



Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Forráskódok változásainak vizsgálata

Szabó Richárd

Programtervező Informatikus MSc

Témavezető: **Tóth Melinda**

Egyetemi tanársegéd

Budapest, 2013

TARTALOMJEGYZÉK

1. Szöveges diff algoritmusok	4
1.1. Standard diff	4
1.2. Patience diff	8
1.3. Továbbfejlesztett soronkénti diff	10
1.4. Verziókezelő rendszerek	11
2. Undo/redo modellek	13
2.1. Refaktorálások specialitásai	13
2.2. Undo/redo modellek	14
2.2.1. Lineáris undo	14
2.2.2. Nemlineáris undo	15
2.2.3. Szigorított lineáris undo	16
2.2.4. Szelektív undo	17
2.3. Speciális modell	17
2.3.1. A modell felépítése	17
2.3.2. Konfliktusok kezelése	19
2.3.3. Változások számítása	19
2.4. Esettanulmányok	20
3. Szintaxisfa diff	23
3.1. Fák összehasonlítása	23
3.2. Parse transform-ok vizsgálata	24
3.3. Összehasonlító algoritmus	25
3.4. Absztrakt kód	27
4. Összehasonlítás	29
Irodalomjegyzék	39

KIVONAT

Programok forráskódjának változásait többféle feladatban is, többféle megközelítésben vizsgálnunk kell. Egyszerűbb esetekben elegendő lehet, hogy manuálisan megnézzük egy diff eszköz segítségével két fájl, vagy egy fájl két változata közti különbséget. Leggyakrabban ezt a feladatot azonban inkább a verziókezelő rendszerekbe beépített diff szolgáltatás segítségével végezzük. Egy másik fontos terület a verziókövetés mellett a szövegszerkesztők, fejlesztőeszközök undo/redo funkciója. Ennek megvalósításához ugyanis szintén szükséges, hogy a fájlokon bekövetkező változásokat valamilyen módon monitorozzuk, kezeljük. Ebben az esetben már jóval finomabban, nagyobb részletességgel kell kezelnünk a változásokat, azonban elképzelhető, hogy bizonyos feladatoknál egy ennél is mélyebb szintre kell lépnünk. Ilyen lehetőség, ha vizsgálat alapjául a forrásszöveg helyett a kód szintaxisfáját választjuk. Dolgozatomban a változások vizsgálatának ezen módszereivel foglalkozom, ezeknek a lehetőségeit, tulajdonságait, sajátosságait elemzem és vetem össze egymással.

BEVEZETÉS

Programok forráskódjainak változásait több feladatban is fontos kezelniük, értelmeznünk, vizsgálnunk. A legegyszerűbb erre a célra használt eszköz a diff program. Ezzel egyszerű szöveges alapú soronkénti összehasonlítást végezhetünk két fájl között. Egy ennél komplexebb alkalmazás, mikor egy szövegszerkesztőben, fejlesztő eszközben monitoroznunk kell a fájlokat, számolni kell az azokon bekövetkezett változásokat, hogy biztosítani tudjuk az undo/redo funkciót. Bizonyos feladatokban szükséges, hogy a szöveges alapú vizsgálatról egy mélyebb szintre lépjünk, és az összehasonlítást szintaxisfák között végezzük. Ilyen feladat megoldására lehet igény, mikor egy adott programozási nyelvre specializált eszközben akarunk felismerni mélyebb – például függőségi – összefüggéseket, vagy transzformációk hatását. Az Erlang nyelvben például a parse transform fordítási idejű transzformációra ad lehetőséget, amivel a szintaxisfán tetszőleges átalakításokat végezhetünk. Ahhoz, hogy egy kód megértést támogató, statikus elemzést végző eszközben egy ilyen transzformáció hatását elemezni tudjuk, szükséges, hogy a szintaxisfa szintjén vizsgálódjunk.

Ahogy látjuk különféle területeken különböző célja van az összehasonlításnak, változások vizsgálatának, ennek megfelelően pedig különböző megoldások szükségesek.

Egyszerű esetekben elegendő lehet a szöveges alapú, soronkénti vizsgálat a már említett diff eszközzel. Ennek egy továbbfejlesztését, specializációját a refaktorálás undo/redo-t megvalósító szakdolgozatomban [17] már bemutattam. Ez szintén szöveges alapon vizsgálódik, viszont a sorszintű összehasonlításnál finomabban, részletesebben kezeli a módosításokat.

Ennél is mélyebb szintű, nagyobb részletességű és pontosságú vizsgálatot végezhetünk, ha az összehasonlítást a program szintaxisfáján végezzük. Ezzel a témával parse transform-okról szóló TDK-dolgozatomban [19] foglalkozom.

Diplomamunkámban a különböző összehasonlítási lehetőségeket összegzem, ezek eredményeit vizsgálom.

1. SZÖVEGES DIFF ALGORITMUSOK

Első fejezetben a hagyományos, szövegalapú összehasonlításról, a különféle diff algoritmusokról írok.

1.1. *Standard diff*

Hagyományosan a diff [26] alatt a unix rendszereken megszokott, fájlok összehasonlítására használt eszközt értjük. Ez a parancssori program soronként összeveti a két megadott szöveges fájlt, majd megadja a megegyező részeket, illetve az eltéréseket. A leggyakoribb feladat, amire használjuk az eszközt, egy fájl két különböző változatának összehasonlítása. Ez jelenik meg pl. a verziókezelő rendszerekben is. A különbségek kiszámításához a program a leghosszabb közös részsorozat problémáját oldja meg. Ebben a feladatban a két fájl – sorszintű vizsgálat esetén – sorok két sorozatának tekintjük és meg akarjuk találni a megegyező elemek – tehát sorok – leghosszabb sorozatát. Az eredmény egy lista lesz az eltérő sorokról: milyen sorokat kell törölni, vagy beszúrni ahhoz, hogy az egyik fájlból megkapjuk a másikat, és fordítva. Ettől kicsivel eltérő megközelítésben a problémára egy úgynevezett edit szkriptet is megadhatunk, ami beszúrási és törlési műveletek sorozatát tartalmazza, amik végrehajtásából az eredeti fájlból megkaphatjuk az új fájlt. Az első problémában a leghosszabb közös részsorozatot keressük, ilyen edit szkript esetén pedig a legrövidebbet, egy minimálisat. A két különböző megközelítés eredménye könnyen átvihető egymásba, így például a diff programtól is kérhetjük az eredményt edit szkript formájában. Ezenkívül sok más lehetséges formátum is van a program kimenetének megjelenítésére.

A leghosszabb közös részsorozat problémájának nincs általános, egységesen elfogadott megoldása, azonban röviden ismertetem azt, amit a diff program is használ. Az alapötlete a megoldásnak a következő: menjünk előre mindkét fájlban, amíg eltérést nem találunk; majd valamilyen módon keressünk tovább, amíg újra egyező sorokat nem találunk. A keresésnek a módja azonban nem triviális, szükséges heurisztikák alkalmazása, hogy elfogadható hatékonyságú megoldást kapjunk. Bizonyos egyszerű heurisztikák viszonylag kisszámú eltérés és kevés ismétlődő sort tartalmazó fájlok esetén jól működnek, azonban bonyolultabb esetekben ronthat a hatékonyságon. Egy lehetőség például, ha az eltérés esetén felső korlátot adunk arra, hogy

mennyit lépünk előre mindkét fájlban a következő egyezés megtalálásához.

A feladatra több dinamikus programozási megoldás ismert. Ezeknek az algoritmusoknak a hatékonyságát különböző hasítási technikákkal, a sorok ekvivalenciaosztályokba való rendezésével, logaritmikus keresés általi összefésülésekkel, dinamikus memórafoglalással, és más módszerekkel is növelhetjük.

Tekintsünk egy egyszerű példát a diff működésére. Adott két fájl, a sorait az egyszerűség kedvéért csak karakterekkel jelöljük. A soronkénti vizsgálat miatt megtehetjük ezt, ugyanis az összehasonlítás alapegysége egy sor. A kiindulási (a) állapotban a fájl tartalmát jelölje a következő sorozat:

a b x y f h j

Tekintsük ennek a fájlnek egy módosított (b) változatát:

e a b w z l f

Ekkor a `diff a b` kimenete a következő:

```

diff a b
0a1
> e
3,4c4,6
< x
< y
---
> w
> z
> l
6,7d7
< h
< j

```

Ez az eredmény a következőt jelenti. Az eredeti fájl virtuális 0. sorába beszúrtunk – a, mint append – egy e karaktert, ami az (b) állapotban megjelent. A > karakter tehát a második fájlból származó sorokat jelöli, a < pedig az eredetiben meglevőket. A 3,4c4,6 azt jelenti, hogy a 3-tól 4-ig terjedő sorokat lecseréljük – c a change angol szót jelöli – a második fájl 4-től 6-ig terjedő soraira. Ezután sorolja fel a lecserélendő sorokat, majd a --- elválasztójel után az új, másik fájlból származó sorokat. Végül pedig töröljük – d, mint delete – a 6. és 7. sorokat. A d utáni 7 azt jelöli, hogy a második fájlban a 7. sorba kellene szűrni ezt a tartalmat, ha a második fájlt akarnánk átalakítani az elsőnek megfelelő tartalmúvá.

Ahogy említettem, a diff kimenetét számos formátumban meg lehet jeleníteni. Ez a fenti a legegyszerűbb **normal** nevű formátum, de létezik például a **context** formátum is, ami a változott sorok néhány soros környezetét is megmutatja az eredményben. Forráskódok frissítésének, módosításának terjesztéséhez, patch-ekhez ez a

```
context-a.cpp
#include <iostream>

using namespace std;

int main()
{
    int v;
    v = 2 * 4;
    cout << "Result: " << v;
    return 0;
}
```

```
context-b.cpp
#include <iostream>

using namespace std;

int main()
{
    int v;
    v = 2 * 4 + 6;
    cout << "Result: " << v;
    cout << "Bye!";
}
```

megszokott, szabványos forma. Ennek megfelelően a unix **patch** programja is ezzel a formátummal dolgozik. Vegyük a két következő egyszerű c++ forrást:

Ekkor a diff kimenete context formátumban:

Az első két sor a két fájl utolsó módosításának idejét írja le. A *-gal jelölt információk az első fájlhoz tartoznak, a - pedig a második fájl adatait jelöli. Először leírja az első fájl változásait az 1-11. sorokban, ezt jelöli a ***** 1,11 *****. Utána hasonlóan leírja a második fájl kapcsolódó részleteit, változásait az 1-9. sorokban. Több változás esetén ilyen csonkokban, részletekben a környezettel megadva mutatja a különbségeket. A konkrét részletben a !, + és - jelek mutatják a konkrét eltéréseket. A ! azt jelenti, hogy az adott sor egy módosított állapotban mindkét fájlban megtalálható. Ilyen például a v változónak való értékadás a 8., illetve a 6. sorban. A + egy sor beszúrását, a - pedig törlést jelöl.

E példa kapcsán szeretném felhívni a figyelmet egy problémára, amire később még visszatérünk. A diff úgy véli az első fájl **return 0;** tartalmú sorának a második fájlban a **cout << "Bye!";** sornak feleltethető meg. Sokkal kézenfekvőbb, ésszerűbb válasz lenne azonban, ha azt mondanánk, hogy az első fájlból töröltünk egy sort, után pedig történt egy beszúrás.

