókezelő rendszerét vizsgáltam meg jobban. Mivel

lényegi megoldást nem nyújtott a problémára, így az Eclipse Platformon belül két ilyen jellegű eszközt vizsgáltam meg a CDO-t és az EMFStore-t. Hogy objektívebb képet kapjak egy fizetős megoldást is megvizsgáltam, a MetaCase MetaEdit+ szoftverét.

Végül pedig egy saját eszközt hoztam létre, amihez a tanszéken fejlesztett IncQuery és ezt kiegészítő EVM (Event-driven Virtual Machine) eszközöket használtam fel. Az elkészült alkalmazás szakterület független modell zárolást tesz lehetővé. Az eszköz valódi hasznát az jelenti, hogy ellentétben az összes többi Eclipsen belüli kollaborációs megoldással, ez tulajdonság alapú zárolást hajt végre, így jóval kényelmesebb és hatékonyabb munka végezhető vele.

Dolgozat célja

A szakdolgozatomban kitűzött célok tehát a következők:

- Megismerkedni több EMF modellek kollaborációs szerkesztését támogató eszközzel, majd összehasonlítani őket funkcionalitásuk terén
- Felkutatni az EMF modellek zárolási lehetőségeinek mikéntjét
- Megvalósítani egy példa alkalmazást tulajdonság alapú zárolásra és bemutatni képességeit

Dolgozat struktúrája

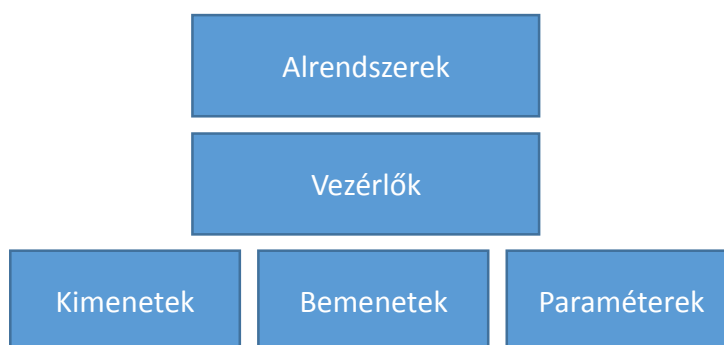
A 2. fejezetben dolgozatomban motiváló problémáját fogom bemutatni. Ezek után a 3. fejezetben az általam elkészített alkalmazáshoz kapcsolódó és felhasznált technológiákat, eszközöket mutatom be. A 4. fejezetben megoldásomban kulcs fontosságú komponenseit, és működésének fő logikáját ismertetem. Az 5. fejezetben részletesen bemutatnám az eszköz működését, majd az 6.-ban értékelem azt, és mérések segítségével elemzem hatékonyságát. A 7. fejezetben a témával kapcsolatos megoldásokat hasonlítok össze, majd ezek tükrében a 8. fejezetben lehetséges továbbfejlesztési irányokat sorolok fel.

2 Esettanulmány

Ahhoz hogy bemutassam alkalmazásom valós felhasználást a MONDO egyik partnerétől, az IKERLAN[3] kutató és fejlesztő cégtől származó modellt fogom felhasználni. A felhasznált modell egy szélérőmű vezérlőrendszerét testesíti meg, fa gráf formájában. A következő részben a modell felépítését és alapvető elemeit mutatom be.

Szélérőmű irányítási rendszer modellje

A modell által definiált szélérőmű vezérlő egységekből épül fel. Ezen egységeknek van a rendszer felé ki és bemenetük és rendszer által meghatározott paraméterük. A rendszer fő szeparációs egysége az alrendszer. Ezen alrendszerek alá a vezérlők sorakoznak be. Minden egyes elem azonosításáról egy egyedi leíró gondoskodik. Egy szélérőmű hierarchikus szerkezetét az 1. ábra mutatja.



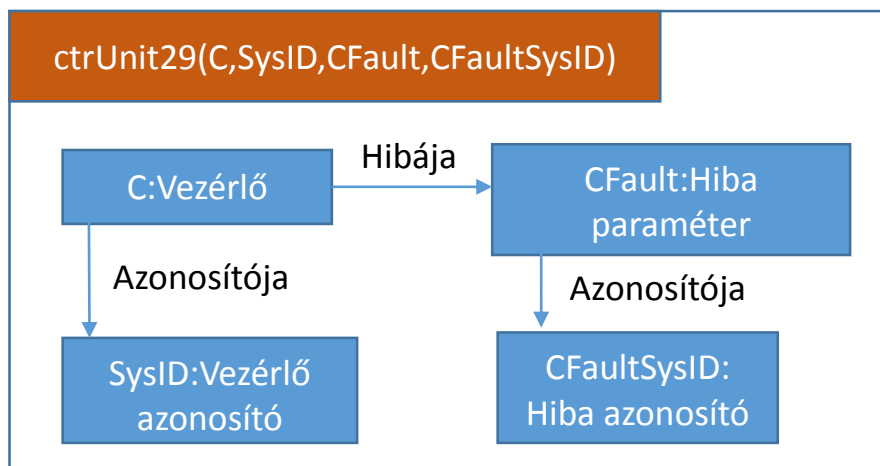
1. ábra Szélérőmű felépítése

Zárolás megvalósítása szélturbina példánymodellen

Tételezzük fel, hogy a modellen dolgozó mérnököknek nem csak egy 1-1 modell elemet kell zárolniuk, ha nem egy komplexebb logikát követő megoldásra van szükségünk.

Két zárat szeretnének elhelyezni, az elsőnek (2. ábra) a következő feltételeket kell teljesítenie:

- 29-es típusú ControlUnit és ID-ja zárolása CtrlUnit_232 azonosító esetén
- A hozzátartozó hibák és azonosítója zárolása



2. ábra Első deklaratív zár minta

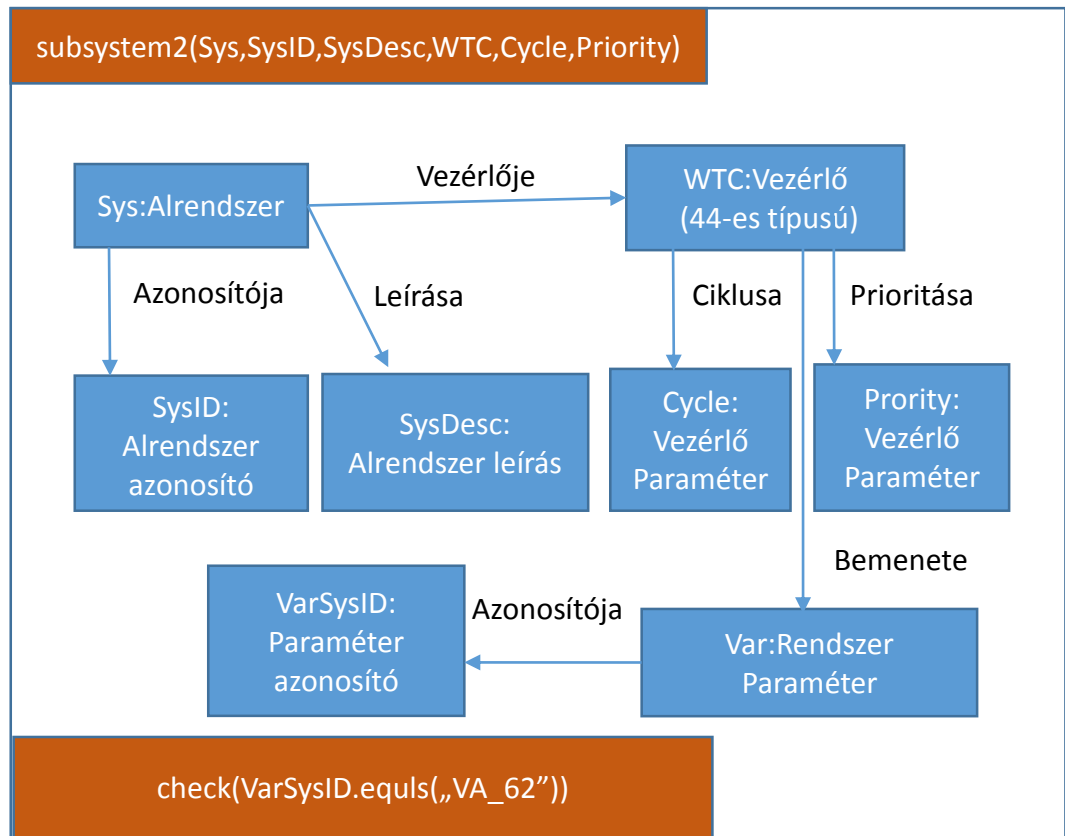
Az ábrán a függvény paraméterei, lentebb pedig a paraméterek megszorításait láthatjuk. Az ilyen deklaratív függvény működését tulajdonképpen felfoghatjuk úgy, hogy az összes lehetséges bemenet közül, a változók azon kombinációi lesznek az eredményhalmazban, amelyekre a fentebbi kapcsolat rendszer és típus megkötések igazak lesznek. A zár működése szempontjából fontos, hogy azon objektumok és tulajdonságok kerülnek zár alá, amelyek a függvény eredményhalmazában szerepelni fognak.

A fentebbi mintához való megszorítás pedig:

SysID=CtrlUnit_232

A második zár (3. ábra) esetében már komplexebb zár igények állnak fenn:

- A „Subsystem 07” azonosítóval rendelkező alrendszer, ID, Description tulajdonsága
- Ezen alrendszeren belüli azon 44-es típusú vezérlők Cycle és Priority zárolása ahol az Input__iInput „VA_62” azonosítójú SystemVariable típusra mutat



3. ábra Második deklaratív zár minta

Az alkalmazott megszorítás:

SysID=Subsystem 07

Mivel itt a VarSysID objektumot nem szeretnénk volna zárolni, ezért a paraméter listán nem jelenhetett meg a mintánál. Így azonban értékére csak záron belül tudunk megadni megszorítást, a check kulcsszó használatával.

3 Háttér technológiák

Ebben a fejezetben a munkám során felhasznált technológiákat, eszközöket szeretném bemutatni. Mivel az általam elkészített megoldás lényegében egy lehetséges kiegészítése egy verziókezelő rendszernek, így elsőre is ezek közül szeretnék bemutatni kettőt, a GIT és SVN verziókezelőket. Ezek után az Eclipse Platformot fogom bemutatni és általam használt szolgáltatásait.

Verziókezelő Rendszerek

A verziókezelő rendszerek fő lényege, hogy a szoftverfejlesztő az által írt program forráskódját megfelelően tudja menedzselni. Értjük ez alatt a forrás rendszeres mentését, nyomon követést, igény esetén egy pontra visszaállítása, illetve egyéb forrás manipulációs lehetőségeket. Mindezeket manapság kollaboratív környezetben valósítják meg a rendszerek, tehát több személy is dolgozhat ugyanazon a forráson egy időben. Jelenleg a piacon két fő rendszer a domináló, ezeket fogom most bemutatni részletesen.

3.1.1 SVN

Az SVN[4] eredeti nevén Apache Subversion egy központi, centralizált verziókezelő rendszer. Ebben az esetben a felhasználók egy közös központi szervert használnak. A változtatásaikat ide töltik fel, illetve a mások által végrehajtott változtatásokat is innen töltik le. Fontos megjegyezni, hogy minden egyes feltöltést egy letöltés előz meg, hisz a lokális módosítást muszáj összefésülni a szerveren lévő jelenlegi kóddal, majd az összefésült verzió kerül továbbításra a szerver felé, és ez válik majd az aktuális változattá a szerveren. Az összefésülést vagy a rendszernek sikerül önmagától megoldania, vagy ha a rendszer feloldhatatlan problémába ütközik, akkor a felhasználónak kell feloldania a konfliktust.

3.1.2 GIT

A GIT[5] egy elosztott, decentralizált rendszer. Ez azt jelenti, hogy ebben az esetben minden egyes felhasználó rendelkezik egy saját tárolóval, ami képes a központi tárolónak használt szervertől teljesen függetlenül működni. Ennek számos előnye van, például átláthatóbbá válhat a fejlesztés az által, hogy a kisebb változtatások csak a helyi tárolóba kerülnek, az ezekből álló komplexebb egységet alkotó változtatások pedig már

felkerülhetnek a központinak jelölt tárolóba. Ezáltal a központi változat mindig konzisztens állapotban lehet. Másik nagy előnye a biztonság, mivel itt minden felhasználó rendelkezik egy tárolóval, és azok mind rendelkeznek a fejlesztés történetével, így a központi elem kiesése esetén sem fordulhat elő adatvesztés, hisz a többi tároló ugyan úgy őrzi a fejlesztés állapotait. Ezek után a GIT néhány fontosabb műveletét mutatom be.

3.1.2.1 Commit

A helyi tárolóban megvizsgálja az állományok régi és új állapotait, majd ennek differenciájából születik meg a commit. A commithoz még egy szöveges leírás is tartozik, ez rendszerint egy pár szavas leírás ami commit által érintett változásokat összegezi. A művelet végrehajtása után a tárolóban a legújabb verzió lesz az aktív verzió. Fontos hogy a művelet elvégzésekor, csak azok a fájlok vesznek részt a procedúrában, amelyek ki lettek rá jelölve, más szóval indexelve lettek.

3.1.2.2 Push

Ez a művelet egy másik távoli repository állományait tudja felfrissíteni a mi saját lokális példányunkból. A távoli tárolót tipikusan a központinak használt tároló személyesíti meg. Ha a rendszer konfliktusba ütközik a művelet során, és nem tudja a két forrást egymásba olvasztani, akkor nekünk kézzel kell feloldani a konfliktusokat, majd commit és újra push műveletet végrehajtani.

3.1.2.3 Fetch

A távolinak megjelölt tárolóban található olyan változásokat töltene le a mi tárhelyünkre, amikkel mi nem rendelkezünk. Fontos, hogy ez esetben a lokális munkapéldányunkat ez nem változtatja meg, ahhoz hogy változtatások azon is megjelenjenek egy merge utasítás is szükséges még. Ez a parancs főleg akkor lehet hasznos, ha mások által végrehajtott változásokat felül szeretnénk vizsgálni és eldönteni, hogy mely változások jelenjenek meg a saját munka példányunkban.

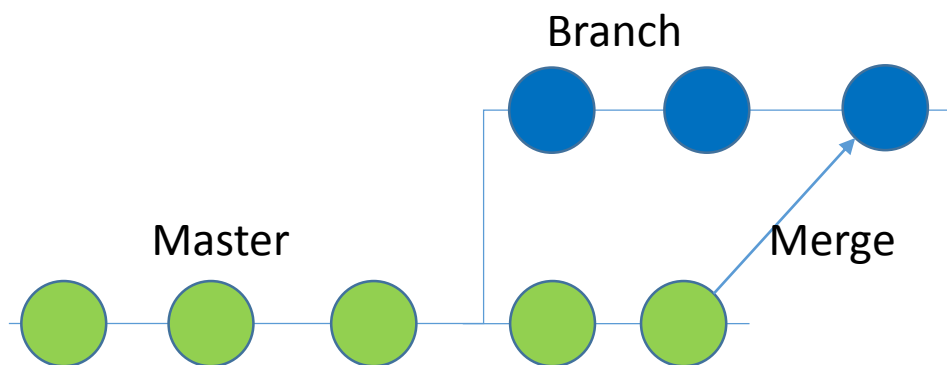
3.1.2.4 Pull

Lényegében ez a parancs a fetch majd a merge végrehajtását jelenti.

