Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Modellek összehasonlítása és egyesítése

Vajna Miklós (AYU9RZ), I. évf, (MSc) mérnök inf. szakos hallgató
Konzulens: Horváth Ákos, MIT

Szolgáltatásbiztos rendszertervezés szakirány
Önálló laboratórium 1 összefoglaló
2009/10. II. félév

Tartalomjegyzék

1. Bevezetés

A modellvezérelt fejlesztés alapötlete, hogy szoftverek fejlesztése során minél kevesebb programkódot írjunk meg közvetlenül, és minél több esetben egy modellből generáljuk azt. Ennek számos előnye van:

- Ha a kódgenerátor rendelkezik valamilyen tanusítvánnyal, akkor helyes modell esetén a generált kód is automatikusan megfelel adott elvárásoknak.
- A modellek magasabb szintű leírást tesznek lehetővé, mint ha közvetlenül a program forráskódját írnánk.
- A követhetőséget (traceability) megkönnyíti, láthatjuk, hogy agy adott kódsor a modell mely részének felel meg, majd az mely követelmény teljesítése érdekében szükséges.
- Forráskód esetében a formális ellenőrzés nehézkes, míg mérnöki modellekből jóval egyszerűbb olyan matematikai modelleket előállítani, melyeken már formális ellenőrzést végezhetünk.

E paradigma egyik égető problémája, hogy míg hagyományos fejlesztés során a forráskód verziókezelése (összehasonlítás és egyesítés) megoldott, addig modellek esetén erre a célra még nem érhetőek el a gyakorlatban is jól használható általános eszközök.

A félév során megvalósított megoldás során olyan elérhető eredményekre kívánunk támaszkodni, melyek elméleti háttere tudományos cikk formájában elérhető, valamint hozzáférhető a ténylegesen megvalósított szoftver kódja.

Az elkészült megvalósítás, mint azt később látni fogjuk egy modell két pillanatkép-szerű (snapshot) verzióját hasonlítja össze, és gráfokon belül fákon működik.

A dokumentum a továbbiakban a következő részekre tagolható. E fejezet maradék része rövid bevezetést ad a VIATRA2 keretrendszerbe, valamint a félév során megoldandó feladatra. A második fejezet megvalósítást megelőző irodalomkutatás eredményeit részletezi. A harmadik fejezetben ismertetésre kerül az FMES algoritmus. A negyedik fejezet fejti ki az implementáció részleteit. A dokumentum végét a tesztelésről és jövőbeli lehetőségekről szóló fejezetek alkotják.

1.1. A VIATRA keretrendszer

A tanszéken fejlesztett VIATRA (VIsual Automated model TRAnsformations) [1] keretrendszer is ilyen modellekkel dolgozik. A VIATRA modellreprezentációja meglehetősen egyszerű, a modellek elemei a ModellElementek, és ezeknek két típusa van:

- Az Entity egy gráf csúcsának feleltethető meg.

- A Relation pedig egy gráf két csúcsát összekötő élnek.

A félév elején azért esett erre a keretrendszerre a választás, mert ha ezen a keretrendszeren sikerül jó modell összehasonlító / egyesítő algoritmust alkotni, akkor egyrészt azt más keretrendszerre átültetni már nem lehet nagy komplexitású feladat, másrészt a VIATRA rendszerbe számos egyéb metamodellnek megfelelő modell példányt lehet importálni (számos importer már most is rendelkezésre áll), valamint egy új importer megírása se jelenthet megoldhatatlanul nehéz feladatot.

1.2. Megoldandó feladatok

A félév során megoldandó feladatokat a következő részekre lehetett bontani:

- Megismerkedni a modellek összehasonlításának és egyesítésének nehézségeivel
- Tanulmányozni a probléma megoldásához használható algoritmusokat
- Egy konkrét algoritmust portolni / implementálni a VIATRA2 rendszerben
- Tesztelhetőséget lehetővé tenni

2. Háttérismeret

Tekintve, hogy a modellvezérelt szoftverfejlesztés témakörében nem voltam jártas a félév során, kézenfekvőnek tűnt, hogy mielőtt nekiállnék egy saját megoldás elkészítésének, először az ide vonatkozó szakirodalmat áttekintsem. Rövidesen kiderült, hogy modellek összehasonlításával, vagy ennél általánosabban gráfok izomorfiájának vizsgálatával már nagyon sokan foglalkoztak, ezekből igyekeztem a gyakrabban előforduló problémákkal megismerkedni, illetve az ezekre adható megoldásokat megismerni.

A félév során olvasott cikkek, illetve kipróbált szoftverek részletes leírása előtt hasonlítsuk össze először röviden a cikkeket:

Cikk	Típus	Rövid leírás
[2]	probléma leírás	Rávilágít arra, hogy a modellek összehasonlítása mennyivel bonyolultabb, mint a forráskódoké.
[3]	konkrét eredmény	Egy megoldást ad arra, hogy gráfok hasonlóságát hogyan lehet mérni.
[4]	probléma leírás	Szűkítve a témakört, UML modellek összehasonlításának nehézségeit részletezi.
[5]	prezentáció	Nem tartalmaz probléma-leírást vagy megoldást, lehetséges megoldások típusait vázolja.
[6]	konkrét eredmény	Lényegében a [3] cikk későbbi verziója, jelentősen bővítve és továbbgondolva.
[7]	probléma leírás	Felveti, hogy fontos, hogy egy összehasonlító algoritmus bármilyen metamodell fölött működni tudjon.
[8]	probléma leírás	Modellek összehasonlítását a verziókezelők oldaláról közelíti meg.
[11]	konkrét eredmény	A később tárgyalt FMES algoritmushoz hasonló megoldást ismertet.
[13]	konkrét eredmény	Egy modellek kezelésére kifejlesztett verziókezelő.
[15]	konkrét eredmény	A változás-vezérelt összehasonlításra egy lehetséges megoldás.