A program kimenetét a fentiekén túl meg lehet jeleníteni egy párhuzamos formában is, amikor a két fájl eltéréseit egymás mellett jeleníti meg. A fenti példára a side-by-side kimenet a következő:

Egy negyedik lehetőség a kimenet formátumára az úgynevezett edit szkript, ami

```

_____ diff -c context-a.cpp context-b.cpp _____
*** context-a.cpp      2013-05-01 11:50:09.000000000 +0200
--- context-b.cpp      2013-05-01 11:54:33.000000000 +0200
*****
*** 1,11 ****
    #include <iostream>

- using namespace std;
-
    int main()
    {
        int v;
!       v = 2 * 4;
        cout << "Result is: " << v;
!       return 0;
    }
--- 1,9 ----
    #include <iostream>

    int main()
    {
        int v;
!       v = 2 * 4 + 6;
        cout << "Result is: " << v;
!       cout << "Bye!";
    }

```

```

_____ diff -y context-a.cpp context-b.cpp _____
#include <iostream>                                #include <iostream>

using namespace std;                               <
                                                    <
int main()                                         int main()
{                                                  {
    int v;                                         int v;
    v = 2 * 4;                                   |   v = 2 * 4 + 6;
    cout << "Result is: " << v;                   |   cout << "Result is: " << v;
    return 0;                                    |   cout << "Bye!";
}                                                  }

```

közvetlenül használható az `ed` szövegszerkesztőben. Ezt végrehajtva az eredeti fájlban megkapjuk a módosított fájlt.

További részletes leírás a különböző formátumokról, paraméterezési lehetőségekről a diff manual-jában [26] található.

wdiff A standard `diff` eszköz egy kiegészítése a `wdiff` program, amivel a soronkénti vizsgálat helyett szóalapú összehasonlítást végezhetünk a fájlkon. A szó már megszokott módon ebben az esetben is a fehér szóközök közötti, attól eltérő karakter sorozatot jelenti. A `wdiff` két fájl összehasonlításakor ideiglenes fájlokat készít, amik soronként tartalmazzák az eredeti fájl szövegét. Ezután az ideiglenes fájlokra meghívja a hagyományos `diff`-et, majd annak az eredményét felhasználva egy olyan kimenetet ad, amin jobban látható az eredeti fájl közti szavankénti eltérés. A fenti C++ nyelven írt példákra a `wdiff` a következő kimenetet adja:

```
wdiff context-a.cpp context-b.cpp

#include <iostream>

[-using namespace std;-]

int main()
{
    int v;
    v = 2 * [-4;-] {+4 + 6;+}
    cout << "Result is: " << v;
    [-return 0;-]
    {+cout << "Bye!";+}
}
```

Az eredményben a `[-` és `-]` közötti részek jelzik a törölt szavakat, amelyek csak az eredeti fájlban szerepelnek, a `{+` és `+` közti szavak pedig az új, beszúrt elemeket jelölik.

Ezen a példán láthatjuk, hogy a `wdiff` jóval pontosabb, finomabb eredményt ad, mint a soronkénti összehasonlítás. Ennek – ahogy később az undo modellekről szóló fejezetnél láthatjuk – bizonyos feladatokban nagy jelentősége van, azonban pl. a verziókezelő rendszereknél túlságosan költséges lenne a változások ilyen finomságú eltárolása, túlságosan nagy chanelog-okat eredményezne a szavankénti vizsgálat.

1.2. Patience diff

A következőkben egy speciális diff algoritmust mutatok be, amit patience jelzővel jelölnek. Az algoritmust a Bazaar nevű verziókezelőben használják.

A megszokott diff algoritmushoz hasonlóan ez is a leghosszabb közös részsorozat problémáján alapul, azonban a pontosabb eredmény érdekében nem kezel egyenran-

gúan minden sort. A programozási nyelvekben vannak ugyanis bizonyos gyakran ismétlődő szintaktikus elemek, amik a diff szempontjából könnyen „bezavarhatnak”, konfliktushoz vezethetnek olyan esetben, amikor valójában nem kellene konfliktust eredményezniük. Ezzel szemben olyan egyedi sorok, mint egy függvény deklarációja sokkal jobban meghatározzák egy fájl karakterét, sokkal nagyobb jelentősége van ezen sorok a változások vizsgálata szempontjából. Ennek a figyelembevételével az algoritmus először csak az egyedi sorokra végzi el a közös részsorozatok keresését, majd utána dolgozza fel az ilyen jelzőként szolgáló sorok közti részeket.

A konfliktusok, a gyakran ismétlődő elemek problémájának szemléltetésére vegyünk egy egyszerű példát: egy fájl végéről törölünk egy függvényt, majd a fájl elejére beszúrjuk. Tehát egyszerűbben mondhatjuk azt is, hogy a fájlban a függvényt átmozgattuk egyik helyről egy másikra. Ebben az esetben a diff algoritmusok nagy valószínűséggel a kapcsos zárójeleket tekintenék támpontoknak a függvények helyett, azokat próbálnák egymáshoz illeszteni. Így például egy `F1 F2 F3` függvénysorozat `F4 F1 F2` függvényekkel való illesztésénél a kapcsos zárójelek miatt a következő párosításokat kapjuk eredményül: `F1-F4`; `F2-F1`; `F3-F2`. Ehelyett a kézenfekvő válasz az lenne, ha az `F4`-et új, beszúrt elemnek, az `F3`-at pedig töröltnek tekintenénk. Ez tehát ilyen egyszerű esetekben is durva eredményhez és konfliktushoz vezethet, holott ez a probléma könnyen feloldható lenne. Ilyen problémák különösen gyakran előfordulhatnak, ha a BSD/Allman kódolási stílust alkalmazzuk, amiben a kapcsos zárójelek új sorba kerülnek, egysoros utasítások köré is teszünk zárójeleket és gyakoriak az üres sorok a különböző blokkok, egységek között.

Tekintsünk egy másik példát, ahol a patience diff algoritmus szintén jobb eredményt ad a standard diff-nél. Először nézzük a patience diff kimenetét:

```
void func1() {
    x += 1
}

+ void functhreehalves() {
+     x += 1.5
+ }
+
    void func2() {
        x += 2
    }
```

Ez tűnik a kézenfekvő, logikus megoldásnak, azonban a többi diff gyakran a következő eredményt adja:

```
void func1() {  
    x += 1  
+ }  
+  
+ void functhreehalves() {  
+     x += 1.5  
+ }  
  
void func2() {  
    x += 2  
}
```

Láthatjuk tehát, hogy itt is a kapcsos zárójelek okoztak más, durvább eredményt. Az eredmény így is helyes, azonban az első változat sokkal inkább kézenfekvő és elfogadható.

Algoritmus leírása A következőkben megadom az algoritmus lépéseit.

1. Illesszük egymásra mindkét fájlban az első sort. Ha megegyeznek, lépünk tovább a második sorra, majd a harmadikra, és tovább mindaddig, amíg eltérő sorokat nem találunk.
2. Tegyük ezt hasonlóan az utolsó sorokkal, hátulról előrefelé haladva a fájlokban, amíg itt is különbséget találunk.
3. Keressük meg azokat a sorokat, amelyek mindkét fájlban pontosan egyszer szerepelnek, majd oldjuk meg ezekre a legkisebb közös részsorozat problémáját.
4. Alkalmazzuk az 1. és 2. lépéseket a megtalált sorok közötti részekre.

Összefoglalva tehát az algoritmus egyszerű alapötlete az, hogy az egyedi sorokat, amik a tényleges, fontos részeket tartalmazzák, mintegy jelzőkként tekinti. Ezzel elkerüli azt, hogy a gyakran ismétlődő, tartalmi szempontból egyébként kevésbé jelentős sorok, mint a nyitó, csukó kapcsos zárójelek konfliktust okozzanak olyan esetekben, amikor valójában nincs konfliktus az érintett területen.

Ez persze nem jelenti azt, hogy minden esetben jobb eredményt adna a többi diff algoritmusnál, vagy gyorsabb lenne, hogy ne lennének olyan inputok, amire rosszabb eredményt ad, azonban nagyon egyszerű az alapötlete, könnyen módosítható, finomítható, és a fenti eseteket jobban kezeli a standard diff-nél.

1.3. Továbbfejlesztett soronkénti diff

A [3] cikkben bemutatásra kerül egy a standard `unix diff`-re épülő, annál pontosabb eredményt adó soronkénti diff algoritmus. A cikk kifejezetten a verziókezelő

rendszerek szempontjából vizsgálja a változások kezelésének problémáját, ezért az itt bemutatott algoritmus ebből a szempontból add jobb, pontosabb eredményeket. Az új algoritmus első lépésben a standard `diff` segítségével megkeresi, mely sorok maradtak változatlanul. Ezután egy távolságmétrikával összehasonlítja a változott-nak ítélt sorokat. Itt azonban egyenként hasonlítja össze őket, hanem a szomszédos sorokból csoportokat képez, majd a két oldal így kapott csonkjait páronként összeveti egyással. Harmadik lépésben az algoritmus veszi a legközelebb álló párokat, majd ezekre végez egy sorszintű összehasonlítást egy újabb metrikával. Egy megadott hasonlósági küszöb feletti sorokat változottnak minősít, tehát nem töröltnek, vagy beszúrtnak. A maradék sorra egy újabb körben alkalmazza a 2. és 3. lépéseket. Az iterációk számát, a hasonlósági értékek küszöbével együtt paraméterként adhatjuk meg.

A következő cikkből vett ábra egy egyszerű példán szemlélteti az algoritmus működését. Az első lépés kiszűri a változatlan sorokat, a bal oldal változott sorait megjelöli töröltként, a jobb oldalét pedig beszúrtnaként. Második lépésben az ezekből a sorokból képzett csoportra alkalmaz egy metrikát. Ezek közül kiválasztva a legmagasabb értékűeket, elvégzi a soronkénti összehasonlítást. Láthatjuk, hogy már egy iteráció után jóval pontosabb eredményt kapunk: megtalálja azokat a sorokat, amelyek csak át lettek mozgatva – `printf("A");` és `printf("B");` –, sőt azokra is jó találatot ad, amelyek ténylegesen csak módosítva lettek, nem pedig törölve, vagy beszúrva: `int b[];` helyett `int b[]={1,2};`, valamint `foo(c,b);` helyett `b=foo(c,b);`. Az eredeti 5 törölt és 4 beszúrt sor helyett egyetlen ténylegesen törölt sor maradt.

1.4. Verziókezelő rendszerek

A diff algoritmusok egyik legfontosabb alkalmazási területét a verziókezelő rendszerek jelentik. Ezek a fájlok aktuális állapota mellett minimális helyfoglalásra törekedve eltárolják a különböző verziókhoz tartozó változásokat, metainformációkat. Egy változás rögzítésekor a rendszer valamely diff algoritmus segítségével kiszámítja a legutóbbi állapothoz képest történt változásokat. A különböző rendszerek különböző konfigurációs lehetőségeket nyújtanak, különféle algoritmust használnak, vagy gyakran megengedik egy külső diff eszköz használatát is, azonban leggyakrabban a standard `diff` programot használják, vagy annak egy saját megvalósítását. Az említett okok miatt a sorszintű vizsgálat szintén alapértelmezés.

SVN [43] Talán a leggyakrabban használt verziókövető rendszer az SVN. Ez eleinte a hagyományos Unix/GNU `diff` eszközt használta, később azonban speciális igények miatt saját diff megvalósításra tért át. A lehetőség továbbra is megvan a

standard diff használatára, sőt a `--diff-cmd` kapcsoló segítségével tetszőleges külső diff programot is használhatunk. A saját belső algoritmus a már ismertetett leghosszabb közös részsorozat problémájának megoldásán alapul. A megoldás alapjául Myers diff algoritmusát szolgál, annak egy módosított, specializált változatát implementálták a rendszerben.

git [44] Az SVN-hez hasonlóan a git is rendelkezik saját diff algoritmussal, és engedni külső program használatát is. Alapértelmezésben szintén Myers algoritmusát használja, azonban beépítetten tartalmazza a patience diff algoritmust is. Ezt opcionálisan lehet kiválasztani. Ezenkívül képes megjeleníteni a különbségeket word-diff algoritmussal is.

bazaar [14] Ez a rendszer alapértelmezésben a patience diff algoritmust használja.

darcs [40] A Darcs verziókezelő rendszer több tekintetben is eltér a hagyományos rendszerektől. Egyik fő különbség, hogy a rendszer egészét, annak aktuális állapotát nem fájl-fák, fájlok verziói írják le, hanem patch-ek, változások egy halmaza. A változások kezelése, konfliktusok feloldása, összefésülések ennek megfelelően teljesen más módon történik, melynek alapja az úgynevezett patch-algebra [39]. Ennek egyik lényeges tulajdonsága, hogy a változások között nem áll fenn egy időbeli sorrend. Ehelyett a patch-ek között részbenrendezési reláció áll fenn a függőségek alapján. Ennek oka, hogy például két különböző felhasználótól származó patch érintheti egy fájlban ugyanazt a területet, de ebben a helyzetben nem adhatunk elsőbbséget egyik vagy másik változásnak, egyenrangúnak kell tekinteni őket. Ennek következtében a konfliktusok kezelése jóval bonyolultabbá válik, mintha kihasználhatnánk a változások között fennálló időrendi sorrendet. A patch-algebra konfliktusok kezelésére egy olyan algoritmust ad, mely nem teljes, sőt bizonyos esetekben exponenciális futási idejű.

