Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatika Tanszék

Szoftvermodellek szöveg alapú összehasonlítása

TDK dolgozat

Készítette

Somogyi Ferenc Attila

Konzulens

Dr. Asztalos Márk

2015. október 26

Tartalomjegyzék

Ábrajegyzék

Kivonat

A szoftverfejlesztés területén egyre nagyobb szerepet kapnak a modellek. Korábban leginkább dokumentációs célokra használták őket, de ma már többek közt felhasználhatók követelmények formális leírására, ill. (fél-)automatizált kódgenerálásra is. A modellek szöveges formában is leírhatók különböző nyelvtanok segítségével. Gyakran egyszerűbb egy modell szöveges reprezentációját szerkeszteni, mint valamilyen grafikus felületen keresztül módosítani azt, különösen akkor, ha nagyméretű modellről van szó. Ez a szerkesztési mód azért is lehet kényelmesebb, mert nagyon hasonlít a hagyományos forráskód alapú fejlesztéshez. Ez persze csak akkor igaz, ha rendelkezésre áll egy jól olvasható, átlátható nyelvtan és egy hatékony szöveges fejlesztői környezet is.

A forráskód alapú fejlesztésnél a csapatmunkát különböző verziókezelő rendszerek támogatják. Ezek a rendszerek képesek a forráskód két eltérő verziója közötti különbségeket felderíteni, illetve ezeket a különbségeket feloldva előállítani az összefésült kimeneti fájlt. A folyamat általában fél-automatizált, vagyis a legtöbb esetben a felhasználó beavatkozása nélkül megy végbe, de előfordulhatnak olyan esetek is, amikor a felhasználót is be kell vonni a folyamatba.

A szöveg alapú összefésülés megvalósítható modellek szöveges reprezentációi esetén is. A különbség a forráskód összehasonlításhoz képest az, hogy modellek szöveges reprezentációitól elvárható, hogy azok helyesek legyenek, mivel nincs értelme hibás modelleket összehasonlítani. Forráskód esetén viszont a kód szemantikai ellenőrzése egy-egy forrásfájl alapján sokszor nem lehetséges, ezért tipikusan nem követelhetjük meg annak helyességét. Modellek esetén tehát, a szövegből felépített szintaxisfa jelentős többletinformációt rejt az összehasonlítás során. Ezáltal egy tisztán szöveg alapú, szintaktikai összehasonlítás helyett egy mélyebb, szemantikai összehasonlítást is el lehet végezni. Modellek szöveges reprezentációi esetén erre az összefésülésre még nem létezik általánosan alkalmazható megoldás.

A dolgozatban ismertetek egy általam kidolgozott megoldást szoftvermodellek szöveges reprezentációinak szintaxisfa alapú összehasonlítására. Módszerem független konkrét modellező környezetektől és – bizonyos megkötések mellett – a szöveges reprezentációt leíró nyelvtantól is. Dolgozatomban továbbá bemutatom az elméletben kidolgozott módszernek egy gyakorlati megvalósítását egy általam írt modell-összefésülő alkalmazás formájában.

Abstract

Models are becoming more important in the area of software development. While they were mainly used for documentation purposes in the past, nowadays they can be used for a wider variety of tasks, including requirements specification and (half-) automated code generation. With the use of a formal grammar, a model can also be described in textual form. It is often more convenient to edit models through this textual form rather than a graphical user interface, especially if the model in question is large in size. The reason for this is that editing source code and editing the textual representation of a model are very similar tasks, provided that there is an easy-to-use Development Environment available for the latter one.

During the course of source code based development, team work is supported by version control systems. These systems identify the differences between two versions of a source code file, and create a merged file by solving all differences between the two input files. This process is usually half-automated, which means that user input is not required in most cases, but there may be some exceptional cases where it is required.

This merging process can also be applied to textual representations of a model. Compared to source code differencing, the difference lies in the fact that it can always be assumed that the models - and therefore their textual representations - are semantically correct, because there is no point in comparing two incorrect models. In the case of source code differencing, most of the time, the semantic correctness of the code cannot be decided based on a single file, therefore it cannot be assumed that the code is correct. Thus, when working with models, the syntax tree built from the textual representation of a model carries a lot more information that can be used during the merging process. Because of this, a deeper semantic analysis can be achieved instead of a syntactic one. There are no existing general solutions to this merging process in the case of textual representations of a model.

In this paper, I am presenting a method which solves the differencing and merging problem of textual representations of a software model. This method is also independent of the modelling environment and - with certain conditions - the formal grammar used to describe the textual representation. I am also presenting a practical use of this method in the form of a model-merging application written by me.

1. Bevezető

Az utóbbi időben a szoftvermodellek egyre nagyobb térnyerése figyelhető meg a szoftverfejlesztés világában. Ennek oka, hogy a modellek többek közt felhasználhatók dokumentációs célokra, követelmények formális leírására, vagy (fél-)automatizált kódgenerálásra is. A modellek népszerűségét az is magyarázza, hogy manapság olyan modellfeldolgozók és modellszerkesztők állnak rendelkezésre, amelyek egyre jobban megoldják a kódgenerálás problémáját, valamint egyszerűen használhatók.

A modelleket gyakran valamilyen grafikus felületen keresztül jelenítjük meg, de lehetőség van azok szöveg alapú szerkesztésére is. Nagyobb modellek esetén a szöveg alapú szerkesztés sokkal hatékonyabbnak és kényelmesebbnek bizonyulhat, mint a grafikus felületen történő szerkesztés. Ez a szöveg alapú szerkesztési mód, valamint az ezt támogató feldolgozó környezetekkel szemben támasztott követelmények párhuzamba vonhatók a hagyományos programozási nyelveken írt alkalmazások forráskódjának a kezelésével. A párhuzam az alábbiakban nyilvánul meg:

- Adott egy jól definiált nyelvtan.
- A kényelmes szerkesztéshez szükség van egy hatékony szerkesztőfelületre is, amely alapvető kényelmi funkciókkal rendelkezik (pl. automatikus kódkiegészítés).
- Fel kell dolgoznunk, és értelmeznünk kell a szöveget. Fel kell ismernünk a szintaktikai és szemantikai hibákat is.
- A szerkesztés szempontjából fontos kényelmi lehetőségeket is biztosítanunk kell (pl. átnevezés támogatása).

Forráskód alapú fejlesztés során a csapatmunkát verziókezelő rendszerek támogatják (pl.: Tortoise SVN [1], Git [2]). Ezek feladata, hogy egy forrásfájl két verziója közötti különbségeket feloldva előállítsanak egy összefésült verziót. Ez a folyamat kiterjeszthető modellek szöveges reprezentációira is, de a feladat speciálisabb a forráskód összehasonlításhoz képest. Arról, hogy ez miben nyilvánul meg, később lesz szó.

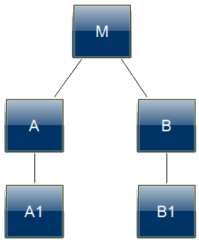
1.1 Problémafelvetés

A továbbiakban modellek szöveges reprezentációiról fogunk beszélni, ezért először nézzük meg, hogy hogyan lehet egy modellt szöveges formában leírni. Mivel a modellt ezen a szövegen keresztül szerkeszteni is szeretnénk, ezért a leképezésnek kétirányúnak kell lennie,

vagyis a szöveget fel is kell tudnunk majd dolgozni, és értelmezni is tudnunk kell azt. Ezért a szöveget valamilyen formális nyelvtannal írjuk le. Feldolgozás során a szövegből először egy absztrakt szintaxisfa (AST [3]) építhető fel, amely már sokkal könnyebben feldolgozható, mint a nyers szöveg.

Nézzük ezt meg egy egyszerű példán keresztül! Tegyük fel, hogy gráf alapú modellekkel dolgozunk, ahol csomópontjaink és éleink vannak, a csomópontokhoz pedig attribútumok rendelhetők. Legyen egy modellünk (M), ami álljon két csomópontból (A és B), a csomópontoknak legyen egy-egy attribútuma (A1 és B1). Mindkét attribútum legyen egy egész szám (vagyis típusuk *int*). Tegyük fel, hogy van egy nyelvtanunk, ami alapján előállítottuk a szöveget a modellből. Feldolgozás során a szövegből először felépítjük a szintaxisfát, ami alapján már sokkal könnyebben beazonosíthatóak a modell elemei, illetve azok tulajdonságai.

```
model M;

node A
{
    int A1;
}
node B
{
    int B1;
}
```



1. ábra: Szintaxisfa felépítése

Az 1. ábra bemutatja, hogy a szintaxisfára való leképzés sokkal pontosabb képet ad a modellről, mint a szöveg, mert a modell elemei könnyen beazonosíthatóak, valamint a modell hierarchia viszonyai (pl. A gyereke A1) is megjelennek a feldolgozás során.

Az ábrán látható szöveg az egyszerűség kedvéért egy (elméleti) mintanyelvtan alapján lett előállítva. A modell szöveges leképzése a következő módon áll elő: megadjuk a modell nevét, majd a csomópontokat a *node* kulcsszóval feltüntetve kapcsos zárójelek között. A csomópontok attribútumait *„típus név;”* formátumban adhatjuk meg. A későbbi példákban is ezt a nyelvtant fogjuk használni, minden alkalommal feltüntetve, ha egy új nyelvi konstrukciót használunk.

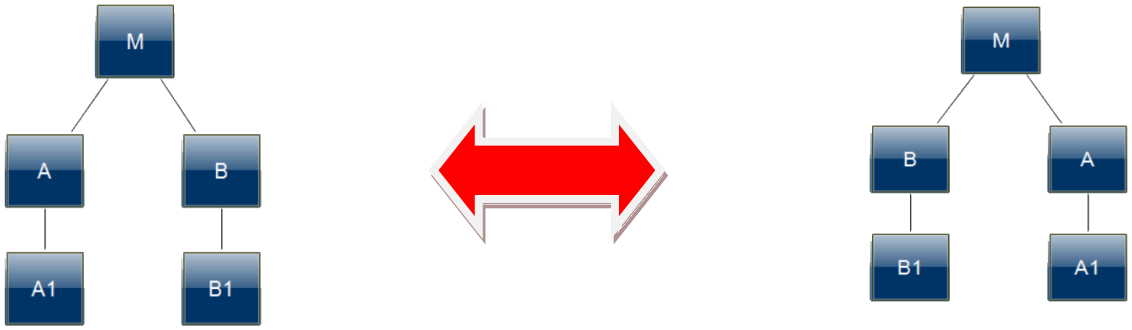
A tisztán szöveg alapú és a szintaxisfa alapú összehasonlítás közötti különbség szemléltetésére szolgál a 2. ábra és a 3. ábra. Két különbséget láthatunk: egyrészt az A és B csomópont sorrendje fel van cserélve, másrészt az A1 attribútum típusa nem egyezik meg (*int* és *string*).

```
model M;

node A
{
    int A1;
}
node B
{
    int B1;
}
```

```
model M;

node B
{
    int B1;
}
node A
{
    string A1;
}
```

2. ábra: Szövegek összehasonlítása

A 2. ábra a tisztán szöveg alapú összehasonlítást ábrázolja. Ekkor az összehasonlítás során nem használható fel a szöveg mögött rejlő tartalom, vagyis csak szintaktikai összehasonlításról beszélhetünk. Az összehasonlítás eredménye a következő: az A csomóponthoz tartozó szöveg az egyik helyen B előtt szerepel, a másik helyen pedig B után. Azt viszont nem tudjuk, hogy mindkét szöveg az A csomópontot hivatott jelképezni, vagyis azt sem tudjuk felismerni, hogy A helyet cserélt B-vel, mert csak szöveges összehasonlítást végzünk. Éppen ezért azt sem tudjuk felismerni, hogy A1 típusa az, ami megváltozott, mert az A-hoz tartozó két szöveg között nem látjuk a kapcsolatot.



3. ábra: Szintaxisfák összehasonlítása

A 3. ábra a szintaxisfa alapú összehasonlítás bemutatására szolgál. Ha a két fát hasonlítjuk össze, akkor rögtön látszik, hogy két különbség is van a két szöveg között. A és B sorrendjének változása a fa felépítéséből következik. A1 típusának eltérése akkor derül ki, ha összehasonlítjuk a két A1-et reprezentáló részfához tartozó szöveget *(„int A1;”* és *„string A1;”).* Tehát a szöveg alapú összehasonlítással ellentétben pontosan és jól elkülönítve tudjuk kezelni a különbségeket. Ez az összehasonlítás szempontjából egy rendkívül előnyös tulajdonság.

Az összehasonlítás és összefésülés során tehát a szintaxisfa alapú összehasonlítás jelentős előnyökkel jár a tisztán szöveg alapú összehasonlításhoz képest. Ki tudjuk használni azt, hogy a szöveges reprezentációkból épített két szintaxisfa elemei megfeleltethetőek egymásnak, ezáltal az összehasonlítás sokkal pontosabb lesz.

1.2 A kitűzött célok

A probléma egy modell két szöveges reprezentációja esetén az összefésült szöveg előállítása, ami a két bemenet közötti összes különbség feloldását tartalmazza. Az összehasonlítás sokkal hatékonyabb és pontosabb lesz, ha a szövegekből felépített szintaxisfákat használjuk az összehasonlítás során, a tisztán szöveg alapú összehasonlítás helyett. A kitűzött feladat az, hogy egy módszert adjunk ennek a problémának a megoldására. A cél az, hogy mindezt minél hatékonyabban, általánosabban és pontosabban megoldjuk.

A forráskód összehasonlításhoz képest egy fontos különbség, hogy a modell szöveges reprezentációi esetén megkövetelhetjük a szövegek szintaktikai és szemantikai helyességét. A szövegek szintaktikai és szemantikai helyességének megkövetelése azzal magyarázható, hogy modelleket csak úgy van értelme összehasonlítani, ha azok helyesek. Ezt alátámasztja, hogy a legtöbb modellező keretrendszer (pl.: Eclipse EMF [4], GME [5], Visual Studio Modeling SDK [6], VMTS [7] stb.) garantálja, hogy a grafikus felületén szerkesztett modellek szemantikailag is helyesek lesznek. Bár most modellek szöveges reprezentációit hasonlítjuk össze, az összehasonlítás akkor is a modellek közötti különbségeket fogja kimutatni, csak szintaxisfa és szöveg szintjén. Egy verziókezelő rendszer esetén sem lenne értelme hibás modellekkel foglalkozni. Érdemes viszont megemlíteni, hogy verziókezelő rendszereknél hagyományos forráskód esetén hibás kód előfordulhat. Még ha egy verziókezelő rendszer csakis szintaktikailag helyes kódot fogadna el (már ez sem jellemző), egy tipikus programozási nyelv esetén nagyon nehéz megmondani egy forrásfájl alapján, hogy szemantikailag is helyes lesz-e a kód (például azért, mert a forráskód több fájlba van szétosztva). Ez az alapvető különbség a kitűzött feladat és a forráskód összehasonlítás között. Mivel megkövetelhetjük a bemeneti szövegek helyességét, ezért garantált, hogy mindig helyes szintaxisfákat fogunk kapni, amiket aztán össze is tudunk hasonlítani. Ezért az összehasonlítás sokkal pontosabb lesz, mivel a szintaxisfák több információt adnak, mint forráskód összehasonlítás esetén. A pontosság a korábban említetteknek megfelelően abban nyilvánul meg, hogy a két szöveg közötti különbségekhez egy szemantikai jelentés is társul, mivel be tudjuk azonosítani a modell elemeit (illetve az őket reprezentáló fákat).

Tehát csak akkor van értelme a kitűzött feladattal foglalkozni, ha a szöveges reprezentációk helyesek. Ekkor mindig felépíthetőek belőlük a helyes szintaxisfák. A két bemenethez tartozó szintaxisfákat felhasználva beazonosíthatók a modell elemei a szövegben. Ezek a beazonosított elemek nem feltétlenül lesznek azonosak a modell tényleges elemeivel, ez az adott nyelvtantól függ, hogy mennyire pontos a leképezés. Ez azt jelenti, hogy a szövegben nem egy tökéletes másolatot találunk a modell szerkezetéről szöveges formában, hanem a kényelmesebb használhatóság miatt különféle rövidítésekkel és kényelmi funkciókkal is találkozunk, amelyek miatt a szintaxisfa szerkezete jelentősen eltérhet a modell logikai szerkezetétől. Ilyen rövidítésekre és kényelmi funkciókra jó példa a megjegyzések, white space karakterek használata, vagy a modell több elemének egy logikai egységbe való tömörítése. Az elméleti mintanyelvtanunk esetén az egyszerűbb megértés érdekében nem vesszük figyelembe az ilyen kényelmi funkciókat. Fontos megemlíteni, hogy ettől még a szintaxisfa tartalmazza a modell logikai szerkezetét is, csak más formában. A szintaxisfa alapján az összehasonlítás sokkal pontosabb lesz, mintha csak a szövegeket hasonlítanánk össze. Ha a fenti feltételek nem teljesülnének, akkor nem tudnánk kihasználni a nyelvtan által nyújtott lehetőségeket, tehát a probléma ugyanaz lenne, mint forráskód összefésülés esetén, vagyis nem lenne kidolgozható egy általános és hatékonyabb megoldás a feladatra.