3.1.2.5 Merge

A fejlesztés során gyakran előfordul, hogy egyszerre akár több problémát is meg kell oldanunk párhuzamosan vagy több verziót is fenn kell tartani a fejlesztendő

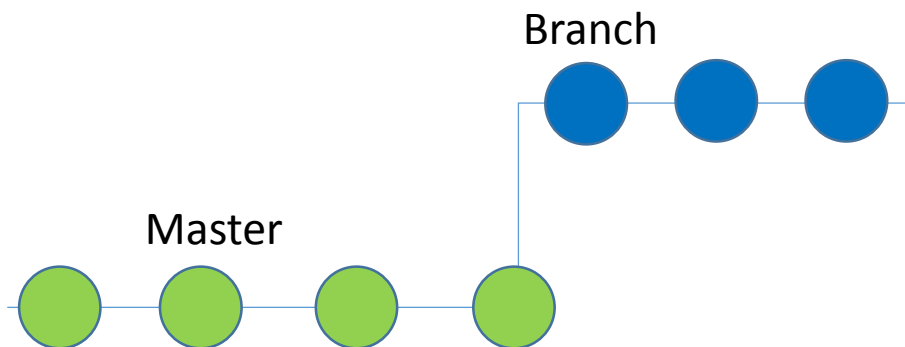
alkalmazásból. Ekkor a verziókezelő úgynevezett branch az az ág eszközéhez nyúlnak. Ez lehetőséget biztosít arra, hogy egy adott szakaszban több felé szétválasszanak egy kódbázist, majd lehetőség szerint később akár össze is fésüljék őket. Jó példa lehet erre, hogy ha a munka folyamán az alkalmazásból létezik egy teszt és egy stabil ág. Ez esetben a fejlesztések a teszt ágban történnek, majd ha ott jónak bizonyulnak, akkor ennek az ágnak a változásai bekerülhetnek a stabil ágba. Ezt a két ág közötti összefésült a merge utasítás végzi el, a 4. ábra szerint.



4. ábra Merge művelet

3.1.2.6 Rebase

Az előző utasításhoz hasonlóan ez is összefésülést végez, azonban egészen más megközelítésben. A különbség abból adódik, hogy míg a merge esetében a másik branch új változásai időrendben előrehaladva egy új commitként jelenik meg, addig a rebase esetén a mi águnk alapjának veszi a kiválasztott ágat (5. ábra). A jelenlegi branch commitjai ennek a „tetején” jelennek meg, tehát a rebase-re jelölt ág változásai időrendben hátrébb lesznek, a mi jelenlegi águnk commitjainál. Ennek számos előnye és hátrányai is lehet, egyik fő aspektusa a nyomon követhetőség. Rebase esetén lényegében feltétel nélkül elfogadjuk a másik ág változásait, míg ha merge utasítást választjuk, nagyobb szabadsággal élhetünk. Pont emiatt is a merge átláthatatlanabb hisz túl sok fölösleges járulékos napló bejegyzést jelenthet, ellenben a rebase ami nem zavarja meg ilyen szempontból a fejlesztés nyomon követhetőségét.



5. ábra Rebase művelet

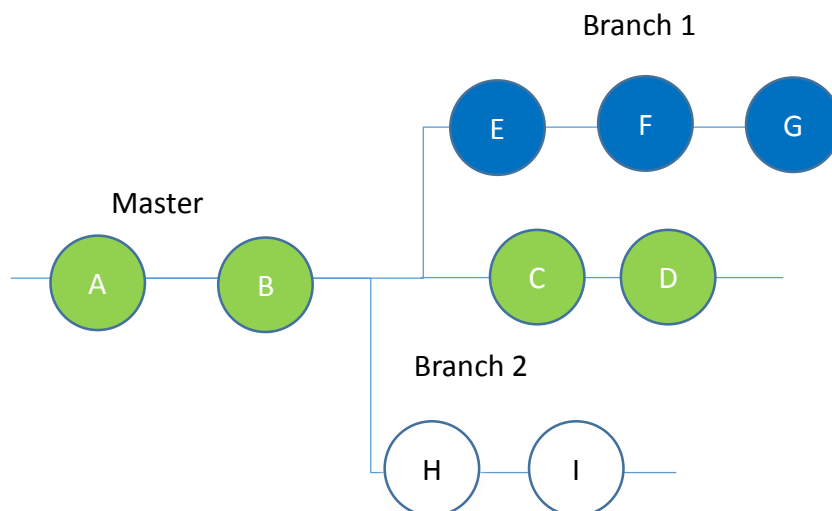
3.1.2.7 Reset

Ezzel az utasítással visszaállíthatatlanul vehetünk fel egy másik commitolt állapotot a tárolónkban. Ennek egy felhasználó barátiabb verziója a revert utasítás, ez esetben már visszatérhetünk az épp elhagyott pontra is később.

3.1.2.8 Cherry pick

Ebben az esetben lehetőségünk nyílik arra, hogy egy adott ágba egy másik ágból átmozgassunk commitokat.

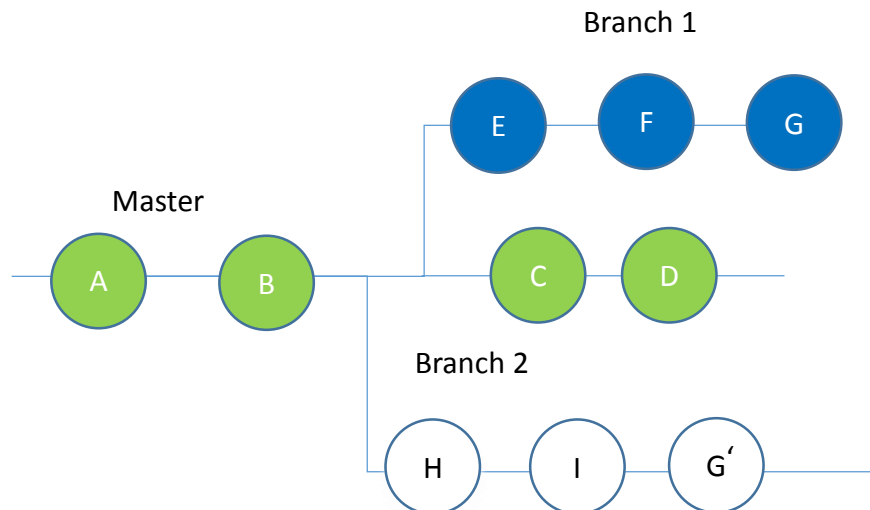
Vegyük a következő kezdeti állapotot:



6. ábra Cherry pick kezdeti állapot

Itt az 1-es ágban történt G jelzésű commit változásait át szeretnénk vinni a 2-es ágba is. Erre tud megoldást nyújtani a cherry pick művelet.

Ha végrehajtjuk a cherry pick műveletét, akkor a következő lesz az állapot:



7. ábra Cherry pick művelet végrehajtása után

Így az F és G commit közti változtatások átkerültek a 2-es ágba, G' néven.

Fontos megjegyezni, hogy a művelet csak akkor valósítható meg, ha rendelkezésre áll megfelelő közös ős.

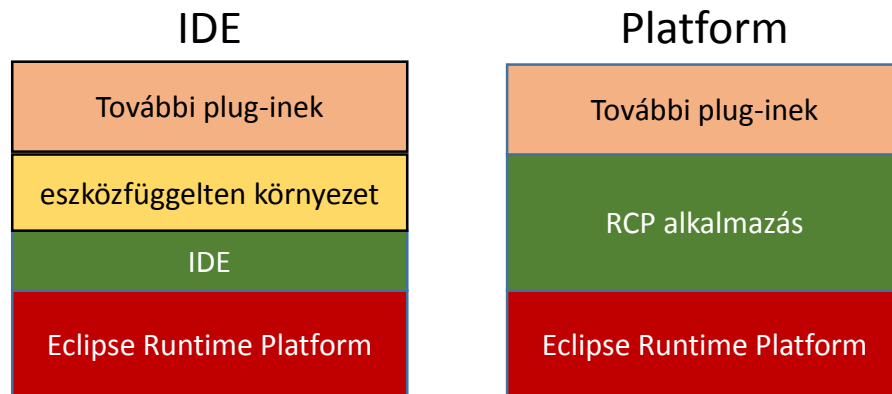
Eclipse

Az Eclipse[6] magát egy nyílt forráskódú közösségnek nevezi, ami magában foglal eszközöket, projekteket és az ezeket létrehozó fejlesztők csoportosulásait. Mindezt pedig a szintén Eclipse nevű alapítvány irányítja, hangolja össze. Fő projektjük az Eclipse Platformnak nevezett környezetük. Lényegében majdnem az összes Eclipse technológia ennek segítségével érhető el, illetve futatható.

Eclipse Platform

A Platform[7] számos keretrendszert és szolgáltatás definiál, amik segítségével már könnyen alkalmazásokat lehet készíteni. Ilyen eszközök például a fontosabbakat említve a nagy testreszabhatósággal rendelkező grafikus felület (workbench, SWT), projekt alapú erőforrás kezelés, erőforrások figyelése, univerzális nyelv független fordító és hibakereső eszköz, több felhasználós verziókezelő rendszerek integrálhatósága (Team API) illetve dinamikus modulkezelő rendszer (OSGi). Ezen szolgáltatások egy részéből áll össze a RCP az az Rich Client Platform, lényegében ez adja meg a lehetőségét arra, hogy vastag kliens alkalmazásokat készítsünk. Fontos megjegyezni, mivel a platform erős modularitással rendelkezik így az RCP használata nem zárja ki más Eclipse alapú

eszközök használatát sem. Többek közt az Eclipse IDE, a fejlesztői környezet is erre a platformra épül. A 8. ábra a fejlesztői környezet és a platformra épülő eszközök viszonyát mutatja be.



8. ábra Eclipse IDE és Platform felépítése

3.1.3 Eclipse PDE

A Plug-in Development Environment[8] röviden PDE az Eclipse egy olyan eszköze, ami lehetőséget biztosít Eclipse IDE beépülő modulok, szolgáltatások, modul frissítő oldalak és RCP alkalmazások elkészítéséhez. A PDE szerteágazó OSGi eszközöket is tartalmaz, így környezetet biztosíthat komolyabb komponens alapú programozásra is. Én mivel csak Eclipse beépülő modult más néven plug-int készítettem így csak az ezzel kapcsolatos funkciókat mutatnám be.

Ahhoz hogy egy Eclipse plug-int hozzassunk létre, ennek megfelelő típusú projektet kell létrehozni. A projektben automatikusa létrejön a plugins.xml nevű fájl. Ez a plug-in legfontosabb fájlja. Itt történik meg a plug-in tulajdonságainak meghatározása. Itt adhatjuk meg modul nevét, azonosítóját és egyéb alap leíróját. Lehetőségünk van, a modulunkhoz aktivátor osztályt is hozzárendelni, ekkor a kiválasztott osztálynak az AbstractUIPlugin nevű osztályból kell leszármaznia. Az itt felüldefiniált start és stop függvény akkor hívódik meg, ha modulunkat elindította a rendszer illetve leállította.

A plugins.xml fájljában kötelezően meg kell adni, hogy az adott plug-in melyik másik modulokat is használja még fel. Itt még lehetőségünk van csak osztály package megnevezése által megnevezni a kívánt komponenseket, ekkor a rendszer maga deríti fel, hogy melyik másik modulban található a hivatkozott package. A leíró fájl egy másik

általam használt funkciója az Extensions az az a kiterjesztések rész. Itt van lehetőségünk regisztrálni, hogy modulunk mivel szeretné kiegészíteni a rendszert. A kiegészítés tipikusan abban valósul meg, hogy az adott osztály megvalósítja a kiválasztott interfészt majd az Extensions résznél regisztrálásra kerül.

3.1.4 OSGi

Az OSGi[10] (Open Services Gateway initiative) egy olyan felsőbb szintű szolgáltatási réteget valósít meg a Java virtuális gépe fölött, ami lényegében operációs rendszer szintű folyamat kezelést tesz lehetővé. Két fontos aspektusa a rendszernek a szeparáció illetve az erőforrások dinamikus megosztása. A rendszer egységként az úgynevezett batyut (bundle) kezeli. Akár csak egy folyamatnak, ennek is vannak állapotai és életciklusa. Fontos tulajdonsága még egy ilyen modulnak, hogy a futtatásához követelményeket, más komponenseket határozhat meg, illetve ő is kiajánlhat szolgáltatásokat más modulok felé. Maga az Eclipse platform is OSGi szolgáltatást nyújt, épp ezért a könnyű átjárhatóság végett, az Eclipse-en belül egy plug-in az OSGi rendszerén belül egy bundle-nek feleltethető meg.

3.1.5 EMF

Az eszköz bemutatása előtt fontos tisztázni két fogalmat, ezek a metamodel és a példánymodell. A metamodel a modellek strukturális felépítését, lehetséges tulajdonságainak keretet adó modell, míg a példánymodell ezen leírásnak eleget tevő adatokat tartalmazó modell.

Az EMF[12] hosszúnevén Eclipse Modeling Framework egy modellezésre használt eszköz. Lehetőséget nyújt modellek magas fokú szerkesztésére és modellekből többféle célra felhasználható kód generálására. Ilyen generált eszköz lehet például a modell alkalmazásban egyből felhasználható kódja, különböző adat módosítást segítő eszközök, illetve grafikus szerkesztői felületek.

Az EMF segítségével létrehozott modellek neve Ecore modellek. Ezek valójában metamodellek amiből a rendszer később képes szerkesztőt, kódot illetve egyéb eszközöket generálni. Később e generált eszközök segítségével készíthetjük el a kívánt példány modellünket.