A változások kiszámítására egyszerűen külső eszközként a Unix-os diff programot használja.

2. UNDO/REDO MODELLEK

Az undo/redo funkció fontos szolgáltatás a grafikus felületű alkalmazásoknál, szövegszerkesztőknél. Az egyre kényelmesebb felhasználói felületeken ugyanis egyre könnyebben előfordul az is, hogy az ember hibát ejt, vagy gyorsan váltótat valamit, amit később inkább vissza szeretne vonni. Egy programfejlesztés során is gyakran előfordul, hogy a felhasználó egy végrehajtott változtatást, műveletet vissza akar vonni, mert más úton kívánja azt megoldani, amit változtatott, vagy pedig egyszerűen elrontott valamit.

Ebben a fejezetben a változások kezelését az undo/redo funkció szempontjából vizsgálom: milyen szempontokat kell figyelembe venni egy szövegszerkesztő, fejlesztőeszköz undo műveletének megvalósításához, kiemelten foglalkozva a refaktorálások bonyolult, összetett változtatásaival. A témával szakdolgozatomban [17] és TDK-dolgozatomban [18] foglalkoztam. A problémát, illetve az általam erre készített megoldást ezek alapján fogom ismertetni.

2.1. Refaktorálások specialitásai

A refaktorálás [22] a forráskód olyan transzformációját jelenti, ami egy fájlt több ponton is érinthet, egyszerre lehetnek benne törlések, beszúrások, sőt egy ilyen művelet akár egyszerre több fájlt is érinthet. Ezenkívül a visszavonásánál konfliktusok is felléphetnek, amik tovább nehezítik az ilyen változások helyes, biztonságos kezelését.

A RefactorErl [23] egy az egyetemünkön fejlesztett refaktoráló, kódmegértést statikus elemzéssel támogató eszköz. Ennek egy hiányossága volt a refaktorálások megfelelő undo/redo művelete, ugyanis csak egy korlátozott funkcionalitást biztosított, ami nem kezelte a konfliktusokat, sőt csak a legutóbbi refaktorálás visszavonására volt lehetőség. Ebben az esetben konfliktust jelenthetett például, ha az adott refaktorálás után még egy szerkesztés is történt valamelyik érintett fájlban. Ekkor egy visszavonás után a szerkesztés változásai is eltűntek.

Ezek figyelembe vételével egy olyan kiegészítést készítettem az alkalmazáshoz, ami egyúttal különálló programként is képes működni, kellőképp általános ahhoz a felülete és a működése is, hogy más eszközökhöz is hozzá lehessen csatolni. Ezt bizonyítja, hogy az első Emacs-interfészhez [38] integrált verzió után rövidesen elkészült

a Vim-integráció [37] is. Ehhez pedig a program nem igényelt lényegi módosítást.

A haskell [35] nyelven megírt, BufferServer nevű program egy FingerTree [34, 33] nevű adatszerkezetben egyenként eltárolja az éppen kezelt fájlokat. A tárolt fájlokat monitorozza, minden változáskor kiszámítja az érintett fájl(ok)on bekövetkezett módosításokat, azokat eltárolja egy történeti listában. A megvalósítás fontos pozitív tulajdonsága, hogy a változások szövegrészletei szintén FingerTree-ben vannak eltárolva, ezzel tovább növelve a művelet hatékonyságát. Egy változás visszavonásakor az eltárolt információk alapján előállítja a megfelelő állapotot a fájlokban, szükség esetén ekkor történik meg a konfliktusok kezelése, a változások összefésülése is.

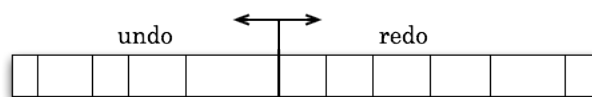
2.2. Undo/redo modellek

A refaktorálásokra specializált saját modell ismertetése előtt röviden bemutatok néhány általánosabb undo/redo modellt [25], amelyek fontosak voltak a speciális kialakításában.

Számos általános felhasználású undo modell létezik, melyek alapvetően két csoportra bonthatóak: lineáris undo, nemlineáris undo. A lineáris undo a végrehajtott parancsok egy sorozatát rögzíti, ahol közvetlenül csak az utolsó művelet vonható vissza, csak lineárisan lépkedhetünk a különböző állapotok között. Ezzel szemben, a nemlineáris undo megengedi, hogy a felhasználó rögtön egy korábbi változtatást vonjon vissza, a későbbiek visszavonása nélkül, feltéve, hogy az aktuális állapotban a parancs értelmesen végrehajtható. A modellek egy harmadik nagy csoportja – valójában ezt is tekinthetjük egyfajta nemlineáris undo-nak – a szelektív undo. A két fő csoport után erről is írok néhány mondatot.

2.2.1. Lineáris undo

A lineáris undo modell minden végrehajtott parancsot egy történeti-listában tárol, mely elméletben korlátlan hosszú lehet. A lista végén levő parancs visszavonható. Visszavonáskor a történeti-listából törlődik, a redo-lista elejére kerül. Ez a művelet mindaddig megismételhető, amíg a lista ki nem ürül. Az utolsó visszavont változtatás, mely a redo-lista elejére került, újra végrehajtható, az undo-lista elejére tehető. Az új parancsok a redo-lista változtatása nélkül az undo-lista elejére kerülnek. A felhasználó ilyen módon a parancsok sorozatát bejárhatja (2.1. ábra). Egy redo parancsot tekinthetünk egy undo undojának, azonban ebben a modellben a redo parancsok csak implicit módon vannak jelen, nem részei a történeti-listának, az érintett parancsot a redo-listában külön tároljuk el.



2.1. ábra. undo-redo lista

2.2.2. Nemlineáris undo

A felhasználó elvárhatja, hogy közvetlenül ne csak az utoljára végrehajtott művelet visszavonására legyen lehetőség. Könnyen előfordulhat ugyanis, hogy már több újabb parancsot végrehajtott, miután egy hibát észrevesz a módosításban. Többfelhasználós alkalmazásokban az undo kényelmes használatához nélkülözhetetlen, hogy az utolsónál korábbi parancs visszavonható legyen, hiszen a felhasználó számára lokálisan utolsó művelet globálisan tekintve nem feltétlenül az utolsó, közben más felhasználó is végrehajthatott parancsokat. A következőkben bemutatok néhány jelentősebb nemlineáris undo modellt.

Szkript modell

A szkript modell a felhasználói műveleteket parancsok egy szkriptjének szerkesztésével modellezi. A felhasználó a szkriptet szerkesztve kitörlésével tetszőleges parancsot visszavonhat, a rendszer egyszerűen újra végrehajtja a parancsokat a kezdőállapottól indítva. Egy másik lehetséges, jóval hatékonyabb módszer, ha csak a törölni kívánt parancsig visszamenően visszavonjuk a változásokat, majd annak kivételével újra végrehajtjuk a visszavont parancsokat. A szkript modell egyszerű, jól működő, azonban felhasználói felületét tekintve nem a legmegfelelőbb, különösen ha grafikus felületű alkalmazásnál használjuk.

US & R modell

Ez a modell a fenténél egyszerűbb felülettel rendelkezik. A lineáris undo-hoz hasonlóan rendelkezik egy undo és egy redo művelettel, ezen felül egy szkip művelettel, mellyel egy parancs redo-ja átugorható. Ilyenkor az adott parancs nem törlődik, csak megjelölésre kerül. Új parancsok felvételekor a redo lista tartalma megmarad, így tulajdonképpen egy történeti fa keletkezik. Ez a fa valójában egy irányított gráf, hiszen különböző ágakból indulva lehetséges a parancsokat újra végrehajtani, új éleket létrehozni. Bár ez a modell működését tekintve egyszerű, a történetet leíró szerkezet rendkívül bonyolulttá, nehezen áttekinthetővé válhat, használata kényelmetlen.

Hármas modell

A hármass modell szintén három műveletet definiál, de harmadik művelete a szkip helyett a forgatás. A modellhez felhasznált adatszerkezet egy egyszerű undo és egy redo lista párosa. A visszavont parancsok a redo listába kerülnek, a forgatás művelete pedig a redo lista utolsó elemét a lista elejére teszi. Forgatással kiválasztható a következő újra végrehajtandó parancs, illetve a következő visszavonandó parancs helyét is kiválaszthatjuk. Ezek alapján megvalósítható egy tetszőleges, a listában levő parancs visszavonása: a kívánt parancsig visszamenően undo-t kell alkalmazni, forgatni kell a redo listát, majd néhány redo-t kell alkalmazni. Egy a redo listában levő műveletet a már említett módon, forgatásokkal lehet elérni, majd végrehajtani. A hármass modellről az előző két modellhez hasonlóan azt mondhatjuk, egyszerű működési mechanizmusa ellenére, kényelmetlen felülete miatt nem a legmegfelelőbb modell.

2.2.3. Szigorított lineáris undo

A fenti modellek mindegyikének problémája, hogy azt feltételezik, bármelyik parancs újra végrehajtható értelmesen a program más állapotában is, mint amiben eredeti végrehajtáskor volt. A problémát szemlélteti a következő, rajzolóprogramból vett példa: a felhasználó rajzol egy kört, ezt átmozgatja egy másik helyre. Ezután visszavonja a mozgatást, törli az objektumot. Az előző nemlineáris modellekben ebben a helyzetben lehetséges volna a mozgatás műveletének redo-ja, azonban ez hibát okozna, hiszen az objektum, amire a mozgatás parancsa hivatkozik, nem létezik.

Ilyen és ehhez hasonló problémák elkerülhetők a lineáris undo szigorított változatában, mely teljesíti a következő tulajdonságot (*biztonságos végrehajthatóság*): egy parancsot mindig abban az állapotában hajtunk újra végre, mint amiben eredetileg volt, és mindig abban az állapotban vonjuk vissza, amiben az adott parancs eredeti végrehajtása után volt. Az állapot fogalma ezekben a modellekben mindig a program undo művelet szempontjából érdekes jellemzőinek együttesét jelenti.

A lineáris undo egyetlen esetben szegi meg a fenti tulajdonságot, amikor úgy adunk hozzá új parancsot az undo listához, hogy a redo-lista nem üres. Ennek a konfliktusnak egy lehetséges, egyszerű, bevett feloldása, ha ilyen esetben egyszerűen töröljük a redo-lista tartalmát.

A szigorított lineáris undo modell szöveges fájl változásaira specializálva következőképp képzelhető el: a végrehajtott módosításokat – absztrakt formában – eltároljuk egy listában, ami a fájl történetét reprezentálja. A lista elején a legutóbbi változás található, ami következő alkalommal visszavonásra kerülhet. Ha ez megtörténik, az undo listából kitöröljük, inverzét berakjuk a redo lista elejére. Az utolsó visszavont, azaz a redo lista elején levő művelet megismételhető, melynek inverze

így újra az undo lista elejére kerül. Az új változások folyamatosan az undo listába kerülnek. Ezenkívül a biztonságos végrehajthatóság kikötését is teljesítjük a fent leírt módszerrel.