1. táblázat. A félév során olvasott cikkek összehasonlítása

Valamint ez egyes szoftvereket:

Szoftver	Állapot	Rövid leírás
[1]	Aktívan fejlesztett	Konkrét működő modell-összehasonlító algoritmust nem tartalmaz, de van rá lehetőség
[9]	Aktívan fejelsztett	Működő algoritmust tartalmaz, de csak EMF metamodellel működik
[10]	Aktívan fejlesztett	Objektum-orientált verifikálás céljából kezel gráfokat, gráf-összehasonlító megoldást tartalmaz.
[18]	Karbantartott	2002-ben fejlesztett eszköz, mely séma nélküli XML fájlokat hasonlít össze és egyesít.

2. táblázat. A félév során kipróbált szoftverek összehasonlítása

A fejezet további részében tehát ezeknek a munkáknak a részletes ismertetése következik, két részre csoportosítva: az egyes cikkek részletezését követően az utolsó négy alfejezet konkrét szoftverekkel foglalkozik.

2.1. Ericsson

Az Ericsson egy ipari esettanulmányából [2] kiderül, hogy miért is annyira égető probléma az, hogy sok esetben nincsenek megfelelő eszközök modellek összehasonlításához és egyesítéséhez. A cikkben részletezik, hogy az Ericsson már alkalmazza a modellvezérelt fejlesztés technikáját, és bár összességében hatékonyabbnak tartja mint modellek nélkül fejleszteni, mégis nagy nehézséget jelent nekik, ha valamilyen konfliktust kapnak a modellek verziókezelése során. Ilyenkor egy *merge szobának* keresztelt helységbe terelik a fejlesztőket, ott nagy képernyőkre kivetítik a modellek egyes verzióit, melyt megbeszélést tartanak arról, hogy mi is lehetne az egyesített verzió. Tehát annak ellenére, hogy a technológia növeli a költséghatékonyságot - állítja az Ericsson -, lenne még mit fejlődni, azt gondolják, hogy megoldható lenne, hogy az ilyen modellegyesítési problémák nagyrésze vagy automatizálható lenne, vagy ha szoftveres támogatást kapnának hozzá a fejlesztők akkor az ehhez hasonló konflikusfeloldások jóval egyszerűbbé válnának.

A cikk elolvasása után továbbá világossá válik számunkra, hogy:

- A modellek egyesítésére várhatóan az elkövetkezendő években egyre nagyobb igény lesz.

- A forráskódok egyesítéséhez használt algoritmusok nem használhatóak erre a problémára, leszámítva azon kevés eseteket, mikor a modelleket kézzel írjuk le, nem pedig valamilyen grafikus szerkesztőt használva.

- A probléma azért nehéz, mert az ideális megoldás bármilyen metamodellel rendelkező két modellpéldányon működne, a jelenleg elérhető megoldások pedig csak egyadott metamodellre adnak használható eredményt.

A konzultáción arra a következtetésre jutottunk, hogy alapvetően két megközelítés lehetne sikeres:

- Figyelni, hogy milyen változtatásokat végeztek a fejlesztők (pl. refactoring, Foo osztály átnevezése Bar-ra), majd az egyik modellen végrehajtani a másik modellen elvégzett műveletek sorozatát, így kapva az egyesített eredményt, vagy

- Csak a kezdeti és végállapotot venni figyelembe pillanatkép (snapshot) jelleggel, és arra egy 3-way merge megoldást készíteni.

Mindkét megoldásnak megvannak a maga előnyei és hátrányai.

Az első megoldás előnye, hogyha például egy adott osztály átnevezése 100 sor kód módosításával jár, a művelet tárolása (az eredményezett változtatások tárolása helyett) sokkal kompaktabb változáslistát eredményez. Hátránya, hogy nem feltétlenül felcserélhetőek az egyes műveletek. Maradva az osztály átnevezési problémánál, ha A fejlesztő még a régi nevével hivatkozik egy hozzáadott kódsorban az osztályra, akkor azt az előtt kell alkalmazni, hogy B azt átnevezte. Ha A is nevezett át osztályokat, és B is adott hozzá kódsorokat melyek még a régi nevével hivatkoznak az osztályokra, akkor egy olyan triviális megoldás, mely csak egymásután egyszer A, aztán B (vagy fordítva) műveleteit alkalmazza nem vezet megoldásra. Ha analógiát keresünk a forráskódokat verziózó rendszerekkel, a Darcs verziókezelő ilyen algoritmussal rendelkezik. (Ott ezt a problémát úgy oldják meg, hogy az egyes változástípusok mögött precíz matematikai szemantika van, cserébe viszont a gyakorlatban ez nem mindig ad elfogadható gyorsaságú eredményt az algoritmus nagy komplexitása miatt.)

A második megoldásnak a legszembetűnőbb előnye, hogy nem érzékeny a változtatások számára, hiszen csak az első és a végső állapotot veszi figyelembe. Továbbá ez az egyetlen lehetőségünk ha gyorsan szeretnénk az összehasonlítást elvégezni és a modell szerkesztő nem támogatja az operációk rögzítését. Természetesen cserébe ilyenkor gondjaink lehetnek abban az esetben ha a refactoring mértéke nem elhanyagolható az egyéb változtatásokhoz viszonyítva. (Forráskód kezelésénél a Git rendszer alapértelmezett egyesítő algoritmusa ilyen: olyankor is adhat konfliktust mikor az operációk rögzítésével az egyesítés automatizálható lenne, viszont az egyesítő algoritmus gyorsaságával szemmel láthatólag nincsenek gondok.)