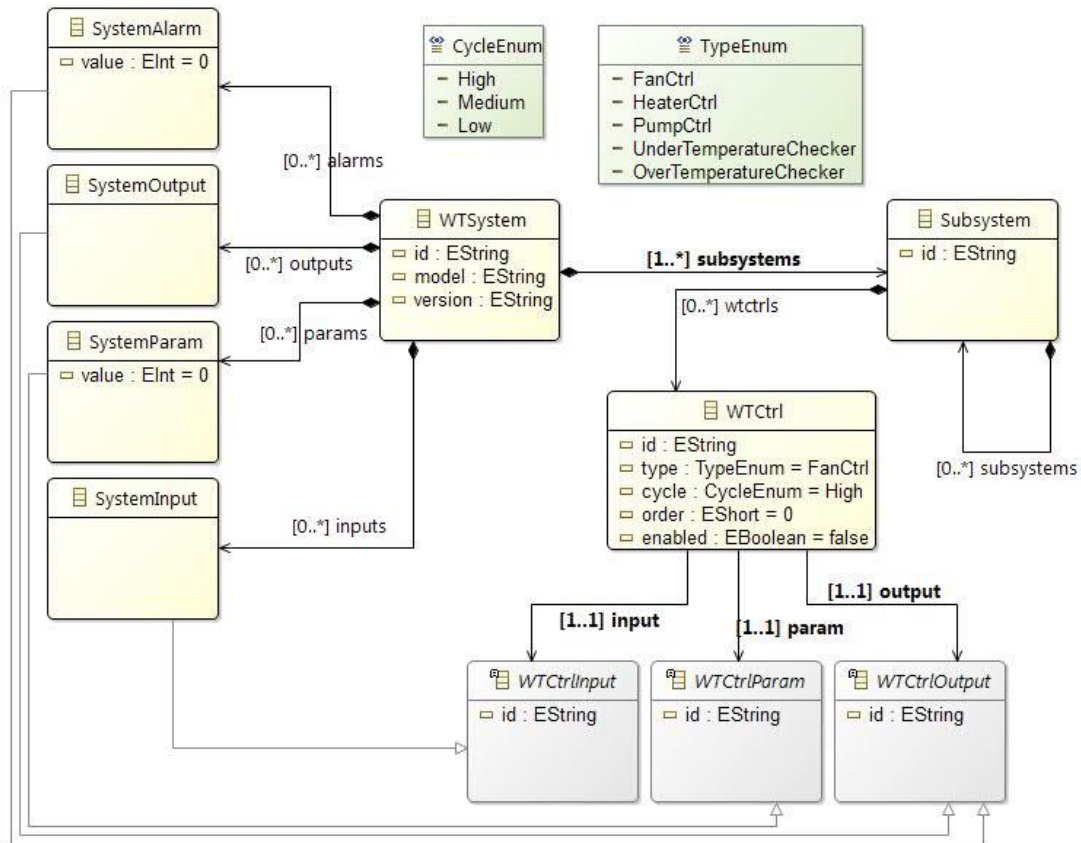
Az Ecore modellekben a következő alap típusokat használhatjuk fel:

- EObject: Az összes EMF által generált elem e típus leszármazottja. Tulajdonképpen a Javában lévő `java.lang.Object` megfelelőjének tekinthető az EMF rendszerén belül.
- EPackage: Ez fogható fel a modellek legfelsőbb szintű tárolójának, a modellek összes eleme egy ilyen típusú példányhoz tartozik.
- EClass: Az osztályok modellezésére használja a rendszer. A modellből generált kódban, valójában egy Java interfész és azt megvalósító osztály testesíti meg.
- EAttribute: Objektumuk névvel és típussal ellátott elemeit modellezi.
- EReference: Névvvel és multiplicitással ellátott hivatkozás osztályok között. Lehetőség szerint asszociációt és aggregáció is jelképezhet.

Az Ecore modellekből képzett példánymodellek fa struktúrába szervezett fájlként tárolódnak el. Ezen fájlokat az EMF rendszerén belől Resource típusú besorolást kapnak, ezek tipikusan egy gyűjtő objektumban helyezkednek el, amit ResourceSet-nek nevezünk. A rendszer a modellek azonosítására az úgynevezett URI (Uniform Resource Identifier) technikát használja fel.

A rendszer lehetőséget biztosít arra is, hogy a modelleken történő változásokat nyomon kövessük. Az eljárás az EMF-en belül adapterek segítségével történik, az objektumok változásai alap szinten megfigyelhetőek. Ilyen események (Notification) például az elem törlés, új elem megjelenése, módosítás.

Az esettanulmányban bemutatott modell Ecore diagramja a következő:



9. ábra Motivációs példa, szélturbina ecore metamodel

A metamodel gyökéreleme WTSYSTEM (Wind Turbine) típusú. Ebben találhatóak meg az alrendszerek (Subsystem) illetve az az egyéb rendszer által használt eszközök. Az ábrán nem szereplő egyéb ilyen eszközök:

- Fault (Hiba)
- Timer (Időzítő)
- Variable (Változó)
- Error reaction (Hiba válasz)

A rendszer közvetlen eszközeivel, a WTSYSTEM fő objektum tartalmazási viszonyban áll. Az alrendszereket (Subsystem) szintén tartalmazza a gyökérelem. Az ezekben található vezérlők ezért már csak pusztán hivatkozás útján érik el a nekik kellő rendszer objektumokat.

A modell összes elemén található egy SysID leíró, ennek segítségével egyedi módon azonosíthatóak. Az objektumok további értékei tipikusan előre meghatározott értékkészletből, enumerációkból származnak, pl. az ábrán is látható CycleEnum.

3.1.6 IncQuery

Az EMF modellekkel való munka során nagy problémát jelenthet a rajtuk való hatékony keresés megvalósítása. Ehhez nyújt segítséget az IncQuery[13] nevű eszköz, ami az előbb bemutatott, modellfigyelő adaptert valósít meg. Egyik fő sajátossága hogy a kéréseinket deklaratív nyelven kell megfogalmazni majd rendszer a kérésekhez megfelelő imperatív kódot generál és futtat. A keresés végrehajtáshoz szükséges kód generálódhat már fordítási időben, de lehetőség van futás időben is beolvasni a deklaratív mintát majd egyből felhasználni azt, így felhasználási terület független megoldások is hatékonyan megvalósíthatóak.

Az eszköz működése során, mint azt a neve is sugallja, inkrementális technikát használ a minták illeszkedésének a megtalálásra. Ezt a Rete algoritmus felhasználásnak segítségével éri el, ami jól ismert a szabály alapú rendszerek területén. A rendszer bemenetként megkapja a modellt, illetve az EMF rendszertől jövő notificationöket amik az adott modellre vonatkoznak. Ezek és a deklaratív minta alapján egy Rete szabály kiértékelő hálózat jön létre, ami feldolgozza az adott elemeket, majd megalkotja az eredményt. Ezen eredményeket egy automatikusan generált típushelyes hozzáférési réteg követ, ami az alkalmazások felé jelent kimenetet. A Rete hálózat nem szűnik meg addig, amíg az eredményére szükség van. Így a modell változása esetén a rendszer ugyanúgy megkapja az értesítéseket, és a kimeneti eredményhalmazt tovább fejleszti. Innen ered a megoldás inkrementális mivolta.

A motivációs példában bemutatott záruk mintái a következőképp néznek ki IncQuery-ben:

```
import HTTP://WTSpec/2.0

pattern ctrUnit29(C, SysID, CFault, CFaultSysID)
{
    CtrlUnit29.sysId(C, SysID);
    CtrlUnit29.Fault__fFault(C, CFault);
    SystemFault.sysId(CFault, CFaultSysID);
}
```

```

pattern subsystem2 (Sys, SysID, SysDesc, WTC, Cycle, Priority) {
    Subsystem.sysId (Sys, SysID);
    Subsystem.description (Sys, SysDesc);
    Subsystem.itsWTCs (Sys, WTC);

    CtrlUnit44.cycle (WTC, Cycle);
    CtrlUnit44.priority (WTC, Priority);

    CtrlUnit44.Input__iInput (WTC, Var);
    SystemVariable.sysId (Var, VarSysID);
    check (VarSysID.equals ("VA_62"));
}

```

Elsőre is minden fájl elején szükségünk van importálni azon metamodellek nevét, amiket használni szeretnénk. Később e modellek elemeire tudunk majd hivatkozni.

A minta deklarálása a pattern kulcsszóval kezdődik, majd a neve és a paraméter lista következik. A függvény törzsében egy mintát alakíthatunk ki az importált modell típusai és tulajdonságai alapján. Itt több illeszkedést is meghatározhatunk, ha ugyan az a változó több helyen is megjelenik, akkor nyilván szűkül annak a változónak az eredményhalmaza. A pattern eredményként csak a paraméterlistán megjelölt változók eredményhalmazával tér vissza. Ahol szükségünk van egy változóra azonban értéke számunkra lényegtelen, azt az '_' karakterrel kell kezdeni. A mintánál a megfogalmazása során lehetőség van még már meglévő mintát újra felhasználni, logikai műveleteket alkalmazni, illetve tranzitív pattern is deklarálható.

3.1.7 EVM

Az EVM[14] (Event-driven Virtual Machine) az IncQuery egy olyan kiegészítése, ami lehetővé teszi, hogy modellen könnyen folyamatos vizsgálatot végezhessünk. A rendszer ezt szabályok segítségével éri el.

A rendszer működtetéséhez a következő elemek szükségesek:

- Egy ütemező, ami a megfelelő IncQuery engine objektumot kap meg.
- Végrehajtási séma, ami az ütemezőnek szab meg feladatokat.
- Szabályspezifikációk, amik a végrehajtási sémának adnak meg célokat. Egy szabályleírás magában foglalja a megfigyelni kívánt objektumok mintázatát (IncQuery minta), és a megfigyelt objektumok különböző típusú változásaihoz társított feladatokat (Job).

Az eszköz használatával tehát az IncQuery funkcionalitása nagyban nő, hisz hatékony megoldás nyújt a rendszer valós idejű nyomon követésére. Így jó alapot

biztosíthat valós idejű modell validációra, szinkronizációra vagy épp kollaboráció megvalósítására.

3.1.8 Egyéb Eclipse által biztosított eszközök

3.1.8.1 Beépülő modul automatikus indítása

Az Eclipse lehetőséget biztosít arra, hogy a rendszer elindítása után, bizonyos modulok mindenképp elinduljanak. Ehhez a modulon belül az egyik osztálynak meg kell valósítani az `IStartup[9]` nevű interfészt és implementálni kell az `earlyStartup` függvényt. Ez a metódus a rendszer betöltődése után hívódik meg, tehát a rendszer alap szolgáltatásai, komponensei már biztosan elérhetőek. Így a plug-in kikerülheti a rendszer alapértelmezett komponens betöltődési szabályát a lazy loadingot, ami csak igény eseti betöltődést jelentene.

3.1.8.2 Erőforrás jelölés

A rendszer használata során tipikusan előforduló eset, hogy bizonyos erőforrásokhoz bizonyos információt kell csatolni, például szintaktikai hiba a fájlban, erőforrás nem elérhető stb. A rendszerben minden egyes erőforrás az `IResource[9]` interfészt valósítja meg, ennek pedig a `createMarker` nevű függvényével társíthatunk jelzést az adott elemhez. A társított információk szinte bárhol megjelenhetnek, a projekt nézetben, az elem szerkesztőjében stb., de ezek tipikusan szintén megjelennek a rendszerszintű Problems az az a problémák nevezetű nézetben.

3.1.8.3 Feladatok ütemezett végrehajtása

Az Eclipse régebbi verzióinál problémásak voltak a rendszert kiegészítő egyéb műveletek végrehajtásai. Tipikus problémaként jelent meg, hogy az egyes műveleteket nehézkesen lehetett ütemezni, kölcsönös kizárás megvalósítása problémás volt, mi több gyakran előfordult az is, hogy a folyamat lefutása közben a rendszer nem reagált egy ideig, hisz a folyamat blokkolta azt. Ezek kiküszöbölésére hozták létre a `Job[9]` osztályt. Használata emlékeztethet a `Thread` osztályra, azonban jóval többet tud nála. Akár csak a `Thread` delegált `Runnable` interfészénél, itt is egy `run` metódust kell megvalósítani, azonban itt egy `IProgressMonitor` típusú objektumot is kap a metódus. A `ProgressMonitor` segítségével a felhasználó számára jelezhetünk, hogy a végrehajtás épp milyen fázisban van, illetve kezelhetjük segítségével, ha épp a felhasználó megszakította a feladatot. Egy `job` fő tulajdonságai közé tartozik, hogy mennyire kritikus a lefutása (`priority`),

rendszerszintű feladat-e, tehát szükség van-e arra hogy a felhasználót értesítsük a végrehajtásról, illetve hogy milyen más feladatokkal futtatható együtt, melyekkel zárják ki egymást (SchedulingRules). A job objektumok végrehajtását komplexebben a statikus Platform osztálytól elkért IJobManager interfészt megvalósító objektummal lehet. Ilyen műveletek lehetnek például a feladatok újra ütemezése, futás időben történő feladatokhoz tartozó dinamikus erőforrás zárolás vagy épp feladatok végrehajtása közben megjelenő események figyelése.

3.1.8.4 Feladatok egyszeri végrehajtása

Akár csak a Job esetében az IHandler[9] interfészt megvalósító osztályok is tipikusan egy komplexebb műveletet valósítanak. Azonban míg az előző esetében a hangsúly tipikusan a többszörös, ütemezett lefutáson van, itt azonban inkább egyszeri lefutásról beszélhetünk. Ahhoz hogy egy osztály megvalósíthassa ezt a tulajdonságot az IHandler interfészt kell megvalósítania. Ebben több függvény közül a legfontosabb az execute nevű függvény, ez hívódik meg végrehajtáskor. Az interfészt megvalósító osztályt a rendszertől elkért, IHandlerService típusú objektum segítségével lehet futtatni. Ez mellett azonban lehetőség van implicit módon is meghívni az adott osztályt. A platformon belül jó példa erre a menüpontok vagy billentyűzet események kötése ilyen utasításokhoz. Ekkor a rendszer az adott trigger eseményre, például egy menüpont kiválasztása, meghívja a hozzárendelt utasítást.

3.1.8.5 Beállítások kezelése

Mint minden komplexebb platform esetén, itt is szükség volt egy központosított beállítások kezelőre. Ennek egyik legnagyobb előnye, hogy így a rendszer az összes beállítást egy központi helyen tudja kezelni, mind fizikailag és mind grafikus felületileg. Ahhoz hogy egy plug-in képes legyen a rendszer ilyen szolgáltatását használni a plugins.xml-ben egy osztályt kell beregisztrálni, aminek az ősoosztálya AbstractPreferenceInitializer[9] kell, hogy legyen. Itt két függvényt kell felüldefiniálnunk. Az initializeDefaultPreferences nevűt a rendszer akkor hívja meg, ha elsőre szeretne hozzáférni az itt kezelt értékekhez, vagy alapbeállításhoz szeretné a rendszer helyezni az értékeket. A másik függvény, a setProperties pedig akkor hívódik meg, ha a rendszer az értékeket szeretné beállítani. Egy fontos pontja lehet az ilyen beállítás kezelőknek, hogy szinten mindig tartalmazzanak egy figyelőt, ami kiértékesíti bizonyos általunk meghatározott érték változások esetén. Ez azért hasznos mert így kényelmesen,

futás időben lehetőségünk nyílik megváltoztatni a modulunk működését, funkcionalitását.

Ahhoz hogy a beállítások megtudjanak jelenni az Eclipse beállítások menüpontjában, szükség van arra, hogy a `plugins.xml` fájlba beregisztráljunk egy `IWorkbenchPreferencePage` interfészt megvalósító osztályt. Itt, mint az Eclipse többi grafikus felületénél az SWT könyvtár segítségével építhetünk felületet a beállításainknak.