Ez a módszer egyszerű szövegszerkesztő undo/redo funkciójának megvalósításához jól használható, azonban a refaktorálás lehetséges konfliktusainak lekezelésére a refaktorálás undo/redo-hoz módosítást, bővítést igényel. Refaktorálás undo-nál a megkövetelt tulajdonság megszegése akkor fordul elő, ha egy több fájlos refaktorálás visszavonásakor az egyik érintett fájlban nem ez a refaktorálás a legutóbbi módosítás. Szemléletesen a fenti, rajzolóprogramban leírt példához hasonlóan akkor léphet fel konfliktus, ha valamely változásban egy korábbi változás eredményére van hivatkozás: például, ha egy refaktorálás utáni változás hivatkozik egy a refaktorálás által létrehozott változóra. A stabil tulajdonság sérülése esetén meg kell vizsgálnunk, hogy értelmes módon meghagyhatjuk-e a refaktorálás utáni változtatás(oka)t, vagy azokat is vissza kell vonnunk. Ehhez a vizsgálathoz azonban szükséges, hogy a változásokat az egymással való összehasonlításhoz egy közös szintre hozzuk, ugyanis a fájl különböző állapotában következtek be.

2.2.4. Szelektív undo

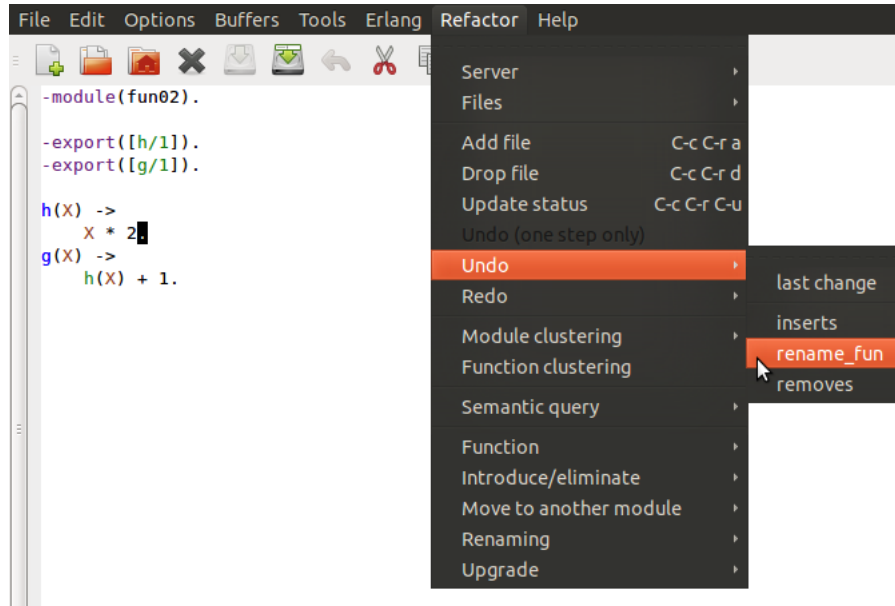
A szelektív undo tulajdonképpen egy a nemlineáris undo-k csoportjába tartozó modell, ami az eddigiekhez képest funkcionalitásban ad egy jelentős többletet. Ennek az alapötlete az, hogy a felhasználónak megengedhetjük, hogy tetszőleges korábbi parancsot válasszon ki a történeti-listából és beszúrjon egy új parancsot, ami visszafordítja a kiválasztott parancs hatását. Amennyiben egy korábbi parancs ilyen kiválasztása és visszavonása nem értelmes, a felhasználónak nem engedjük meg annak kiválasztását. Speciálisan egy refaktoráló eszköz esetén ez azt jelentené, hogy a felhasználónak felajánlanánk az utolsó néhány változást a történeti listából, a listából kiválaszthatja, melyik változást kívánja visszavonni. Amennyiben a kiválasztott parancs konfliktusmentesen visszavonható, egy a változással ellentétes hatású parancs a redo-listába beszúrásra kerül. Ellenkező esetben visszautasítjuk a felhasználó kérését, vagy az adott változást követő módosításokat szintén visszacsévéljük.

A RefactorErl-ben megvalósított undo/redo funkció már ezt a szolgáltatást is tartalmazza. Ennek a felülete látható a 2.2 ábrán.

2.3. Speciális modell

2.3.1. A modell felépítése

Saját modellem a szigorított lineáris undo refaktorálásokra specializált, bővített változata. Ennek megfelelően minden fájlunk eltároljuk a tartalmát, a módosításokat



2.2. ábra. Szelektív undo a RefactorErl-ben

pedig egy undo-redo listapárosban tároljuk, melyben egy konkrét változás elemi módosítások sorozataként jelenik meg. Egy elemi módosítás egy beszúrást, vagy egy törlést jelent, annak kezdőpozíciójával és az érintett szöveggel. Szemléltetésnek megadom az adatszerkezetek haskell nyelvű kódját.

```
data URList = URList (undo :: [Change], redo :: [Change])
data Change = Change (changedId :: Int, updates :: [Update])
data Update = Insert (updatePoint :: Point, updateString :: String)
              | Delete (updatePoint :: Point, updateString :: String)
```

Az `URList` jelöli magát a történetet reprezentáló listapárt. A változásokat a `Change` adatszerkezetben tároljuk. Ezekben nemcsak a módosítások listáját tároljuk el, hanem ellátjuk őket egy azonosítóval is, amivel összetudjuk kötni a több fájlos refaktorálások fájlonkénti változásait. Ezt tehát egy csoport azonosítónak is nevezhetjük. Az elemi módosításokat jelölő `Update` típus az eddigieknek megfelelően egyszerűen egy beszúrást, vagy egy törlést jelent a módosítás kezdőpozíciójával és tartalmával.

A szövegek tárolása a már említett `FingerTree`-ben történik. Ez egy általános felhasználási célú, kiegyensúlyozott keresőfa. A feladat szempontjából a legfontosabb tulajdonsága a hatékony keresés, elérés mellett a dinamikussága, ugyanis rendkívül hatékony – $\Theta(\log(\min\{n, m\}))$ – a vágás, illetve összefűzés művelete, ami a mi esetünkben a törlésnek, illetve beszúrásnak felel meg. Mivel szövegpufferként használjuk, a fájlokon pedig a szerkesztés miatt gyakoriak a módosítások, ez fontos tulajdonsága az adatszerkezetnek.

2.3.2. Konfliktusok kezelése

Mint már említettem, refaktorálások visszavonásánál konfliktusok is felléphetnek a különböző változások közt. Ez olyan esetben fordulhat elő, ha az éppen visszavonni kívánt refaktorálás valamelyik érintett fájlban nem az utolsó. Például lehetséges, hogy a refaktorálás után még szerkesztettük az egyik fájlt, vagy akár végrehajtottunk még egy refaktorálást. Ekkor meg kell vizsgálni, hogy vissza lehet-e vonni úgy az adott refaktorálás hatását, hogy a későbbi változást megtartsuk és értelmes, konzisztens is maradjon a kód. Szemléletesen azt mondhatjuk, hogy akkor lép fel konfliktus az undo-nál, ha a két vizsgált változás érint közös területet a fájlban. Ez nagy valószínűséggel szemantikusan is eredményez egy függőséget. Például a későbbi változás hivatkozhat egy azt megelőző által létrehozott változóra. Az általam választott megoldás előnye, hogy nem végez szemantikus ellenőrzéseket, hanem egyszerűen szöveges alapon leellenőrzi, hogy a módosítások pozíciói átlapolnak-e. Ez jóval hatékonyabb, mégis lefedi a szemantikus függőségek esetét is.

A változások konfliktusosságának pontos definíciója, a konfliktust ellenőrző, illetve az összefésülést végző algoritmus leírása megtalálható a TDK-dolgozatomban. A haskell nyelvű implementáció pedig publikus, a RefactorErl eszköz részét képi, elérhető a projekt oldalán.

2.3.3. Változások számítása

A lineáris undo-nak megfelelően minden lépésben kiszámítjuk az érintett fájlokban történt változásokat. Ehhez a már többször említett Myers-féle algoritmus egy haskell-es implementációját használom. Alapértelmezésben, az egyszerű szerkesztéseknél karakterenkénti összehasonlítást végzünk. Az algoritmus a fájl tartalmának megfelelő sorrendben megadja, mik azok a szövegrészletek, amelyek csak az egyik, vagy csak a másik állapotban találhatók meg. Az így kapott eredményt a modellben feldolgozzuk, egy absztraktabb formára átalakítjuk: a csak az elsőben szereplő változást törlésnek tekintjük, az új változatban szereplőket pedig beszúrásnak. Az algoritmus az átalakítás során előrehaladva folyamatosan számolja a változások pozícióit is. Így az undo/redo funkciót könnyen meg tudjuk valósítani, ugyanis egy ilyen absztrakt módosítás végrehajtása a megfelelő pozícióra való beszúrást, vagy attól a megfelelő hosszú szöveg törlését jelenti. Az undo végrehajtásakor a műveletet egyszerűen invertálnunk kell, azaz a törlésből beszúrást készítünk, a beszúrásból törlést.

Egyszerű, szerkesztési módosításoknál használhatjuk a szöveg karakterszintű összevetését, hiszen ez általában néhány karakter beszúrását jelenti, a karakterek szintjén történő vizsgálat megfelelő finomságú eredményt ad. Ezzel ellentétben refaktorálás során nem célszerű karakterek szintjén végezni az összehasonlítást, hiszen itt fel kell

készülni a komplexebb, egy fájlban belül több pontra kiterjedő változásokra is, ahol a karakteres vizsgálat túlságosan finom részletességű eredményt adna, ami később a módosítások konfliktusának vizsgálatánál jelentene hátrányt. A problémát szemléltetik a következő rész példái. Ehelyett célszerű a vizsgálatot szavak szintjén végezni, azaz a szöveget fehérszóközök mentén felvágni, és az így kapott szeleteket tehát nem karakterenként vizsgálni, hanem teljes egészében. Ezzel elérjük, hogy a kiszámított változások ne legyenek túl finomak, és ne akadjanak össze a pozíciók konfliktus vizsgálatánál olyan esetekben is, amikor valójában nincs konfliktus.