Összességében tehát megállapíthatjuk, hogy minkét technika lehet sikeres, nem érdemes egyiket se elvetni.

2.2. Gráfok hasonlóságának mérése

Zager és Verghese [3] cikkéből kiderül, hogy a modellek összehasonlításának problémája megfeleltethető gráfok összehasonlításának [1]. Ehhez a matematikának széles fegyvertára van, ugyanakkor heurisztikák nélkül a probléma túl nehéz ahhoz, hogy a gyakorlatban elfogadható idejű eredményt kapjunk. Egy heurisztika lehet egy iteratív algoritmus, melyben egy G_a és G_b gráfban lévő pont akkor hasonló, ha a szomszédaik hasonlóak, az élek pedig akkor ha a forrás/cél pontok hasonlóak.

A cikk részletesen foglalkozik azzal, hogy hogyan tudjuk megtalálni, hogy két gráfban hol vannak hasonló részek (similarity). A részgráfok esetén tervezési döntés, hogy a részben hasonlókkal vagy csak az izomorfakkal akarunk foglalkozni a párosítás kialakítása során (matching).

2.3. UML modellek hasonlósága

Ohst, Welle és Kelter [4] cikke konkrétan UML modellekkel foglalkozik. Kiemeli annak fontosságát, hogy ha két modellt összehasonlítunk, akkor az érdektelen adatokat figyelmen kívül kell hagyni, például az egyes modell elemek pozícióját (layout info). Foglalkozik a vizuális megjelenés problémájával, erre mutat pozitív és negatív példát is. Kiemeli, hogy lényeges különbség lehet modellek között, hogy van-e az elemeknek egyedi azonosítójuk (UUID), ennek hiányában a probléma perfekt megoldása sajnálatos módon NP-teljes. Hivatkozik a Rational Rose termékre, melyben egyfelől van modell összehasonlítási / egyesítési funkcionalitás, másrészt viszont az egyesítés során előkerül a modell konkrét XML-alapú szintaxisa, ennek ismerete nélkül aligha tudunk sikeres egyesítéseket véghezvinni.

Az eddigi cikkekből a következő szempontok derültek ki az összehasonlító algoritmusokat illetően:

- Történettel (history) foglalkozó / nem foglalkozó algoritmus
- Egyedi azonosítást lehetővé tevő elem létezése / nem létezése
- Az algoritmus metamodell-specifikus-e
- A fejlesztő grafikus felületet kap-e vagy a konkrét szintaxist is ismernie kell

[1] Feltéve, hogyha a pillanatkép-jellegű hozzáállást követjük.

2.4. Gráfok hasonlóságainak típusai

Gráfok hasonlóságának alapkérdéseivel foglalkozik Zager és Verghese prezentációja [5], melyben a hasonlóság következő fajtáit határozzák meg:

- Izomorfia (csak igen/nem a válasz)
- Szerkesztési távolság: hány lépés kell ahhoz, hogy az egyik gráfból eljussunk a másikba.
- Maximális / minimális mérete a részgráfoknak / szupergráfnak

2.5. Gráfok párosítása

Zager MSc dolgozata [6] a következő újdonságokkal szolgál:

- Úgy is neki lehet állni a problémának, hogy a gráfokat különféle módokon mátrixokként ábrázoljuk és ezeket hasonlítjuk össze.
- A feljebb említett iteratív hasonlóság-kereső módszerét itt részletesen kifejti.

2.6. Szakterület-specifikus összehasonlítás

Lin, Gray és Jouault cikke [7] azt hangsúlyozza, hogy fontos, hogy az algoritmus metamodelltől független legyen, a cikkben ismertetett *DSMDiff* tetszóleges domain-specifikus nyelven íródott modellpélányokat kíván összehasonlítani. Magáról a DSMDiff-ről nagyon keveset sikerült kideríteni, pedig a cikk szerint ez nem csak egy terv, hanem implementálták is.

2.7. Transzformációk tesztelése és verziókezelés

Lin, Zhang és Gray cikke [8] problémafelvetési céllal íródott, nem tartalmaz konkrét megoldásokat. A 2004-es írásban felvetett problémák egy rész már megoldódott, mivel részben a CVS problémái voltak ezek, melyek közül számosat az SVN már ma is megold, például a projekt-szintű verziózást a fájl-szintű helyett.

A fenti probléma-leírások és megoldás-vázlatok alapján az is napirendre került, hogy mely már létező projekteket lenne érdemes kipróbálni ebben a témakörben.

A két első jelölt:

- EMF Compare [9]
- Groove [10]

A probléma sokadik megfogalmazása tehát: az egyesítő algoritmus jó lenne, ha nem egyetlen metamodell fölött tudna működni, cserébe viszont ha semmit se tudunk a metamodellről akkor nem nagyon lehet jól egyesíteni. Erre megoldás lehet, ha a metamodell információi is az algoritmus inputját képezik a két modellpélány mellett. Egy másik megoldás lehet, ha a metamodell információi alapján tudunk kódot generálni, és így tetszőleges metamodell számára könnyen elkészíthető a működő algoritmus-implementáció.

Ezen kívül még probléma, hogy az algoritmusnak nagy elemszámú modelleket is jól kell tudnia kezelnie, a jelenlegi megoldások nagy része néhány ezer elem felett már nem végez gyakorlatban megengedhető időn belül.

2.8. Változás-vezérelt összehasonlítás

A tanszéken született cikk [15] tudását végül nem használtam fel a későbbiekben a félév során, mivel nem változás-vezérelt algoritmust implementáltam.

2.9. X-Diff

Olyan gráfok összehasonlítását végző algoritmusokat kerestem, melyek fák (körmentes gráfok). A szakirodalom ilyen téren már kisebb mértékben gazdag, pedig számos metamodell-csoport (például az összes EMF által kezelt) szigorú tartalmazási hierarchia szerint épül fel, tehát faként reprezentálható.