Felmerülhet a kérdés, hogy miért nem magukat a modelleket hasonlítjuk össze a szöveges reprezentációjuk helyett. A válasz erre az, hogy ebben az esetben sokkal nehezebb egy általános és pontos megoldást adni a problémára. Ezen a területen már vannak kidolgozott megoldások, de ezek vagy az általánosságra, vagy a pontosságra fókuszálnak. Általánosság alatt itt azt értjük, hogy minél többféle konkrét modellre működjön a megoldás, pontosság alatt pedig azt, hogy minden különbséget pontosan felismerünk és beazonosítunk a két modell között. Ha a szöveges reprezentációkat hasonlítjuk össze, akkor viszont az általánosság és pontosság nem zárja ki egymást, ahogy azt később látni fogjuk. Továbbá megmarad a szöveges szerkesztési mód előnye is, miszerint nagyobb modelleket kényelmesebb szöveges formában szerkeszteni.

Dolgozatomban egy olyan általam kidolgozott általános módszert fogok bemutatni modellek szöveges reprezentációinak összehasonlítására és összefésülésére, amely független konkrét modellező környezetektől, valamint – bizonyos követelmények támasztása mellett – a szöveges reprezentációt leíró nyelvtantól is. Ez úgy valósul meg, hogy a rendszer a nyelvtan feldolgozóját egy külön komponensként kapja meg a bemenetén. A feldolgozónak néhány

olyan alapvető funkcionalitást kell nyújtania, amit az összehasonlítás során veszünk majd igénybe. Tehát egy interfészt definiálunk a nyelvtan feldolgozója számára.

A módszer ismertetése után egy általam írt alkalmazást is bemutatok, ami ezen a módszeren alapszik. Az alkalmazás által használt nyelvtan konfigurálható, tehát a program megvalósítja a nyelvtanfüggetlenséget, és konkrét modellező környezettől sem függ. A program a bemenetén kapott, az adott nyelvtanhoz tartozó két szöveget összehasonlítja, és a felmerülő különbségeket automatikusan megpróbálja feloldani. Az automatizált feloldás után a felhasználót is bevonja a folyamatba. Mindehhez egy kényelmes grafikus felhasználói felületet is biztosít.

1.3 A dolgozat tartalmi áttekintése

A felvetett problémára eddig nem létezett általános megoldás, de az összehasonlítás és összefésülés területén számos olyan eredmény van, ami a problémához közel áll. Ezért először (1.4. fejezet) a témához legközelebb álló eredményeket nézzük meg. Ezt követően a kidolgozott módszert ismertetem (2. fejezet), alfejezetenként külön tárgyalva a módszer egyes részeit. Ezután a módszer gyakorlati alkalmazását is bemutatom (3. fejezet) egy általam írt program formájában. Végül összefoglalom a dolgozatot (4. fejezet), valamint megemlítek néhány további kutatási és továbbfejlesztési irányt is.

1.4 Kapcsolódó irodalom

Szövegek összehasonlítása egy hosszú ideje kutatott feladat, aminek a középpontjában az *LCS* (Longest Common Subsequence) probléma áll [8]. Az *LCS* probléma központi kérdése az, hogy hogyan lehet két szöveg leghosszabb közös részét megtalálni. A legtöbb népszerű szöveg összehasonlító algoritmus és program (pl.: *GNU Diff Utils* [9]) az *LCS* probléma megoldásán alapszik. A szöveg összehasonlítás forráskódok esetén elterjedt változata a *Patience Diff* algoritmus [10]. A *Patience Diff* lényege az, hogy az egyszer előforduló leghosszabb közös részekből indul ki, mivel azok forráskód esetén nagy valószínűséggel az egyedi kódrészleteket fogják visszaadni.

Modellek összehasonlításánál a legtöbb megoldás UML modellekkel foglakozik. Ezek a megoldások magukat a modelleket hasonlítják össze, és nem a szöveges reprezentációikat. Ennél a problémánál a legelterjedtebb módszer valamilyen hasonlósági metrikákon alapuló algoritmus használata [11] [12]. Szintén egy népszerű megoldás az *Eclipse EMF* modelljeit összehasonlító *EMF Compare* [13]. Az *EMF Compare* egy skálázható megoldást ad, ahol a

felhasználó szabadon kiterjesztheti az összehasonlítási módszert, de a modelleket nem szöveges formában hasonlítja össze. Mivel mi a modellek szöveges reprezentációival foglalkozunk, az *EMF Compare* által nyújtott módszer a problémát nem oldja meg.

Egy modellek általános összehasonlításával foglalkozó cikk [14] az összehasonlító algoritmusokat négy fő kategóriába sorolta az összehasonlítás alapja szerint. Ezek az alábbiak: (1) statikus azonosító alapú, (2) dinamikus azonosító alapú, (3) hasonlósági metrikákat alkalmazó, valamint (4) nyelv-specifikus algoritmusok. Bár a cikk nem a modellek szöveges reprezentációinak összehasonlításával foglalkozik, a besorolásnak itt is van alapja. A kidolgozott módszer egy dinamikus azonosító alapú algoritmust használ, mivel a nyelvtan feldolgozója segít az összehasonlításban, amit előtte be kell konfigurálni. De elképzelhető egy hasonlósági metrikákon alapuló megoldás is, ami szintén nyelvtan független, nem szükséges konfigurálni, de ennek ára a pontatlanabb összehasonlítás lenne. Ezekről részletesebben a szintaxisfa illesztésről szóló fejezetben (2.2) lesz szó.

Szintaxisfák összehasonlítására is többféle megoldás létezik. Az egyik fő irány a szemantikai alapú összehasonlítás [15]. Ez kihasználja a szintaxisfák sajátosságait, illetve a reprezentált elemek tulajdonságait is. A másik irány a fa átalakítása valamilyen más formába (általában XML), majd az új forma alapján való összehasonlítása [16]. Ekkor viszont a legtöbb esetben elvesznek a szintaxisfák sajátosságai, és szöveg alapú összehasonlítás lesz a feladatból. A szintaxisfa összehasonlítás a legtöbb esetben forráskód analízissel jár együtt. A forráskód összehasonlítás a jelenlegi feladattól eltér, mivel forráskód esetén a szintaxisfák nem minden esetben helyesek, míg modellek esetén megköveteljük, hogy a leírt szöveg helyes legyen, vagyis egy jó modellt reprezentáljon.

A hozzánk legközelebb álló problémával R. v. Rozen és T. v. d. Storm cikke [17] foglalkozik. A probléma ebben az esetben is modellek szöveg alapú összehasonlítása, de a megoldás a viselkedési modellekre (pl. állapotgépek) lett kitalálva. Az összehasonlítás szöveges alapon, egy segéd gráf felhasználásával megy végbe. Az összehasonlítás során a viselkedési modell által leírt procedurális változásokon van a hangsúly. A megoldás nem jól kezeli a szövegben elmozdult elemeket, viszont dinamikus és nyelvtan független. A mi problémánkra ez a megoldás nem jó, mivel tudnunk kell kezelni a sorrend változását is, és nem viselkedési modellekkel dolgozunk.

2. Módszer modellek szöveg alapú összehasonlítására

Ebben a fejezetben a modellek szöveg alapú összehasonlításának problémájára kidolgozott módszerrel ismerkedünk meg. A módszer logikailag elkülöníthető részeit külön alfejezetenként tárgyalva mutatom be. A fejezet csak elméleti síkon mutatja be a módszert, annak egy konkrét gyakorlati megvalósítása a gyakorlati megvalósításról szóló (3.) fejezetben található.

2.1 Nyelvtan független összehasonlítás

A módszer kidolgozásakor egy fontos szempont volt, hogy az általános legyen, vagyis ne függjön konkrét modellező környezettől vagy nyelvtantól. Ezért a modell két szöveges reprezentációja mellett bemenetként szükségünk van egy olyan feldolgozóra, amely bizonyos műveleteket végre tud hajtani. A nyelvtanhoz mindig kell, hogy tartozzon egy saját feldolgozó, hiszen a modell szöveges formába való átalakítását, valamint a szöveg modellé alakítását el kell tudni végezni. Az összehasonlításhoz szükséges műveleteket ez alapján könnyű előállítani. A feldolgozó használatával a módszer nem függ konkrét modellező környezettől vagy nyelvtantól. A műveletek, amiket a feldolgozónak el kell tudnia végezni, az alábbiak:

1. Szöveg feldolgozása, szintaxisfa felépítése
2. Szintaktikai és szemantikai ellenőrzés: helyes-e a szöveg által reprezentált modell?
3. Annak eldöntése, hogy két részfa szemantikailag ugyanazt jelképezi
4. Annak eldöntése, hogy két részfa szövege közötti különbség formai vagy tartalmi

Az első két művelet az összehasonlítás alapjául szolgál, hiszen ahhoz, hogy megkapjuk a szintaxisfát, fel kell dolgoznunk a szöveget, valamint ellenőriznünk kell azt is, hogy a szöveg, illetve az általa reprezentált modell helyes-e. Ezek nélkül nem tudnánk kihasználni a helyes szintaxisfa nyújtotta előnyöket.

A 3. művelet a szintaxisfák illesztéséhez szükséges, míg a 4. műveletnek a konfliktuskezelés során lesz jelentősége. Ezekről később bővebben is lesz szó.

A fent definiált feldolgozó használatával a megoldás általános lesz, vagyis bármilyen modellező környezet és nyelvtan esetén alkalmazható. A feldolgozó használatának hátránya, hogy a nyelvtan feldolgozóját ki kell terjeszteni a megkövetelt műveletekkel, ami implementációs többletköltséggel járhat. Ez a legtöbb esetben nem jelentős, mivel a felsorolt

műveleteket a legtöbb feldolgozó már eleve tartalmazza, legfeljebb nincsenek explicit kivezetve a kimenetükre. A feldolgozó használatának előnye, hogy az összehasonlítás sokkal pontosabb lesz. A megoldás általános marad, de az összehasonlítás mindig testreszabott lesz a feldolgozó használata miatt, hiszen a feldolgozó mindig az adott nyelvtanra fog vonatkozni.

2.2 Szintaxisfa illesztés

Az összehasonlítás legelső lépéseként meg kell állapítanunk, hogy a két szöveges reprezentációból felépített két szintaxisfa egyes részfái miként feleltethetőek meg egymásnak. Ehhez a korábban említett feldolgozó segítségét vesszük igénybe. A feldolgozó két fa esetén eldönti, hogy azok szemantikailag ugyanazt jelentik-e, ami legtöbbször azt jelenti, hogy ugyanazt a modell elemet reprezentálják-e.

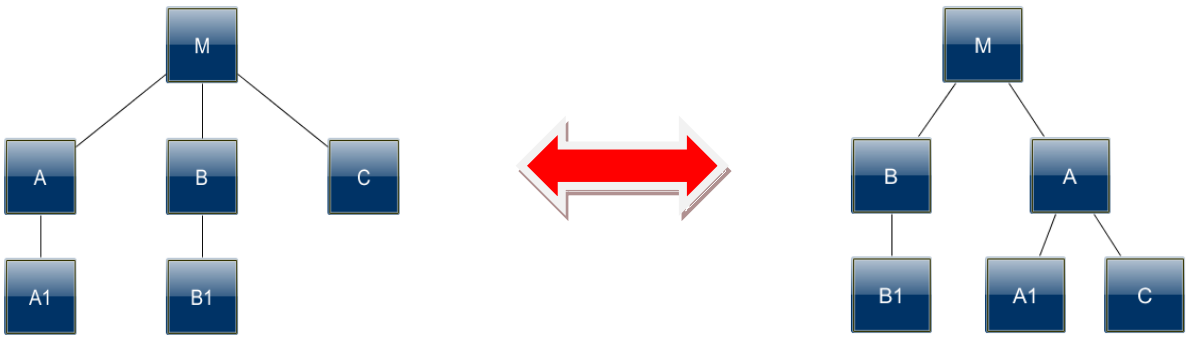
A szintaxisfa illesztő algoritmusnak annyi a dolga, hogy valamelyik szintaxisfát bejárva megkeresse a másik szintaxisfában a részfák párjait. Ezt szintenként teszi meg, így nem kell az egyik fa minden részfáját a másik fa minden részfájával összehasonlítania, tehát az algoritmus gyorsabb lesz. Ha az algoritmus egy részfához nem talál párt, a részfát (illetve annak minden gyerekét) megjelöli, mint új fa, aminek a konfliktusok felismerésénél lesz majd jelentősége.

Előfordulhat az is, hogy egy részfa párja a másik szintaxisfában nem ugyanazon a szinten helyezkedik el. Az algoritmus teljesítménybeli okok miatt csak az azonos szinten lévő részfákat hasonlítja össze, tehát ha a részfa párja nem ezen a szinten található, akkor azt új fának fogja jelölni. Emiatt az illesztés végén az új fákat is össze kell hasonlítani egymással, de ez összességében még mindig kevesebb összehasonlítást igényel, mintha minden részfát összehasonlítanánk minden másik részfával.

Ezek után nézzük meg formálisan leírva a szintaxisfa illesztés algoritmusát!

```
Algoritmus SZINTAXISFA_ILLESZTÉS (Fa1, Fa2)
1  Gyerekek1 ← GYEREKEK(Fa1);          // a fa gyerekeinek listáját adja vissza
2  Gyerekek2 ← GYEREKEK(Fa2);
3
4  for ∀ i ∈ Gyerekek1 do
5    for ∀ j ∈ Gyerekek2 do
6      if ILLESZKEDIK(i, j) then
7        ILLESZT(i, j);                // párok beállítása
8        SZINTAXISFA_ILLESZTÉS(i, j);  // rekurzió
9    end for
10 end for
11
12 NemIllesztettFák1 ← NEM_ILLESZTETT(Fa1);
13 NemIllesztettFák2 ← NEM_ILLESZTETT(Fa2);
14 ILLESZTÉS_PRÓBA(NemIllesztettFák1, NemIllesztettFák2);
15 end
```

Lássuk egy példán keresztül a szintaxisfa illesztés folyamatát, a korábbi példa nyelvtanunknál maradva! Az egyszerűség kedvéért tegyük fel, hogy a nyelvtan olyan modelleket ír le, amelyeknél a csomópontok és attribútumaik neve mindig egyedi kell, hogy legyen. Tehát a feldolgozó két csomópont vagy attribútum fa esetén akkor mondja azt, hogy azok szemantikailag ugyanazt jelentik, vagyis illeszthetők, ha megegyezik a nevük. Ezek után nézzük az alábbi szintaxisfákat, melyeket a 4. ábra ábrázol!



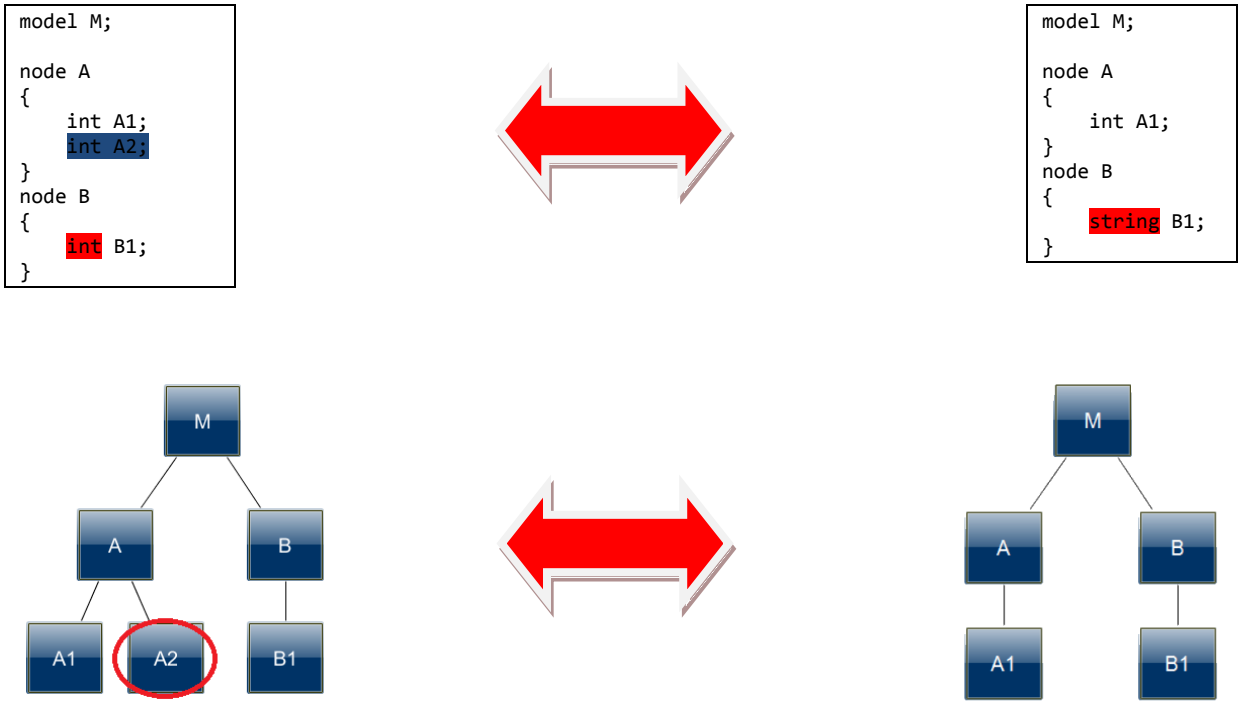
4. ábra: Szintaxisfa illesztés

Az algoritmus végigmegy az egyik szintaxisfa részfáin, legyen ez a bal oldali szintaxisfa. A-t és B-t M gyerekei között megtalálja mindkét fában, tehát összeilleszti őket. A1-et és B1-et hasonló módon szintén összeilleszti. C-t viszont az egyik fában M gyerekei között találja meg,

míg a másik fában A gyerekeként szerepel (pl. a modell tartalmazás relációja miatt). Tehát két fa lesz, aminek az algoritmus nem találja a párját, mégpedig a két C-hez tartozó részfa. Ezek után az algoritmus az összes új fát megpróbálja egymással illeszteni, így végül a C párját is meg fogja találni, tehát ebben az esetben egy nem illesztett (új) fa sem lesz.