3.1.8.6 Projekt leíró használata

Ahhoz hogy az Eclipse rendszerén belül komolyabb projektkezelést valósítsunk meg, tipikusan információt kell hozzárendelnünk a projektekhez. Egy lehetőség lehet, hogy az adott információkat a `project`-en belül fájlokban tároljuk, azonban így nem jól különülnek el a projekt metaadatai és a valós erőforrások, amiken dolgozni szeretnénk. Erre adhat megoldást a `Project Nature`[9] nevű szolgáltatása a rendszernek. Saját projekt környezet definiálása után lehetőségünk nyílik a projektet megjelölni, majd különféle információk társítani a projekthez. E szolgáltatás használata igen gyakori, hisz a rendszer által létrehozott projektről is ennek segítségével derül ki, hogy például Java vagy `c++` projektek-e. Annak ellenére, hogy a rendszeren belülről egy jól elkülönített adatokról beszélünk, valójában a projektek ilyen jellegű leírója minden egyes projekt gyökér könyvtárában található meg, a „`project`” fájlban.

3.1.8.7 Nézet és Perspektíva

Ha megoldásunkat szeretnénk grafikus megjelenítéssel is kiegészíteni, akkor ehhez egy `ViewPart` osztályból leszármazó osztályt kell megvalósítanunk. A plug-in leírójába való beregisztrálás után az SWT eszközei segítségével felépíthetjük a grafikus felületünket, majd az meg is jelenik az Eclipse „`Show View`” ablakában kiválasztható opcióként. Azonban ha egy feladat elvégzése során tipikusan több nézetre van szükségünk, azokat könnyen egy úgynevezett perspektívába gyűjthetjük össze. A perspektívát maga a felhasználó is összeállíthatja vagy fejlesztési időben is lehetőség van összeállítani. Ehhez az `IPerspectiveFactory`[9] megvalósítása majd a modul leíróba való regisztráció szükséges.

3.1.8.8 Grafikus felület és adatok összekötése

A `JFace`[9] az Eclipse Platform SWT grafikus könyvtárat kiegészítő MVC az az modell, nézet vezérlő megvalósítása. Lényegében ez lehetőséget biztosít arra, hogy az

adott vezérlőhöz, adatokat kössük és bármelyik megváltozása esetén a másik fél követi ezt a módosítást. Többek között ilyen grafikus elem JFace keretein belül a fanézet, lista, táblázat is. Az eszközkönyvtár az SWT-ben lévő elemekre épül rá, a JFace segítségével megvalósított komponenseket az SWT által nyújtott interfészekon keresztül is elérhetőek, így nagy rugalmasságot biztosítva a két API közt.

Jersey

A Jersey[15] egy olyan server és kliens oldali komponenst nyújt, ami REST szolgáltatások használatát teszi lehetővé. Kompatibilis implementációt nyújt a JAX-RS szabvánnyal, így más, a szabványt megvalósító komponensekkel is szabadon használható.

3.1.9 REST

Hálózati kommunikáció terén az egyik legelterjedtebb architektúra a REST[16]. A felépítés lényege hogy a kliensek kéréseket küldenek a szerver felé, majd az feldolgozza őket és válasz küld vissza. Tipikusan HTTP protokollt használnak a REST megvalósítására hisz az általa használt címzési struktúra, az URL nagy kifejező készséget biztosít a kérések megvalósítására. A kliens felé továbbítandó válaszok formátumára nincs megkötés, azonban a legtöbbször a hatékonyság miatt valamilyen kompakt formátumot használnak például XML vagy JSON.

Az architektúra fő tulajdonságai:

- Kliens – szerver felépítés: A kliens és a szerver jól elkülöníthetők egymástól, azok kommunikációja egy egységes interfész segítségével valósul meg.
- Állapotmentesség: A szerver nem tárolja el az egyes kliensek állapotait, nem veszi figyelembe azt, hogy régebben a klienssel mit kommunikált. Épp ezért a kliensnek mindig pontosan specifikálni kell az ő állapotát a kérése mellett, hogy megfelelő választ kaphasson.
- Gyorsítótárazhatóság: A rendszer lehetőséget biztosíthat arra, hogy az egyes lekérések eredménye eltárolódjon majd újbóli lekérés esetén a régebbi eredmény kerüljön feldolgozásra. Ez nagyban csökkentheti a válaszidőt, azonban bizonyos esetekben elavult adatokkal való munkához

vezethet, így fontos specifikálni, hogy mely adatok kerülhetnek gyorsítótárazás alá és melyek nem.

- Erőforrások kezelése: Minden egyes erőforrásnak egyértelmű, egyedi azonosítója és elérése van. A kliensnek lehetősége van ezen erőforrásokat lekérdezni, illetve azokon műveleteket végezni. A kliens kérései mindig önleírók, egyértelműen határozzák meg a szándékot.

A REST műveletei a kliens oldaláról:

- Get: Egy erőforrás vagy azzal kapcsolatos adatok lekérése.
- Put: Egy elem tulajdonságainak megváltoztatására.
- Post: Új példány felvétele a rendszerbe.
- Delete: Elem törlésére szolgáló művelet.

A műveletek paraméterei HTTP esetén az elérni kívánt cím végén jelennek meg. Azonban e protokoll használata esetén a művelet megnevezése a kérés fejében, nem pedig a címében szerepel.

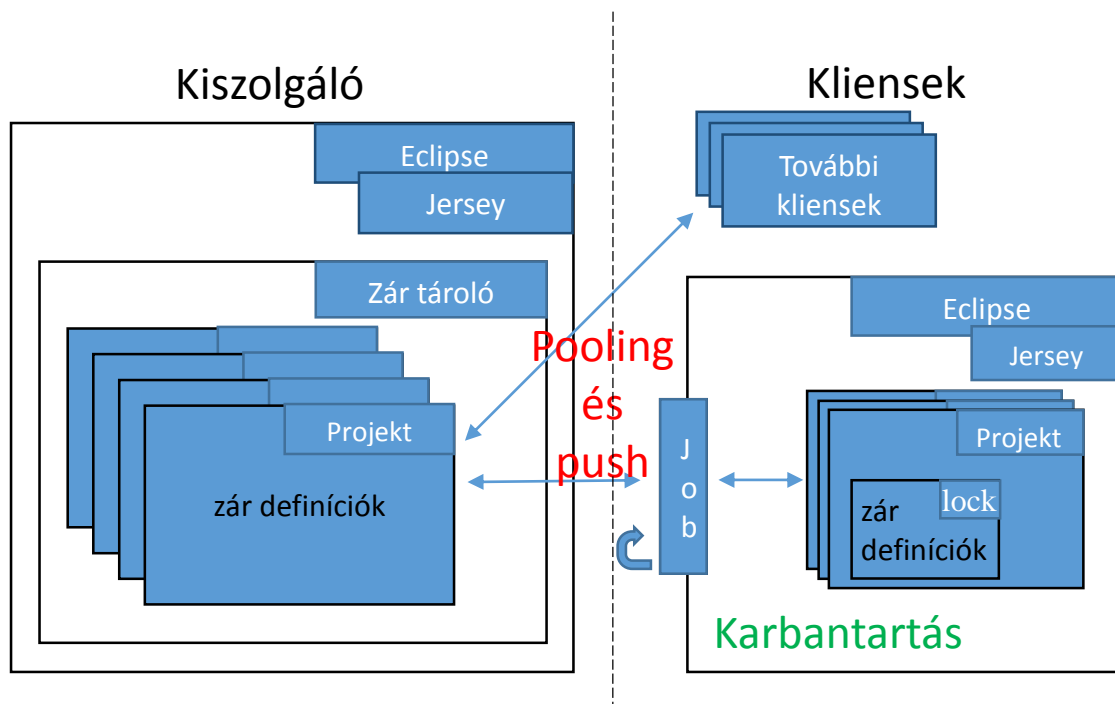
3.1.10 JSON

Teljes nevén JavaScript Object Nation[17] mint a nevéből is mutatkozik egy JavaScriptből átvett leíró szerkezet. Fő erőssége, hogy sikeresen egyesít olyan tulajdonságokat, mint az ember számára is könnyű olvashatóság, kompakt méret és magas ábrázoló készség. Az objektumok mellett még két struktúra ábrázolása is megoldott a formátumban. Ezek a lista és a párosított kulcs-adat halmazok. Így lényegében alkalmas bármilyen programozási nyelvvel való használatra, hisz ezen adatszerkezetei segítségével nagy lefedést biztosít bármely nyelv elemeire. A formátumot főleg a web világában használják, adatok küldésére és fogadására viszonylag kis méretigénye és gyors feldolgozhatósága miatt.

4 Áttekintés

Munkám során több technológiát és módszert is alkalmaztam, így most itt ezek kulcs pontjait mutatom be, majd a következő fejezetben részletesen is áttekintem megoldásom menetét.

Kliens – Szerver felépítése



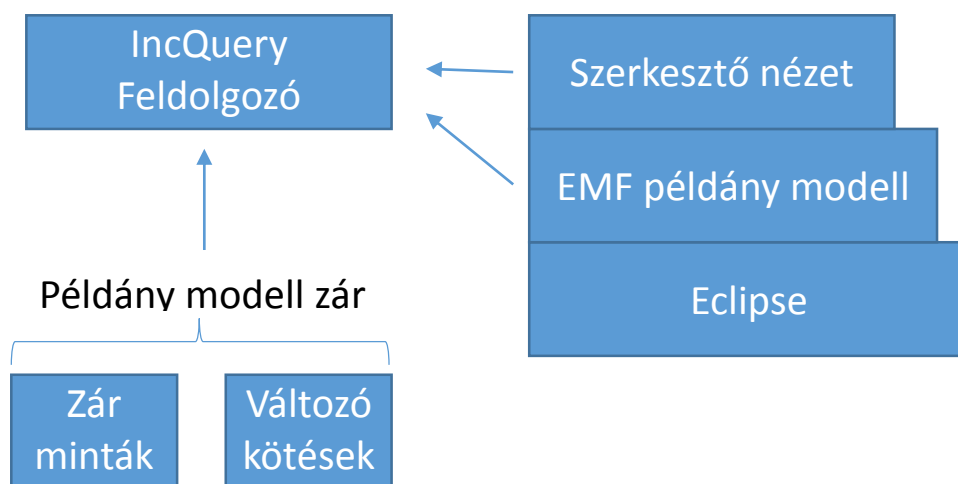
10. ábra Kliens - Szerver felépítése

Munkám lényegében egy verziókezelő rendszer kiegészítése, így adott volt, hogy kliens – szerver (10. ábra) architektúrát kell alkalmaznom a feladat megoldására. A szerver megvalósításom tehát egy kiegészítésének tekinthető a verziókezelő rendszernek, logikusan ugyanazon a szerveren futnak. A munka folyamán csak is a zárokat alkotó fájlokkal dolgozom e végett fontos, hogy kliens oldalon a zárok fájljait kivegyük a más rendszer hatásköréből, és csak az én megoldásom kezelje ezeket a konfliktusok elkerülése végett. Szerver oldalon a zárok külön könyvtárban, egyedi projekt azonosító alapján vannak tárolva, kliens oldalon pedig az egyes projektek „lock” nevű könyvtárában helyezkednek el.

Mindkét oldalon adott volt Eclipse futtatókörnyezet az előzetes igények alapján, így ehhez kellett találnom egy kommunikációt megvalósító technológiát. Ehhez a Jersey-

t használtam fel. Bár REST tulajdonságai alapján nem feltétlen a legjobb választás, de az Eclipse OSGi rendszerébe való könnyű integrálhatósága végett mellette döntöttem. A technológiából fakadóan, hogy folyamatos adatcsere történhessen, poolingot, az az folyamatos adat lekérdezést alkalmaztam. Ezt kliens oldalon a rendszer által nyújtott Job eszközzel valósítottam meg. Az általam definiált job objektum feladata az, hogy időközönként ellenőrizze le az adott projektben lévő zárat és hasonlítása össze őket a szerveren találhatóval, ha a szerveren talál frissebb zárat, akkor töltsen le a felhasználónak. Ez az ellenőrzés tipikusan 100-5000 milliszekundumos időközönként valósul meg. Ha helyben új zárunk van, azt csak kézzel tudjuk a többi felhasználó részére eljuttatni, ehhez publikálnunk (push) kell az adott projekt zárait a szerver felé, majd a többi kliens letölti azt.

Kliens működése



11. ábra Kliens működése

Ahhoz hogy modelleket zárolni tudjunk, a modell mellett szükségünk van a zárat alkotó fájlokra is. Ezek a projekt „lock” nevű könyvtárában helyezkednek el, a modellekkel megegyező néven, de más kiterjesztéssel. Az egyik a modellhez tartozó mintákat tárolja, a másik az e minták paramétereire kötött változókat. Ezen két fájl együttese adja ki a zárat illetve záratokat, bármely hiánya nélkül a zárolás nem lehetséges. A modellfájl bárhol lehet a projekten belül, egyedül a neve az, ami egyedivé teszi, és a zárákhoz köti azt.

Ahhoz hogy a zárolási funkciót el tudjuk érni, meg kell nyitnunk egy modellt egy EMF szerkesztőben, majd a rendszerem ezt érzékeli, és a felderíti a megfelelő lock állományokat. Mivel az EMF szerkesztőknek van egy olyan tulajdonságuk, hogy képesek

mások számára kiszolgáltatni az általuk használt modellt, így minden adott lesz ahhoz, hogy a zárainkat alkalmazhassuk. A kliens felépítését a 11. ábra mutatja. A szerkesztő érzékelése esetén a saját zárolási nézet betöltődik, majd zárok adhatóak meg. Ahhoz hogy a projekt szinkronizálhatóvá váljon, szükségünk van arra, hogy a projekthez megfelelő leíró, project nature-t is hozzáadjuk, ez elmulasztása esetén, a szerver felé történő utasítások nem lehetségesek.

A már zárolt modell esetén a megállapítása annak, hogy az adott felhasználói művelet sérti-e egy lock alatt lévő objektumot, az a következő képen történik (12. ábra):

1. a felhasználó a szerkesztőben végrehajtja a műveletet
2. a módosítás megjelenik a szerkesztő belső modelljében (ResourceSet)
3. az EVM ezt a módosítást érzékeli, és a változáshoz előre definiált trigger (Job) lefut
4. vizsgálat történik arra nézve, hogy az adott változás érint-e zár alatt lévő objektumot
5. a szerkesztő utasítás végrehajtási kezelője értesül arról, hogy a művelet jóváhagyható-e, vagy visszavonandó



12. ábra Zár felderítésének folyamata

5 Megvalósítás

A fejezetben az általam elkészített eszköz megvalósításának mikéntjeit fogom bemutatni. Első részben a szerver megvalósítását fogom bemutatni, majd a kliensnél alkalmazott megoldásokat.