Az egyik legérdekesebb Wang, DeWitt és Cai cikke [11], melyben az *X-Diff* algoritmust vázolják. Az algoritmust alapötlete nagyon hasonlít a később tárgyalt *FMES* algoritmushoz, viszont ehhez nem találtam nyilvánosan elérhető működő kódot.

2.10. VIATRA2

A következő lépés a VIATRA2 rendszerrel való ismerkedés volt. Mivel egy hallgató már korábban foglalkozott modell összehasonlítási témával, így először az ö algoritmusát [12] próbáltam ki. Ez a keretrendszerrel való ismerkedéshez kiváló feladat volt, az algoritmus viszont az én inputomra lefagyott (*NullPointerException*).

Ezt követően a VIATRA Java API-jával ismerkedtem, ennek keretében elkészült egy egyszerű (hello world) model comparator[14], ami csak összehasonlítja az egyes entitások nevét, relációinak számát, és ez alapján tekinti egyezőnek az egyes entitásokat vagy sem.

2.11. Odyssey-VCS

Ezzel párhuzamosan megpróbáltam felvenni a kapcsolatot az *Odyssey-VCS*[13] szerzőivel, melynek célja egy modellek kezelésére kitalált EMF-alapú verziókezelő lenne. Sajnálatos

módon a szoftver első blikkre (out of the box) nem működött, a forráskódot meg többszöri levélváltás után se sikerült végül megkapni, bár határozott elutasítást nem kaptam (kutatási célokra).

2.12. EMF Compare

Utána az EMF Compare-rel ismerkedtem. Az org .eclipse .emf .compare .examples .standalone csomagban van egy parancssorból használható (standalone) példa, ez volt az amit a konzin véletlenül kitöröltünk fordítási hiba miatt (mondván, hogy az csak példa). Mivel ez egy jó belépési pont, letöltöttem újra, megjavítottam a függőségeit (dependency) és így már lefordult. Innen már látszik, hogy az EMF esetén az összehasonlítás két részből áll: először van egy *match* rész, ez találja meg a hasonló részeket - *MatchService.doMatch()*, 2 EObject-et tud összehasonlítani -, majd ezután jön a diffelés, ami a különböző részeket találja meg - *DiffService.doDiff()* ami egy MatchModel-t var, amit a *MatchService* ad vissza -, végül az ez által adott DiffModel-en tud lefutni a merge - *MergeService.merge()*.

Tehát ami először számunkra érdekes lehet, az az, hogy a *MatchService* hogy működik. Nyilván ez is strategy patternnel van megoldva, az *IMatchEngine* interfász implementációjával. Az EMF beépítetten egyetlen implementációt ad, ez pedig a *GenericMatchEngine*. Ez egy remek 2000 soros implementáció, a részletek megismerésére idő hiányában nem vállalkoztam, de egy rövid összefoglaló [16] leírja, hogy milyen metrikákat használ.

Hasonló módon a match modelből a diff egy IDiffEngine-t implementáló osztállyal oldható meg, erre a kódban az implementáció a *GenericDiffEngine*. (Szinten komplexebb, 900 sor.) Végül az utolsó lépés interfésze az *IMerger*, ennek egy implementációja a *DefaultMerger* (350 sor).

Tesztesetek tekintetében statikus modelljeik nincsenek, viszont van egy *EcoreModelUtils .createModel()*, amit lehet paraméterezni, hogy milyen tesztmodelleket generáljon. (Egy egyszerű könyvek-szerzők modell n az n-hez kardinalitású kapcsolattal, az egyes attribútumok értékei véletlenszerűek, de az egyedi azonosítást biztonsító *xmiid* attribútumok stimmelnek, így a matchernek meg kell találnia az egyező elemeket a modellben.) Ezt teszteli a TwoWayContentMatchTest. Ezen kívül van még másik 2 teszt:

TwoWayModelMatchTest: az előző tesztben használt modellel dolgozik ez is, nézi, hogy alapesetben minden elemet össze tud párosítani, aztán módosít egy szerzőt, és várja, hogy azt a szerzőt már ne tudja megtalálni (legyen hozzá *UnMatchElement*), mivel a kettőnek más a könyvlistája. (*TwoWayResourceMatchTest*: hasonló, de resourcematch-et használ.)

Tehát a 3 match típus:

- ContentMatch: komparálás tartalom alapján (testvérek és szülők figyelmen kívül hagyása)

- ModelMatch: normál komparálás (figyelembe veszi az előbb figyelmen kívül hagyott relációkat)
- ResourceMatch: ez EObject-ek helyett Resource-okra fut. A dokumentáció alapján úgy tűnik, hogy a Resource gyakorlatilag egy perzisztens tároló, lehetnek benne EObject-ek, tehát ez algoritmikusan nem nagyon különbözik a fenti 2 match típustól.

Ha interaktív összehasonlítás a cél, akkor a hivatkozás-jegyzékben említett [17] bejegyzés hasznos lehet.

2.13. Groove

A Groove mögött dolgozó gráf-összehasonlító algoritmus leírása[21] újabb ötleteket adhat a saját összehasonlító tervezésénél. A szerző is megjegyzi, hogy gráfok izomorfiáját alapesetben nem lehet polinomidőben vizsgálni. McKay algoritmusát használja, mely a következőt adja: egy gráfon és egy páronként nem-izomorf gráfokat tartalmazó halmazon dolgozik, és azt keresi, hogy a halmazból bármelyik elem izomorf-e a gráffal. Ha nem, akkor értelemszerűen felveszi az elemet a halmazba. Az algoritmus közelítő ötletek segítségével polinom-idejű. Az ötletek a következők:

- A gráf redukálása (izomorfia szempontjából lényegtelen elemek eltávolítása) vizsgálat előtt.
- A gráfok lenyomatának (hash) számítása, így ha a lenyomat-függvény gyors, az izomorfia konstans időben eldönthető.
- A lenyomatot készítő függvény a gráf színezését használja fel építőkőként.