2.3 Konfliktusok felismerése

Az összehasonlítás második lépése a két szintaxisfa közötti összes különbség (konfliktus) megkeresése. Konfliktus alatt a két szintaxisfa közötti elemi eltérést értjük. A konfliktusokat ahhoz a részfához rendeljük, ahol az eltérés található. Ennek az összefésülés során lesz jelentősége. A pontos azonosítás érdekében a konfliktus mindig a legbelső részfához tartozik. Ez a mintanyelvtanunk szempontjából annyit jelent, hogyha például egy csomópont attribútumával kapcsolatban találunk egy konfliktust, akkor a konfliktus az attribútumot reprezentáló fához fog tartozni, és nem a csomópontot reprezentáló fához.



5. ábra: Konfliktusok szemléltetése

Az 5. ábra a konfliktus fogalmának szemléltetésére szolgál. Láthatjuk, hogy a két szöveg között két különbség (konfliktus) van, A2 beszúrása az egyik oldalon, valamint B1 típusának változása. A2 a szintaxisfák szerkezetében is jelen van, míg B1 típusváltozása nem jelenik meg a szintaxisfák szintjén. A két konfliktusunk az A2-t, valamint B2-t reprezentáló fához fog tartozni, mivel ezek beljebb vannak a fában, mint A és B.

2.3.1 Konfliktusok feloldása

A konfliktusokat az összefésülés során fel kell majd oldanunk, hogy előállíthassuk az összefésült szöveget. Az a célunk, hogy minden konfliktushoz hozzárendeljünk egy szöveget. Ez lesz a konfliktus feloldása. Az összefésülés során a konfliktushoz tartozó fa szövege helyett a feloldás szövegét használva oldjuk majd fel a konfliktusokat.

Az a cél, hogy az összefésülést minél automatizáltabban végezzük el, tehát a felhasználót a lehető legkevesebbszer kelljen bevonni a folyamatba. Ahhoz, hogy ez megvalósulhasson, előre definiált feloldásokat kell adnunk a konfliktusokra. Ehhez természetesen előre definiált konfliktustípusok is szükségesek. A konfliktustípusokról a feloldások bemutatása után lesz szó. Először nézzük meg, hogy milyen módon rendelhetünk feloldásokat egy konfliktushoz. Egy konfliktushoz mindig csak egy aktív feloldás lehet hozzárendelve. Az aktív feloldás szövege fog megjelenni az összefésült szövegben. Az előre definiált feloldási módok elősegítik az automatizált működést és választási lehetőségeket kínálnak a felhasználó számára.

A konfliktusokhoz rendelhető feloldási módok a következők:

1. **Automatikus feloldás:** Legfeljebb egy lehet belőle. Az összefésülés során automatikusan ezt fogjuk használni, ha az adott konfliktushoz létezik ilyen feloldás.
2. **Alternatív feloldás:** Akár több is lehet belőle. Nem rendeljük hozzá rögtön a konfliktushoz, tehát az összefésülés során nem használjuk automatikusan.
3. **Manuális feloldás:** A felhasználó tetszőleges szöveget adhat meg a konfliktus feloldására. Részletesebben az 2.5 fejezetben lesz róla szó.

2.3.2 Konfliktustípusok

A módszer kidolgozásakor az általánosság mellett a pontosság és a minél automatizáltabb összefésülés is fontos szempontok voltak. Ahhoz, hogy a módszer pontos legyen, minden lehetséges konfliktust fel kell tudni ismernünk. Ahhoz, hogy az összefésülés minél automatizáltabb legyen, minél több konfliktushoz kell automatikus feloldást rendelnünk. Mindebből következik, hogy olyan konfliktustípusokat kell bevezetnünk, amelyek minden lehetséges eltérést lefednek, ami két szintaxisfa között előfordulhat. Továbbá törekednünk kell arra is, hogy a lehető legtöbb ilyen típushoz rendeljünk automatikus feloldást is.

A továbbiakban 3 + 1 konfliktustípust fogok bemutatni, amelyek a fenti kritériumokat teljesítik. Az első három típus lefed minden lehetséges eltérést, míg a negyediknek a gyakorlati felhasználás során lesz fontos szerepe. A konfliktustípusokat az alábbi szempontok szerint fogom jellemezni:

- **Leírás**: általános bemutatás.
- **Felismerés**: annak bemutatása, hogy hogyan ismerhető fel a típus.
- **Feloldás**: előre definiált automatikus és / vagy alternatív feloldások megadása.
- **Példa**: egy egyszerű példa, a már ismert mintanyelvtanunkat használva.

2.3.3 Eltérő szöveg

Leírás

Ez a fajta konfliktus akkor lép fel, ha a szintaxisfa illesztés során két, összeillesztett fához tartozó szöveg nem egyezik meg. Tehát ez a konfliktustípus csak akkor jelenik meg, ha a fák szövegét hasonlítjuk össze, mivel a szintaxisfák szerkezetében ez a konfliktus nem jelenik meg. A szövegek közti különbség formai vagy tartalmi lehet. Formai különbség alatt tipikusan whitespace karaktereket vagy kommenteket értünk, míg a tartalmi különbség a modell szempontjából szemantikai eltérést jelent. De ez persze nyelvtanonként változhat. Annak eldöntése, hogy a különbség formai vagy tartalmi, a megkövetelt feldolgozó feladata, így a különbséget minden nyelvtan esetén fel tudjuk ismerni.

Ez a konfliktustípus tehát a mindkét szintaxisfában reprezentált modell elemek szemantikai változását (pl. attribútum értékének változása), valamint a szövegek formai eltéréseit (pl. white space karakterek, kommentek) reprezentálja. A szintaxisfák szerkezetét ez a típus nem befolyásolja.

Felismerés

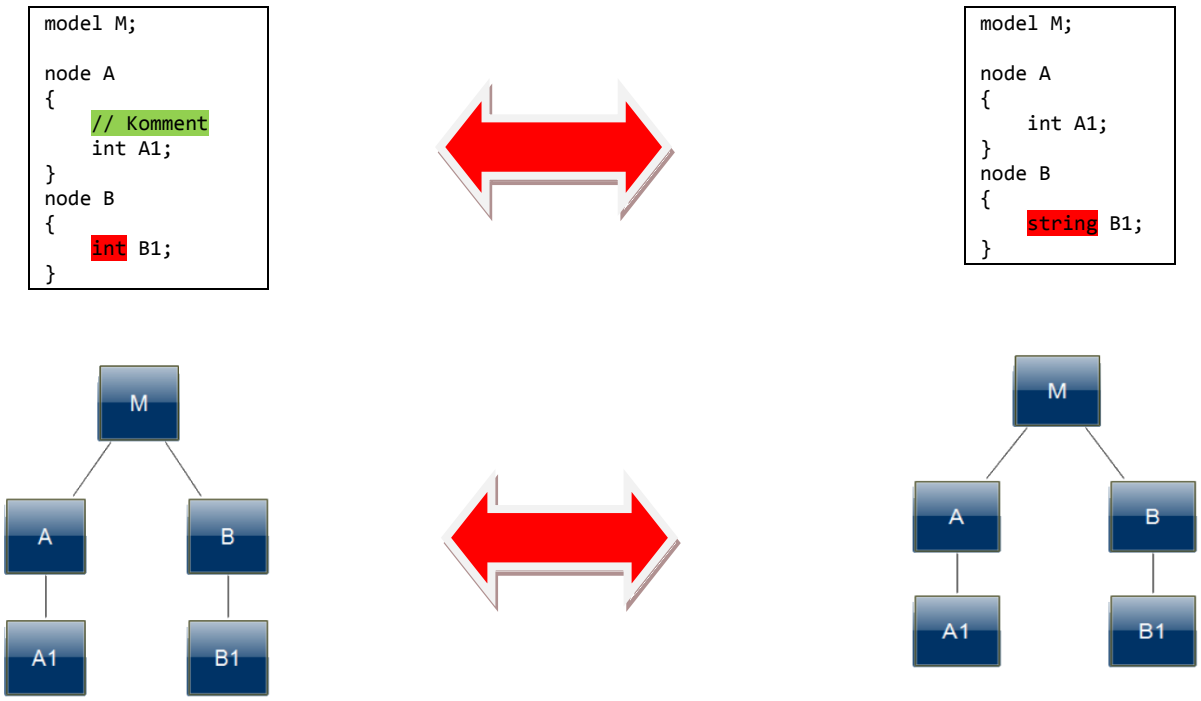
Ahhoz, hogy ezt a konfliktustípust felismerjük, minden, a szintaxisfa illesztés során sikeresen összeillesztett párt egyesével össze kell hasonlítanunk. Minden párra meg kell határoznunk a két fa szövegét, majd ezeket szöveges módon össze kell hasonlítanunk. Ha eltérés van a két szöveg között, meg kell kérdeznünk a feldolgozótól, hogy az eltérés tartalmi vagy formai.

Feloldás

A konfliktushoz rendelt feloldások függenek attól, hogy a két fa szövege között csak formai különbség van, vagy pedig van tartalmi különbség is. Csak akkor rendelhetünk automatikus feloldást a konfliktushoz, ha csak formai különbség van a két fa szövege között.

- **Automatikus feloldás (csak formai különbség):** A hosszabb szöveg. Ennek oka az, hogy a hosszabb szövegben lesznek a tipikus formai elemek (főleg kommentek), amiket általában szeretnénk megtartani. Ha mindkét szövegben vannak formai elemek, és szeretnénk, hogy mindkét szöveg formai elemei megmaradjanak, akkor manuálisan kell megadnunk a feloldást. Ez a kevésbé valószínű eset, ezért a hosszabb szöveg lesz az automatikus feloldás.
- **Alternatív feloldás (csak formai különbség):** A rövidebb szöveg.
- **Automatikus feloldás (van tartalmi különbség):** Ekkor nincs automatikus feloldás, hiszen nem tudhatjuk, hogy melyik eset a valószínűbb, mivel szemantikai eltérésről van szó. Tehát mindenképpen felhasználói beavatkozás szükséges.
- **Alternatív feloldások (van tartalmi különbség)**: Mindkét fa szövege egy-egy alternatív feloldás lesz.

Példa



6. ábra: Eltérő szöveg típusú konfliktus

A 6. ábra jól szemlélteti, hogy az eltérő szöveg típusú konfliktus a részfákhoz tartozó szövegeknél jelenik meg. A példán a két szintaxisfa szerkezete ugyanaz, de a részfákhoz tartozó szövegek különböznek. Nézzük meg, hogyan történik a konfliktusok felismerése ebben az esetben!

Össze kell hasonlítanunk az összes olyan részfát, amihez találtunk párt a szintaxisfa illesztés során. Ebben az esetben az összehasonlítást az összes részfára el kell végezni, mert mindegyik részfához találtunk párt. Először az M-hez tartozó részfát nézzük meg, melynek a szövege mindkét helyen megegyezik, tehát itt nincs konfliktus. Ezután következik az A csomópontot reprezentáló részfa. Itt már van különbség a két szöveg között. A feldolgozó azt mondja, hogy a különbség formai, mivel csak egy kommentről van szó. Tehát ebből egy olyan konfliktus keletkezik, aminek automatikus feloldása a bal oldali (a hosszabb) szöveg, alternatív feloldása pedig a jobb oldali (rövidebb) szöveg. A B fánál is eltérést veszünk észre, de itt azt is észrevesszük, hogy az eltérés a B gyerekének, B1-nek a szövegében keresendő. Tehát a konfliktus B1-hez fog tartozni, és nem B-hez. Mivel ebben az esetben a feldolgozó azt mondja, hogy tartalmi különbségről van szó (az attribútum típusa változott), ezért egy-egy alternatív feloldásként hozzárendeljük a konfliktushoz a két, B1-hez tartozó szöveget.

Tehát a példánkban két konfliktust találtunk: egy formai és egy tartalmi különbséget tartalmazó eltérő szöveg típusú konfliktust. Az egyikhez egy automatikus és egy alternatív, míg a másikhoz két alternatív feloldást rendeltünk hozzá, a fent leírtak alapján.

2.3.4 Új részfa

Leírás

A szintaxisfa illesztés során azokat a részfákat, amelyekhez nem találtunk párt, megjelöltük, mint új fákat. Ezek jelképezik azokat az elemeket, amik a modell egyik változatában (vagyis az egyik szöveges reprezentációban) benne vannak, míg a másikban nincsenek jelen. Ez a típusú konfliktus mindig egy ilyen új részfához tartozik.

Felismerés

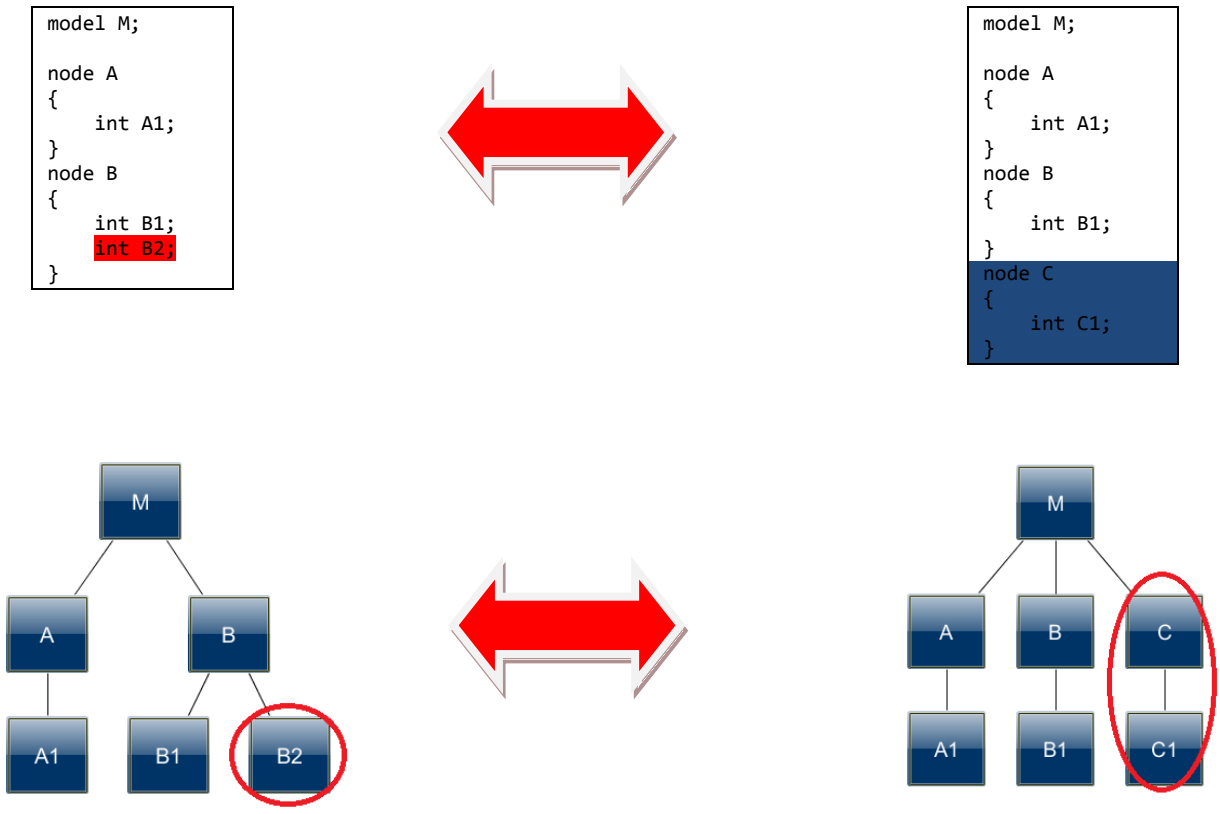
A szintaxisfa illesztés során megjelöltük azokat a részfákat, amikhez nem találtunk párt. Az összes ilyen fához tartozni fog egy ilyen konfliktus.

Feloldás

Ennél a típusnál fontos kérdés, hogy az új fa hol helyezkedik majd el az összefésült szövegben. Erre a problémára több megoldás is létezik (pl. a szöveg végén, a megelőző fa után stb.), az, hogy melyiket használjuk, csak preferencia kérdése, nincs rá általánosan elfogadott megoldás. Itt azt az automatikus megoldást használjuk, hogy az új fa mindig az eredeti szövegben az őt megelőző (legelső nem új) fa után kerül bele az összefésült szövegbe. Ez azt jelenti, hogy az új fa igyekszik megtartani a régi relatív pozícióját, vagyis úgy próbáljuk elhelyezni, hogy az összefésült szövegben is az eredetileg azt megelőző fa után következzen. Ezt az információt (a megelőző fát) célszerű a konfliktus feloldásához rendelni.