Server

A szerver egy OSGi keretrendszeren belüli bundle-ként került megvalósításra, így a kiszolgáló könnyen karbantartható illetve igény esetén később egyéb dinamikus szolgáltatásokkal is kiegészíthető majd. Mivel a szerver a Jersey segítségével lett megvalósítva, elindulása után készen várja a feldolgozandó HTTP üzeneteket. A kiszolgáló 3 féle utasításra tud reagálni, ezek pedig:

- projecthez zár feltöltés
- adott projekthez tartozó záruk lekérdezése
- adott projekt egy zárjának letöltése

A továbbiakban e funkciókat mutatom be részletesen.

5.1.1 Projekthez zárfeltöltés

Ez a funkció a szerver gyökerének címéből nézve „/upload” címén érhető el, mivel itt a művelet a szerver felé fájlátvitel, ezért post típusú üzenetnek határoztam meg. A művelet 4 paramétert vár:

- a projekt egyedi azonosítóját, amit a kliens határoz meg
- a feltöltendő fájl nevét
- a fájl utolsó módosítási dátuma
- a fájl reprezentáló InputStream típusú objektum

Végrehajtás során a szerver a tárolónak jelölt könyvtárban létrehoz egy könyvtárat a projekt azonosítójával, majd elhelyezi oda a bejövő fájlt a paraméterben megadott névvel. Ha a fájl már régebben létezett, akkor a rendszer felülírja azt.

5.1.2 Adott projekthez tartozó záruk lekérdezése

A funkció a „/projectFiles” címen érhető el és mivel ez egy kliens felé küldött üzenet így get típusú. A választ a kiszolgáló JSON formátumban adja meg, tartalma pedig a paraméterben megadott azonosítóhoz tartozó projekt fájlnevei és a fájlok utolsó módosítási dátumai.

5.1.3 Projekt egy zárjának letöltése

A kliensek a „/download” címen érhetik el ezt a get típusú funkciót. Paraméterként a projekt azonosítót és az azon belül kívánt fájl nevét várja. Feldolgozás után válaszként a paraméterben megjelölt elem File objektumként kerül vissza a kérő felé.

5.1.4 Funkciók összefoglalás

Összefoglalva tehát, a szerver által nyújtott szolgáltatások a következő tulajdonságokkal rendelkeznek:

Funkció neve	REST típus	Elérési útja	Paraméterei
zárfeltöltés	Post	/upload	projekt azonosító, fájlnev, fájl módosítási dátuma és maga a fájl
záruk lekérdezése	Get	/project	projekt azonosító
zárletöltés	Get	/download	projekt azonosító, fájlnev

13. ábra Szerver funkciók

Kliens

5.1.5 Projekt leíró

Ahhoz hogy megfelelően tudjuk használni a modult, a kívánt projektekhez hozzá kell adni a plug-in által biztosított projekt leíró. Ehhez a projekt almenüjében a Team>Locks> Add EMFGit nature opciót kell kiválasztani. Az opció kiválasztása után projektleíró fájlban megjelenik a mi bejegyzésünk, illetve a projekt gyökerében létrejön egy fájl „.emfgit” névvel, ez a projekt egyedi azonosítóját tartalmazza.

Ahhoz hogy a menüpontok megfelelő helyen jelenjenek meg a plugins.xml fájlban, az „org.eclipse.ui.menu” típusú objektumot kellett deklarálni. Elsőre egy menuContribution típusút, aminek a locationURI paraméterében a következőnek kell

szerepelnie „popup:org.eclipse.ui.popup.any?after=additions”. Majd ennek egy menu típusú gyermeket kellett megadni „Team” felirattal és „team.main” azonosítóval. Így az ez alá beszúrt menük mindig a Team API opcióinak megfelelő helyen jelennek meg az előbb említett API használata nélkül.

Miután a projekt leírójába belekerült a mi bejegyzésünk, ugyan már a projekt jelölve van számunkra, azonban mivel kollaboratív környezetben vagyunk így indokolt lenne az adott projekt egyediségét is biztosítani. Mivel a nature csak jelölő információkat engedélyez, ezért egy fájlt kellett létrehoznom, ami majd tárolja az egyedi kulcsot. A fájl neve azért kezdődik ’.’ karakterrel, mert így a rendszer automatikusan elrejti az Eclipse rendszerén belül, így nem zavarja a felhasználót. A fájl tartalma pedig egy szimpla soros szöveg, amit a Java UUID eszközével generáltam. Ez az eszköz 128 bites véletlen azonosítót képes generálni, így alkalmas projektek egyediségének a biztosítására.

Mivel az előbbi biztosítja a projekt egyediségét ahhoz, hogy egy csoport ugyan azon projekten tudjon dolgozni, az adott helyeken a megfelelő UUID-nek kell szerepelnie. Kollaboratív környezetben ideálisan ezt úgy tudják legkönnyebben megvalósítani, hogy valaki létrehozza a kívánt tartalmat, majd azt feltölti egy verziókezelő rendszerbe, a többiek pedig letöltik azt.

5.1.6 Zárak definiálás

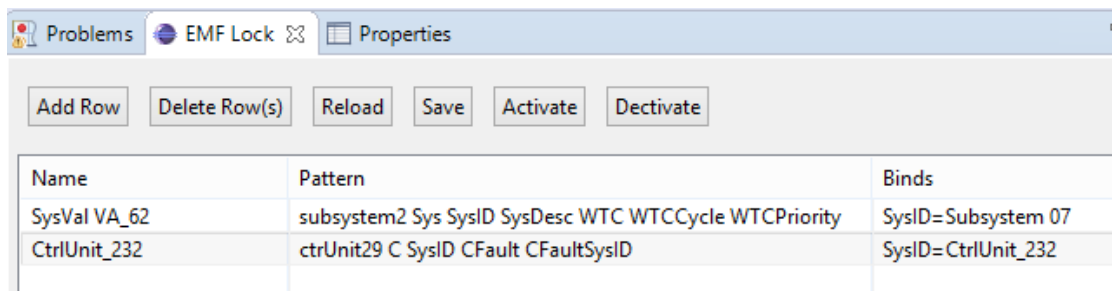
A zárak használatához meg kell nyitni egy EMF szerkesztőt, ekkor a rendszer ezt érzékeli, majd feltölti az EMF Lock nevű nézet tartalmát. Ezt legkönnyebben az EMFGit nevű perspektívából tudjuk elérni. A nézet mindig a legutolsó aktív szerkesztő kontextusához tartozik.

A nézet (14. ábra) a következő funkciókkal bír:

- Add Row: új, üres sort ad hozzá a listához
- Delete Row(s): az épp kijelölt sorokat távolítja el
- Save: az aktív modellnek megfelelő nevű „binds” fájlba kisorosítja a zárat
- Activate: aktiválja a listában lévő zárat a modellen
- Deactivate: eltávolítja a zárat a modelltől

A táblázat oszlopai:

- Name: a definiált zár neve, a rendszer ezzel hivatkozik rá, üres név esetén a zár figyelmen kívül kerül
- Pattern: a modellel megegyező nevű „.eiq” fájlban található IncQuery mintázatok listája, a kiválasztott alapján fog a zár működni
- Binds: Itt a kijelölt pattern változóira lehet megkötéseket tenni. Az elvárt formátum: patternVáltozó=érték. Az esetlegesen több változó kötése esetén, a megszorítások vesszővel választandóak el. Az itt megadott esetlegesen több változó mindig szűkíti a keresési teret, a változók és viszonyban vannak egymással.
- Enable: True/False értékeivel ki illetve bekapcsolhatjuk az adott zárat.



Name	Pattern	Binds
SysVal VA_62	subsystem2 Sys SysID SysDesc WTC WTCCycle WTCPriority	SysID=Subsystem 07
CtrlUnit_232	ctrlUnit29 C SysID CFault CFaultSysID	SysID=CtrlUnit_232

14. ábra Kliens zár definiáló nézete

5.1.7 Zárak megvalósítása

5.1.7.1 Nézet detektálás

Ahhoz hogy a számunkra megfelelő szerkesztő jelenlétét érzékelni tudjuk több dologra is szükség van. Elsőre is fel kell iratkoznunk a workbench ablakában történő változásokra. Ezt a következő kódrészlettel lehet megvalósítani:

```
final IWorkbench workbench = PlatformUI.getWorkbench();
IWorkbenchPage page = workbench.getActiveWorkbenchWindow()
    .getActivePage();
page.addPartListener(listener);
```

Az első sorban elkérjük a workbench objektumot a rendszertől majd az eszköz aktív oldalára egy nézetfigyelő interfészt (IPartListener) megvalósító objektummal iratkozunk fel. Az interfésznél a partActivated függvényt valósítjuk meg, a többire nincs

szükségünk. A függvényen belül lényegében egy szűrést végzek, azt vizsgálom, hogy az aktivvált lett ablakrész megvalósítja-e az `IEditingDomainProvider` interfészt. Azért erre vizsgálom, mert az EMF modellek szerkesztőinél bár nem kötelező, de ajánlott ezt az interfészt megvalósítaniuk. Többek között az alapértelmezett generált modell szerkesztő is megvalósítja ezt. A fő haszna ennek az interfésznek az, hogy a `getEditingDomain` függvényen keresztül elérhetővé válik a szerkesztő `EditingDomain` objektuma. Ezzel az objektummal ugyanahhoz a modellhez tudunk hozzáférni, mint a szerkesztő maga, így értesülni tudunk a szerkesztési folyamatokról még a modell elmentése előtt. Először a változtatások az `EditingDomain` által biztosított `ResourceSet`-en jelennek meg, majd később a változást eredményező művelet a szintén a dómén által biztosított `CommandStack` objektumon jelenik meg. Az utasítás stack egy `CommandStackListener` típusú objektummal kerül megfigyelésre, illetve szerkesztésre. A `ResourceSet` objektumot pedig az `IncQuery` - EVM figyeli meg. Így lehetőség nyílik arra, hogy az EVM segítségével hatékonyan megfigyeljük a modell változásait, majd ha nem kívánt változás lép végbe, akkor azt az utasítást a stacken visszavonhatjuk.

5.1.7.2 IncQuery - EVM

Ahhoz hogy elérhessük az EVM szolgáltatását a következőt kell tennünk a mi esetünkben:

```
EMFScope emfScope=new EMFScope(resourceSet) ;

engine=      AdvancedIncQueryEngine.createUnmanagedEngine(emfScope);

schedulerFactory = Schedulers
.getIQEngineSchedulerFactory(engine);

executionSchema = ExecutionSchemas
.createIncQueryExecutionSchema(engine, schedulerFactory);
```

Először is egy `EMFScope` objektumot kell létrehozni, erre azért van szükség, mert az újabb `IncQuery` verziókban már nem EMF modelleken is lehetséges dolgozni, ez egyfajta adapterként működik. Az `engine` megkapja ezt paraméterként, majd létrehozuk az ütemezőt, végül pedig a végrehajtási séma jön létre.

Ahhoz hogy fenti rendszer működésre tudjuk bízni, szükségünk van még egy query mintára is. Ezt jelen esetben mivel nem szakterület specifikus megoldást készítettem, futás időben olvassuk be, majd a rendszer készít belőle alkalmazható mintát. Normál esetben a query minta már fordítási időben elérhető, így ebből már akkor elkészül

az alkalmazható verzió. Az átalakításra ugye azért van szükség, mert a minta deklaratív módon van leírva, végrehajtani azonban csak imperatív kódot tudunk.

A minta beolvasása után le kell kezelnünk az általa lefedett adathalmaz változásait. Ehhez az EVM a Job objektumokat ajánlja fel. Egy jobnál lehetőségünk van meghatározni, hogy az adathalmazon milyen típusú változáshoz szeretnénk triggerelni. Ilyen változás például az új elem megjelenése, elem frissülése stb. Egy példa job objektumra:

```
Job<IPatternMatch> jobAppeared = Jobs.newStatelessJob(
    IncQueryActivationStateEnum.APPEARED,
    new IMatchProcessor<IPatternMatch>() {

        @Override
        public void process(IPatternMatch match) {
            handlePatternMatch(match);
        }
    });
```

A jobok definiálása után már csak egy szabály specifikációt kell létrehozni, ami a mintát és a hozzá kapcsolt job objektumokat egyesíti, majd hozzá adni ezt a specifikációt a végrehajtási sémához. Ha az összes kívánt szabályszerkezetet hozzáadtuk a sémához, fontos hogy a sémán mindenképp hívjuk meg a `startUnscheduledExecution` függvényt, ugyanis ennek hatására fognak helyesen, megfelelő időben lefutni a jobok.

5.1.7.3 Zár

A zárat egy általam létrehozott osztály írja le, 4 paraméterük van, a zárak definiálása pontban bemutatottaknak megfelelően. Az osztály egy fontos függvénye az `isMatchWithEventAtom` ami egy `IPatternMatch` típusú objektumot vár, az előzőleg bemutatott `Job` bemenő paraméterének megfelelően. A zárak e függvényét a job objektumok használják fel, minden egyes job lefutásakor az összes definiált záron végigmegy, majd a zár visszaadja a függvénye által, hogy őt fed-e az `IPatternMatch` objektum vagy nem. A fedés eldöntése egyszerű, hisz az `IPatternMatch` is tartalmazza, hogy pontosan melyik `IncQuery` minta miatt futott le, és hogy melyik objektumot érinti. A zár esetén pedig pont ezeket az információkat kell megadni, a `pattern` és a `binds` mezőkben. Tehát ha a két objektum ezen értékei megegyeznek, akkor az utasítás stack kiértékelendő, ha pedig nincs egyezés, akkor hagyjuk lefutni az utasítást.

5.1.7.4 Helyes IncQuery minták deklarálása záarakhoz

Ahhoz hogy az IncQuery mintát feltudjuk használni záarakhoz egy fontos kitételnek teljesülnie, kell. A deklarált minta paramétereinek tartalmaznia kell azon objektumokat, amiken a záarak szeretnénk elhelyezni, és azon paramétereiket is szerepeltetni kell, amelyeket zárolni szeretnénk. Egy rossz és helyes példán keresztül mutatom be, hogy mi ennek az oka.

Helyes példa:

```
pattern subsystem(Sys, SysID) {  
    Subsystem.sysId(Sys, SysID);  
}
```

Rossz példa:

```
pattern subsystem2(SysID) {  
    Subsystem.sysId(_Sys, SysID);  
}
```

A két példa mindkét esetben a SysID-re illeszkedik paraméter szerint, azonban az első esetben az objektum is megvan nevezve a paraméter listán, aminek a SysID a tulajdonsága. Továbbá feltételezzük, hogy a két query-re ugyanolyan típusú jobok vannak ütemezve, appear, disapper és update trigger feltétellel. Ha lefuttatjuk ugyanazon az eredményhalmazon a két queryt ugyanazokat a SysID értékeket kapjuk meg a jobokon belül, azonban megfigyelhetjük, ha két Sys objektumnak ugyan az a SysID értéke, akkor a második esetben csak egyszer jelenik meg ez a SysID. Ez önmagában még nem jelent problémát, azonban ha például megjelenik egy ugyanolyan értékű SysID-vel rendelkező objektum, ami már szerepelt régebben az eredményhalmazban, akkor a második esetben egyik job sem fut le, míg az első esetben lefut az appear job. A jelenség oka az, hogy míg az első esetben az új eredménynek a Sys mindenképp egyediséget biztosít, hisz mint objektum egyedi és biztos nem szerepelt eddig az eredményhalmazban, míg a második esetben a SysID-re ez nem teljesül.