2.4. Esettanulmányok

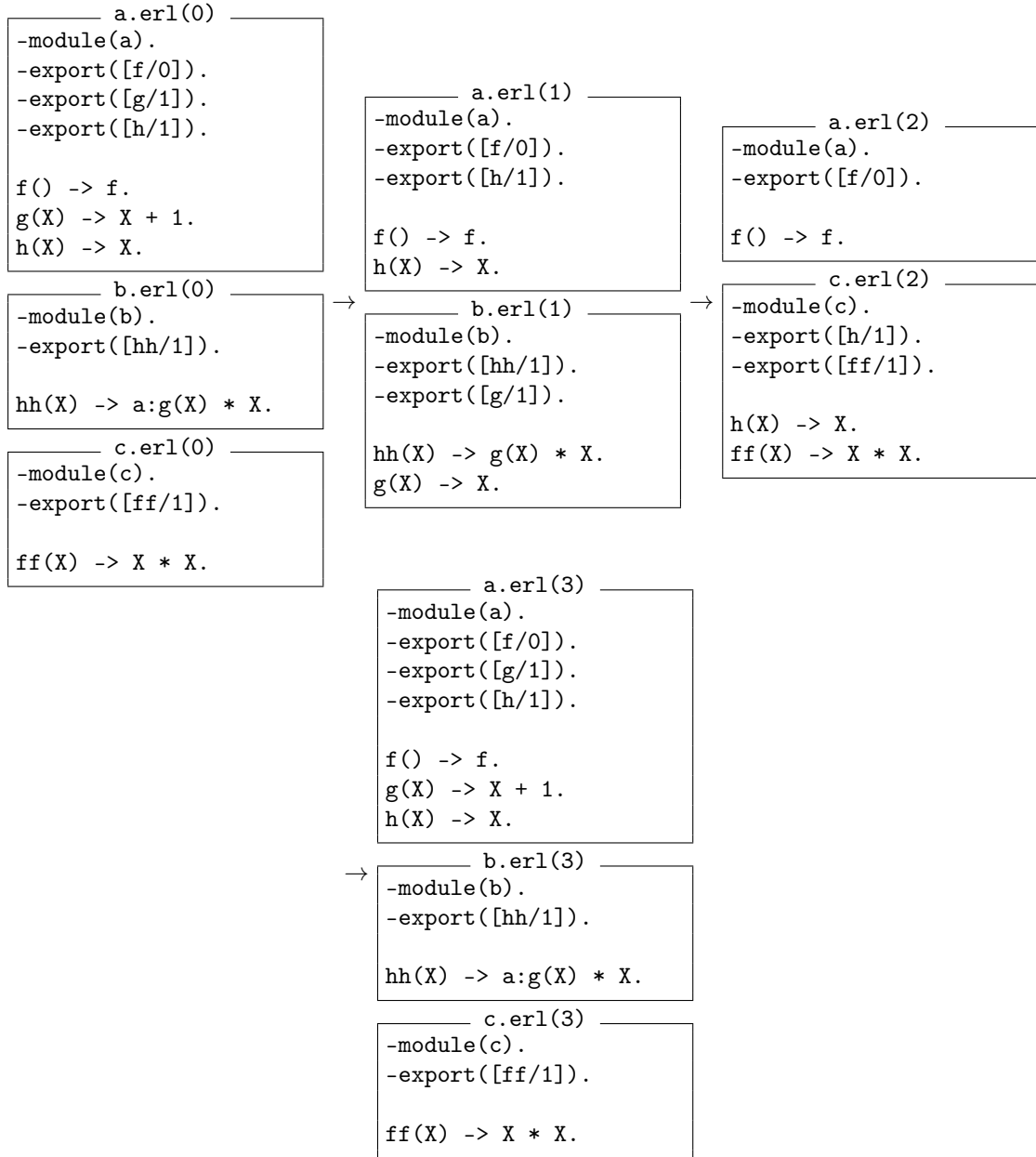
Az alábbiakban néhány példán keresztül szemléltetem a leírt modell működését. A példákban leírt pontokat az ábrák fejlécében levő számok segítségével lehet nyomon követni. A példákon keresztül megmutatom, milyen különbséget jelent, hogy a változások számításakor karakterenként végezzük az összefésülést, vagy szavanként, a fehérszóközök mentén felvágva. A szemléltetéshez megadom a kiszámított változást is mindkét esetben az első refaktoráláshoz.

1. *Példa (2.3. ábra)* A változásokat karakterenként való összehasonlítással számoljuk.

1. A g függvényt áthelyezzük az a modulból a b modulba. A kiszámított változás: `Insert (25 "g/1]).\n -export(["`. Láthatjuk, hogy az algoritmus az export-listában a változást nem egyszerűen egy sor törlésének tekinti, hanem egy ennél finomabb eredményt ad a változásra.
2. A h függvényt áthelyezzük az a modulból a c modulba.
3. Ezután visszavonjuk a refaktorálást a b modulban: a két (1., 2.) változás által érintett terület látszólag nem lapol át az a modulban, azonban a karakterenkénti vizsgálat túl finom eredményt ad már az első refaktorálásnál, így konfliktus lép fel a pozíciók vizsgálatánál. Mindkét refaktorálás visszavonásra kerül, a fájlok az eredeti (0.) állapotnak megfelelő állapotba kerültek.

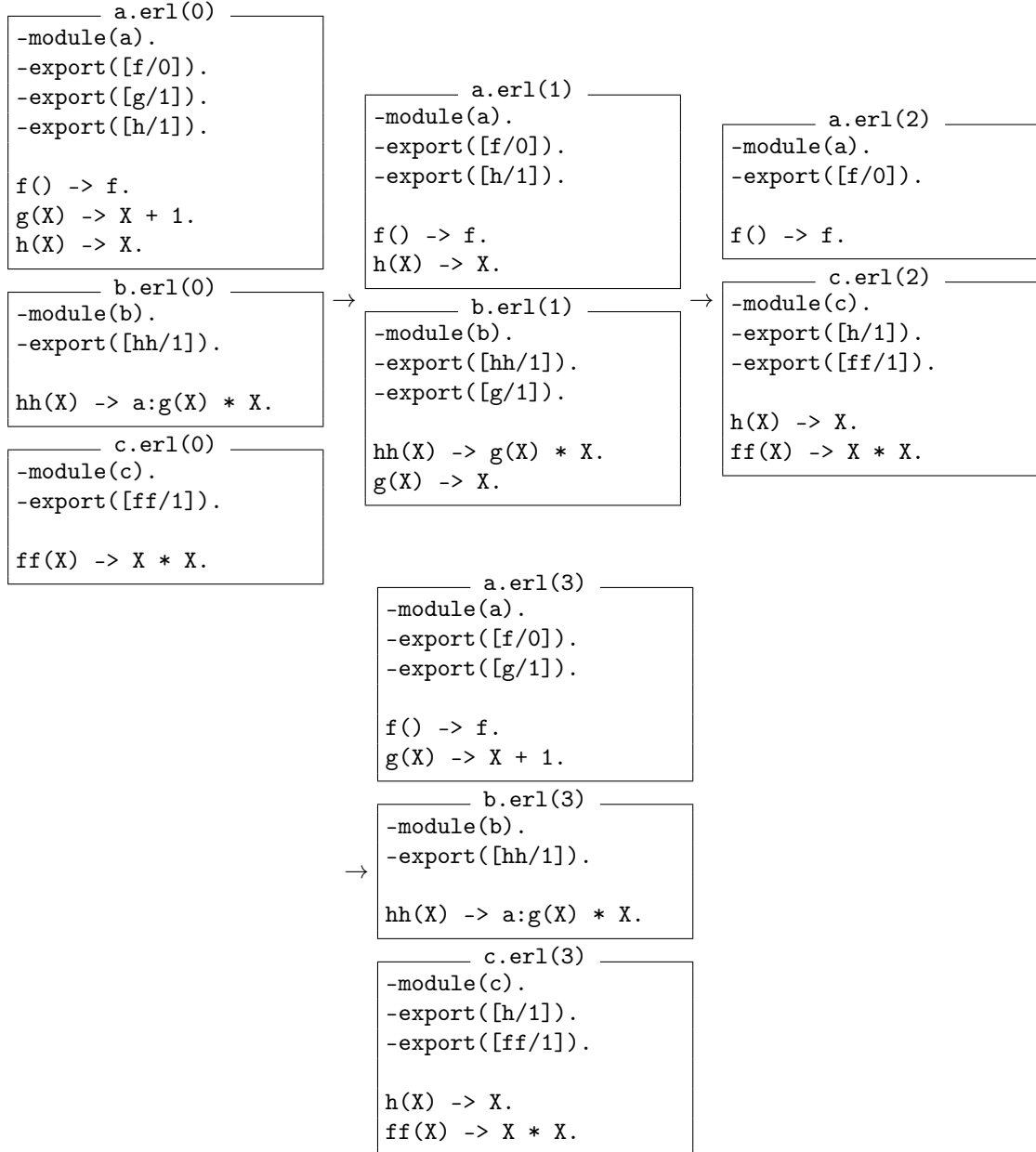
2. *Példa (2.4. ábra)* A változásokat szavanként való összehasonlítással számoljuk.

1. A g függvényt áthelyezzük az a modulból a b modulba. A változások számításának eredménye: `Insert (16 "-export([g/1]).\n")`. Az algoritmus az export-listában a változást elvárásainknak megfelelően egy sor törlésének tekinti.
2. A h függvényt áthelyezzük az a modulból a c modulba.



2.3. ábra. Összehasonlítás karakterenként (konfliktus)

3. Ezután visszavonjuk a refaktorálást a *b* modulban: a két (1., 2.) változás által érintett terület nem lapol át az *a* modulban. A durvább vizsgálat most jobb eredményt ad az előzőhöz képest, valóban nincs konfliktus a pozíciók vizsgálatánál. Az *a* modulba visszakerül a *g* függvény, az időben ezután következett, *c* modul is érintő változás hatása pedig megmarad, sikeres összefésülés történt az *a* modul két változása közt.



2.4. ábra. Összehasonlítás szavanként (sikeres összefésülés)

3. SZINTAXISFA DIFF

3.1. Fák összehasonlítása

Az eddig vizsgált szöveges alapú összehasonlításoknál egy magasabb absztrakciós szintre lépve a szintaxisfa alapján is végezhetjük két forráskód összehasonlítását. Fa típusú adatszerkezetek összehasonlításának egyre nagyobb szerepe van, egyre több területen van igény ilyen jellegű feladat elvégzésére. Napjainkban egyre növekszik az XML-alapú dokumentumok, szolgáltatások felhasználása. Ezekben gyakran van igény a változások felismerésére, különbségek kiszámítására, verziókövetésre, szinkronizálási feladatok elvégzésére. Mivel az XML-dokumentumok fa-szerkezetűek, ezekben a feladatokban szintén fákat kell összehasonlítanunk [7]. Fontos lehet még fák változásainak kezelése bioinformatikai feladatokban – például DNS vizsgálatoknál –, automatikus tételbizonyításoknál, képelemzési feladatokban, fordítók optimalizálásánál. Bizonyos esetekben egy projekt programjainak verziókövetését is szintaxisfa alapon végzik. Erre akkor lehet szükség, ha mélyebb összefüggéseket akarunk felismerni egy program két verziója között. A szöveges alapon történő összehasonlításnak ugyanis ezen a területen korlátozottak a lehetőségei. Ilyen kérdések lehetnek például változók, függvények, típusok törlése, vagy hozzáadása, ehhez kapcsolódó információk. A szakirodalom foglalkozik általános fa adatszerkezetek összehasonlításával, azonban a konkrét feladatok mindig specializációt, optimalizációt igényelnek. A témában a fák összehasonlítására használt angol fogalmak a következők: tree difference [8], tree edit (distance) [6], tree matching [9], tree alignment [5].

A fontosabb modellek mindegyike definiál néhány alpműveletet, primitívet, amiknek a segítségével egy fa átalakítható egy másik fába. Egy ilyen műveletsozaratot nevezünk edit szkriptnek. Ez egyben egy leképezést is definiál két fa között. Az alpművetek más diff algoritmushoz hasonlóan itt is a beszúrás, törlés, frissítés. Ezekhez adódik még a mozgatás, ami azonban gyakran a többi művelettel kerül leírásra. Két fa összehasonlításánál általában több helyes edit szkriptet is előállíthatunk, azonban a cél egy minimális szkript előállítása. Ehhez minden alpművelethez rendelnek egy költséget, így egy szkript költsége az azt leíró műveletek költségeinek az összege lesz. A minimalizálást tehát ezen az alapon kell érteni.

A fa csúcsainak illesztésére gyakran valamilyen szöveges távolságmetrikákat – például Levenshtein távolság – használnak. Ebben általában külön kell kezelnünk a

fa leveleit és a belső csúcsokat. A leveleknél alkalmazhatunk, a belső csúcsok hasonlóságát pedig a részfában levő közös levelek száma alapján becsüljük meg. AST-k összehasonlításánál ezeket a szabályokat tovább pontosíthatjuk, ha a csúcsokat felcímkézzük típusokkal is. A típusok elképzelhetők egy imperatív nyelv esetén: IF – if-utasítás; THEN – then blokk; MI – metódushívás. Így a levelek hasonlóságához megkövetelhetjük, hogy azonos legyen a típusuk. Az algoritmusok pontossága nagyban függ a választott metrikáktól, és korlátoktól. Gyakran szükség van több iterációra is.