- **Automatikus feloldás:** Az új fa szövege. Az a gyakoribb eset, hogy az új fát szeretnénk az összefésült szövegben látni, azzal szemben, hogy nem szeretnénk belerakni.
- **Alternatív feloldás:** Üres szöveg. Vagyis ekkor a fa nem fog belekerülni az összefésült szövegbe. Ez azt jelenti, hogy azért van egy új fa az egyik szintaxisfában, mert azt a másikból kitöröltük, és ezt a kitörlést szeretnénk megtartani.

Példa



7. ábra: Új részfa típusú konfliktus

A szintaxisfa illesztés során a bal oldali fában a B2, míg a jobb oldali fában a C és C1 lettek új faként megjelölve. A B2-t és C-t reprezentáló fához tartozni fog egy-egy új részfa típusú konfliktus. Automatikus feloldásként a fentiek alapján a B2, illetve C szövegét kapják meg, míg alternatív feloldásként egy üres szöveget. Az üres szöveg azt jelenti, hogy a fát nem adjuk majd hozzá az összefésült szöveghez. Mivel C1 a C gyereke, és C is egy új részfa, amihez már tartozik egy konfliktus, ezért C1-hez már nincs értelme új konfliktust definiálni.

2.3.5 Áthelyezés és megváltozott sorrend

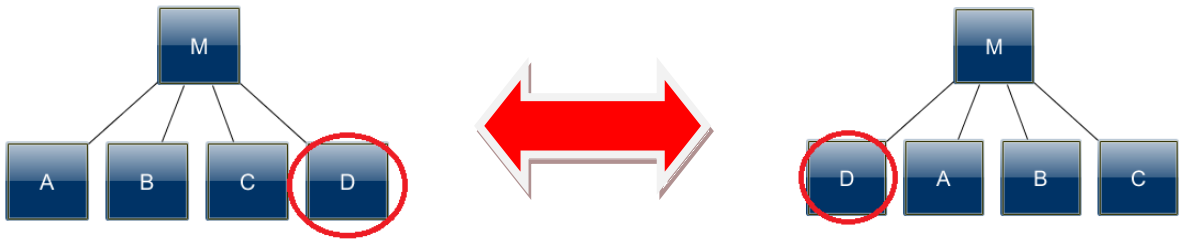
Leírás

A következőkben két, egymáshoz szorosan kapcsolódó konfliktustípust fogok bemutatni, melyek mindketten a részfák sorrendjének megváltozását jelképezik.

Az első sorrenddel kapcsolatos típus az áthelyezés. Áthelyezésről akkor beszélünk, ha egy részfa, melynek van párja, más helyen található meg az egyik szintaxisfában, mint a párja a másikban. Ez általában csak sorrendbeli különbséget jelent, de előfordulhat, hogy szemantikai eltérés is van mögötte. Például ha a modellben létezik tartalmazás reláció, és egy csomópont az egyik fában egy másik csomópont gyerekeként jelenik meg (mint tartalmazás), míg a másik fában nem, az szemantikai eltérést jelent. Mivel ezt nem tudjuk nyelvtan független módon megállapítani, a két esetet (szemantikai és nem szemantikai különbség) együtt kezeljük az áthelyezés típussal.

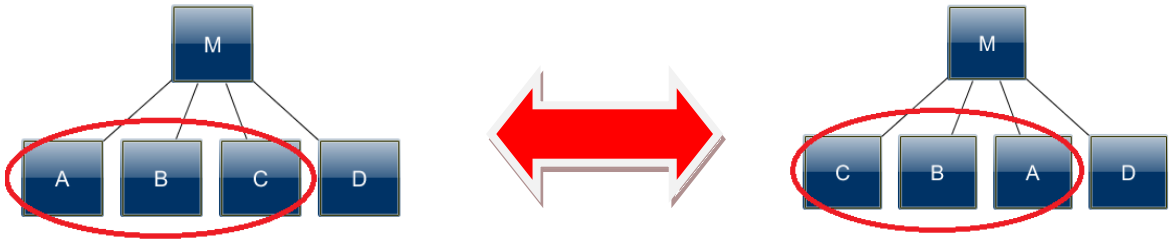
A második, sorrenddel kapcsolatos konfliktustípus a megváltozott sorrend. Bevezetésének az oka a gyakorlati felhasználás kényelmesebbé tétele. Tehát ez a típus a működés szempontjából akár el is hagyható, az összehasonlítás pontos, de sokkal átláthatatlanabb lenne nélküle.

Az áthelyezés és megváltozott sorrend közötti különbségek szemléltetésére szolgál a 8. ábra és a 9. ábra.



8. ábra: Áthelyezés típusú konfliktus

A 8. ábra az áthelyezést szemlélteti. Mivel itt négy fa sorrendjéről van szó, nem használjuk a megváltozott sorrendet. Helyette két áthelyezés konfliktusunk lesz, mivel A és D helye is megváltozott.



9. ábra: Megváltozott sorrend típusú konfliktus

A 9. ábra a megváltozott sorrendet mutatja be. Itt az ABC fák sorrendje változott meg, CBA lett helyette. Ha csak áthelyezésekkel dolgoznánk, akkor két konfliktusunk lenne: C a B elé, valamint A a B mögé kerülne. Megváltozott sorrend használata esetén egy konfliktusba tömöríthetjük az eltérést (ABC ⇔ CBA), ezáltal az összehasonlítás átláthatóbb lesz. A megváltozott sorrend típusú konfliktust két, vagy három egymást követő fa egymáshoz viszonyított sorrendjének megváltozása esetén használjuk. A típust be lehetne vezetni ennél több fára is, de a gyakorlatban háromnál több fa esetén már kevésbé lesz átlátható, illetve a felismerés is bonyolultabbá válik.

Természetesen a sorrend kezelését többféleképpen is meg lehet oldani, például be lehetne vezetni egy olyan konfliktustípust is, ami két, nem egymást követő fa cseréjén alapszik. Ez is megoldaná a fenti esetet, de nincs olyan megoldás, ami bonyolult eseteknél is mindig átlátható lesz. Én azért döntöttem a megváltozott sorrend alkalmazása mellett, mert a két, vagy három fa egymáshoz viszonyított sorrendbeli változása meglehetősen gyakori, és ez a gyakorlati felhasználás során kényelmesebbé teszi az összehasonlítást.

Felismerés

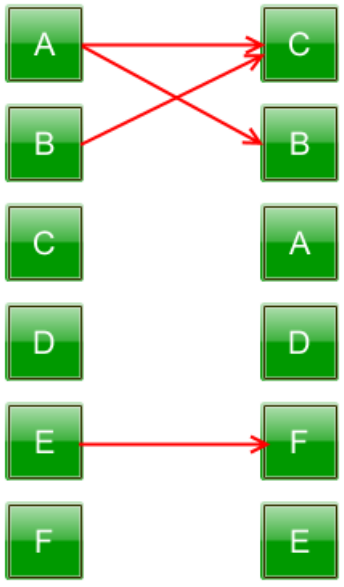
Áthelyezés kétféleképpen fordulhat elő. Az egyik eset, hogy a részfa párja nem ugyanazon a szinten van a másik szintaxisfában (pl. tartalmazás reláció). Ezt az esetet egyszerű felismerni, a szintaxisfák szerkezetének vizsgálatából adódik.

A másik eset az, amikor nem megváltozott sorrendről van szó, vagyis nem két vagy három fa egymáshoz viszonyított sorrendjében történt változás. Tehát először nézzük, hogyan ismerjük fel a megváltozott sorrend típusú konfliktust!

A sorrend vizsgálatának alap ötlete az, hogy az egyik szintaxisfa (mivel az összehasonlítás szimmetrikus, mindegy. hogy melyik) minden részfájára vizsgáljuk meg az ugyanazon a szinten lévő többi részfa pozícióját. Ha egy részfa az egyik szintaxisfában a vizsgált részfa előtt található, míg a másikban a részfa után helyezkedik el, akkor találtunk egy sorrendbeli eltérést. Nézzük meg ezt a folyamatot egy példán keresztül a 10. ábra alapján!

Az egyszerűség kedvéért nem a szokásos szintaxisfa jelölést használom. A bal oldali oszlop a bal szintaxisfa egy szinten található részfáit ábrázolja (ABCDEF), míg a jobb oszlop ugyanezt ábrázolja a jobb oldali szintaxisfában (CBADFE). A probléma bármilyen sorrend jellegű feladatra kiterjeszthető. A piros nyilak az ábrán a sorrendbeli eltéréseket jelölik. Lássuk, hogy hogy kaptuk meg ezeket az eltéréseket!

Menjünk végig a bal oldali részfán, kezdve A-val. A bal oldalon A után B, C, D, E és F állnak, míg a jobb oldalon D, F és E. Tehát A-B és A-C között egy-egy eltérés lesz, mivel a jobb oldalon megelőzik A-t, de a bal oldalon nem. B után a bal oldalon C, D, E és F áll, míg a jobb oldalon A, D, F és E. A B-A eltérést felesleges megjelölnünk, hiszen a szimmetria miatt ugyanazt jelenti, mint A-B, amit már megtaláltunk. Ezért csak a B-C lesz új eltérés. A többi fa esetén ugyanígy járunk el. Még egy eltérést fogunk találni, ez pedig az E-F lesz.

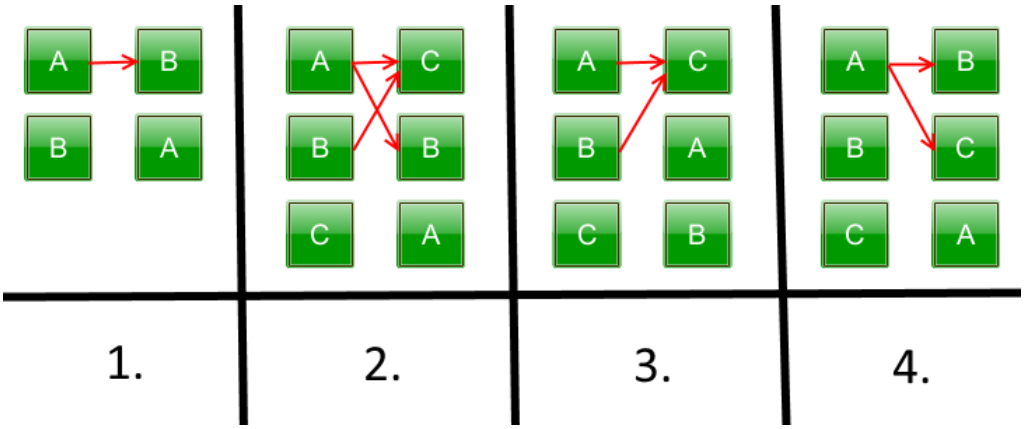


10. ábra: Sorrendbeli eltérések szemléltetése

Ezek után annyi a dolgunk, hogy megállapítsuk, milyen eltérések lépnek fel, amikor két vagy három egymást követő részfa sorrendje változik meg. Ha ezt megállapítottuk, akkor az összehasonlítás során ezeket a minta eltéréseket kell megkeresni, hogy felismerjük a

megváltozott sorrend típusú konfliktusokat. A kimaradt eltérésekből pedig megállapíthatók az áthelyezések is.

A 11. ábra összefoglalja az összes olyan esetet, amikor megváltozott sorrend típusú konfliktusról beszélhetünk.

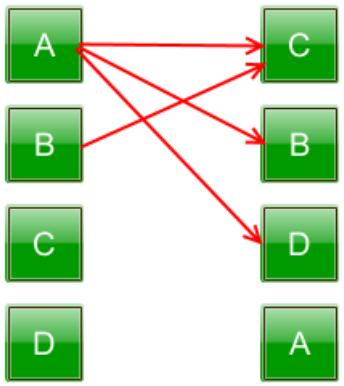


11. ábra: Megváltozott sorrend előfordulása

A felismerés során tehát az alábbi eltérés mintákat kell keresnünk:

1. A-B
2. A-B, A-C, B-C
3. A-C, B-C
4. A-B, A-C

Fontos kiemelni, hogy a fenti minták csak önmagukban érvényesek. Vegyük például a következő eltéréseket: A-B, A-C, B-C, A-D. Ebben látszólag benne van a fentiek közül a 2. minta (A-B, A-C, B-C). A 12. ábra bemutatja, hogy ez valójában mit is jelent.



12. ábra: Minta eltérések szemléltetése

Vagyis ekkor az A-t áthelyeztük (A-B, A-C, A-D eltérések), és a B-C helyet cseréltek egymással (1. minta). Az A áthelyezését onnan ismerjük fel, hogy több, mint két eltérés van, ahol az A szerepel a bal oldalon, tehát az ABC nem lehet három egymást követő részfa. Ezért mindig az áthelyezések felismerésével kezdjük a folyamatot, majd a megmaradt eltéréseket illesztjük a fenti mintákra.

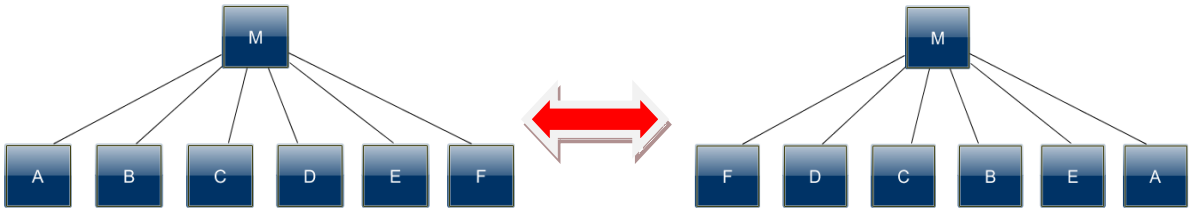
Feloldás

Az új részfa konfliktustípushoz hasonlóan ennél a két típusnál is fontos, hogy tudjuk, hogy az összefésült szövegben hol fog elhelyezkedni a fa (illetve megváltozott sorrend eseten a fák). Tehát a fát (vagy fákat) megelőző fát itt is hozzá kell rendelni a feloldáshoz.

Automatikus feloldás: A bal oldali fa vagy fák szövege. Az, hogy a bal vagy a jobb szöveget vesszük, lényegében mindegy, a lényeg az, hogy az összefésülés automatikusabb lesz, ha a sorrendbeli konfliktusokra tudunk automatikus feloldást adni. Ezért valamelyik szöveget automatikus feloldásként adjuk meg, ami most a bal oldali lesz.

Alternatív feloldás: A jobb oldali fa vagy fák szövege.

Példa



13. ábra: Sorrend kezelése

Nézzük meg, hogy milyen sorrendbeli eltéréseket találunk a két szintaxisfa második szintjén található részfák (ABCDEF ⇔ FDCBEA) között! A felismerést a fentiek alapján elvégezve a bal oldali szintaxisfából kiindulva az alábbi eltéréseket találjuk:

- A-B, A-C, A-D, A-E, A-F
- B-C, B-D, B-F
- C-D, C-F
- D-F
- E-F

Az eltérések között megtaláljuk a B-C, B-D, C-D hármast, ami illeszkedik a felismerésnél ismertetett 2. mintára (A-B, A-C, B-C). Ebből jó eséllyel egy megváltozott sorrend típusú konfliktus lesz, de ehhez előbb meg kell vizsgálnunk a maradék eltéréseket is. Az A-B, A-C, A-D, A-E és A-F eltérések A áthelyezésére, míg az A-F, B-F, C-F, D-F, E-F eltérések F áthelyezésére utalnak. Tehát két áthelyezés (A és F), valamint egy megváltozott sorrend (BCD ⇔ DCB) típusú konfliktusunk lesz.