3. A kiválaszatott módszer: FMES

3.1. Diff XML

A DiffXML nevű eszközt azért tartom kiválónak a labor szempontjából, mert kiterjedt tesztesetgyűjteménnyel rendelkezik XML modellekből. Egy rövid kiértékelés alapján az eszköz tudja is amit hirdet magáról: XML modelleket tud összehasonlítani és foltozni (patch), a metamodell figyelembevétele nélkül.

Másik nagy előnye, hogy mivel fákon dolgozik, gyorsabb, mint azon algoritmusok melyek tetszőleges gráfokkal képesek dolgozni. Figyelembe véve, hogy sok modell esetén a metamodell előírja a szigorú tartalmazási hierarchiát (pl. EMF), ez a megkötés sok modellpéldány esetén nem jelent gondot.

Felhasználói szemszögből úgy működik, hogy 2 XML modellt kér bemenetként és kimenetként egy olyan XML modellt ad, mely operációk listáját tartalmazza, mellyel el lehet jutni az első modellből a másodikba. Ezt az eszköz szerkesztési szövegnek (edit script) hívja. Így tehát egy tipikus 3-utas egyesítés (3-way merge) megoldható a következő lépésekkel:

- A közös ős és *A* között végzünk összehasonlítást, ebből lesz egy folt
- A foltot alkalmazzuk *B*-re, a kimenet lesz az egyesített eredmény

A probléma a következő: ugyan jól formázott lesz az eredmény, de nem biztos, hogy a sémának is megfelel (valid).

További gond, hogyha ütközés lenne (például x-et módositjuk egyszer A-ra, másszor B-re), akkor azt nem veszi észre, mivel az operácio csak azt írja, hogy "töröld ki az elemet, írj bele A-t". Így aztán attól függően A v B lesz az eredmény, hogy melyiket hasonlítjuk és melyiket foltozzuk.

Azt is megnéztem, hogy B-t hasonlítva és A-t foltozva a kapott egyesítési eredmény azonos az előbbi módon kapott egyesítési eredménnyel, ami bizakodásra ad okot az háttérben munkálkodó algoritmust tekintve.

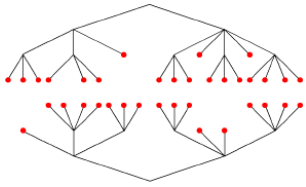
3.2. FMES

A DiffXML eszköz összehasonlító moduljában alapértelmezett esetben egy FMES nevű algoritmus oldja meg a tényleges összehasonlítást.

A formátum, amit kimenetként használ a diffxml és bemenetként a patchxml le van írva a szerző disszertációjában[18], de csak egyszerűen olvasva is meglehetősen intuitív. Ebből következik, hogy a patchxml nem különösebben bonyolult, egyszerűen csak végrehajtja a műveleteket a bemenetről. Az érdekes rész a diffxml.

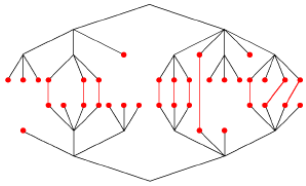
A FMES algoritmusnak a precíz leírása megtalálható hivatkozások között[19]. Az algoritmus alapötlete egy iteratív megoldás, mely a következő módon valósul meg: Definiált egy két csomópontot összehasonlító függvény.

Megkeressük a fa leveleit:



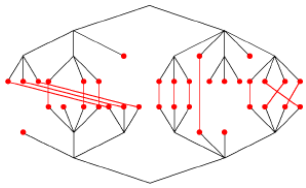
1. ábra. A fa leveleinek keresése

Ezek közül megkeressük azokat, melyek megegyeznek és azonos útvonalon elérhetőek:



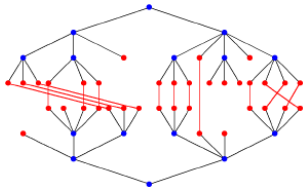
2. ábra. Párosítás egyező úton

Ha ezzel megvagyunk, a maradék levelek között keressünk olyan egyező csomópontokat melyek eltérő útvonalon érhetőek el, de megegyeznek:



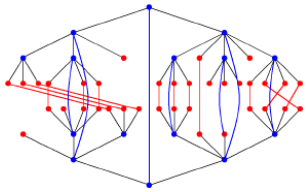
3. ábra. Párosítás eltérő úton

Ezután tekintsük azt a gráfot, melyet a már párosított pontok elhagyásával kapunk, és keressük meg ennek a leveleit:



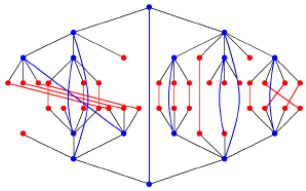
4. ábra. Második iteráció: levél-keresés

Futtassuk erre iteratív módon újfent az algoritmust, tehát keressük meg az azonos úton elérhető egyező leveleket:



5. ábra. Második iteráció: párosítás egyező úton

Majd keressük meg az eltérő úton elérhető egyező leveleket:



6. ábra. Második iteráció: párosítás eltérő úton

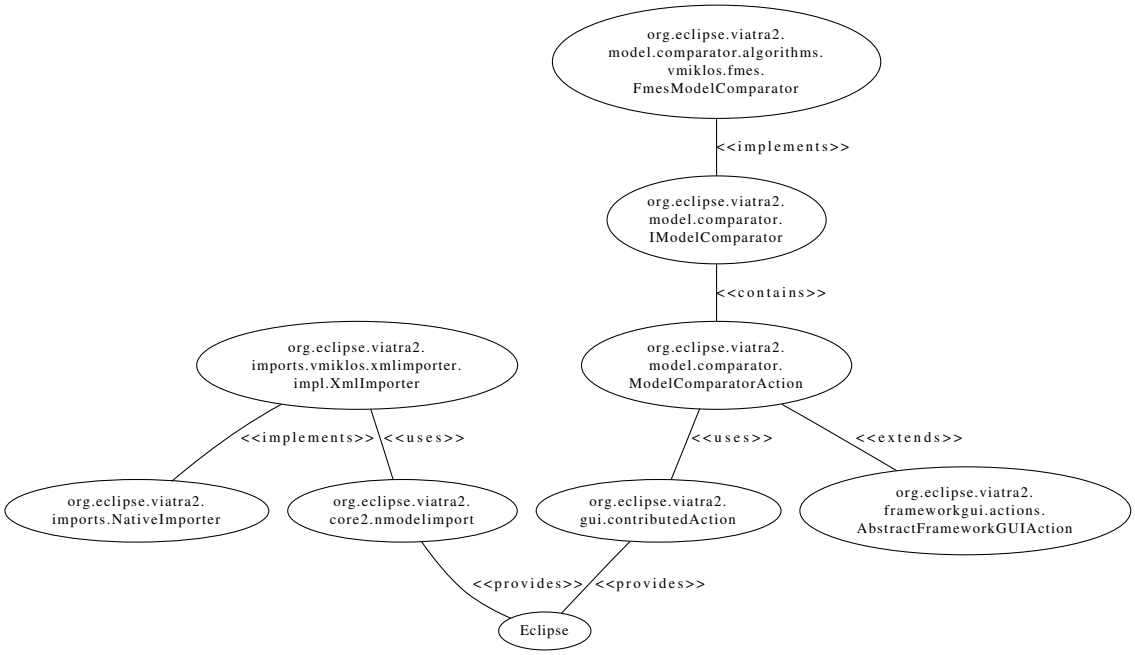
Az iteráció folytatható egészen a gyökérelemig.

4. Implementáció

Ebben a fejezetben két VIATRA plugin implementációját ismertetem: Az xml importert valamint az FMES összehasonlítót.

Az xml importer egy VIATRA importert valósít meg. Ez egy Eclipse plugin, mely egyrészt az *org.eclipse.viatra2.core2.modelimport* kiterjesztési pontot (extension point) használja, másrészt Java szinten az *org.eclipse.viatra2.imports.NativeImporter* interfészt implementálja.

A modell összehasonlító felépítése a következő: egy általános modell összehasonlító Eclipse plugin használja az *org.eclipse.viatra2.gui.contributedAction* kiterjesztési pontot, mely Java szinten az *org.eclipse.viatra2.frameworkgui.actions.AbstractFrameworkGUIAction* osztályból öröklődik, majd ez definiál egy saját *org.eclipse.viatra2.model.comparator.IModelComparator* interfészt, melyet az egyes algoritmusoknak implementálnia kell. Ezt az interfészt valósítja meg az általam implementált *org.eclipse.viatra2.model.comparator. algorithms.vmiklos. fmes.FmesModelComparator* is.



7. ábra. Az implementáció architektúrájának váza

4.1. VIATRA XML importer

Egyszerű importert készítettem a releváns dokumentáció[22] alapján.

A cél az volt, hogy a diffxml teszteseteit be lehessen húzni VIATRA-ba és ne kelljen kézzel felvenni az entitásokat, ezzel el is készültem.

Egyszerű alatt azt kell érteni, hogy csak az *entity* meg *text* típusú csomópontokat figyeli és az alapján IEntity-ket hoz létre a tartalmazási hierarchiát figyelembe véve, ellenben az

attribútumokkal már nem foglalkozik.

Ez a munka az SVN-ben szintén elérhető[23].

4.2. FMES illesztése VIATRA-hoz

Az FMES VIATRA-hoz illesztése (portolása) során az első lépés a a párosítás implementálása volt. A párosítás és a két modell alapján a következő lépés a szerkesztési szöveg előállítása lesz. Ebből először a beillesztő és a törlést biztosító részt készítettem el.

Egy egyszerű modellre a kimenet a következő:

```
origin.a matches working.a
origin.a.b matches working.a.b
origin.a.x unmatched
origin.a.z matches working.a.z
origin.a.b.y matches working.a.b.y

working.a matches origin.a
working.a.X unmatched
working.a.b matches origin.a.b
working.a.z matches origin.a.z
working.a.b.y matches origin.a.b.y

Applying: <insert name="X" parent="a"/>
Applying: <delete node="a.x"/>
```

Az implementáció[20] ezen állapotában a két legfontosabb hiányosság, hogy nincsenek kezelve az áthelyezések, valamint a relációkat nem veszi figyelembe a két csúcsot összehasonlító függvény.

Tekintsük át tehát, hogyan oldható meg az áthelyezések detektálása, és ennek mi az alapötlete:

- Ha van egy ModelElement ami már párosított (tehát van megfelelője a másik modellben), de
- A szülője nem párosított (tehát törlésre fog kerülni), akkor
- Tegyük át a ModelElementet a *szülő(párja(elem))* alá.