A problémát tovább fokozza, ha módosítás történik az eredményhalmazon belül. Nincs eszköz arra nézve, hogy egy művelet miből mibe alakította át az eredményhalmazt. A frissítéseket kezelő job csak az új, megváltozott eredményt kapja meg, a régi értékekhez nem férhet hozzá. Így nem tudjuk eldönteni, hogy a műveletet vissza kell-e hívni az utasítás stackről vagy nem. Erre a problémára szintén megoldást jelent az első verzió, hisz a másodikkal szemben, ha egy módosítás történik az eredményhalmazon, a második esetében csak az update job fut le, míg az első esetben az update mellett az appear és a

disapper is biztosan lefut. Ennek az oka szintén az, hogy a tulajdonos objektum a query eredményhalmazába egyediséget biztosít. Jelen esetben pedig az apper és a disapper job azon értékeket fogja megkapni, amik a módosítási művelet során eltűntek vagy megjelentek, így a műveltről eldönthető hogy érint-e zárolt objektumokat vagy nem.

5.1.8 Beállítások

Ahhoz hogy a modulunkhoz beállításokat is tudjunk kezelni a már bemutatott preference eszközöket kell használni. A kliens 3 beállítást kezel. Ezek a zárat kezelő szerver címe, a pooling engedélyezése illetve a pooling ütemezésének a ciklusa. Ahhoz hogy a rendszer újraindítása nélkül az adott beállítások életbe lépjenek egy konvenciót kell alkalmazni. A beállításokon aktivált `IPropertyChangeListener` figyelő változás esetén módosít egy központinak jelölt értéket, majd azokon a helyeken ahol ezen beállítás értéke felhasználandó innen hivatkozva éri el az adatot, így mindig a friss értékkel dolgozhat az adott folyamat.

5.1.9 Jelölök használata

A modul működése során 3 féle jelzést továbbítok a felhasználó felé.

Ezek:

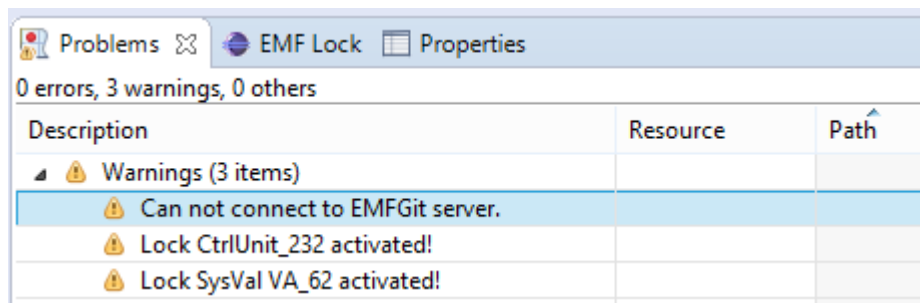
- kapcsolat nem létesíthető a szerverrel
- szerverre történő publikáció hiba
- zár aktív

Megvalósításuk mindegyik esetben hasonló, a hibának megfelelő üzenetet társítok a jelölőhöz, majd warning, figyelmeztető tulajdonságot adok meg neki. Mivel minden jelölőt egy erőforráshoz kell csatolni, így univerzálisan a platform statikusan elérhető workspace objektumához társítom.

Az egyes esetekben a következő üzeneteket jelenítem meg felhasználónak:

- Szerver kapcsolati probléma: „Can not connect to EMFGit server.”
- Publikálási probléma: „Can not publish locks.”
- Aktív zár esetén: „Lock [zár neve] activated!”

Példa kapcsolati hiba és két zár aktivitása esetén:



15. ábra Szerver kapcsolati és zár aktivitás visszajelzések

5.1.10 Jersey használata

5.1.10.1 letöltés – feltöltés

Ahhoz a Jersey-t le vagy feltöltéshez megfelelően tudjuk használni a következő kódrészletet kell használni:

```
Client client=Activator.getClient();

String url=Preferences.ServerAddress;

WebResource resource =
client.resource(url).path("/download").queryParams("projectID",
projectID)
.queryParam("filename", fileName);

File response=resource.accept(MediaType.APPLICATION_OCTET_STREAM)

.get(File.class);
```

Az első sorban elkérem a központosított kliens objektumot a modul Activator osztályától, majd pedig a szerver elérhetőségét a beállításokat megvalósító osztálytól. A következő sorban megadom az elérni kívánt erőforrást, cím és paraméterek által. Ezek után a válasz fogadása jön, itt megadom, hogy fájlt fogok válaszként visszakapni a get típusú kérésre. A feltöltés esetén a WebResource objektumig megegyezik a használat, azonban utána a következő műveleteket kell végrehajtani:

```
Builder post =
resource.entity(fileToUpload,MediaType.APPLICATION_OCTET_STREAM );
post.post();
```

Mint látszik, itt egy post művelet hajtódik végre, jelölésre került az erőforrás, a feltöltendő fájl, illetve hogy milyen típusú objektumot töltünk fel.

5.1.10.2 Pooling

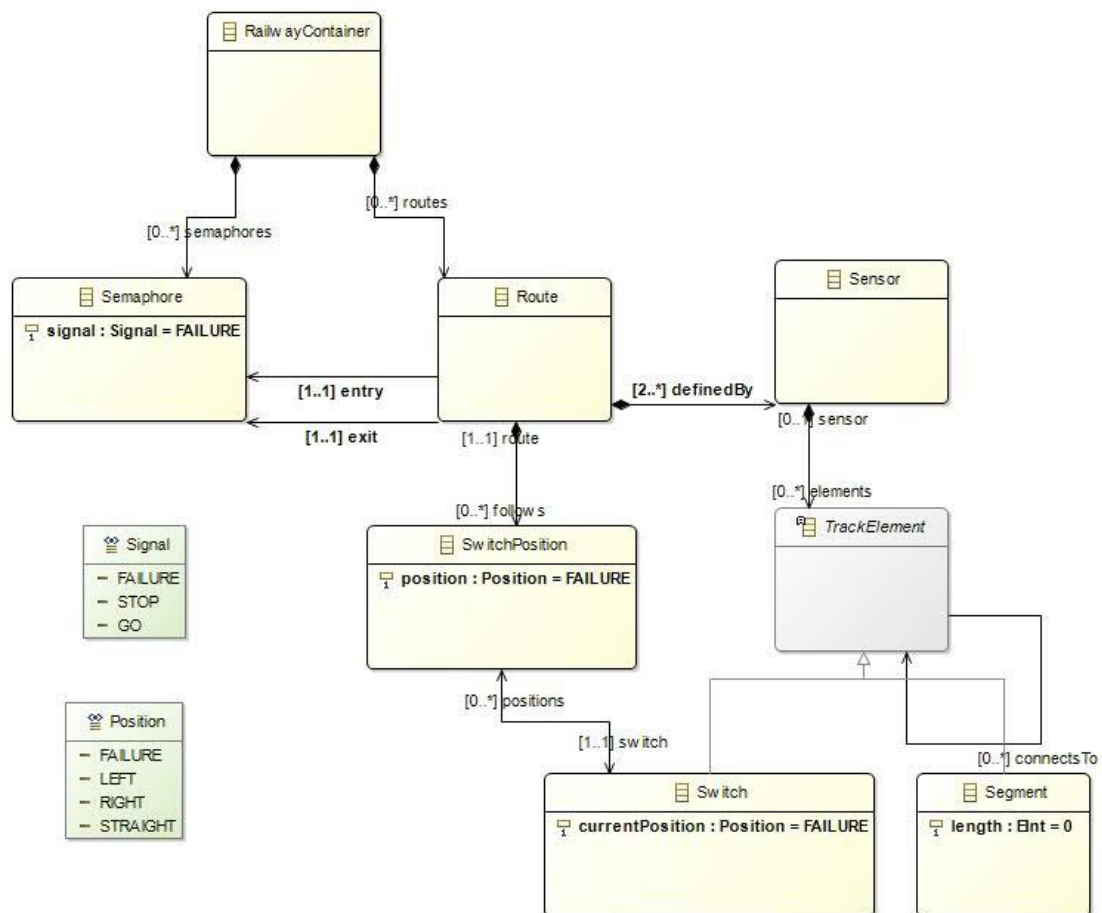
A szerver folyamatos ellenőrzése a következő Eclipse job által történik:

```
job=new Job("Check locks on server") {  
  
    @Override  
    protected IStatus run(IProgressMonitor monitor) {  
        try{  
  
            monitor.beginTask("Check locks ", 1);  
  
            refreshActiveProjectsLocks();  
  
            monitor.done();  
            return Status.OK_STATUS;  
        }finally{  
            schedule(Preferences.PoolingInterval);  
        }  
    }  
};  
  
job.setPriority(Job.SHORT);  
job.setSystem(true);
```

Mint már említettem lényegében egy okosabb szálkezelési eszközt kapunk. Lehetőségünk van jelezni a folyamat állapotát, ezt a monitor objektumon keresztül tehetjük meg, az ütemezést pedig a finally ágban végrehajtott schedule függvény meghívásával ajánlott végrehajtani. Mivel az itt végrehajtott feladat egy viszonylag rövid ideig tartó feladat, így alacsony prioritásra állítottam be és rendszerfeladatként állítottam be. Ez azt jelenti, hogy a felhasználó felé a grafikus felületen nem jelenik meg értesítés a munka lefolyásáról.

6 Kiértékelés

Ahhoz hogy az elkészített megoldásom használhatóságát felmérjem méréseket végeztem. Ehhez a tanszéken fejlesztett Train Benchmark framework[11] nevű eszközt használtam fel. Ez az eszköz képes egy vasúti rendszer modelljét (16. ábra) legenerálni majd felmérni a modellen végzet műveletek idejét különböző háttértár technológiák esetén. Jelen esetén én csak a modell generáláshoz használtam az eszközt, a méréseket saját magam végeztem el.



16. ábra Generált modell felépítése

A modell generálása esetén a „Repair” nevű opciót használtam, ez a következő paramétereket használta fel a példánymodell elkészítése során:

```

maxSegments = 5;
maxRoutes = 20 * SIZE;
maxSwitchPositions = 20;
maxSensors = 10;
posLengthErrorPercent = 10;
switchSensorErrorPercent = 4;

```

```
routeSensorErrorPercent = 10;
semaphoreNeighborErrorPercent = 8;
switchSetErrorPercent = 10;
connectedSegmentsErrorPercent = 5;
```

SIZE (utak száma)	50	100	200	500
Méret (MB)	40	78	155	382

17. ábra A felhasznált modellek utjainak száma és fizikai méretük

A mérés során felhasznált záruk mintái:

```
pattern route(R, RID) {
    Route.id(R, RID);
}

pattern sensor(S, SID) {
    Sensor.id(S, SID);
}
```

A SIZE paraméter 50,100,200 és 500as értékek mellett vizsgáltam meg. A hozzájuk tartozó modellméretet a 17. ábra mutatja. Kisebb értékek esetén a Java rendszer nem determinisztikus szemétgyűjtője adta mérési hibák miatt értelmetlen lett volna a mérés, nagyobb modellek használata esetén pedig már kevés volt a memóriám. A zárat paraméter lekötés nélkül használtam fel, így a route nevű minta viszonylag kicsi, pár ezres számú eredményhalmazzal dolgozott, míg a sensor pattern az előző halmaz kb. negyvenszeres számosságával volt, így már a legkisebb vizsgált esetben is 44 ezer volt az eredményeinek a száma. A méréseim során a modul használatához szükséges memóriát, a záruk aktiváláshoz szükséges időt és záruk használata közbeni törlés műveletet figyeltem meg.

A méréseimet a következő környezetben végeztem:

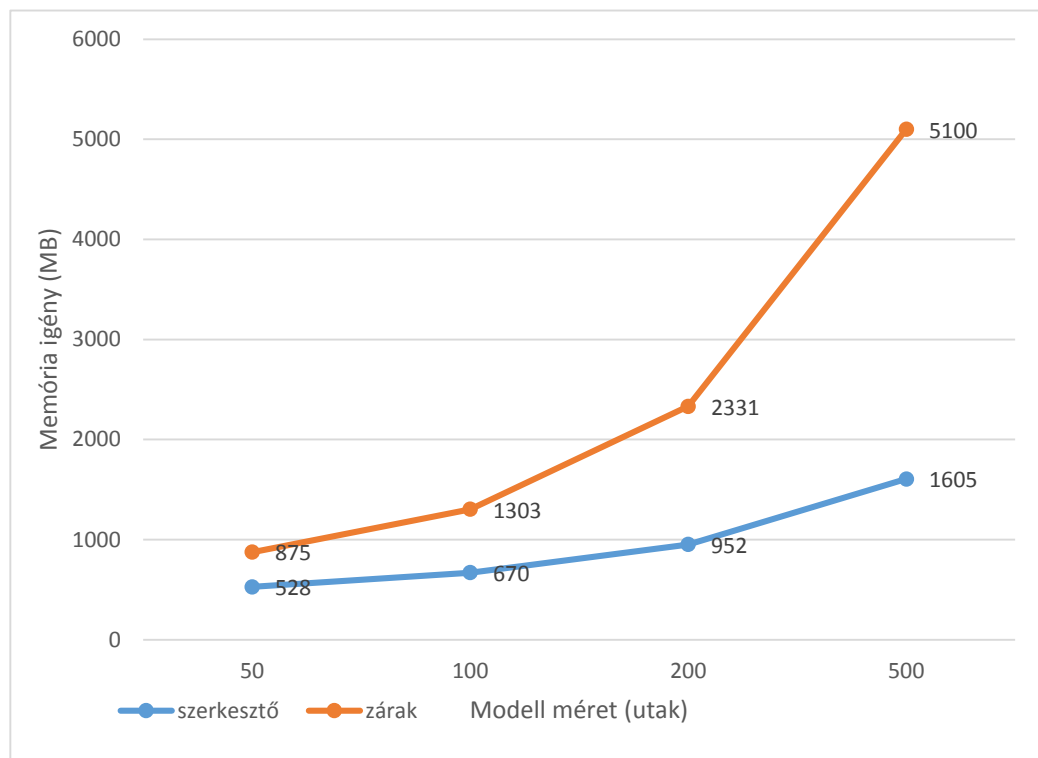
- CPU: i5-3210M – 2 mag, 4 szál, 2.5GHz
- RAM: 8GB DDR3
- Háttértár: Samsung 840 Evo 250GB
- Operációs rendszer: Windows 8.1 Pro 64bit
- Runtime Eclipse paramétereit: -Xms40m -Xmx6144m

Az Eclipse paramétereinél az –Xms kapcsoló a rendszer kezdetleges memória lefoglalást, az –Xmx pedig a lehetséges maximális memóriát jelenti. Mivel a rendszer

alapértelmezésként 40MB és 512MB értékeket használ, ez nagyban korlátozta volna az alkalmazás teljesítményét, így a konfigurációm által biztosított maximumot állítottam be.