2.3.6 A konfliktusfelismerő algoritmus

Az elmondottak alapján már fel tudjuk ismerni az összes eltérést, ami a két szintaxisfa, illetve két szöveg között bekövetkezhet. A nem illesztett fák mindegyikéhez egy új részfa típusú konfliktust rendelünk. Az illesztett fák szövegét megvizsgálva keletkeznek az eltérő szöveg típusú konfliktusok. Ezután megvizsgáljuk a sorrendbeli eltéréseket. Az áthelyezések az eltérések és a nem egy szinten található illesztett fák alapján jönnek létre. A megváltozott sorrend típusú konfliktusokat az eltérés minták alapján azonosítjuk be. Minden konfliktushoz megadjuk a feloldásokat is. A konfliktus felismerés folyamata a következő kódrészletben látható.

```
Algoritmus KONFLIKTUSOK_FELISMERÉSE(Fa1, Fa2, Szöveg1, Szöveg2)
1  NemIllesztettFák ← NEM_ILLESZTETT_FÁK(Fa1, Fa2);
2  IllesztettFák ← ILLESZTETT_FÁK(Fa1, Fa2);
3
4  for ∀ f ∈ NemIllesztettFák do
5    Új_RÉSZFA_KONFLIKTUS(f);
6  end for
7
8  for ∀ f ∈ IllesztettFák do
9    Eltérések ← ELTÉRÉSEK_HOZZÁADÁSA(f);
10   if SZÖVEG_KÜLÖNBÖZIK(f, f párja) then
11     BelsőFa ← LEGBELSŐ_ELTÉRŐ_FA(f);
12     ELTÉRŐ_SZÖVEG_KONFLIKTUS(BelsőFa, BelsőFa párja);
13   end if
14 end for
15
16 Minták ← MINTÁK_FELISMERÉSE(Eltérések);
17 ÁTHELYEZÉS_KONFLIKTUSOK(Eltérések, Minták);
18
19 for ∀ m ∈ Minták do
20   EltérőSorrendűFák ← SORREND_NEM_EGYEZIK(m, IllesztettFák);
21   MEGVÁLTOZOTT_SORREND_KONFLIKTUS(EltérőSorrendűFák);
22 end for
23 end
```

2.4 Összefésülés

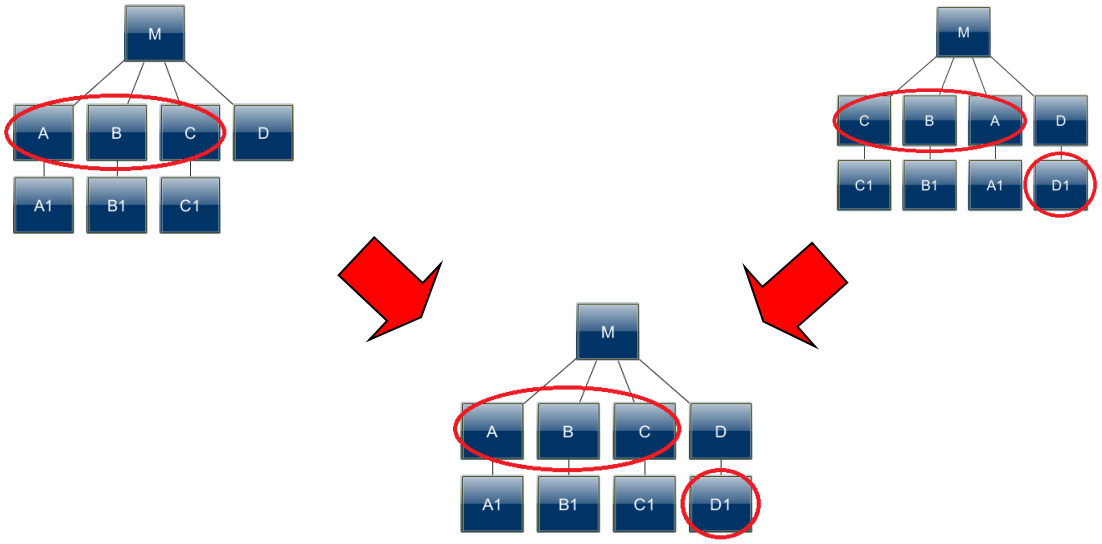
Az összehasonlítás elvégzése után kerül sor az összefésülésre. Az összefésülés célja, hogy az összehasonlítás során felismert konfliktusokat feloldva előálljon az összefésült (kimeneti) szöveg. A továbbiakban az összefésülés folyamatát fogjuk részletesen megnézni.

2.4.1 Az összefésülés folyamata

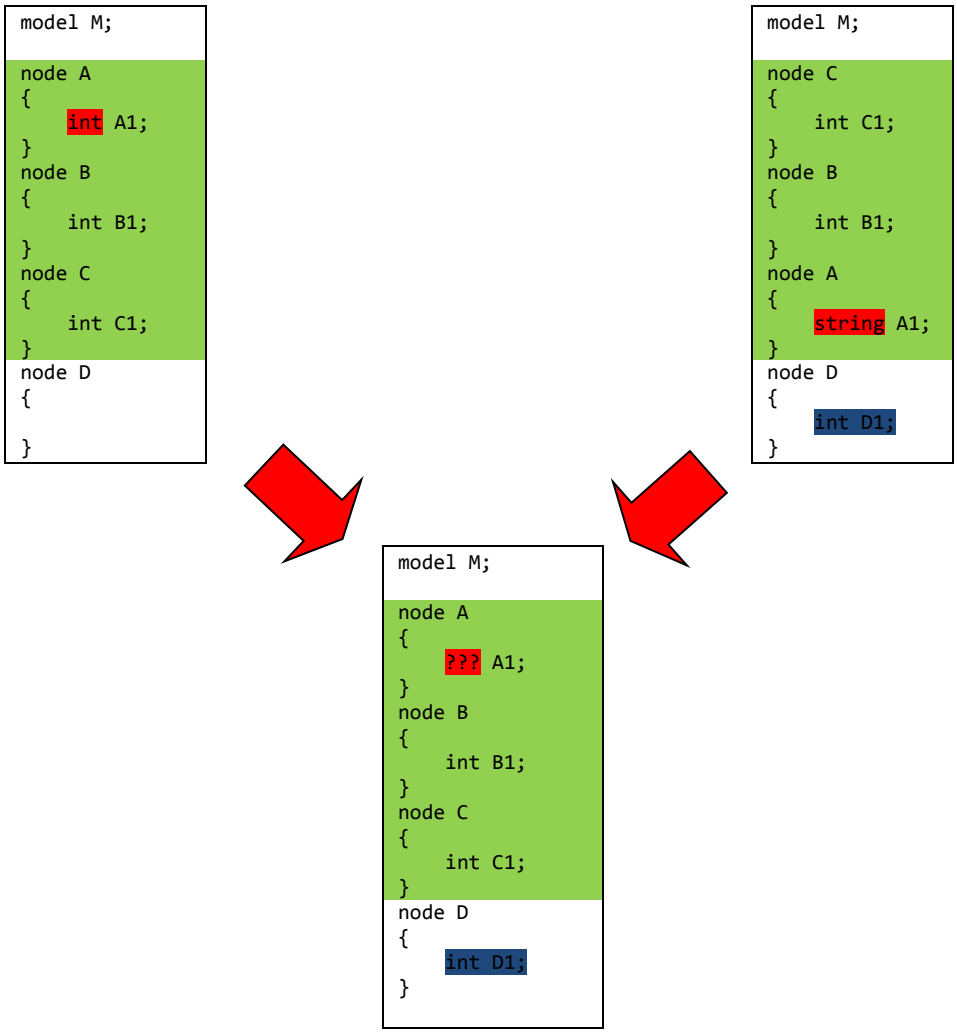
Az összefésült szöveg a konfliktusok feloldása után áll elő. Kiindulásként vegyük a bal oldali szöveget, mivel a sorrenddel kapcsolatos konfliktustípusoknál a bal oldalt vettük automatikus feloldásnak. Ezután vegyük azokat a konfliktusokat, amelyekhez van hozzárendelve automatikus feloldás. Ahogy azt korábban láttuk, minden konfliktus egy adott részfához tartozik. Tehát minden automatikus feloldással rendelkező konfliktus esetén megkeressük az összefésült szövegben a hozzá tartozó részfát, majd annak szövegét lecseréljük az automatikus feloldás szövegére. Olyan új részfa típusú konfliktusoknál, ahol az új részfa a jobb oldali fában található, az azt megelőző fát kell megkeresnünk az összefésült szövegben (ezt megjegyeztük, ahogy arról a konfliktustípus tárgyalásánál szó volt), és utána kell beszúrnunk az új fa szövegét. Tehát megkaptuk az összefésült szöveg automatikusan előállítható részét.

Az összefésülés bemutatása

A14. ábra és a 15. ábra egy összefésülést mutatnak be a szövegek és szintaxisfák alapján is. Tanulmányozzuk tovább az összefésülés folyamatát ezen a példán keresztül!



14. ábra: Szintaxisfák összefésülése



15. ábra: Szövegek összefésülése

Az összehasonlítás során az alábbi konfliktusokat ismertük fel:

1) az ABC ⇔ CBA sorrendet (**megváltozott sorrend**)
2) D1 attribútum hozzáadását D-hez (**új részfa**)
3) A1 típusának (int ⇔ string) eltérését (**eltérő szöveg**)

Az 1. konfliktus esetén a konfliktus típusa miatt az automatikus feloldás a bal oldali sorrendet (ABC) foglalja magában, ezért kezdetben az összefésült szövegben is ez fog megjelenni. A 2. konfliktus esetén az automatikus feloldás az új fa szövegét tartalmazza, melyet a régi szövegben megelőző fa (D) után kell beszúrni az összefésült szövegben. A 3. konfliktushoz nincs automatikus feloldás hozzárendelve, az összefésült szövegben egy jelölő szöveg található helyette, amely azt jelképezi, hogy a felhasználónak muszáj beavatkoznia, vagyis választania egy feloldást.

Az összefésülés egy inkrementális folyamat, ami azt jelenti, hogy az automatikus szöveg előállítása után a felhasználó a konfliktusok feloldásait egyesével megváltoztathatja, így az összefésült szöveg is változni fog. Láthatjuk, hogy a példánk 2. konfliktusához nincs automatikus feloldás hozzárendelve, ezért itt mindenképpen szükség van a felhasználó beavatkozására, hogy válasszon egyet a két alternatív feloldás közül (int vagy string). Az összefésülés addig nem véglegesíthető, amíg van olyan konfliktus, amihez nem tartozik aktív feloldás.

Természetesen a felhasználó azoknak a konfliktusoknak az aktív feloldását is megváltoztathatja, amelyekhez van automatikus feloldás hozzárendelve. Például, ha az ABC sorrend helyett a CBA sorrendet szeretnénk az összefésült szövegben látni, akkor megváltoztathatjuk az 1. konfliktus aktív feloldását, hogy ezt elérjük.

Eddig még nem beszéltünk arról, hogy hogyan határozzuk meg az egyes konfliktusok helyét az összefésült szövegben. Erre szükségünk van ahhoz, hogy a feloldások szövegét a megfelelő helyre tudjuk beszúrni az összefésült szövegben. A legkézenfekvőbb megoldás az, hogy a konfliktusokhoz tartozó részfákat leképezzük az összefésült szövegből felépített szintaxisfában található megfelelő részfákra. Ez a megoldás viszont azért nem jó, mert nem lehetünk benne biztosak, hogy az összefésülés bármely pillanatában felépíthető az összefésült szövegből a szintaxisfa, vagyis hogy a szöveg mindig szemantikailag és szintaktikailag helyes lesz. Tehát a konfliktusok iteratív feloldása miatt nem mindig tudjuk felépíteni az összefésült szövegből a szintaxisfát. Még ha mindig fel is tudnánk építeni a fát, nem biztos, hogy teljesítmény szempontjából előnyös lenne. Ha megnézzük a példánkat, az automatikus feloldások beillesztése után például egy olyan szöveget kapunk, ami szintaktikailag nem helyes. A jelölő szöveg, amely a 3. konfliktus miatt van a szövegben (mivel ahhoz nincs automatikus feloldás), szintaktikailag nem értelmezhető a nyelvtan szempontjából. A helyzetet az is bonyolítja, hogy manuális feloldást is rendelhetünk egy konfliktushoz, aminek helyességében megint csak nem lehetünk biztosak.

Ettől még nem kell elvetnünk a konfliktushoz tartozó fa ötletét, csak ki kell egészítenünk a fa összefésült szövegbeli abszolút pozíciójával. Az abszolút pozíciót a konfliktusok feloldásainak változása ellenére már karban tudjuk tartani, tehát a konfliktushoz tartozó fa helye mindig behatárolható lesz az összefésült szövegben is, csak nem a felépített szintaxisfa alapján találjuk azt meg. Ez a fajta megoldás újabb problémákat vet fel. Mi történik például, ha az előzőeknél maradva az ABC sorrendet CBA-ra változtatjuk? A változtatás magával

vonja az A-hoz tartozó fa, és ezzel együtt az A1-hez tartozó fa változását is a szövegben, tehát a 3. konfliktus pozíciójára hatással van az 1. konfliktus megváltozása. Ezt, és számos hasonló helyzetet le kell tudnunk kezelni ahhoz, hogy az összefésülés működhessen. A továbbiakban az egyes konfliktustípusok egymásra hatásairól lesz szó, vagyis arról, hogy hogyan befolyásolja egy adott típusú konfliktus aktív feloldásának a változása a többi konfliktushoz tartozó fák pozícióit.

2.4.2 Konfliktustípusok egymásra hatásainak vizsgálata

Az előbb tehát láthattuk, hogy nem elég a konfliktushoz tartozó fát vagy fákat meghatározni az összefésült szövegben, mivel a szöveg helyessége nem garantálható, vagyis nem építhető fel belőle minden esetben a teljes szintaxisfa. Ezért a konfliktushoz tartozó fa mellett a fa kezdő- és végpozícióját is számon tartjuk a szövegben. Amikor egy konfliktus aktív feloldása megváltozik, meg kell vizsgálnunk, hogy ez milyen hatással jár a konfliktushoz tartozó fa vagy fák pozíciójára, valamint a többi konfliktusra. Tehát iteratív módon mi „építjük fel" az összefésült szintaxisfát a fák egymáshoz viszonyított sorrendjének, valamint szövegbeli pozíciójuknak a karbantartásával. Érdemes megjegyezni, hogy nem szükséges minden fa helyzetét karbantartani, csak azokra vagyunk kíváncsiak, amelyek valamilyen módon kapcsolódnak egy konfliktushoz. Ez teljesítménybeli okokból előnyös. Fontos kiemelni, hogy az iteratívan „felépített" szintaxisfa természetesen nem lesz egyenlő a nyelvtan feldolgozója által felépített fával, az általunk karbantartott fát csak az összefésülés során használjuk. Ezért az összefésülés végén meg kell kérdezni a feldolgozót, hogy az összefésült szöveg helyes-e. Ezt amúgy is érdemes megtennünk, erről az 2.5 fejezetben bővebben is lesz szó.

Az 1. Táblázat egy összefoglalót tartalmaz arról, hogy mit kell megváltoztatnunk, ha egy adott típusú konfliktus egy másik, adott típusú konfliktusra hat.

A táblázatból látszik, hogy nagyon hasonló esetekről van szó, ami azért előnyös, mert ki tudunk dolgozni egy egységes módszert az egymásra hatások megoldására. A konfliktustípusok ismertetésénél már volt szó volt arról, hogy a megelőző fát (ha van) is tárolnunk kell a konfliktus feloldásában. Erre azért van szükség, hogy tudjuk, hogy a sorrendben hova kell a feloldás szövegét beszúrni. A táblázatban lévő „előző fák" kifejezés erre utal. A megváltozott sorrend esetén az egymásra hatás azért van implicitként megjelölve, mert itt csak a sorrendhez tartozó fák pozíciójával foglalkozunk. Ha egy másik konfliktushoz egy olyan fa tartozik, ami a sorrendben található (vagy egy sorrendben található fának a

gyereke), az egymásra hatás automatikusan meg lesz oldva azzal, hogy a sorrendbeli fák (illetve azok gyerekeinek) pozícióit frissítjük.

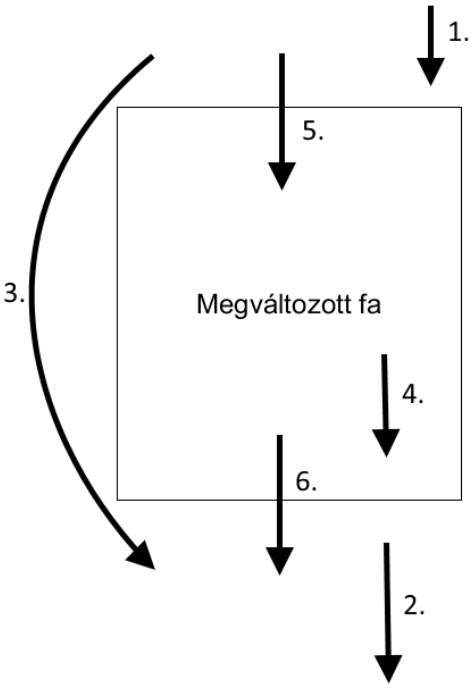
		Ráhatás			
		Eltérő szöveg	Új részfa	Megváltozott sorrend	Áthelyezés
Változás	Eltérő szöveg	fa pozíciója	fa + előző fa pozíciója	fák + előző fák pozíciója	fa + előző fák pozíciója
	Új részfa	fa pozíciója	fa + előző fa pozíciója	fák + előző fák pozíciója	fa + előző fák pozíciója
	Megváltozott sorrend	fa pozíciója (implicit)	fa + előző fa pozíciója (implicit)	fák + előző fák pozíciója (implicit)	fák + előző fák pozíciója (implicit)
	Áthelyezés	fa pozíciója	fa + előző fa pozíciója	fák + előző fák pozíciója	fa + előző fák pozíciója

1. Táblázat: Konfliktusok egymásra hatása

Mikor lép fel egymásra hatás?

Ezek után nézzük meg, hogy mikor hat két fa egymásra, valamint hogy ez pontosan milyen hatással van a fák pozíciójára. Ezt a konfliktusok egymásra hatásainak vizsgálatánál fogjuk felhasználni. Lássuk, hogy két fa hogyan helyezkedhet el egymáshoz képest a szövegben! Ezt a 16. ábra szemlélteti. Az ábra közepén az a fa látható, amelynek a szövege megváltozott. A nyilak egy (másik) fa lehetséges kezdő- és végpozícióját jelölik a szövegben a megváltozott szövegű fához viszonyítva.

Nézzük meg, hogy hogyan változik meg a megváltozott fa pozíciója, amikor egy új aktív feloldást kap, vagyis megváltozik a szövege. A helyzet rendkívül egyszerű, az új szöveg hosszából kell levonnunk a régi szöveg hosszát, és ennyivel kell korrigálnunk a végpozíciót. Például ha a fa szövege „string A1” volt, és „int A1” lett belőle, akkor a végpozícióhoz hozzáadott összeg 7 – 9 = -2 lesz, tehát ebben az esetben csökkent a végpozíció, vagyis rövidebb lett a fához tartozó szöveg. A továbbiakban ezt az értéket **eltolásnak** nevezzük.