Ezen kívül teszteseteket készítettem három tesztesetet:

- Egyszerű esetet ahol csak beillesztés ill. törlés van

- Áthelyezés-detektálást igénylő esetet
- Kombináltat

Kimenet a kombinált esetre:

```
test1a.a matches test2a.a
test1a.a.b matches test2a.a.b
test1a.a.d matches test2a.a.d
test1a.a.e matches test2a.a.e
test1a.a.f unmatched
test1a.a.b.1 unmatched
test1a.a.b.4 matches test2a.a.b.4
test1a.a.b.c matches test2a.a.b.c
test1a.a.b.c.3 matches test2a.a.b.c.3
test1a.a.d.5 matches test2a.a.d.5
test1a.a.e.6 matches test2a.a.e.6
test1a.a.f.7 matches test2a.a.g.7

test2a.a matches test1a.a
test2a.a.9 unmatched
test2a.a.b matches test1a.a.b
test2a.a.d matches test1a.a.d
test2a.a.e matches test1a.a.e
test2a.a.g unmatched
test2a.a.b.4 matches test1a.a.b.4
test2a.a.b.c matches test1a.a.b.c
test2a.a.b.c.3 matches test1a.a.b.c.3
test2a.a.d.5 matches test1a.a.d.5
test2a.a.e.6 matches test1a.a.e.6
test2a.a.g.7 matches test1a.a.f.7

Applying: <insert name="9" parent="a"/>
Applying: <insert name="g" parent="a"/>
Applying: <move name="a.f.7" parent="a.g"/>
Applying: <delete name="a.f"/>
Applying: <delete name="a.b.1"/>
```

4.3. Dokumentáció

Az FMES kód kapcsán felmerült, hogy ezt vagy azt miért úgy csináltam, és a válasz az volt, hogy azért, hogy a diffxml-es fmes kódtól minimálisan térjen csak el a VIATRA felett futó kód. Ennek leginkább akkor volt jelentősége, mikor még az áthelyezések detektálása nem működött. Ez az állapot ottvan SVN-ben (r1360), viszont most kitisztítottam, ami egész pontosan a következőket jelenti:

- Az összes osztály és metódus megjegyzésekkel (javadoc) van ellátva.
- Az olyan metódusok vagy kódrészek melyek "esetleg a jövőben hasznosak lehetnek, de jelenleg nem használtak", törlésre kerültek.
- Az xmlimporter kód esetén nem voltak hosszú használatlan kódrészek, de a dokumentáció hiányzott, ezt elkészítettem hozzá.

Az összehasonlító osztályainak viszonya a következőképpen alakul. A végrehajtás az *Fmes Model Comparator* osztályból indul, ez egy párosítást végez az *Match.easyMatch()* hívással majd létrehozza egy *EditScript* példányt a párosítás eredménye alapján. Az *easyMatch()* a párokat egy *ModelElementPairs* osztályban tárolja, a párosítás során az egyes elemeket a mélységükkel együtt egy *ModelElementDepth* osztályba csomagolja. Ezen osztályok rendezését valósítja meg a *ModelElement DepthComparator* osztály. Fontos még a *Match.compare ModelElements()* metódus, mely az egyes elemek összehasonlítását végzi, és ahol jelenleg a típusellenőrzés hiányzik. Visszatérve a kiinduló osztályunkhoz, az *EditScript.create()* metódus végzi el a tényleges szerkesztő script létrehozását. Ennek során egy változáslistát jelképező *IDelta* interfészt megvalósító *DULDelta* osztály metódusait kell meghívni. A lista építése során szükségünk van egy olyan FIFO-ra, melynél nem csak adott csomópontot, hanem kizárólag annak gyerekeit is hozzá tudjuk adni, ezt valósítja meg a *ModelElementFifo* osztály. A *DULDelta* hívások során (beillesztés, törlés, áthelyezés) az keletkező XML-beli konkrét stringek külön, a *DULConstants* osztályban találhatók meg, valamint minden egyes lépésben a keletkező XML új eleme a standard kimeneten is megjelenik, ezt végzi a *DOMOps* osztály. Ezen kívül már csak a hibakezelést megvalósító *Delta InitialisationException* és *DocumentCreationException* osztályok maradtak.

Ezen kívül elkészítettem a labor végén esedékes prezentáció fóliáit.

5. Tesztelés

A tesztelés módja a következő: Az XML modelleket rá kell húzni a modelltérre (ehhez szükséges a korábban leírt XML importer) és a fájlnévből a kiterjesztés elhagyásával képzett névvel ellátott, legfelső szintű elem alá kerül létrehozásra a modell VIATRA-s reprezentációja. Ha ezt legalább két modellre megtettük, akkor futtatható a modell összehasonlító úgy, hogy a modelltéren jobb gombbal kattintunk, majd kiválasztjuk a *Contributions* majd *Run model comparator* menüpontokat. Az ekkor megjelenő ablakban pontosan két modellt kell kiválasztani, majd az OK gombra kattintva indítható az összehasonlítás.

A korábban leírt egyszerű teszt bemeneteken kívül egyéb tesztelést nem végeztem, összetettebb méréseket végezni véleményem szerint majd akkor lenne érdemes ha funkcionalitás szempontjából az algoritmus már teljes: amíg a metamodell kezelése nem készül el, valamint a relációkat nem vesszük figyelembe elvérható mértékben addig úgy gondolom nincs értelme részletes diagramokat készíteni arról, hogy pontosan mennyi időt igényel adott modelleknek az összehasonlítása. A fejlesztés során használt kis bemenetekre a futási idő elhanyagolható (kevesebb, mint 1 másodperc) volt.

A félév során a tesztelés és dokumentáció készítés céljára használt környezet a következő volt:

E dokumentáció alapjául a félév során vezetett wiki oldalak[24] [25] szolgáltak. A dokumentáció tördelése a a tanszék által küldött fedlap alapján a TEX 3.141592 verziójával készült, Frugalware Linux operációs rendszeren. A munkához az Eclipse 3.5-ös (Galileo) verzióját használtam, míg a VIATRA rendszert SVN-ből, a r1353-at. Erre (és nem egy stabil kiadás használatára) azért volt szükség, mert a félév elején még nem volt elérhető a VIATRA 3.1-es kiadása, az előző kiadás pedig már kifejezetten ősinek számított, a konzulens se javasolta a használatát.