3.2. Parse transform-ok vizsgálata

2013-as TDK-dolgozatomban [19] parse transform-ok elemzési lehetőségeit vizsgáltam. Ebben az általam választott módszer szintén szintaxisfák összehasonlításán alapul.

A parse tranform [20] az Erlang [36] nyelv egy lehetősége, amellyel a fordítás folyamatában, a forrás előfeldolgozása, szintaktikus ellenőrzése után tetszőleges átalakításokat végezhetünk a program szintaxisfáján. Ezzel az eszközzel bármely szintaktikusan helyes Erlang kódhoz tetszőleges új szemantikát rendelhetünk.

Lehetséges alkalmazásai ennek az eszköznek különféle szintaktikus cukorkák implementálása a nyelvre. Egy létező felhasználás például az Erlang nyelv bonyolult rekord szintaxisa helyett lehetővé teszi az egyszerűsített, imperatív nyelveknél megszokott jelölést.

A probléma vizsgálatához a motivációt az adta, hogy a RefactorErl nem volt felkészítve parse transform-ok vizsgálatára, nem vette figyelembe annak esetleges változtatásait. Így lehetséges, hogy az az új szemantika, amit egy transzformáció vezetett be, elveszett a programozó szeme elől, nem tudta a kód érintett részeinek valódi jelentését megérteni. Ezért szükséges, hogy a program elemzése során valahogy kezeljük a parse transform-okat is, figyelembe vegyük annak változtatásait, így tároljuk el és jelenítsük meg a kódot az adatbázisban.

Ennek a megoldására több módszert is választhatunk. Ezek közül az egyik legkézenfekvőbb a következő. Meg kell vizsgálnunk a transzformáció előtti, leellenőrzött szintaxisfa és az átalakított szintaxisfa közötti különbségeket. A különbségek alapján különböző heurisztikákkal, illesztésekkel megpróbáljuk megadni a kapcsolatot az eredeti és az átalakított fa csúcsai között. Ezzel később meg tudjuk mutatni a felhasználónak a két állapot közötti kapcsolatokat, segítjük megérteni a transzformáció hatását, a kód tényleges szemantikáját. A választott módszer előnye, hogy nem szükséges a transzformációs kódot értelmeznünk, anélkül próbáljuk kitalálni annak a kódra való hatását, hogy a szemantikáját vizsgálnunk kéne.

Az eredeti és a transzformált szöveget a hagyományos diff algoritmussal is össze-

hasonlíthatnánk, mi azonban szöveges szintről egy magasabb absztrakciós szintre lépünk: szintaxisfákat hasonlítunk össze. Csak így lehetséges ugyanis, hogy az összehasonlításnál olyan szemantikus összefüggéseket is felismerjünk, amit egy hagyományos szöveges vizsgálatnál nem tudnánk felismerni. Felismerhetjük például, hogy a transzformáció egy beszúrási művelete nem egy egyszerű beszúrással volt, hanem egy meglevő csúcs másolása. A feladatot tehát tekinthetjük két szintaxisfa összehasonlításának. Egy másik megközelítésben pedig a problémát a verziókezelés irányába is elterelhetjük. Az eredeti és az átalakított kódot két verzióknak tekinthetjük és speciális módon a különbségeket a szöveges hasonlítás helyett a szintaxisfa alapján számítjuk ki.

A probléma nehézségéhez az is hozzátartozik, hogy – mint említettem – az átalakításra vonatkozóan nincs semmilyen megkötés, így szélsőséges esetben az is elképzelhető, hogy nincs semmilyen tényleges kapcsolat a két kód között. Ez az eset például egy konstans transzformációnál adódik. Ezeket figyelembe véve megszorításokat teszünk a lehetséges kapcsolatokra, olyan összefüggéseket próbálunk felismerni, amelyek értelmes – és létező – parse transform-okban előfordulhatnak, előfordulnak.

3.3. Összehasonlító algoritmus

A dolgozatomban megadtam egy az adott feladatra, Erlang nyelv szintaxisfájára [24] specializált algoritmust, amely megadja két szintaxisfa közti különbséget, pontosabban egy leképezést ad a kettő közt. Az algoritmus párhuzamosan haladva egyenként összehasonlítja az eredeti és az átalakított szintaxisfa csúcsait és megpróbálja azokat egymásra illeszteni. Az összehasonlítás során a fa elemeit rekurzívan vizsgáljuk a gyökér szintjéről az alsóbb szintekre, a levelek felé haladva. Egy lépésben két csúcs összehasonlításánál három illeszkedési szabályt definiálunk.

Ekvivalens típusok Az első szabály illesztésének előfeltétele a két csúcs típusának megegyezése. Ha ez teljesül, elkezdjük rekurzívan lebontani, az adott típusnak megfelelően megvizsgáljuk a részfák egyezőségét. Az egyes csúcsok összehasonlításánál azonban nem követelünk meg minden esetben teljes egyenlőséget, az illeszkedés vizsgálatához egy speciális hasonlósági metrikát alkalmazunk. Infix kifejezéseknél például megengedhetjük, hogy az operátorok különbözőek legyenek.

Ezzel a szabállyal ismerhetjük fel például változók, függvények átnevezését, literalok értékének megváltoztatását, illetve a két csúcs teljes egyezőségét is.

Szerkezeti hasonlóság Különböző típusú csúcsok esetén először azt nézzük meg, hogy szerkezetileg ekvivalensek, vagy legalább hasonlóak lehetnek-e. Először ki-nyerjük mindkét csúcsból a részfákat, amit paramétereknek tekinthetünk. Ezek

például listánál, rendezett n-eseknél a tartalmazott elemek, vagy egy függvényhívásnál a paraméterek. Ezután megvizsgáljuk, hogy az első csúcs paramétereinek halmazát tartalmazza-e a második csúcs paramétereinek halmaza. Egy-egy elem összehasonlításánál itt nem egyszerű egyenlőségvizsgálatot végzünk, hanem rekurzívan összehasonlítjuk őket az algoritmussal.

Ezzel a szabállyal a következő csúcsok illeszkednek: $[X, Y] - \{Y, X, \text{ok}\}$; $[X, Y] - f(X, Y)$. Ahogy a két példában is láthatjuk, itt tehát a csúcs tartalma lényegileg nem változott meg, megvannak az eredeti elemek. Szemléletesen úgy fogalmazhatunk, hogy csak a „csomagalolása”, a külső szerkezete lett átalakítva – például listáról tuple-re, vagy listáról egy függvényhívásra –, még nem is került be egy újabb szint a fában. Ezenkívül azt is megengedjük, hogy új elemekkel bővüljön a részfa.

Szerkezeti tartalmazás A harmadik szabályban rekurzív lebontást alkalmazunk: megvizsgáljuk, hogy egy szintet lebontva fennállhat-e az első típusú illeszkedés, vagy a szerkezeti hasonlóság. Ezt a lebontást mindkét irányban megpróbáljuk, tehát lejjebb megyünk egy szintet az első paraméterben és így hasonlítjuk össze a másik oldal aktuális szintjével, illetve a másik oldal részfáit is összevetjük az első aktuális szintű részfával. Ebben a lépésben tehát csak leveszünk egy szintet a csúcs szerkezetéből, lemegyünk a fában egyel alsóbb szintre, és itt újra sorban megpróbáljuk illeszteni mindegyik szabályt. Így olyan transzformációkat ismerhetünk fel, amelyek új szintet, vagy szinteket vezetett be, illetve valamilyen értelemben csúcsokat vont össze. Példák a harmadik szabály illeszkedésére: $[X, Y] - f([X, Y])$; $\{X, Y\} - [\{X, Y\}, \text{ok}]$.

Az utóbbi két szabály ötlete a fordítóprogramokban ismert ablak-optimalizációból [16] származik. Ennek a lényege, hogy egy kisebb mintát, egy ablakot karakteresen megpróbálunk illeszteni a kódban, keressük annak előfordulásait. Erre leginkább a második szabályunk hasonlít, azonban a probléma vizsgálata közben arra jutottunk, hogy bővítéssel több és általánosabb kapcsolatot is felismerhetünk. Ahelyett, hogy egy mintával való teljes egyezést keresnénk, megengedjük, hogy egy bővebb kapcsolat, szerkezeti hasonlóság álljon fenn. Így az előző két szabály kombinációjával, egymás utáni alkalmazásával újabb illeszkedésekre is eljuthatunk:

$$[X, Y] - f(\{X, Y\}); \{X, Y, \text{ok}\} - f([X, Y]).$$

Ha a fenti három lehetőség közül egyik sem teljesül, az algoritmus az éppen vizsgált páros második elemet egy lehetséges beszúrásnak tekinti, azonban egy második körben újra megpróbálja illeszteni. Ezután az eredeti kódban még nem lép tovább, hanem az éppen vizsgált elem összehasonlítja az új kód következő elemével is. Tehát a bal oldalon maradunk az aktuális elemnél, a jobb oldalon pedig tovább lépünk.

Ha a rekurzív vizsgálat során az eredeti kód listájából elfogynak az elemek, a jobb oldal maradék csúcsát beszúrt elemeknek vesszük, fordított esetben a baloldali elemeket töröltnek.

Miután egyszer végimentünk ezen a vizsgálaton, végzünk egy újabb iterációt azokon az elemeken, amelyeknek nem sikerült biztosan illeszkedő párt találnunk.

3.4. Absztrakt kód

Az alábbiakban megadom az algoritmus egy egyszerűsített leírását. Ehhez pszeudokódot használok.

Az algoritmus első két paramétere a két összehasonlítani kívánt fa. Harmadik paramétere a rekurzív hívásoknál használt lista, ugyanis az eredmény kiszámításához akkumulatív technikát használunk. Először a 2. és 3. sorokban lekérdezzük az éppen vizsgált X és Y csúcsok típusát. Ezután a 4. sorban megpróbáljuk az első szabályt illeszteni a két csúcsra. Ennek feltétele, hogy a típusuk megegyezzen. Ha teljesül, megkezdjük a csúcsok rekurzív összehasonlítását, lebontását. Amennyiben nem teljesül illeszkedési feltétel, továbblépünk a következő szabályra, ami a szerkezeti hasonlóságot vizsgálja. Ennek illeszkedési feltételét jelzi a 6. sorban `struct_similar` logikai függvény. Ha teljesül a feltétel, betesszük az $\{X,Y\}$ párt az eredménylistába és folytatjuk a vizsgálódást. Ha ez a szabály sem illeszkedett, megpróbáljuk a tartalmazási szabályt mindkét irányban (8., 14.). Ha teljesül valamelyik, egyenként összevetjük a tartalmazni vélt csúcsot a tartalmazó részfáival (9., 15.). Ha egyik irányú tartalmazás sem teljesül, a vizsgált párt egy `insert_maybe` előtaggal az eredménybe tesszük, majd egy újabb iterációban megvizsgáljuk a második elemet, hiszen eddig a vizsgálat során ennek a csúcsnak nem találtunk párt az eredeti csúcsok közt.