Memória igény

E mérés során azt vizsgáltam meg, hogy az Eclipse mennyi memóriát foglal le magának, ha a modelleket az EMF alapértelmezett szerkesztőjével nyitom meg, majd a zárok elhelyezése utáni állapotot is megvizsgáltam. Az előbb bemutatott zárok közül mindkettőt aktiváltam, a méréseket minden egyes modell esetén egy újonnan megnyitott Eclipse-ben vizsgáltam meg. Az eredmények mindig a folyamat végeztével beálló értékeket jelentik. A modellek mellett más nem volt megnyitva, más munkafolyamat nem történt. Az üres, szerkesztői nézetet nem tartalmazó Eclipse kb. 295MB memóriát igényelt, mindig ebben az üres nézetben történt az egyetlen aktuális modell megnyitása.

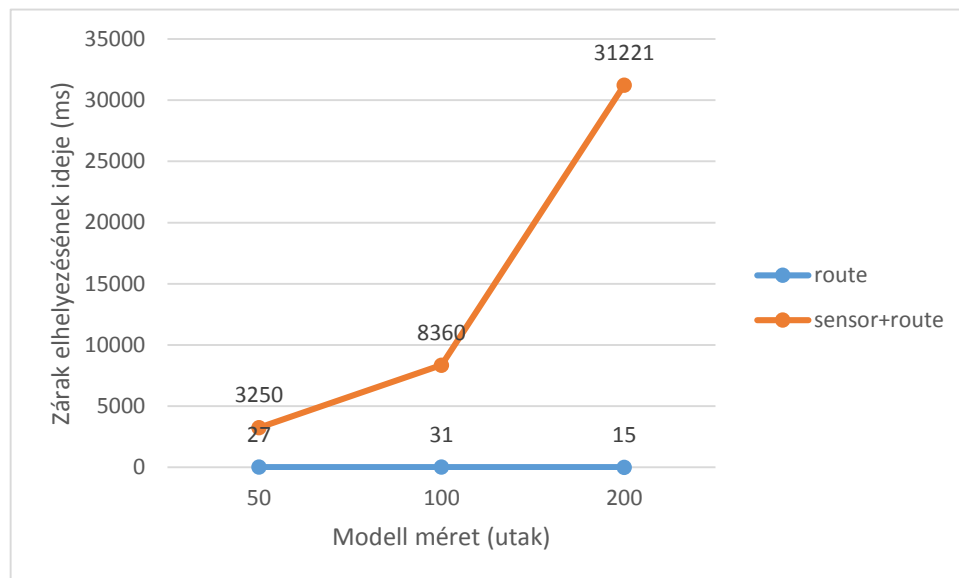


18. ábra Memória használat a tesztelt modellek esetén

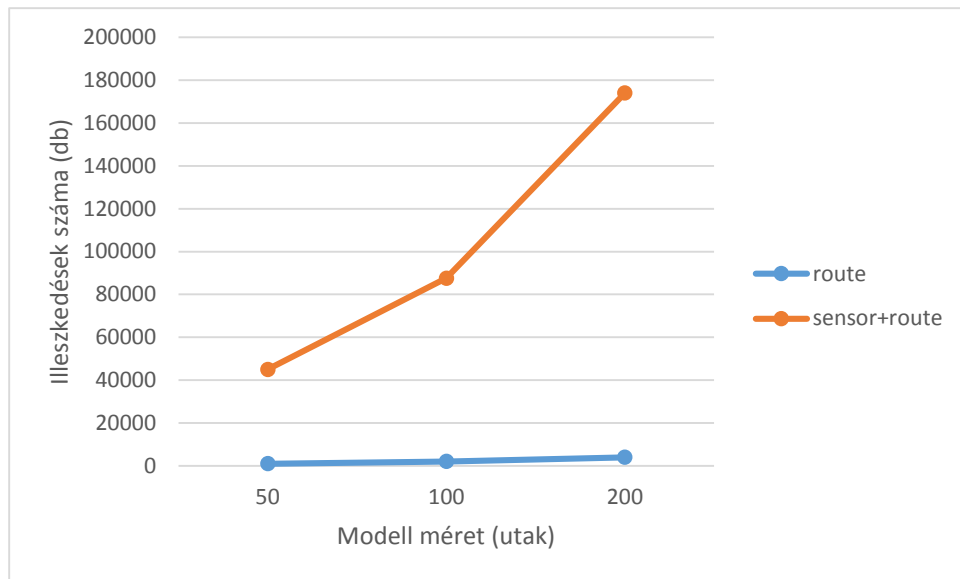
A 18. ábra alapján azt láthatjuk, hogy a szerkesztő és a zárolt modell memória igénye a modell nagyságával exponenciálisan nő. A mérést elvégeztem több esetben is zárok alkalmazása nélkül, pusztán csak az IncQuery minta illeszkedésre, ekkor nem történt lényegi változás az eredmények terén. Az 500as modell esetében bár még a zárok elhelyezése sikeres volt, ezek után azonban a rendszer már nem volt működőképes, nem reagált semmire, így a mérés további részéből ezt a modellt már kihagyom.

Zárak aktiválásához szükséges idő

A következő mérésben azt vizsgáltam meg, hogy a különböző méretű modelleken a zárok elhelyezése mennyi időt vesz igénybe. Ezt a végrehajtási séma aktiválása, a `startUnscheduledExecution` függvény meghívása előtti és utáni rendszeridőpontok differenciája alapján mértem. Ez azért megfelelő pont a mérésre, mert a függvény meghívására az összes, mintára illeszkedő objektum regisztrációra kerül, és a neki megfelelő appear job meghívódik. Külön teszteltem azt az esetet, amikor csak a route, és amikor a route és sensor zárok együtt aktiválódtak. Így lehetőségem van megfigyelni hogyan, alakulnak az eredmények egy kis és egy nagy minta illeszkedési eredményhalmaz esetén. A mért eredményt az alább ábra mutatja:



19. ábra Zárak elhelyezésének ideje

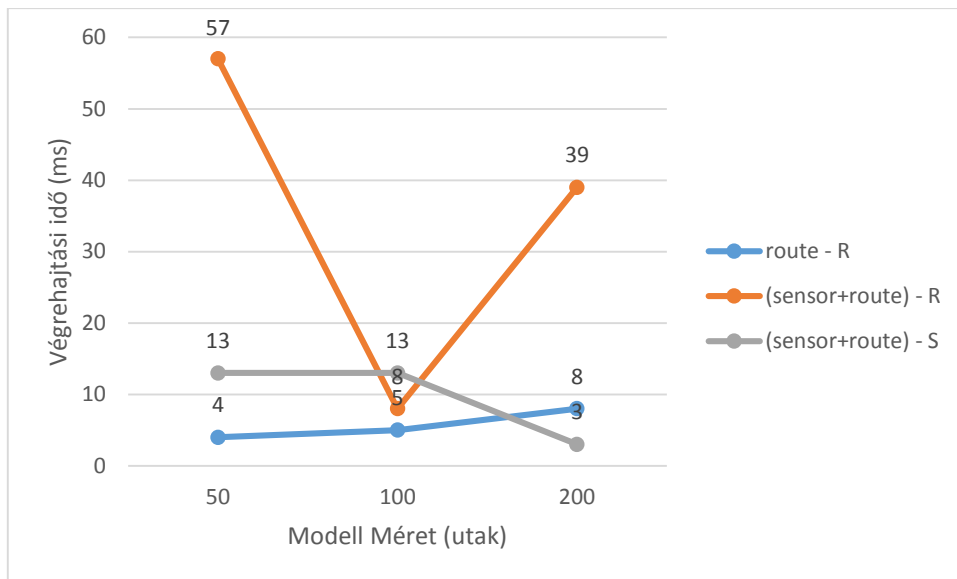


20. ábra Illeszkedések száma

Figyelembe véve, hogy a generált modell lényegében lineáris (20. ábra), tehát nagyobb számú út esetén a modell többi része is ezzel párhuzamosan nő. Így kijelenthető hogy a kezdeti zárlerakási idő nem a modell őszméretétől, ha nem a modellben található illeszkedések számától függ. Akár csak a memória használata esetén, itt is exponenciális növekedést figyelhetünk meg, az illeszkedő objektumok száma exponenciálisan növeli a záruk lehelyezései idejét. Arra hogy nem a modellméret, ha nem csak az illeszkedő objektumok száma okozza ezt, arra az a magyarázat, hogy az IncQuery már a memóriában lévő modellen dolgozik, ezen megvizsgálni egy illeszkedést szinte jelentéktelen költségű, azonban ha illeszkedést talált a rendszert azt nyilván kell tartania már, ami mint látszik, nagy költséggel jár.

Művelet végrehajtása zárolt modellen

Kérdés volt még, hogy már zárolt modell esetén egy zárat sértő művelet feloldása mennyi időt vesz igénybe. Ennek a mérésére az EMF Notification szolgáltatását használtam fel. Ez az eszköz még az IncQuery előtt képes jelezni egy modell változását, így lényegében az itt jelzett változást tekinthetjük a művelet kezdetének. A művelet végét az utasítás stackre való eljutásig mértem.



21. ábra Végrehajtási idő egy törlési művelet esetén

A méréseket végrehajtottam csak a route illetve a route és sensor pattern páros együttes használatával is. A route pattern alkalmazása esetén egy véletlen kiválasztott Route elemet töröltem, a minta páros esetén egy Route és egy Sensor objektum törlését is megvizsgáltam. A 21. ábra alapján, a már zárolt modellen a műveletek végrehajtási ideje minimális.

Ezek után mindegyik modell esetén 100, 200 és 500 véletlenszerű elem törlését is megkíséréltem úgy, hogy lehetőleg minél több Route és Sensor típusú elem legyen a művelet hatáskörében. Az alkalmazott záráktól és modellmérettől függetlenül, 100, 200 és 450 ms körüli értékeket kaptam. Tehát a műveletek végrehajtása jól skálázódik, lineáris.

Összegzés

Mint kiderült a módszer hátrányai a hatalmas memória igény, és a modellek zárolási ideje. A hatalmas memória igényt ugyan minimálisan lehetne csökkenteni egy másik szerkesztő, vagy testreszabott Eclipse használatával, azonban a zárok alkalmazása így is jelentős terhelést jelent. A zárok elhelyezésének az ideje annak ellenére, hogy egy memóriában történő művelet, igen hosszú ideig is eltarthat nagy illeszkedési halmaz esetén. Sajnos ezt tovább csökkenteni nem igen lehet. Másik oldalról, viszont ha a rendszerünk már egyszer elindult, a rajta végzett műveletek folyamatosak, elvárható időn belül végrehajthatók.

7 Kapcsolódó munkák

Mint minden mérnöki munka esetén, itt is érdemes volt már meglévő, hasonló megoldásokat megvizsgálni a felvetett problémára. Először a záruk tipikus megközelítéseit, majd az Eclipse két eszközét a CDO-t és az EMFStore-t hasonlítom össze. Végül ezek funkciót az egyik népszerű fizetős modellezési eszközzel is összevetem.

Záruk megvalósítása általában

A zárat több minőségben is vizsgálhatjuk, ilyen főbb tulajdonságok a művelet zársértésének detektálása, granularitás, záruk egymásra való hatásai, illetve hierarchiakon végzett zárolások mikéntjei.

Azt, hogy egy adott művelet egy zárat sértett-e, kétféleképpen detektálhatjuk, vagy a művelet végrehajtása előtt vagy után detektáljuk az esetleges zársértést. Az első eset a pesszimista, hisz feltételezzük, hogy zár van az adott objektumon, a második eset az optimista eset, megpróbáljuk a műveletet végrehajtani és csak aztán detektálunk.

Granularitás szerint a záruk sokrétűek lehetnek. Verziókezelő rendszerek vagy fájlrendszer esetén a zár a fájlra szól, vagy épp egy adott könyvtárra, de egy modell szerkesztőben már például nem feltétlen. Ott már tipikusan a modell elemeire, részeire helyezhetünk el zárat, így egy komolyabb zárolási logikát alkalmazva.

Záruk esetén gyakori az, hogy az adott műveleteket kiéheztetik egymást, az erőforrások kölcsönösen blokkolják. Így megoldást kell találni arra nézve, hogy a rendszer tovább működhessen. Erre tipikusan a relációs adatbázisok területén láthatunk megoldásokat, ilyen például a kétfázisú zárolás is.

Fontos kérdés még, hogy ha egy adatszerkezetben zárt helyezünk egy csomópontot, akkor mi történik a leszármazottaival, kapcsolódó elemeivel. Itt szintén a relációs adatbázisok területéről példa lehet a fa vagy a figyelmeztető protokoll megvalósítása.

CDO

A CDO[18] az az Connected Data Objects egy olyan megoldás, amely képes az EMF modellek tárolására magas testreszabhatóság mellett. Egyik fő tulajdonsága, hogy

az adatok tárolására számos lehetőségünk van. A rendszer képes használni SQL, NoSQL, fájl vagy akár memória (Ram) alapú háttértárat is. Felépítéséből adódóan ugyan lehetőséget biztosít kollaboratív munkára, de az esetleges konfliktusok megoldásához tipikusan külső eszközöket kell használnunk pl. EMF Diff/Merge.

EMFStore

Az EMFStore[19] egy olyan tároló EMF modellek számára, ami képes verzió követni az egyes modellek változásait. Létjogosultsága nyilvánvaló, hisz a többi verziókövető rendszer kizárólag szöveg alapú, ezáltal nem tudnak megfelelően kezelni olyan fájlokat, amikben szöveg nem program kódot, ha nem egy modellt testesít meg. Ezáltal a rendszer hatékony megoldást tud nyújtani nem csak verzió kezelésre, ha nem kollaboratív munkára is, hisz a modellek összefésülése illetve konfliktus feloldása könnyen megoldható.

EMFStore és CDO összehasonlítása

7.1.1 Adattárolás

A CDO számos tárolási technológiát fel tud használni egészen az egyszerű fájl alapútól a komplexebb adatbázisokig. Előnye abban rejlik, hogy a tárolási mechanizmusa szabadon megvalósítható adapterek útján, így akár eddig nem létező technikákkal is kiegészíthető. Fontos megemlíteni, hogy a rendszer támogatja a lazy loadingot, a késleltetett betöltést is. Ennek a haszna főleg akkor jelentkezik, amikor egy nagyobb modellt szeretnék használni. A technika lehetőséget biztosít arra, hogy ne az egész modellt olvassuk be a háttértárról, ha nem csak egy részét, épp amit használunk.