A VIATRA rendszer fejlesztését Eclipse-ből futtatott Eclipse-ben teszteltem (runtime Eclipse), mely a következő alap VIATRA projekteket tartalmazta:

- org.eclipse.viatra2.core2
- org.eclipse.viatra2.editor
- org.eclipse.viatra2.editor.text
- org.eclipse.viatra2.gtasm.interpreter
- org.eclipse.viatra2.gtasm.interpreter.impl
- org.eclipse.viatra2.gtasm.interpreter.term
- org.eclipse.viatra2.gtasm.model

- org.eclipse.viatra2.gtasm.patternmatcher
- org.eclipse.viatra2.gtasm.patternmatcher.impl
- org.eclipse.viatra2.gtasm.patternmatcher.incremental
- org.eclipse.viatra2.gtasm.patternmatcher.incremental.rete
- org.eclipse.viatra2.gtasm.trigger
- org.eclipse.viatra2.gtasm.typing.model
- org.eclipse.viatra2.gui
- org.eclipse.viatra2.help
- org.eclipse.viatra2.imports.uml2.galileo
- org.eclipse.viatra2.imports.vtml
- org.eclipse.viatra2.loaders.vtcl_lpgparser

Valamint természetesen a már korábban említett org.eclipse.viatra2.imports.vmiklos.xml és org.eclipse.viatra2.model.comparator.

6. Jövőbeli lehetőségek

Számos lehetőséget kellett figyelmen kívül hagyni idő hiányában ezek közül a legfontosabbak:

- XML importer: Séma megkövetelése, így a VIATRA-beli entitásokhoz típusok lennének rendelhetőek.
- Csak az összehasonlító került megvalósításra, ennek kimenete alapján egy modell egyesítő eszközt is lehetne készíteni.
- A két entitást összehasonlító elem finomítható lenne.
- Az FMES fákon dolgozik, egyéb gráf-összehasonlító algoritmusok is elérhetőek (pl. Groove-ból).

Hivatkozások

[1] VIATRA2 (VIsual Automated model TRAnsformations), http://wiki.eclipse.org/VIATRA2

[2] Lars Bendix, Pär Emanuelsson: Requirements for Practical Model Merge - An Industrial Perspective

[3] LA Zager, GC Verghese: Graph similarity scoring and matching

[4] D Ohst, M Welle, U Kelter: Differences between versions of UML diagrams

[5] Laura Zager, George Verghese: Graph similarity

[6] Laura Zager: Graph Similarity and Matching

[7] Y Lin, J Gray, F Jouault: DSMDiff - a differentiation tool for domain-specific models

[8] Y Lin, J Zhang, J Gray: Model Comparison - A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development

[9] Eclipse Modeling Framework Compare, http://wiki.eclipse.org/index.php/EMF_Compare

[10] GRaphs for Object-Oriented VErification, http://sourceforge.net/projects/groove/

[11] Yuan Wang, David J. DeWitt, Jin-Yi Cai: X-Diff - An Effective Change Detection Algorithm for XML Documents

[12] VIATRA2 Model Comparator by Name, https://svn.inf.mit.bme.hu/ viatra/viatra_extra/ R3/mit.bme.hu/ trunk/org.eclipse.viatra2.model.comparator/ src/org/ eclipse/viatra2/ model/comparator/ algorithms/karer/

[13] Hamilton Oliveira, Leonardo Murta, Cláudia Werner: Odyssey-VCS - a Flexible Version Control System for UML Model Elements

[14] Saját kód: VIATRA egyszerű modell összehasonlító, https://svn.inf.mit.bme.hu/ viatra/viatra_extra/ R3/mit.bme.hu/ trunk/org.eclipse.viatra2.model.comparator/ src/org/ eclipse/viatra2/ model/comparator/ algorithms/vmiklos/hello/

[15] Gábor Bergmann, István Ráth, Gergely Varró, Dániel Varró: Change-Driven Model Transformations: Taxonomy and Language

[16] EMF FAQ: GenericMatchEngine leírás, http://wiki.eclipse.org/ EMF_Compare_FAQ #What_kind _of_.22 strategies.22 _use_EMF _compare_.3F

[17] EMF Compare: How to use EMF Compare in Java, http://www.eclipse.org/ forums/ index.php ?t=msg&th=37716 &goto=122763#msg_122763

[18] Adrian Mouat: Tools for comparing and patching XML files, http://prdownloads .sourceforge.net/ diffxml/dissertation.ps ?download

[19] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, Jennifer Widom: Change Detection in Hierarchically Structured Information

[20] Saját kód: FMES VIATRA rendszerre, https://svn.inf.mit.bme.hu/ viatra/viatra_extra/ R3/mit.bme.hu/ trunk/org.eclipse.viatra2.model.comparator/ src/org/ eclipse/viatra2/ model/comparator/ algorithms/vmiklos/fmes/

[21] Arend Rensink: Isomorphism Checking for Symmetry Reduction

[22] Model import and export: Creating model importers, http://wiki.eclipse.org/ VIATRA2/UseCases/ModelExportImport #Creating_model_importers

[23] Saját kód: Egyszerű XML importer, https://svn .inf.mit.bme.hu/ viatra/viatra_extra/ R3/mit.bme.hu/ trunk/org.eclipse.viatra2 .imports.vmiklos.xml/

[24] A labor során vezetett heti összefoglalókat tartalmazó wiki oldal, https:// trac .inf.mit .bme.hu/ eclipse/wiki/ vmiklos_readme

[25] A labor során vezetett részletesebb naplót tartalmazó wiki oldal, https:// trac .inf.mit .bme.hu/ eclipse/wiki/ vmiklos_log