16. ábra: Fák egymáshoz viszonyított helyzete a szövegben

Vegyük sorra az adott eseteket és nézzük meg részletesebben, hogy hogyan változik egy fa pozíciója attól függően, hogy hol helyezkedik el a megváltozott szövegű fához képest! Az egyszerűség kedvéért vezessük be az alábbi jelölésrendszert:

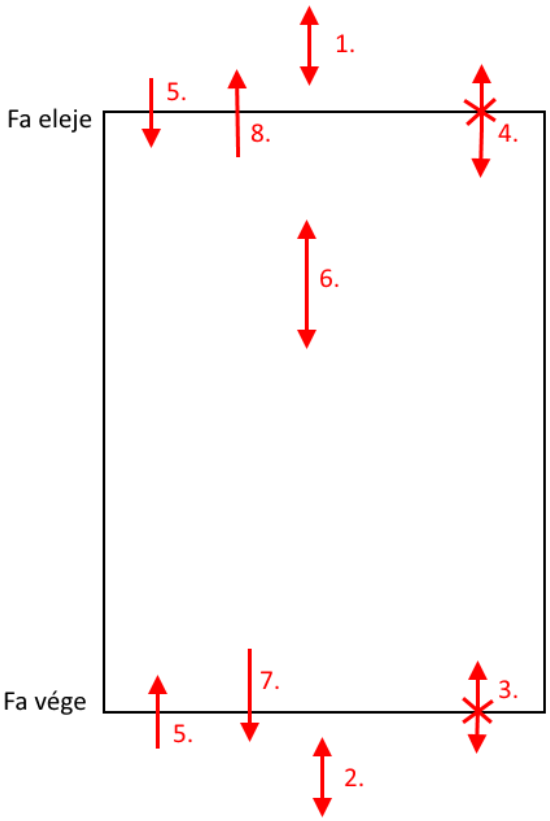
- p_k = a fa kezdőpozíciója
- p_v = fa végpozíciója
- p_k^m = a megváltozott fa kezdőpozíciója
- p_v^m = a megváltozott fa végpozíciója.

Ezek után lásuk a fenti eseteket:

1. $p_k < p_k^m,\ p_v < p_k^m$: A fa a megváltozott fa előtt helyezkedik el. Ekkor a két fa független egymástól, tehát nem következik be változás a pozícióban.
2. $p_k > p_v^m$: A fa a megváltozott fa után helyezkedik el. A kezdő- és végpozíciót módosítani kell az eltolás értékével.
3. $p_k \leq p_k^m,\ p_v > p_v^m$: A fa belsejében helyezkedik el a megváltozott fa. A végpozíciót módosítani kell az eltolás értékével.
4. $p_k \geq p_k^m,\ p_v \leq p_v^m$: A fa a megváltozott fa belsejében helyezkedik el. Meg kell határoznunk, hogy mennyiben érinti a szöveg változása a fát.

5. $p_k \leq p_k^m$, $p_v \leq p_v^m$, $p_v > p_k^m$: A fa a megváltozott fa előtt kezdődik, és benne ér véget. Meg kell határoznunk, hogy mennyiben érinti a szöveg változása a fát.
6. $p_k \geq p_k^m$, $p_v > p_v^m$, $p_k < p_v^m$: A fa a megváltozott fában kezdődik, és utána ér véget. Meg kell határoznunk, hogy mennyiben érinti a szöveg változása a fát.

A 4., 5., és 6. esetnél meg kell határoznunk azt, hogy a szöveg változása érinti- e egyáltalán a fát, és ha igen, akkor milyen mértékben. Ezt a pozíciók és az eltolás segítségével tudjuk kikövetkeztetni, miután eltoltuk a megváltozott fa pozícióját. A folyamat hasonlít a fák egymáshoz viszonyított helyzetének a vizsgálatához. A 17. ábra szemlélteti a lehetséges eseteket.



17. ábra: Szöveg változásának vizsgálata

Ezúttal arra vagyunk kíváncsiak, hogy a szöveg változása (tehát a megváltozott fa változása) hogyan hat a 4., 5., és 6. esetben leírtaknak megfelelően elhelyezkedő többi fára. Tehát az ábra középpontjában ezúttal nem a megváltozott fa áll, hanem az a fa, amire a megváltozott fa ráhatott. Az ábrán a nyilak a szöveg megváltozását jelzik. A lefele mutató nyíl a szöveg hosszának növekedését jelenti, míg a felfele mutató nyíl annak csökkenését. A kétirányú nyíl mindkét esetet lefedi.

Vegyük be az alábbi jelölésrendszert:

- v = a megváltozott szöveg eredeti végpozíciója; vagyis a nyíl kezdőpontja
- v_m = a megváltozott szöveg változás utáni végpozíciója; vagyis a nyíl végpontja
- p_k = a fa kezdőpozíciója
- p_v = fa végpozíciója

Ez alapján lássuk az ábrán látható eseteket:

1. $v < p_k,\ v_m < p_k$: A szöveg változása a fa szövege előtt történt, tehát nem érinti azt.
2. $v > p_v,\ v_m > p_v$: A szöveg változása a fa szövege után történt, tehát nem érinti azt.
3. $v = p_v$: A változás a fa végén kezdődik. Ekkor a fa végpozícióját kell eltolni az eltolás mértékével.
4. $v = p_k$: A változás a fa elején kezdődik. Ekkor a fa kezdőpozícióját kell eltolni az eltolás mértékével.
5. $v < p_k,\ v_m > p_k\ VAGY\ v > p_v,\ v_m < p_v$: A szöveg változása a fa előtt vagy után kezdődik, és benne ér véget. Ez azt jelenti, hogy mind a kezdő, mind a végpozíciót el kell tolnunk az eltolás mértékével.
6. $v > p_k,\ v < p_v,\ v_m \leq p_v,\ v_m \geq p_k$: A szöveg változása a fában kezdődik, és ott is ér véget. Tehát csak a végpozíciót kell módosítanunk.
7. $v > p_k,\ v < p_v,\ v_m > p_v$: A szöveg változása a fában kezdődik, és utána ér véget. Csak a végpozíciót kell módosítanunk.
8. $v > p_k,\ v < p_v,\ v_m < p_k$: A szöveg változása a fában kezdődik, és előtte ér véget. Csak a kezdőpozíciót kell módosítanunk.

Egymásra hatások felismerése és megoldása

A fentiek segítségével már eldönthető, hogy két konfliktustípus között mikor lép fel egymásra hatás. Lássuk, hogy az egyes konfliktustípusok aktív feloldásának változása esetén hogyan történik az egymásra hatások felismerése és megoldása.

- **Eltérő szöveg:** A előbb ismertetett eljárást használjuk, olyan fákat keresünk, amik a fenti 6 (leszámítva a triviális esetet, 5) eset közül valamelyikre illeszkednek, és egy másik konfliktushoz tartoznak. Ebbe beletartoznak az előző fák is, nemcsak azok, amelyekre az adott konfliktus vonatkozik. A pozíció eltolását minden érintett fa gyerekeire is elvégezzük.

- **Új részfa:** Ugyanaz a folyamat, mint eltérő szöveg esetén, azzal a kitétellel, hogy az iteratívan felépített szintaxisfát is módosítanunk kell.
- **Megváltozott sorrend:** Ebben az esetben csak a sorrendhez tartozó fák helyzete változik meg. Itt nemcsak a pozíciót kell frissítenünk, hanem az iteratívan felépített szintaxisfánk szerkezetén is módosítanunk kell, hogy tükrözze az új sorrendet.
- **Áthelyezés:** Áthelyezésnél az áthelyezett fa régi és új helye közötti összes olyan fa pozícióját módosítanunk kell, amelyek egy másik konfliktushoz tartoznak. Ha a fát előrébb helyeztük a szövegben, akkor a közbülső fák kezdő- és végpozícióját növeljük az áthelyezett fához tartozó szöveg hosszával. Ha a fát hátrébb helyeztük, akkor pedig csökkentjük. A végén az iteratívan felépített szintaxisfát is módosítjuk.

Az algoritmus működésének szemléltetése

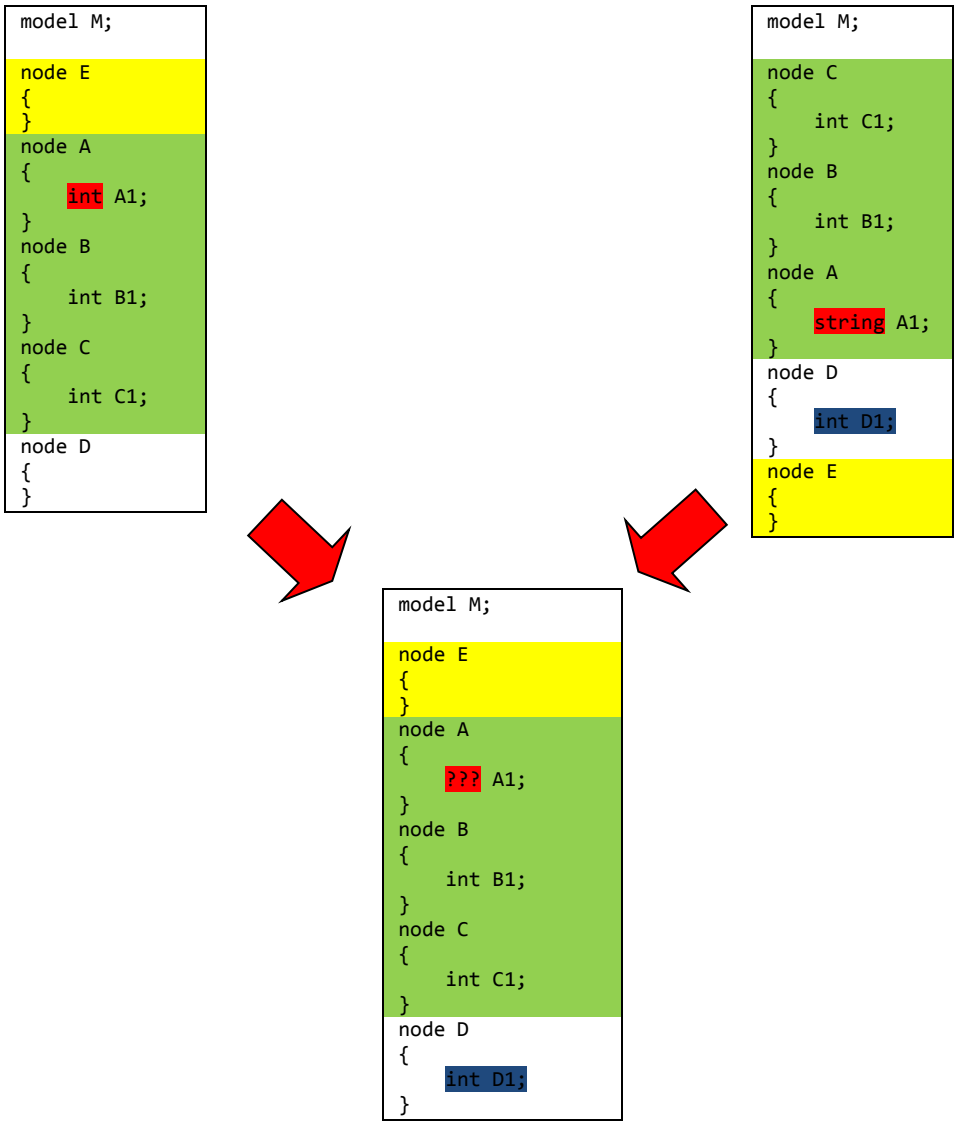
A 18. ábra egy példát tartalmaz, ami alapján az egymásra hatásokat fogjuk tanulmányozni. Az egyszerűség kedvéért csak a két szöveges reprezentációt és az automatikusan összefésült szöveget adtam meg.

Az összehasonlítás során az alábbi konfliktusokat ismertük fel:

1. az ABC ⇔ CBA sorrendet (**megváltozott sorrend**)
2. D1 attribútum hozzáadását D-hez (**új részfa**)
3. A1 típusának (int ⇔ string) eltérését (**eltérő szöveg**)
4. E a D mögül közvetlen az M mögé került (**áthelyezés**)

Nézzük végig, hogy milyen egymásra hatások lépnek fel!

Kezdjük az 1. konfliktussal. Változtassuk meg a sorrendet ABC-ről CBA-ra. Itt csak a sorrendben lévő fákkal kell foglalkoznunk. Egyrészt átrendezzük az iteratívan felépített fánkat, hogy az a helyes sorrendet (CBA) tükrözze. Másrészt frissítjük a fák pozícióit. Ezt a következőképpen tehetjük meg: vegyük a sorrendbeli első fát (C), ennek a kezdő pozíciója az előzőleg első fa (A) kezdőpozíciója lesz, végpozíciója pedig a kezdőpozíció megnövelve a C-hez tartozó szöveg hosszával. Ezután C gyerekeinek pozícióját is frissítjük. Ez jelen esetben csak C1-et foglalja magában. Ezt ismételjük meg B-re és A-ra is. Mivel A frissítése során A1 pozíciója is frissül, ezzel automatikusan megoldjuk a 3. konfliktusra való ráhatást is.



18. ábra: Egymásra hatások tanulmányozása

Nézzük, mi történik, ha a 2. konfliktus feloldását változtatjuk meg, vagyis az új fa (D1) beillesztése helyett annak törlését választjuk. Ekkor az előbb ismertetett eljárást hívjuk meg. D1 végpozíciója megváltozik, mégpedig 0 – 7 = -7-el módosul, mivel a régi szöveg „int D1;” hossza 7, az új szöveg pedig az üres szöveg, aminek hossza 0. Az eljárás során megtaláljuk a D fát, ami az említett 5 eset közül a 3.-ra (D gyereke D1, D1 szövege D „belsejében” található) illeszkedik. D a 4. konfliktushoz tartozik, mivel E egyik feloldásában D előtt található, tehát D végpozícióját is csökkentjük 7-el.

A 3. konfliktus esetén a helyzet hasonlít a 2. konfliktusnál leírtakhoz. Válasszuk a bal oldalhoz tartozó feloldást („int A1;”). Itt A1 végpozíciója 7 – 3 = 4-el fog nőni, mivel a régi szöveg a jelölő szöveg volt („???”). Megtaláljuk az A-hoz tartozó fát, ami az 1. konfliktushoz tartozik, ezért 4-el növeljük a végpozícióját. Megtaláljuk D-t is, ez A1 után kezdődik és a 4.

konfliktushoz tartozik (mint előző fa), tehát 4-el növeljük a kezdő- és végpozícióját D-nek és a gyerekeinek (D1) is. Ezért D1-et már nem növeljük még egyszer.

Végül nézzük meg, mi történik, ha E-t a D utáni helyre helyezzük át (4. konfliktus). Ekkor az E új és régi helye közötti összes fa (A, B, C, D), illetve gyerekeik kezdő- és végpozícióját E szövegének hosszával (8) megnöveljük, mivel mind a négy fa valamelyik konfliktushoz tartozik.

Végül lássuk az összefésülés formális leírását!

```
Algoritmus ÖSSZEFÉSÜLÉS (Fa1, Fa2, Szöveg1, Szöveg2, Konfliktusok)
1  AutoKonfliktusok ← AUTO(Konfliktusok);
2  ÖsszefésültSzöveg ← Szöveg1;
3  ÖsszefésültFa ← FELÉPÍT(Fa1, Fa2);
4
5  // Automatikus összefésülés
6  for ∀ k ∈AutoKonfliktusok do
7    AUTOMATIKUS_FELOLDÁS(k);
8    EGYMÁSRA_HATÁSOK_VIZSGÁLATA(k, Konfliktusok);
9    POZÍCIÓK_FRISSÍTÉSE(Konfliktusok);
10   ITERATÍV_FAÉPÍTÉS(ÖsszefésültFa);
11 end for
12
13  // Iteratív összefésülés
14  while true
15    Parancs ← FELHASZNÁLÓ_VÁLASZT();
16    if Parancs == FeloldottKonfliktus then
17      FELOLDÁS(FeloldottKonfliktus);
18      EGYMÁSRA_HATÁSOK_VIZSGÁLATA(FeloldottKonfliktus, Konfliktusok);
19      POZÍCIÓK_FRISSÍTÉSE(Konfliktusok);
20      ITERATÍV_FAÉPÍTÉS(ÖsszefésültFa);
21    else
22      if Parancs == Véglegesítés then
23        ELLENŐRIZ(ÖsszefésültSzöveg);
24        KÉSZ(ÖsszefésültSzöveg);
25      end if
26    end if
27  end while
28 end
```

2.5 A hibás működés kivédése

A módszer kidolgozása során arra törekedtem, hogy az összehasonlítás és összefésülés minél automatizáltabb módon mehessen végbe, a felhasználót minél kevesebbszer kelljen bevonni a folyamatba. Ennek ellenére – mint minden verziókezelő rendszer esetén – meg kell adni a felhasználónak a lehetőséget arra, hogy felülbírálhassa az automatikus működésű folyamatot. Ezt kétféleképpen teheti meg.