A definíció 22. sorban kezdődő ága azt az esetet jelenti, amikor a baloldalon végigértünk. A jobb oldal maradék elemeit ekkor mind beszúrtnak vesszük. Hasonló a helyzet a 24. sorban, csak ekkor az új elemek fogynak el, így törölt elemeknek tekintjük az eredeti fából megmaradt csúcsokat. Ha mindkét fa vizsgálatán végigértünk (26.), az eredményt az `Acc` listából kapjuk meg.

```
01 diff_tree([X|Xs],[Y|Ys],Acc):
02     T1=node_type(X)
03     T2=node_type(Y)
04     if T1==T2 then
05         Result=diff_tree(Xs,Ys,check(X,Y) ++ Acc)
06     elseif struct_similar(X,Y)
07         Result=diff_tree(Xs,Ys,{X,Y} ++ Acc)
08     elseif contains(X,Y) then
09         SubX=subtree(X)
10         Result=[]
11         for I in SubX do
12             Result+=diff_tree(I,Y,[])
13             Result+=diff_tree(Xs,Ys,Acc)
14     elseif contains(Y,X) then
15         SubY=subtree(Y)
16         Result=[]
17         for I in SubY do
18             Result+=diff_tree(X,I,[])
19             Result+=diff_tree(Xs,Ys,Acc)
20     else
21         diff_tree([X|Xs],[Ys],{insert_maybe, {X,Y}} ++ Acc)
22 diff_tree([],Ys,Acc):
23     {insert,Ys} ++ Acc
24 diff_tree(Xs,[],Acc):
25     {delete,Xs} ++ Ac
26 diff_tree([],[],Acc):
27     Acc
```

4. ÖSSZEHASONLÍTÁS

Az előző három fejezetben megvizsgáltunk a forráskódok változásaival kapcsolatos három területet. Ahogy a bevezetőben is említettem, a különböző feladatokban különböző igények vannak, így különböző megközelítést igénylik a változások kezelésének. Bizonyos feladatokban elegendő, ha szöveges alapon kiszámítjuk a különbséget a két forrás közt, és az a felhasználó számára megfelelően megformázva megjelenítjük. Erre láttuk a standard diff eszközt, illetve annak különböző módosításait, finomításait. Ezen belül is sok paraméterezési, testreszabási lehetőséget láttunk, amivel például megadhatjuk, hogy soronként, vagy szavanként végezzük az összehasonlítást. Ezeknek a módszereknek a fő előnye a hatékonyság, valamint az, hogy a szöveg tartalmától függetlenül, általánosan használható. Bármilyen szövegre, bármilyen nyelvű forráskódra gyors eredményt kaphatunk. Ezen tulajdonságai miatt általánosan használt eszköz a verziókezelőkben, ahol pontosan az a cél, hogy tetszőleges szövegre, tetszőlegesen nagy méretű projektre hatékony eszközt adjunk a változások számításához, megjelenítéséhez.

Ahogy azonban említettük, az egyszerűség, hatékonyság és az általánosság miatt megvannak a korlátai is egy ilyen eszköznek. Erre adtuk példának a refaktorálásokat kezelő undo/redo modellünket. Ebben a feladatban ugyanis a konfliktuskezelés szempontjából a megszokottnál finomabb részletekre volt szükségünk. Egy refaktorálásnál például egy átnevezéssel olyan apró részletek változhatnak meg, amik konfliktuskezelésnél nem szabad, hogy problémát jelentsenek. Egy egyszerű soralapú összehasonlításnál azonban már kisebb feladatokban is konfliktus léphet fel. Hasonlóan problémát jelenthet egy függvény exportálása, vagy átmozgatása. Ehhez a standard diff algoritmust a soronkénti összehasonlítás helyett szavankénti, illetve bizonyos esetekben karakterenkénti összehasonlítással kellett használnunk. Erre pedig már ráépíthettünk egy finomabb konfliktuskezelő, összefésülő algoritmust.

Ezzel a megoldással már jóval finomabban tudjuk kezelni a forráskódon bekövetkezett változásokat, azonban ennek is vannak korlátai, és bizonyos esetekben még finomabb eredményre van szükségünk. Ezekben a feladatokban kell áttérnünk a szöveges alapú vizsgálatról a szintaxisfa alapúra. Azzal pedig, hogy a teljes szintaxisfa a kezünkben van jóval több kapcsolatot fel tudunk fedezni az egyes nyelvi elemek, szövegrészletek közt. Ezzel verziókezelési feladatokban is pontosabb eredményt adhatunk, illetve különösen nagy jelentősége van olyan alkalmazásokban,

amikor ténylegesen – leggyakrabban nem manuális, hanem automatizált – transzformációkat akarunk elemezni, a forráskód két állapotában az egyes részletek között a kapcsolatot, egy leképezést megadni. Ilyen transzformáció lehet egy refaktorálás is, vagy az előző fejezetben ismertetett parse transform-ok. Ha refaktorálás undo/redo megvalósításánál azonban figyelembe kell vennünk, hogy az undo funkciónak az egyszerű szerkesztéseket is kezelnie kell. Ilyen finom részletek, karakterszintű változások kezelésére pedig túlzottan költséges lenne ez az eszköz. Pontosan ez az egyik hátránya is a szintaxisfa alapú összehasonlításnak, hogy minden egyes lépésnél végre kell hajtunk a kód parse-olását, a szintaktikus elemzést. Ez egy nagy verziókezelő rendszernél nagy költséget jelentene. Ezentúl korlátozást jelent az is, hogy szükségünk van az adott nyelv fordítójára, vagy legalábbis egy parser-re, nincs a kezünkben egy általános eszköz. Problémája jelenthet ezentúl az is, hogy egy repository nem mindig konzisztens, lehetséges, hogy hiányoznak bizonyos fájlok az elemzéshez, vagy valamelyik fájl szintaktikusan hibás, nem lehet az elemzést elvégezni.

A következő példákon keresztül mindhárom megközelítésre mutatok egy-egy példát, bemutatom, melyik eszköz hogy reagál az egyes feladatokra. A szemléltetéshez megadom a szöveges `diff` eredményeket, a szintaxisfák összehasonlításának eredményét azonban a nehéz olvashatóság miatt csak szövegesen magyarázom el.

1. Példa (4.1. ábra) Egy egyszerű Erlang modulban az `add/2` függvényt egy refaktorálással átnevezzük `sum/2`-re. Az erre a két fájlra vonatkozó `diff` eredmény a 4.2 ábrán látható. A kimenethez a `diff unified` formátumát használtam. Ez a formátum könnyen összevethető a `wdiff` kimenetével, ami szintén egy helyen, egy fájlban jeleníti meg az eltéréseket.

Az undo/redo modellünkben a változásokat itt szavankénti vizsgálattal számítjuk, mivel refaktorálás történt. Ez a `wdiff` eredményének felel meg, amit a 4.3 ábra mutat. Láthatjuk, hogy itt jóval finomabb eredményt kapunk: a teljes sorok cseréje helyett csak rövidebb szórészleteket veszünk be a változásokba. Az eredményben látszik, hogy az `add` minden – megfelelő – előfordulását `sum`-ra cseréltük le. Konfliktuskezelés szempontjából mindkét eredmény megfelelő, azonban kisebb részletek tárolás szempontjából, illetve az undo végrehajtásánál is előnyösebbek.

Szintaxisfák összehasonlításánál könnyen felismerhetjük, hogy az `add` függvényhívások lelettek cserélve `sum` függvényhívásra.

sum.erl(0)	sum.erl(1)
<pre>-module(sum). add(N) -> N + 1. add(A, B) -> A + B. caller() -> add(add(1, 2), 3), add(10).</pre>	<pre>-module(sum). add(N) -> N + 1. sum(A, B) -> A + B. caller() -> sum(sum(1, 2), 3), add(10).</pre>

4.1. ábra. Refaktorálás – függvény átnevezése

```
diff -u rename-1.erl rename-2.erl
-module(sum).
add(N) -> N + 1.

-add(A, B) -> A + B.
+sum(A, B) -> A + B.

caller() ->
-    add(add(1, 2), 3),
+    sum(sum(1, 2), 3),
    add(10).
```

4.2. ábra. függvény átnevezése – diff

```
wdiff rename-1.erl rename-2.erl
-module(sum).
add(N) -> N + 1.

[-add(A,-]

{+sum(A,+} B) -> A + B.

caller() ->
    [-add(add(1,-]
    {+sum(sum(1,+} 2), 3),
    add(10).
```

4.3. ábra. függvény átnevezése – wdiff

2. *Példa (4.4. ábra)* Ebben a példában szintén egy refaktorálást hajtunk végre: a `firstsimple` modul `moved_fun` függvényét áthelyezzük a `targetsimple` modulba. Ebben az esetben azt láthatjuk, hogy a soronkénti (4.5, 4.7), illetve szavankénti (4.6, 4.8) vizsgálat nem ad jelentősen eltérő eredményt. Ennek az oka egyszerűen az, hogy a változott szövegeket nem darabolják fel fehér szóközök, így lényegében a szavankénti felbontásnál is egy-egy egységnek látjuk a változott területeket.

A szintaxisfák összehasonlításánál megkapjuk, hogy a `firstsimple` modulból törölve lett a teljes `moved_fun` függvény formja, a `targetsimple` modulba pedig beszúrásra került.

<pre> _____ firstsimple.erl(0) _____ -module(firstsimple). -export([moved_fun/2,other_fun/3]). moved_fun(A,B) -> {A+B, A, B}. other_fun(Pid, Param1, Param2)-> Pid ! {moved_fun(Param1,Param2),self()}. </pre>	<pre> _____ targetsimple.erl(0) _____ -module(targetsimple). -export([sim/2]). sim(A,B)-> {A*B, A, B}. </pre>
<pre> _____ firstsimple.erl(1) _____ -module(firstsimple). -export([other_fun/3]). other_fun(Pid, Param1, Param2)-> Pid ! {targetsimple:moved_fun(Param1,Param2), self()}. </pre>	<pre> _____ targetsimple.erl(1) _____ -module(targetsimple). -export([sim/2]). -export([moved_fun/2]). sim(A,B)-> {A*B, A, B}. moved_fun(A,B) -> {A+B, A, B}. </pre>

4.4. ábra. Refaktorálás – függvény áthelyezése

```

_____ diff -u firstsimple.erl(0) firstsimple.erl(1) _____
-module(firstsimple).
--export([moved_fun/2,other_fun/3]).
-
-moved_fun(A,B) ->
-    {A+B, A, B}.
+-export([other_fun/3]).

other_fun(Pid, Param1, Param2)->
-    Pid ! {moved_fun(Param1,Param2),self()}.
+    Pid ! {targetsimple:moved_fun(Param1,Param2),self()}.

```

4.5. ábra. függvény áthelyezése (1) – diff

```

wdiff firstsimple.erl(0) firstsimple.erl(1)

-module(firstsimple).
[--export([moved_fun/2,other_fun/3]).

moved_fun(A,B) ->
    {A+B, A, B}.-]
{+-export([other_fun/3]).+}

other_fun(Pid, Param1, Param2)->
    Pid ! [-{moved_fun(Param1,Param2),self()}.-]
    {+{targetsimple:moved_fun(Param1,Param2),self()}.+}

```

4.6. ábra. függvény áthelyezése (1) – wdiff

```

diff -u targetsimple.erl(0) targetsimple.erl(1)
-module(targetsimple).
-export([sim/2]).
+-export([moved_fun/2]).

sim(A,B)->
    {A*B, A, B}.
+
+moved_fun(A,B) ->
+ {A+B, A, B}.

```

4.7. ábra. függvény áthelyezése (2) – diff

```

wdiff targetsimple.erl(0) targetsimple.erl(1)
-module(targetsimple).
-export([sim/2]).
{+-export([moved_fun/2]).+}

sim(A,B)->
    {A*B, {+A, B}.

moved_fun(A,B) ->
    {A+B,+} A, B}.

```

4.8. ábra. függvény áthelyezése (2) – wdiff

3. *Példa (4.9. ábra)* A következő példában a hagyományos diff és a patience diff eredményét vetem össze egymással. A példában az algoritmusok haskell-es implementációját használtam [29, 30].

A szövegben töröljük a 7. sort; töröljük a **fact** függvényt, és helyette a 2. sortól beszúrjuk a **fib** függvényt. Ennek megfelelően a **fact** függvényhívást lecseréljük **fib**-re (22. sor).