Ezzel szemben az EMFStore csak is kizárólag XMI fájl alapú perzisztenciát támogat. Az XMI[20] lényegében 3 technológiát egyesít, ezek pedig az XML, a fájl struktúráját rögzíti, az UML és MOF pedig az ábrázolt adatok formátumát adja meg. Bár a formátum akár egy ember számára is jól olvasható struktúrához vezet, azonban a gép számára ez nehézkes feldolgozást jelent. A CDO lazy loading megoldásához képest, ahhoz hogy itt dolgozni tudjunk egy modellel, a háttértárolóról teljes egészen be kell azt olvasni, és a memóriában kell tárolni. Így a modellhez való első hozzáférés nehézkes lehet és nagy memória igény is jelentkezhet egy komplexebb modell esetén. Cserébe viszont az első interakció után a modellen végzett műveletek késleltetése kicsi lesz, hisz a memóriában lévő modellen történnek a műveletek.

7.1.2 Kollaboráció

A CDO mint egy központosított tároló, elérhetősége miatt egyértelmű hogy alkalmas lehet megosztott, több felhasználót igénylő munkák kiszolgálására. Azonban mivel, komolyabb modell műveletek, mint pl. a merge nem támogat, így egyértelműen csak online munka esetén lehet hatékony. Bár a rendszer támogat offline módot is, ekkor azonban csak összegyűjti a változtatásokat, majd amikor a kiszolgáló elérhető lesz, végrehajtja azokat. Mivel ez a mód gyakran konfliktuskezelést kíván, külső eszközöket kell bevonni, amik nehezítik a munkát.

Az EMFStore ezzel szemben a több felhasználós környezet igényeit figyelembe véve készült. Műveletei az SVN rendszerhez hasonlóan működnek, ezáltal a mindig offline történnek. Ennek ellenére a rendszer képes online üzenetek közlésére a modell változásairól. Figyelembe véve hogy verziókövetést, magas fokú modell összefésülést, probléma megoldásokat is tartalmaz, egyértelműen hogy kollaboratív munkára ez a megoldás hatékonyabb biztosított eszközei által.

7.1.3 Jogok kezelése, zárolás

Felhasználók azonosítása terén, a CDO több lehetőséget is kínál. Lehetőségünk van egyszerűen, akár egy fájlban is megadni a felhasználókat, illetve komolyabb LDAP (Lightweight Directory Access Protocol) is támogatott, ami már komolyabb, strukturáltabb felhasználó kezelést tesz lehetővé. Az objektumok zárolása esetén is több lehetőség van, objektumokat akár egyesével is zárolhatunk, de lehetőségünk van magasabb szintű szűrő kifejezéseket is deklarálni.

Az EMFStore ezzel szemben szegényesebb funkcionalitással bír. A felhasználók azonosításánál külső megoldást nem lehet alkalmazni, és lényegében lock funkcióval sem rendelkezik a rendszer. Az egyetlen amit tehetünk, hogy az adott projekt esetén jogot adunk felhasználónak vagy egy felhasználók csoportjának, és ezzel kontrollálhatjuk a hozzáférésüket.

7.1.4 Konfliktusok megoldása

Mint azt az eddigiekben már említettem, a CDO esetén külső megoldásokhoz kell folyamodni tipikusan, míg az EMFStore magas szintű összefésülést, modell commit pontra való visszaállítást tesz lehetővé.

7.1.5 Skálázhatóság

Mivel magas háttértár testreszabhatósággal rendelkezik a CDO ezért skálázhatósága nagyban függ a felhasznált technológiától. Mivel főként SQL alapú adatbázis használata az elterjedt így az ilyen rendszerekre alapuló megoldást veszem most figyelembe. Tekintettel arra, hogy általában egy SQL adatbázis jól skálázódik nagyobb adatmennyiség esetén, számos helyen gyorsítótárazást lehet alkalmazni a műveletek során, és hogy a CDO lazy loading technikát is támogatja, egy viszonylag jól skálázható rendszer kaphatunk.

Ellentétben ezzel, az EMFStore pusztán XMI alapú tárolásra ad lehetőséget, és a modellen való művelet esetén mindenképp be kell tölteni a teljes modell a memóriába. Ez az első művelet végrehajtása előtt sok időt tehet ki, azonban később a műveleti idők viszonylag kicsik lesznek. A megoldás hátránya, hogy egy méret fölött biztosan kifutunk a memóriából és a rendszer szinte használhatatlanná válik, hisz a Java szemét gyűjtője folyamatosan „szenvetni” fog.

7.1.6 Összefoglalás

Figyelembe véve a két eszköz tulajdonságait, mindkettőnek vannak célterületet meghatározó tulajdonságaik, amik egyértelműen eldöntik, hogy mire is használhatóak. Az EMFStore esetében egyértelmű, hogy a készítő célja a verziókezelés megvalósítása volt, azonban más funkcióval nem igen rendelkezik a rendszer. Ezzel szemben az CDO sokrétű szolgáltatásai, mint a felhasználó kezelés, zárok megvalósítása, rugalmas háttértár kezelés véleményem szerint szélesebb igényeket fed le és szélesebb réteg által használt megoldás, míg az EMFStore egy szűk réteg eszköze lehet csak.

MetaEdit

Az iparban egy igen népszerű és sokfunkcionalitású modellező eszköz a MetaCase vállalat MetaEdit[21] szoftvercsomagja. A szoftver népszerűségét az igen szerteágazó funkcionális sokszínűségének köszönheti. Ilyen fontosabb funkciók például:

- modellek központi szerverről való elérése
- modellből kód és eszközök generálása
- validáció

- sok felhasználós kollaboratív környezet biztosítása
- modellek verziókövetése
- magas szintű integrálhatóság szinte bármilyen nyelvvel és környezettel

Összehasonlítva az Eclipse által nyújtott eszközökkel az első különbség, hogy ez egy fizetős megoldás, míg az előbbiek mind ingyenesek és nyíltak. Szolgáltatások terén elég jól lefedi a CDO és EMFStore nyújtotta lehetőségeket, azonban néhány, tipikusan testreszabhatósági tulajdonságokban csorbákat szenved.

7.1.7 Modellszerkesztés

A munka folyamán jól elkülönülő metamodel (Workbench) és példánymodell szerkesztővel (Moduler) rendelkezik. Ezen szerkesztők specifikusan csak az egyik feladat elvégzésére vannak kielevezve, így jóval hatékonyabb módon lehet egy adott problémát megoldani.

Annak ellenére, hogy a rendszer jó integrálhatóságot biztosít számos fejlesztői környezetbe, fejlesztés közben a külső eszközök használata problémássá válhat annak ellenére, hogy egy jól definiált kiegészítő interfésszel rendelkezik a MetaEdit. Az eszközök olyan szintű integrálhatóságát, mint az Eclipse rendszer nem tudja megvalósítani. Épp ezért egy komplexebb, szerteágazóbb eszköztárat igénylő fejlesztés során az Eclipse biztosan kényelmesebb és hatékonyabb munkára képes.

7.1.8 Kollaboratív munka támogatása

Rendszerük lényegében a bemutatott két Eclipse eszköz tulajdonságait ötvözi. Képes verziókezelésre, bár ennek a granularitása egész más felfogású, mint az EMFStore esetében. A változásokról pillanatképeket készít, és ezekre lehet esetlegesen visszatérni. Így bár szerényebb lehetőségekkel, de megvalósítja a verziókezelést. Azonban magas fokú zárolási lehetőségekkel is bír, külön objektumokat és struktúrák zárolását is engedélyezi, felhasználóknak pedig széles körű jogok szabhatók meg. Így egyedüli hátránya ebben az aspektusa az offline munka lehetősége az EMFStore-hoz képest.

7.1.9 Integrálhatóság, más környezetekkel való kapcsolat

Az eszköz bár rendelkezik kiegészítő interfésszel, ám ez közel sem közelíti meg az Eclipse funkcionalitását. Ilyen funkciók lehetnek például más aspektusú szerkesztők használata egy időben ugyanarra a modellre, vagy az adatokon végzett több lépcsős

feldolgozás. Másik oldalról viszont, az Eclipse eszközei csak a Java környezetre redukálódnak, míg a MetaEdit lényegében bármely programozási nyelvvel együtt tud működni. Ez pedig igen nagy előnyt jelenthet egy olyan környezetben, ahol az adott problémához a legjobb programozási nyelvet, rendszert használják.

8 Összefoglalás

A szakdolgozat során megismertem és bemutattam a ma használatos népszerű verziókezelő rendszereket, részletesebben bemutattam a GIT fontosabb funkcióit. Ezek után az EMF modellek kollaborációs lehetőségeit vizsgáltam meg, bemutattam az Eclipse CDO és EMFStore eszközét, majd egy külsős fizetős megoldással, a MetaCase MetaEdit szoftverével is összevettem a képességeiket.

Szakdolgozatom további céljaként létrehoztam egy kollaborációt támogató, tulajdonság alapú EMF modell zárolást lehetővé tevő eszközt. Az eszköz hatékonyságát, használhatóságát mérésekkel vizsgáltam meg, majd egy ipari példán, a MONDO IKERLAN tagjától származó szélturbina vezérlési modelljén mutattam be az eszközt.

Továbbfejlesztési irányok

Az elkészített alkalmazásom több lehetséges fejlesztési ponttal is rendelkezik, ezek közül néhány fontosabb alább szerepel.

8.1.1 Nagyobb kifejező erővel rendelkező zárok

Komplexebb lock definíciók lehetősége sokat tudna javítani az eszköz használhatóságán. Bár az IncQuery minta illeszkedési leírása igen szerteágazó funkcionalitással bír, azonban a paraméterek megadásánál semmilyen funkcionalitás nincs a további paraméter lekötésének lehetőségén túl. Itt mindenképp lehetne bővíteni a funkcionalitást logikai műveletekkel, esetleg pár függvénnyel. Távlati célként pedig akár egy külön zárok definiálására szolgáló nyelv megalkotása lenne a legideálisabb, például az Eclipse Xtext eszközével.

8.1.2 Zár sértés esetén visszajelzés

Zárat szegő műveletek esetén jelenleg a felhasználó érdemi visszajelzést nem kap arról, hogy nem sikerült a kívánt művelete, csak azt tapasztalhatja, hogy minden maradt a régiben. Itt egy felhasználóbarát megoldás lenne, ha tudatnánk a felhasználóval, hogy pontosan miért nem tudja az akcióját végrehajtani, melyik zár vagy zárok akadályozza őt.

8.1.3 Intelligens zárelhelyezés

Miközben a felhasználó dolgozik az adott modellen kollaboratív környezetben sokat segítene, ha a rendszer magától tudná zárolni a modell egyes részeit pusztán a felhasználó akciói alapján. Így sok konfliktus lenne elkerülhető illetve hatékonyabb munkát eredményezne, hisz az adott művelet végrehajtása előtt nem kellene a zár elhelyezésével foglalkozni.

8.1.4 Hatékonyabb kommunikáció

Bár a feladat során használt Jersey REST megoldása funkcionalitásban elég volt a követelményeknek, ha a program funkcionalitása tovább nőne, bizonyossággal a technika korlátaiba ütköznénk. A REST alapvetően egy tartalomszolgáltatási eszköz, valós idejű vagy titkosított adatátvitel már nagyon nehézkesen vagy egyáltalán nem lenne megvalósítható segítségével. Jelenleg talán a legjobb eszköz a WebSocket lehetne, hisz egyaránt nyújt magas szintű szolgáltatások, de alacsony szintű adat átvitelre is lehetőséget biztosít, így a kommunikációt teljesen igényünk szerint alakíthatnánk.

8.1.5 Magasabb szintű integráció

Az Eclipse rendelkezik egy úgynevezett Team API-val. Ez az interfész lehetőséget biztosít arra, hogy kollaboratív eszközt valósítsunk meg platformon belül. Ezt használja többek között a Subversive (SVN) és EGit (GIT) eszköz is. Az elkészített alkalmazásom használhatósága nagyban növekedett volna, ha az SVN vagy GIT rendszerekkel kombinálni tudtam volna úgy, hogy ezt az interfészt megfelelően implementálom. Azonban e feladat komplexitása miatt erre nálam nem került sor.

Irodalomjegyzék

- [1] Stephen J. Mellor, Marc Balcer: Executable UML: A Foundation for Model-Driven Architectures ISBN:0201748045
- [2] MONDO Project, [HTTP://www.mondo-project.org/overview](http://www.mondo-project.org/overview), 2015.05.21
- [3] Mondo Project - IKERLAN, [HTTP://www.ikerlan.es/en/what-we-research/projects/mondo](http://www.ikerlan.es/en/what-we-research/projects/mondo), 2015.05.01
- [4] Ben Collins-Sussman, The subversion project: buiding a better CVS, 2002
- [5] Scott Chacon, Ben Straub : Pro Git, ISBN-13: 978-1484200773
- [6] Eclipse Foundation, [HTTPs://eclipse.org/](https://eclipse.org/), 2015.03.04
- [7] Eclipse Foundation, [HTTPs://eclipse.org/eclipse/](https://eclipse.org/eclipse/), 2015.03.05
- [8] Erich Gamma, Kent Beck, Contributing to Eclipse: Principles, Patterns, and Plugins, ISBN-13: 978-0321205759
- [9] Eclipse Foundation, Eclipse wiki, [HTTP://help.eclipse.org/juno/index.jsp?nav=%2F2](http://help.eclipse.org/juno/index.jsp?nav=%2F2), 2015.03.23
- [10] Richard Hall, Karl Pauls, Stuart McCulloch, David Savage, Osgi in Action: Creating Modular Applications in Java, ISBN:1933988916 9781933988917
- [11] BUTE Fault Tolerant Systems Research Group [HTTPs://github.com/FTSRG/trainbenchmark](https://github.com/FTSRG/trainbenchmark), 2015.05.20
- [12] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks: EMF: Eclipse Modeling Framework (2nd Edition), ISBN-13: 978-0321331885
- [13] Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A., Incremental evaluation of model queries over emf models. Model Driven Engineering Languages and Systems, 13th International Conference, October 2010
- [14] Eclipse Project, [HTTP://wiki.eclipse.org/EMFIncQuery/DeveloperDocumentation/EventDrivenVM](http://wiki.eclipse.org/EMFIncQuery/DeveloperDocumentation/EventDrivenVM), 2015.04.24
- [15] Jersey, [HTTPs://jersey.java.net/](https://jersey.java.net/), 2015.05.20
- [16] Jim Webber, Savas Parastatidis, Ian Robinson: REST in Practice: Hypermedia and Systems Architecture ISBN-13: 978-0596805821
- [17] Network Working Group D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), July 2006

- [18] Eclipse project, [HTTP://projects.eclipse.org/projects/modeling.emf.cdo](http://projects.eclipse.org/projects/modeling.emf.cdo), 2015.05.13
- [19] Jonas Helming, Maximilian Koegel, EMFStore: a model repository for EMF models, 2010
- [20] Timothy J. Grose, Gary C. Doney, Stephen A. Brodsky, Mastering XMI: Java Programming with XMI, XML, and UML, ISBN: 0471384291
- [21] MetaCase, [HTTP://www.metacase.com/solution/](http://www.metacase.com/solution/), 2015.05.11