Először is korábban már szó volt a konfliktusokhoz rendelhető feloldások tárgyalása során a manuális feloldás fogalmáról. Ennek lényege, hogy a felhasználó egy tetszőleges szöveget rendelhet bármely konfliktushoz (manuális feloldás), ha valamiért nem elégedett az automatikus és alternatív feloldásokkal. A konfliktus továbbra is az adott részfához fog tartozni, tehát az összefésülés során a szövegbeli pozíciója természetesen ugyanaz marad. Manuális feloldás használata esetén előfordulhat, hogy a felhasználó egy szintaktikailag vagy szemantikailag helytelen szöveget ad meg a konfliktus feloldásaként. Ezért az összefésülés véglegesítése előtt kérdezzük meg a feldolgozót, hogy az összefésült szöveg (mind szintaktikailag, mind szemantikailag) helyes-e. Ha a szöveg helytelen, arról célszerű figyelmeztetni a felhasználót. Az, hogy mit csinálunk ezután (vagyis hogy engedjük-e az összefésülést, és ha igen, mit teszünk az összefésült szöveggel), már a módszer gyakorlati megvalósításától függ.

A hibás működés elleni utolsó – és egyben mindent megoldó – védelem az, hogy engedjük meg a felhasználónak, hogy az összefésülés véglegesítése előtt bárhogyan módosíthassa az összefésült szöveget. Ez a módszer a legtöbb verziókezelő rendszerben is jelen van valamilyen formában. Előnye, hogy ezzel (utólag) bármilyen hiba elkerülhető, hátránya viszont, hogy kényelmetlen, valamint itt is előfordulhat, hogy a felhasználó egy helytelen szöveget ad meg. A helytelen szöveg megadása ellen a korábban elmondottak alapján védekezünk.

A fentiek bevezetésével tehát minden lehetséges hibalehetőséget kivédünk. Ennek ára, hogy a felhasználót mélyebben be kell vonni a folyamatba, de ez minden verziókezelő rendszer esetén előfordul, sőt, megszokott.

3. Gyakorlati megvalósítás

Ebben a fejezetben a kidolgozott módszer egy gyakorlati megvalósítását fogom bemutatni egy általam készített program formájában. Az alkalmazás a bemutatott általános megoldást használja, de az egyszerűség kedvéért jelenleg csak az ANTLR [18] [19] segítségével definiált nyelvtanokat fogadja el a bemenetén. A program jelenleg fájlrendszer alapú. Az alkalmazás a használt nyelvtan feldolgozójával, valamint egy minta példával együtt az irodalomjegyzékben található linken keresztül érhető el [20].

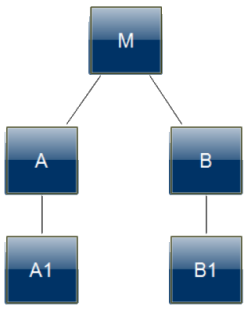
3.1 Visual Modeling and Transformation System

A program fejlesztése során egy konkrét modellező környezetet és nyelvtant használtam mintaként, de a program más modellező környezetekre és nyelvtanokra is fel van készítve.

A VMTS (Visual Modelling and Transformation System [7] [21]) az Automatizálási és Alkalmazott Informatika Tanszéken fejlesztett metamodell alapú modellező és modellfeldolgozó keretrendszer. A metamodellezés lényege, hogy két modell között egy modell-metamodell kapcsolatot adhatunk meg. A metamodell határozza meg, hogy a modell milyen elemeket használhat, illetve azok milyen kapcsolatban állhatnak egymással. Például ilyen kapcsolat van az UML osztálydiagramok és objektumdiagramok között is. A VMTS n-szintű metamodellezést alkalmaz, ami azt jelenti, hogy akárhány szintig megadható a modell-metamodell kapcsolat. A példánknál maradva osztálydiagram esetén csak 2-szintű metamodellezésről beszélünk. A legfelső szint az ún. *RootMeta*, amely meghatározza, hogy milyen elemek adhatók hozzá egy VMTS modellhez. Minden VMTS modell közvetlen vagy közvetett modell-metamodell kapcsolatban áll a *RootMeta*-val. A VMTS egy további, számunkra érdekes tulajdonsága, hogy a modelleket és modell elemeket egy egyedi GUID azonosítóval látja el, a pontos azonosítás érdekében. A VMTS modellek gráfokként írhatók le, ahol a csomópontok az entitások, valamint az entitások között különböző relációk is lehetnek. Ezek közül az egyik a tartalmazás reláció, amely alapján a csomópontokat egyértelmű szülő-gyerek hierarchiába lehet rendezni. Mind a csomópontok, mind a relációk rendelkezhetnek attribútumokkal. Egy attribútumnak neve, típusa és számossága van.

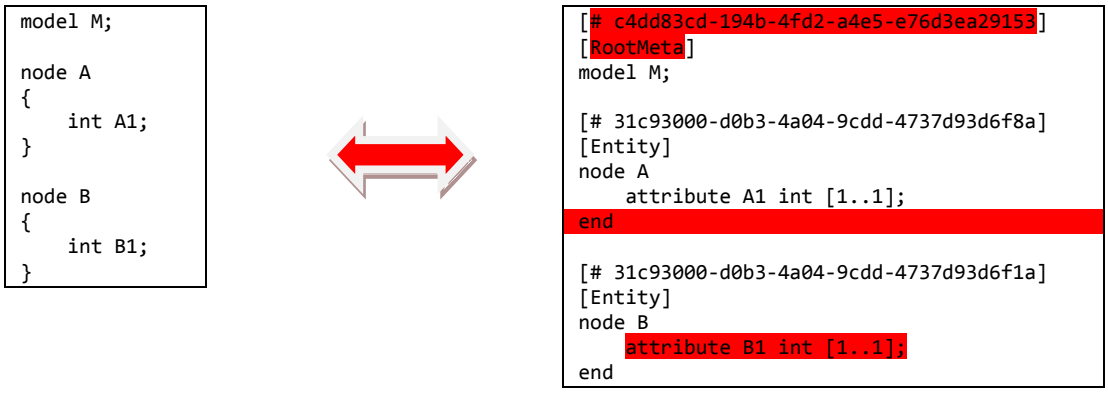
A VMTS alapértelmezetten XML formában vagy egy adatbázisban tárolja a modellek leírásait. Ezek a leírások több részre vannak bontva, ezért nehezen szerkeszthetők manuálisan, csak a VMTS GUI felületén keresztül van mód ezek módosítására. A VMDL (Visual Modelling Definition Language) egy általam kifejlesztett szöveges nyelv, amely

VMTS modellek szöveges formában való megjelenítésére szolgál. A VMDL segítségével a modellek manuális szerkesztése kényelmesebbé válik. A VMDL nyelvtan mellé egy feldolgozó is tartozik, tehát a modellt a szövegen keresztül szerkeszteni is lehet, vagyis a VMDL nemcsak a modell → szöveg, hanem a szöveg → modell konverziót is támogatja. A VMDL-hez egy hatékony szövegszerkesztő is tartozik, ami a hagyományos fejlesztőkörnyezetek funkcionalitásával rendelkezik: automatikus kódkiegészítés, hibakezelés, automatikus kódellenőrzés, valamint egyéb kényelmi funkciók. A VMDL egy olyan leképezést hajt végre a modell és a szöveg között, amely nagyban tükrözi a modell szerkezetét, ezáltal a szöveg sokkal olvashatóbb lesz. Ez például abban nyilvánul meg, hogy az egyes modell elemeket azok típusára utaló kulcsszavak vezetik be (*node, model, attribute* stb.), vagy például a tartalmazott csomópont közvetlenül a tartalmazó csomóponton belül jelenik meg.



19. ábra: Egyszerű szintaxisfa

A dolgozatban az eddig elmondottakon kívül nem fogom a VMDL részletes működését bemutatni, ehelyett szemléltetésképpen hasonlítsuk össze egy nagyon egyszerű példamodell VMDL és az elméleti mintanyelvtanunk által leírt reprezentációját. A 19. ábra egy szintaxisfa által reprezentált egyszerű példamodellt ábrázol. Érdemes megjegyezni, hogy a gyakorlatban (például VMDL esetén is) általában nem ennyire egyszerű a szintaxisfa felépítése (még egy ilyen egyszerű modell esetén sem), a dolgozatomban használt példa szintaxisfák csak elméleti jellegűek. Nézzük meg, hogy milyen szöveg keletkezik ebből a mintanyelvtanunk és a VMDL alapján!



20. ábra: A mintanyelvtanunk és a VMDL összehasonlítása

A mintanyelvtanunk és a VMDL közötti különbségek az alábbiak:

- minden csomópontnak van egy azonosítója, '[# ID]' szintaktikával megadva
- az azonosító alatti '[XY]' konstrukció a meta elemet azonosítja névvel vagy ID-val
- kapcsos zárójelek helyett a VMDL az *end* kulcsszót használja
- az attribútum definiálása az *attribute* kulcsszóval történik, a név és típus mellett meg kell adni a számosságot is (*[1..1]*), mivel ez egy meta attribútum definiálást jelent, vagyis példányosítani csak az eggyel alacsonyabb meta szinten található modell tudja

A fentieknél a VMDL természetesen lényegesen többet tud, de a program bemutatásának szempontjából nekünk elég ezekkel az alapvető dolgokkal tisztában lennünk.

Nézzük, hogy hogyan valósítja meg a VMDL feldolgozója a nyelvtan függetlenségről szóló (2.1) fejezetben megkövetelt műveleteket:

1. **Szöveg feldolgozása, szintaxisfa felépítése:** a feldolgozónak nyelvtantól függetlenül mindig el kell tudnia végezni a műveletet, ha a szöveg → modell konverziót támogatja. A VMDL feldolgozója is támogatja ezt.
2. **Szintaktikai és szemantikai ellenőrzés:** az ellenőrzés végrehajtásához minden nyelvtan esetén szükség van arra, hogy rendelkezésre álljon a modell, amit a szöveg alapján ellenőrizhetünk. Ezt a legegyszerűbb memóriában létrehozni, és a szöveg alapján frissíteni, a modellező keretrendszerek ezt általában támogatják. Mivel a VMTS metamodell alapú, ezért az ellenőrzéshez szükségünk lehet a vizsgált modell metamodelljére is. Ezért a feldolgozó a VMTS segítségével felépíti a metamodellt tartalmazó tárhelyet (ezt a VMTS rendszerben *repository*-nak nevezzük), nekünk csak a metamodellt kell fájlrendszer szintjén átadni. Az ellenőrzés ezután már végrehajtható.

3. **Két fa összeillőségének eldöntése:** először a két fa ANTLR [18] [19] által adott típusát hasonlítja össze, majd ha ez egyezik, megpróbál azonosító, majd név alapján egyezést találni. Ez mindig egy pontos eredményt fog szolgáltatni a nyelvtan felépítése miatt.
4. **Formai vagy tartalmi különbség:** VMDL esetén formai különbség csak komment vagy white space karakter lehet. Ezek reguláris kifejezések használatával egyszerűen felismerhetőek.

Látható, hogy VMDL esetén nem jelent jelentős implementációs többletköltséget a feldolgozótól megkövetelt műveletek kiajánlása az alkalmazás számára.

3.2 Technikai tudnivalók és gyakorlati megfontolások

A következőkben a program gyakorlati használatáról és a gyakorlattal kapcsolatos olyan kérdésekről lesz szó, amik az elmélet szempontjából nem érdekesek. Egyebek között szó lesz a nyelvtan függetlenség gyakorlati megvalósításáról is. A program egy konkrét nyelvtant felhasználva készült el, de fel van készítve más nyelvtanokra is, tehát megvalósítja az elmélet tárgyalása során említett nyelvtan függetlenséget.

Technikai tudnivalók

A program (úgy, mint a VMTS) a .NET keretrendszer segítségével készült el. A felhasznált nyelvtan megadása DLL-ek formájában történik. A program alapértelmezetten a VMDL-t, illetve annak feldolgozóját használja, de lehetőség van másik nyelvtan használatára is. Ehhez az *„App.config”* fájl „*Plugins_Directory”*, illetve „*Parser_DLL_Name”* bejegyzését kell módosítanunk. Az előbbi azt a könyvtárt adja meg, ahol a feldolgozó található, az utóbbi pedig a feldolgozót tartalmazó DLL neve. Az utóbbira azért van szükség, mert a feldolgozó mellett annak esetleges függőségei is ebben a könyvtárban találhatók.

A VMDL feldolgozója esetén meg kell adnunk a modell metamodelljét is, hogy az ellenőrzés megtörténhessen. Ezt a program build könyvtárában fogja keresni. A metamodellt a VMTS DLL formájában kezeli, a *Domains* mappába kell azt beraknunk.

Ha saját nyelvtanunk feldolgozóját szeretnénk kiajánlani a program számára, a betöltött DLL-ben található feldolgozó osztálynak az *IGrammarParser* interfészt kell megvalósítania. A betöltés a *Managed Extensibility Framework* (MEF, [22]) segítségével, DLL injekcióval

történik. Tehát a feldolgozót tartalmazó osztályunkat a következő annotációval kell ellátnunk, hogy a betöltés sikeres legyen*: [Export(typeof(IGrammarParser))]*.

Gyakorlati megfontolások

A feldolgozótól az elméletben megkövetelt négy művelet mellett a program megkövetel egy **ötödik műveletet** is, az egy fához tartozó kényelmesen megjeleníthető azonosító visszaadását. Ez a szintaxisfák és a konfliktusok megjelenítésénél hasznos, a megjelenítés felhasználóbarát lesz. A VMDL feldolgozója a fa nevét adja itt meg, vagy ha az nincs, akkor a fa ANTLR azonosítóját.

A módszer bemutatásakor felmerült a kérdés, hogy mit csináljunk akkor, ha az összefésült szöveget a feldolgozó helytelennek minősíti. A program ebben az esetben figyelmezteti a felhasználót, valamint megkérdezi tőle, hogy mégis szeretné-e elmenteni az összefésült szöveget. A bemenetek ellenőrzésekor, ha a program hibás szöveget talál, nem kezdődik el az összehasonlítás, tehát hibás bemeneti szöveggel nem foglalkozunk, az elméletnek megfelelően.

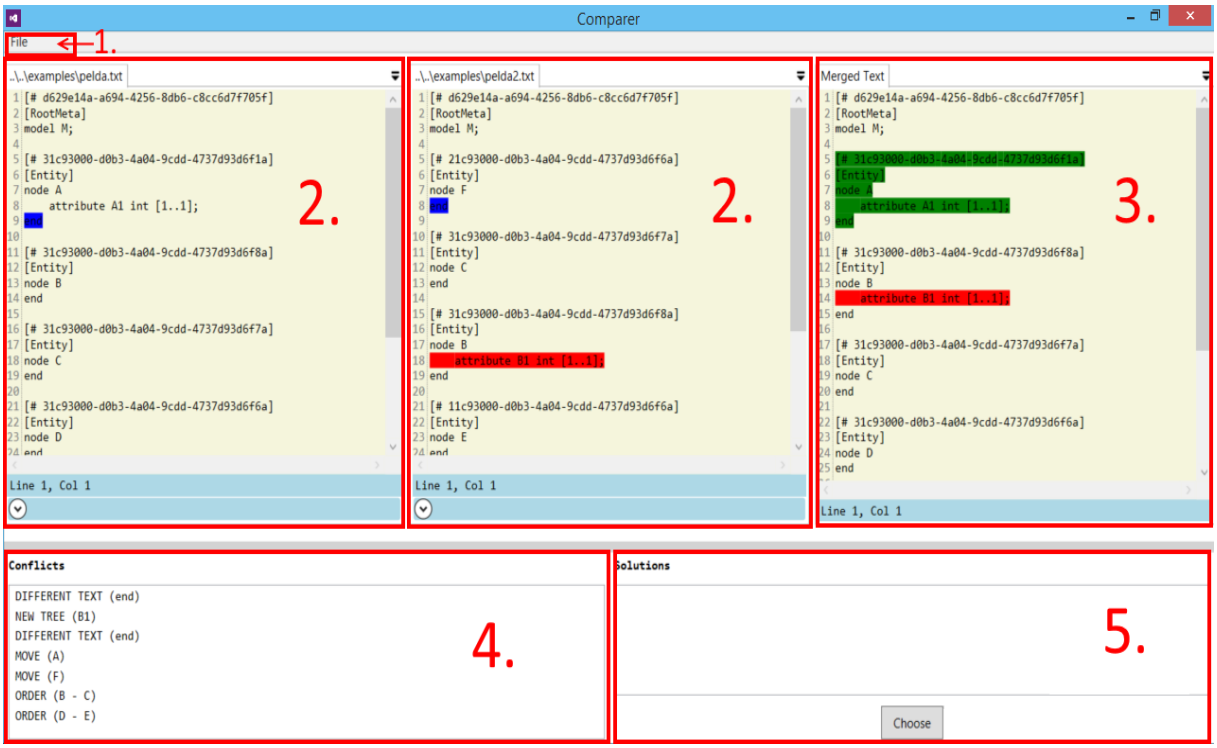
A program nem engedi meg, hogy a megváltozott sorrend típusú konfliktusokhoz saját feloldást adjunk. A többi konfliktushoz adható manuális feloldás. Ennek oka az implementáció egyszerűsítése. Az összefésülés végén, az összefésült szövegben természetesen továbbra is szabadon megváltoztathatunk bármit.