A hagyományos diff algoritmus (4.10) az első fejezetben tárgyalt okok miatt túlzottan „feldarabolja” a változásokat, megpróbálja összepárosítani a **fact** és **fib** függvény közös sorait. Ezzel szemben a patience diff (4.11) tisztább, átláthatóbb, tömörebb eredményt ad: egyszerűen a teljes függvénytörzset töröltnek, illetve beszúrtnak veszi.

context-a.cpp	context-b.cpp
<pre> 01 #include <stdio.h> 02 03 int frobnitz(int foo) 04 { 05 int i; 06 for(i = 0; i < 10; i++) 07 { 08 printf("Your answer: "); 09 printf("%d\n", foo); 10 } 11 } 12 13 int fact(int n) 14 { 15 if(n > 1) 16 { 17 return fact(n-1) * n; 18 } 19 return 1; 20 } 21 22 int main(int argc, char **argv) 23 { 24 frobnitz(fact(10)); 25 }</pre>	<pre> 01 #include <stdio.h> 02 03 int fib(int n) 04 { 05 if(n > 2) 06 { 07 return fib(n-1) + fib(n-2); 08 } 09 return 1; 10 } 11 12 int frobnitz(int foo) 13 { 14 int i; 15 for(i = 0; i < 10; i++) 16 { 17 printf("%d\n", foo); 18 } 19 } 20 21 int main(int argc, char **argv) 22 { 23 frobnitz(fib(10)); 24 }</pre>

4.9. ábra. Egyszerű szerkesztések

```
diff -u context-a.cpp context-b.cpp
- int frobnitz(int foo)
+ int fib(int n)
+ {
-     int i;
-     for(i = 0; i < 10; i++)
+     if(n > 2)
+     {
-         printf("Your answer: ");
-         printf("%d\n", foo);
+         return fib(n-1) + fib(n-2);
+     }
+     return 1;
+ }

- int fact(int n)
+ int frobnitz(int foo)
+ {
-     if(n > 1)
+     int i;
+     for(i = 0; i < 10; i++)
+     {
-         return fact(n-1) * n;
+         printf("%d\n", foo);
+     }
-     return 1;
- }

int main(int argc, char **argv)
{
-     frobnitz(fact(10));
+     frobnitz(fib(10));
+ }
```

4.10. ábra. Egyszerű szerkesztések – diff

```
patience-diff context-a.cpp context-b.cpp
+ int fib(int n)
+ {
+     if(n > 2)
+     {
+         return fib(n-1) + fib(n-2);
+     }
+     return 1;
+ }
+
+ int frobnitz(int foo)
+ {
+     int i;
+     for(i = 0; i < 10; i++)
+     {
-         printf("Your answer: ");
+         printf("%d\n", foo);
+     }
+ }
- int fact(int n)
- {
-     if(n > 1)
-     {
-         return fact(n-1) * n;
-     }
-     return 1;
- }
-
- int main(int argc, char **argv)
- {
-     frobnitz(fact(10));
+     frobnitz(fib(10));
+ }
+ }
```

4.11. ábra. Egyszerű szerkesztések – patience diff

4. *Példa (4.12. ábra)* Egy létező parse transform [21] lehetővé teszi, hogy literállokkal XML-t írjunk a forráskódban, amit aztán átalakít egy megfelelő szerkezetű rekorddá. Ennek a transzformációnak az elemzéséhez kifejezetten a szintaxisfa vizsgálatára van szükségünk, hiszen itt nem is kapjuk meg szöveges formában a transzformált kódot, nem tudjuk az összehasonlítást szöveges alapon elvégezni.

```
Doc = '<greeting>Hello!</greeting>'
```

```
{xmlElement,greeting,  
  greeting,  
  [],  
  {xmlNamespace,[],[]},  
  [],  
  1,  
  [],  
  [{xmlText,[{greeting,1}],1,[],"Hello!",text}],  
  [],  
  "/tmp",  
  undeclared}
```

4.12. ábra. Parse transform – Literal XML

ÖSSZEGZÉS

Diplomamunkámban megvizsgáltam a forráskódok változásai kezelésének három fő területét, azokra három fő módszert. Először ismertettem a megszokott, szöveges alapú összehasonlítást végző diff algoritmust, illetve megmutattam ennek különböző változatait, finomításait, specializációit. Ezeket az eszközöket leggyakrabban verziókezelő rendszerekben használják. Láttuk, hogy már ebben az egy feladatban is többféle speciális igény megjelenhet, többféle finomításra lehet szükség az alap algoritmusban. A paraméterezés történhet a vizsgálat alapjának megválasztásával – sorok, szavak, esetleg karakterek –, illetve távolságmetrikák bevonásával, és arra vonatkozó korlátok finomhangolásával.

Ezek után egy speciálisabb vizsgálatot igénylő feladatot láttunk a fejlesztőeszközök undo/redo funkciójának megvalósításánál. Itt az előző témakörben már megjelent finomítások mellett fel kellett készülnünk a változások konfliktuskezelésére is. Mivel itt vegyesen jelenhetnek meg kisebb, karakterszintű változások, valamint nagyobb változással járó transzformációk, refaktorálások, szükséges volt, hogy vegyesen használjunk karakterenkénti, illetve szavankénti vizsgálatot.

A harmadik fejezetben láttuk, hogy bizonyos feladatokban, ahol mélyebb, szemantikus összefüggéseket kell felismernünk, szükséges, hogy a szövegesnél mélyebb szintű, szintaxisfa alapú vizsgálatra térjünk át. Ehhez megmutattam az Erlang nyelv parse transform-jainak egy szintaxisfák összehasonlításán alapuló elemzését.

Miután külön-külön bemutattam a három területet, illetve azokhoz három módszert, egymással is összevetettem őket, megvizsgálva az egyes megoldások előnyeit, hátrányait, lehetőségeit és korlátaikat. Ennek szemléltetéséhez konkrét példákat vettem, megmutattam, milyen eredményt ad a három különböző megoldás különböző jellegű feladatokban.

Ezek alapján láttuk, hogy a különböző területeken különböző igények, különböző hangsúllyal jelennek meg, ezért különböző megoldások, finomítások, paraméterezések szükségesek. A problémakör vizsgálatának egy lehetséges folytatása lenne az egyes módszerek vegyítése, figyelembe véve az ismertetett szempontokat, igényeket és tulajdonságokat.

IRODALOMJEGYZÉK

- [1] Eugene W. Myers: *An $O(ND)$ Difference Algorithm and Its Variations* Algorithmica 1, 251-266, 1986
- [2] Gerardo Canfora, Luigi Cerulo, Massimiliano Di Penta: *Identifying changed source code lines from version repositories* Proceedings of the 4th International Workshop in Mining Software Repositories, 2007
- [3] Gerardo Canfora, Luigi Cerulo, Massimiliano Di Penta: *Tracking Your Changes: A Language-Independent Approach* IEEE Software, 26(1), 50-57, 2009
- [4] J. W. Hunt, M. D. Mcilroy: *An Algorithm for Differential File Comparison* Bell Laboratories Computing Science, No. 41., 1976
- [5] Tao Jiang, Lusheng Wang, Kaizhong Zhang: *Alignment of trees — An alternative to tree edit* Lecture Notes in Computer Science, 807, 75-86, 1994
- [6] Philip Bille: *A survey on tree edit distance and related problems* Theor. Comput. Sci, 337, 217-239, 337
- [7] Luuk Peters: *Change Detection in XML Trees: a Survey* 3rd Twente Student Conference on IT., 2005
- [8] Beat Fluri, Michale Wuersch, Martin Pinzger, Harald Gall: *Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction* IEEE Trans. Softw. Eng., 33, 725-743, 2007
- [9] Iulian Neamtiu, Jeffrey Foster, Michael Hicks: *Understanding Source Code Evolution Using Abstract Syntax Tree Matching* Proceedings of the International Workshop on Mining Software Repositories (MSR), 1-5, 2005
- [10] Eelco Lempink, Sean Leather, Andres Löh: *Type-Safe Diff for Families of Datatypes*, Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming, 61-72, 2009, ISBN-978-1-60558-510-9
- [11] Patience Diff.
<http://bryanpendleton.blogspot.hu/2010/05/patience-diff.html>, 2013. 05. 14.

-
- [12] Patience Diff, a brief summary.
<http://alfedenzo.livejournal.com/170301.html>, 2013. 05. 14.
 - [13] Patience Diff Advantages.
<http://bramcohen.livejournal.com/73318.html>, 2013. 05. 14.
 - [14] bazaar – version control system.
<http://bazaar.canonical.com/en/>
 - [15] ChangeDistiller - Source code change extraction using AST differencing.
<http://www.ifi.uzh.ch/seal/research/tools/changeDistiller.html>, 2013. 05. 14.
 - [16] Peephole optimization – Wikipedia.
http://en.wikipedia.org/wiki/Peephole_optimization
 - [17] Szabó Richárd: *FingerTree felhasználása refaktorálás undo/redo megvalósítására* Szakdolgozat, ELTE IK, Budapest, 2011
 - [18] Szabó Richárd: *Undo/Redo modellek refaktorálások során a RefactorErl rendszerhez* TDK-dolgozat, ELTE IK, Budapest, 2011
 - [19] Szabó Richárd: *Parse transform-ok vizsgálata* TDK-dolgozat, ELTE IK, Budapest, 2013
 - [20] Erlang – identity parse transform.
http://www.erlang.org/doc/man/erl_id_trans.html, 2013. 05. 14.
 - [21] Literal XML in Erlang with parse_transform/2.
<http://hyperstruct.net/2007/06/26/literal-xml-in-erlang-with-parse-transform-2/>, 2013. 05. 14.
 - [22] Martin Fowler: *Refactoring - Improving the design of existing code*, Addison Wesley, 2000, ISBN-0201485672
 - [23] RefactorErl - Refactoring Erlang programs.
<http://plc.inf.elte.hu/erlang>
 - [24] Erlang – The Abstract Format.
<http://www.erlang.org/doc/apps/erts/absform.html>, 2013. 05. 14.
 - [25] Thomas Berlage: *A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects* ACM Trans. Comput.-Hum. Interact. 1(3), 269-294, 1994
 - [26] GNU diff.
<http://www.gnu.org/software/diffutils>

-
- [27] GNU wdiff.
<http://www.gnu.org/software/wdiff>
 - [28] diff – Wikipedia.
<http://en.wikipedia.org/wiki/Diff>
 - [29] HackageDB: Standard diff.
<http://hackage.haskell.org/package/Diff>
 - [30] HackageDB: Patience diff.
<http://hackage.haskell.org/package/patience>
 - [31] Longest common subsequence problem – Wikipedia.
http://en.wikipedia.org/wiki/Longest_common_subsequence_problem,
2013. 05. 14.
 - [32] Patch (Unix) – Wikipedia.
[http://en.wikipedia.org/wiki/Patch_\(Unix\)](http://en.wikipedia.org/wiki/Patch_(Unix)), 2013. 05. 14.
 - [33] Finger tree - Wikipedia.
http://en.wikipedia.org/wiki/Finger_tree, 2013. 05. 14.
 - [34] Ralf Hinze, Ross Paterson: *Finger Trees: A Simple General-purpose Data Structure*, Journal of Functional Programming, 16:2, 197-217, 2006,
<http://www.soi.city.ac.uk/~ross/papers/FingerTree.html>, 2013. 05. 14.
 - [35] The Haskell Programming Language.
<http://www.haskell.org>
 - [36] Erlang Programming Language.
<http://www.erlang.org>
 - [37] vim – the editor.
<http://www.vim.org>
 - [38] GNU Emacs text editor homepage.
<http://www.gnu.org/software/emacs>
 - [39] Ian Lynagh: *An algebra of Patches*, 2006. 10. 30.
<http://urchin.earth.li/~ian/conflictors/paper-2006-10-30.pdf>, 2013. 05. 14.
 - [40] darcs revision system.
<http://darcs.net>
 - [41] Darcs - Wikipedia.
<http://en.wikipedia.org/wiki/Darcs>

- [42] Camp version control system.
<http://projects.haskell.org/camp>
- [43] SVN – Apache Subversion.
<http://subversion.apache.org>
- [44] git.
<http://git-scm.com>