3.3 A program működése

A program a módszer ismertetése során megismert funkcionalitást valósítja meg:

- Nyelvtan megadása (DLL injekció formájában).
- Bemenet (szöveges reprezentációk) megadása.
- Bemeneti szöveges ellenőrzése a feldolgozó alapján.
- A szintaxisfa illesztés automatikus végrehajtása.
- A konfliktusok felismerésének automatikus végrehajtása.
- Az összefésülés automatizálható részének automatikus végrehajtása.
- A konfliktusok feloldásainak megváltoztatása (iteratív összefésülés).
- Az előző két funkcióhoz az egymásra hatások vizsgálatát is el kell végezni.
- Az összefésülés véglegesítése.
- Az összefésült szöveg ellenőrzése a feldolgozó alapján.

A továbbiakban látni fogjuk, hogy hogyan épül fel a program felhasználói felülete, valamint hogy az említett funkciók hogyan jelennek meg a gyakorlatban. A felhasználói felület elkészítése során a nyílt forráskódú AvalonEdit ([23]) és AvalonDock ([24]) keretrendszereket használtam.

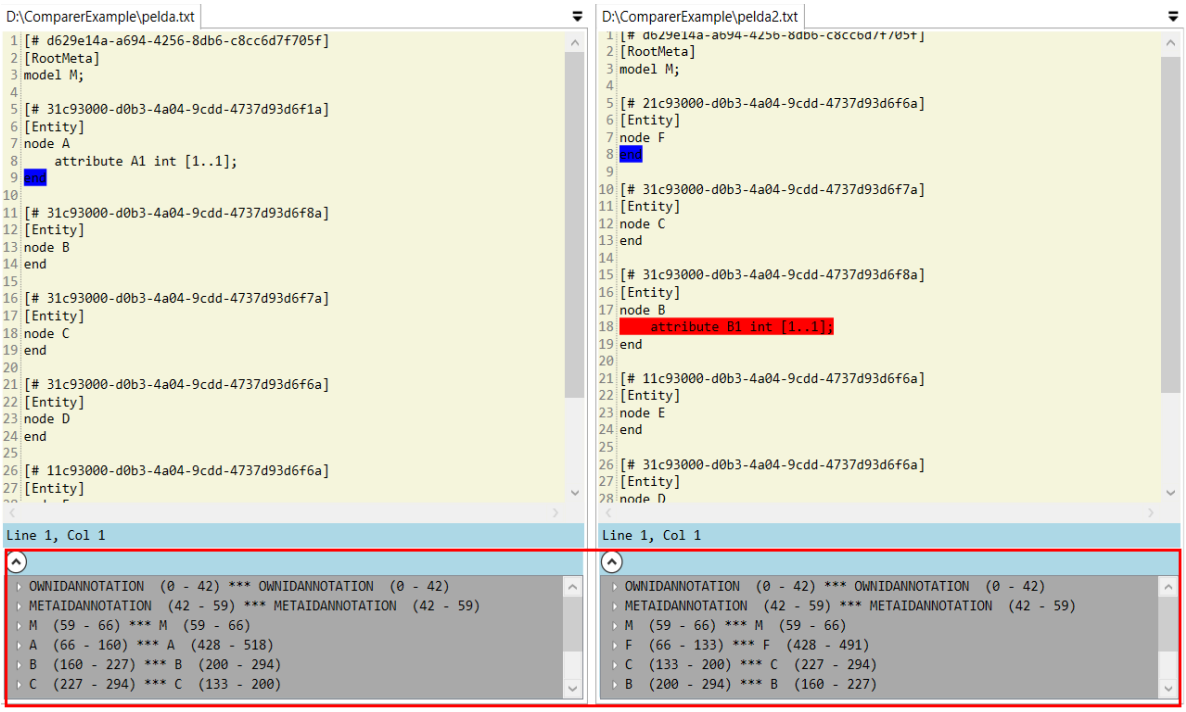


21. ábra: A felhasználói felület

A 21. ábra a program felhasználói felületét mutatja be, miután betöltöttük a két szöveges reprezentációt, és ellenőriztük azokat a feldolgozó segítségével. A felület különböző, logikailag elkülöníthető részekre bontható, melyek az ábrán is be vannak jelölve. Ezek az alábbiak:

1. Itt található a menüsor, ahol a szöveges reprezentációk betöltéséhez, valamint az összefésülés véglegesítésének elindításához szükséges funkciók érhetők el. Az utóbbi csak akkor, ha minden konfliktushoz már tartozik hozzárendelt aktív feloldás. A véglegesítés után végbemegy az összefésült szöveg ellenőrzése is.
2. A szöveges reprezentációk megjelenítésére szolgáló terület. Látható, hogy a konfliktusok helye a szöveges reprezentációkban is megjelenik. Az eltérő szöveg típus kék színnel, az új részfa típus piros színnel jelenik meg. Az áthelyezés és a megváltozott sorrend nincs szín kódolva, a jobb átláthatóság érdekében. A szövegek alatt a hozzájuk tartozó szintaxisfa is megjeleníthető. Ezt a 22. ábra mutatja be. A

felhasználóbarát megjelenítés érdekében a feldolgozó egy megjeleníthető azonosítót is ad minden fára, ha tud. Ez VMDL esetén a fa nevét jelenti, ha létezik.



22. ábra: Szintaxisfák megjelenítése

3. Az összefésült szöveg helye. Itt az áthelyezés típusú konfliktus is szín kódolva van, zölddel. Akár a megváltozott sorrendet is megjeleníthetnénk, de én a jobb átláthatóság miatt ezt nem tettem meg. Az összefésült szöveghez nem tartozik megjeleníthető szintaxisfa a korábban említett okok miatt. Az összefésülés automatizált része a konfliktusok felismerése után automatikusan végbemegy.
4. A konfliktusok listája. A konfliktus típusa után a hozzá tartozó fa vagy fák szerepelnek. Ehhez szintén a feldolgozótól elkért megjeleníthető azonosítót használom fel, hogy felhasználóbarát legyen a megjelenítés. A program a konfliktusokat az ismertetett módon automatikusan felismeri, amint betöltöttük a két szöveget. Egy konfliktust kiválasztva megjelenik a hozzá rendelt feloldások listája, valamint mindhárom szövegben ki lesz jelölve a konfliktushoz tartozó fa szövege. Az ábrán 2 eltérő szöveg, 1 új részfa, 2 áthelyezés és 2 megváltozott sorrend típusú konfliktus látható. Ezek közül a B-C-re vonatkozó megváltozott sorrendet jelöltük ki.
5. A kiválasztott konfliktushoz rendelt feloldások listája. Ha nincs kiválasztva konfliktus, akkor ez üres. Egy feloldásra duplán kattintva megnézhetjük a hozzá tartozó szöveget, valamint manuális feloldás esetén meg is változtathatjuk azt. A feloldások listájából a

lista alatt található gomb segítségével aktív feloldást állíthatunk be a konfliktusra. Ekkor az összefésült szövegben a konfliktus szövege az aktív feloldás szövegére változik, és végbemegy az egymásra hatások vizsgálata, valamint a pozíciók frissítése. Ez az iteratív összefésülés folyamata, ami addig tart, amíg nem véglegesítjük az összefésülést a menüsorból.

4. Összefoglalás

A kitűzött feladat egy olyan módszer kidolgozása volt, amely képes egy tetszőleges szoftvermodell két szöveges reprezentációjának összehasonlítására, illetve összefésülésére, és független a konkrét modellező környezettől vagy nyelvtantól. A problémafelvetés során párhuzamot vontunk a forráskód alapú fejlesztés és a kitűzött probléma között, de azt is láthattuk, hogy a probléma jelentősen eltér a forráskód összehasonlítástól. Az eltérés abban nyilvánul meg, hogy modellektől, illetve azok szöveges reprezentációiktól megkövetelhetjük, hogy azok szemantikailag is helyesek legyenek, míg forráskód esetén ez nem tehető meg. Emiatt a szövegből felépíthető szintaxisfák jelentős többletinformációval rendelkeznek az összehasonlítás során. Tehát az összehasonlítás sokkal pontosabb lehet, mint forráskód esetén. Ez abban nyilvánul meg, hogy a felhasználó az összehasonlítás során a fa által reprezentált modell elemeket látja, vagyis a két szöveg közötti különbségekhez szemantikai jelentés is társul. Láthattuk, hogy bár a szöveg- és modell összehasonlítás területén számos eredmény létezik, modellek szöveges reprezentációival eddig csak keveset foglalkoztak.

A dolgozatban elméleti módszert dolgoztam ki a fenti feladat megoldására.

- Az összehasonlítás a szövegekből felépített absztrakt szintaxisfák (AST) alapján megy végbe. A feldolgozó felelős egyebek mellett a szöveg feldolgozásáért (szintaxisfává alakításáért), valamint a bemeneti- és összefésült szövegek szintaktikai és szemantikai ellenőrzéséért is.
- Az összehasonlítás a szintaxisfák illesztésével kezdődik. Az egyik szintaxisfa minden részfájához megkeressük a másik szintaxisfában található párját, vagyis azt a részfát, ami ugyanazt az elemet reprezentálja. Ebben a feldolgozó nyújt nekünk segítséget.
- Az összehasonlítás második lépése a konfliktusok – a két szintaxisfa közötti eltérések – felismerése. Minden konfliktushoz egy vagy több feloldás tartozik, de mindig csak egy lehet aktív. Minden feloldás egy szöveget tartalmaz a hozzá tartozó konfliktus feloldására. A módszer kidolgozásakor fontos kritérium volt, hogy az minél több konfliktust automatikusan feloldjon, a felhasználó bevonása nélkül. Ezért olyan konfliktustípusokat vezettünk be, amelyek minden esetet lefednek, valamint megkönnyítik az automatizált működést is.
- A konfliktusok felismerése után az összefésülés következik. Az összefésülés a konfliktusok feloldásának folyamata. Az összefésülés első lépéseként az automatikus feloldással rendelkező konfliktusok feloldásai kerülnek bele az összefésült szövegbe.

Ezután minden konfliktus feloldása egyesével felülbírálható. A konfliktusok pozícióját az összefésült szövegben végig karbantartjuk. Ezáltal az összefésülés inkrementális lesz, ami teljesítménybeli előnyökkel jár, valamint az összefésülés folyamata sokkal áttekinthetőbb lesz a felhasználó számára. Az inkrementális működés eléréséhez a konfliktusok egymásra hatásait vizsgáltuk meg és használtuk fel.

- A módszer a hibás működésre is fel van készítve. A felhasználó minden konfliktushoz egy tetszőleges, saját feloldást is megadhat, valamint a véglegesítés előtt az összefésült szöveg is tetszőlegesen szerkeszthető. Az összefésülés végén ezért (is) a feldolgozó segítségével ellenőriznünk kell az összefésült szöveget.

A kidolgozott módszer fontos tulajdonsága, hogy nem függ konkrét modellező környezettől vagy nyelvtantól, mivel csak az összehasonlításhoz szükséges műveleteket követeli meg a nyelvtan feldolgozó egységétől.

A módszer elméleti tárgyalása után láthattuk annak egy konkrét, gyakorlati megvalósítását is. Az elkészített alkalmazás a VMTS-t [7] [21] és VMDL-t használta konkrét modellező környezetként és nyelvtanként, de tetszőleges, az ANTLR [18] keretrendszerben írt nyelvtanra fel van készítve. Az alkalmazás az adott nyelvtan által leírt, a bemenetén kapott két szöveget összehasonlítja, és a módszernek megfelelően minden konfliktust felismer, valamint ahol lehetséges, automatikusan fel is oldja azokat. Ezután a felhasználó egyesével felülbírálhatja a konfliktusok feloldásait, majd véglegesítheti az összefésülést.

4.1 További kutatási irányok és továbbfejlesztési lehetőségek

Végezetül megemlítenék néhány további kutatási irányt és továbbfejlesztési lehetőséget. Fontos megjegyezni, hogy ezek nagy része inkább alternatív iránynak tekinthető, mint továbbfejlesztésnek, mivel a módszer bemutatása során felmerülő problémák közül sokat többféleképpen is meg lehet oldani, és természetesen minden megoldásnak megvannak az előnyei és hátrányai. A teljesség igénye nélkül lásunk néhány példát!

Formai és tartalmi különbségek feloldása

Az eltérő szöveg típusú konfliktusnál volt szó arról, hogy milyen automatikus és alternatív feloldásokat adhatunk meg formai, illetve tartalmi különbségek esetén. Ha a feldolgozótól további műveleteket követelnénk meg, pontosabb lehetne a feloldások definiálása, de ez nem biztos, hogy minden nyelvtan esetén kevés implementációs többletköltséggel járna.

Sorrend más megközelítése

A sorrend kezelésére nagyon sokféle megoldás kitalálható, kezdve azzal, hogy egyáltalán nem kezeljük le azt. Ha nem kezelnénk le a sorrendet, az összehasonlítás sokkal átláthatóbb lenne, de így az olyan modelleket nem tudnánk jól lekezelni, amelyeknél a sorrend szemantikai különbséget is jelent, tehát a megoldásunk nem lenne általános. Ezen kívül számos egyéb megoldás is kitalálható a sorrend változásának felismerésére, illetve kezelésére, például a már említett csere alapú megoldás.

Időbélyeg használata új részfák esetén

Az új részfa típusú konfliktus tárgyalásánál azt mondtuk, hogy legyen az automatikus feloldás a fa szövege (beszúrás), az alternatív pedig az üres szöveg (kitörlés). Ha verziókezelő rendszerekben gondolkodunk, akkor egy másik megoldás lehet, hogy mindig a frissebb időbélyeggel rendelkező szöveg részesül előnyben a konfliktus feloldásakor. Ez persze csak akkor működik, ha a gyakorlatban megvalósított rendszer fájl szinten is hozzáfér a szövegekhez, de ez verziókezelő rendszereknél tipikus. Ez az irány inkább a gyakorlati megvalósítás során jelentős.

Integrálás verziókezelő rendszerbe

A módszer gyakorlati megvalósítása jelenleg fájlrendszer szintjén működik. Egy gyakorlati továbbfejlesztési lehetőség egy tényleges verziókezelő rendszer megvalósítása modellek szöveges reprezentációi számára, ami a kidolgozott módszeren alapszik.

Irodalomjegyzék

[1] *Tortoise SVN.*
http://tortoisesvn.net/ (2015.10.26).

[2] *GIT.*
https://git-scm.com/ (2015.10.26).

[3] A. V. Aho, M. S. Lam, R. Sethi és J. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, 2006.

[4] *Eclipse Modeling Framework.*
https://eclipse.org/modeling/emf/ (2015.10.26).

[5] *Generic Modeling Environment.*
http://www.isis.vanderbilt.edu/projects/gme/ (2015.10.26).

[6] *Visual Studio Modeling SDK.*
https://msdn.microsoft.com/en-us/library/bb126259.aspx (2015.10.26).

[7] *Visual Modeling and Transformation System.*
https://www.aut.bme.hu/Pages/Research/VMTS/Introduction (2015.10.26).

[8] *Longest Common Subsequence (LCS).*
https://www.ics.uci.edu/~eppstein/161/960229.html (2015.10.26).

[9] *GNU Diff Utils.*
https://www.gnu.org/software/diffutils/ (2015.10.26).

[10] *Patience Diff.*
http://alfedenzo.livejournal.com/170301.html (2015.10.26).

[11] Z. Xing és E. Stroulia, „UMLDiff: an algorithm for object-oriented design differencing. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05). ACM, New York, NY, USA, 54-65.," 2005..

[12] U. Kelter, J. Wehren és J. Niere, „A Generic Difference Algorithm for UML Models.. In P. Liggesmeyer, K. Pohl & M. Goedicke (eds.), Software Engineering (p./pp. 105-116), : GI. ISBN: 3-88579-393-8," 2005.

[13] *EMF Compare.*
http://www.eclipse.org/emf/compare/ (2015.10.26).

[14] D. S. Kolovos, A. Pierantonio, D. D. Ruscio és R. F. Paige, „Different models for model matching: An analysis of approaches to support model differencing. In Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (CVSM '09). IEEE Computer Society, Washington, DC, USA, 1-6.," 2009.

[15] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez és M. Monperrus, „Fine-grained and accurate source code differencing. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). ACM, New York, NY, USA, 313-324.," 2014.

[16] *Szintaxisfák XML alapú reprezentációja.*
http://www.srcml.org/ (2015.10.26).

[17] Van Rozen, Riemer, and Tijs Van Der Storm. "Model Differencing for Textual DSLs." BENEVOL 2014-Proceedings of the Belgian-Netherlands Evoluation Workshop.

[18] *ANTLR.*
http://www.antlr.org/ (2015.10.26).

[19] T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, Pragmatic Bookshelf, 2007.

[20] *Az elkészült alkalmazás.*
https://www.aut.bme.hu/Upload/Pages/Research/VMTS/Papers/TextualModelComparer.zip (2015.10.26).

[21] T. Levendovszky, L. Lengyel, G. Mezei és H. Charaf, „A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. Electron. Notes Theor. Comput. Sci. 127, 1 (March 2005), 65-75.," 2005.

[22] *Managed Extensibility Framework.*
https://msdn.microsoft.com/en-us/library/dd460648%28v=vs.110%29.aspx (2015.10.26).

[23] *AvalonEdit.*
http://avalonedit.net/ (2015.10.26).

[24] *AvalonDock.*
https://avalondock.codeplex.com/ (2015.10.26).