Forrásállományok soreredete

Szerző: **Járai Zsolt Sebestyén**,

Babeş-Bolyai Tudományegyetem,

Matematika és Informatika Kar,

Informatika szak, 3. évfolyam

Témavezető: **dr. Darvay Zsolt adjunktus**,

Babeş-Bolyai Tudományegyetem,

Matematika és Informatika Kar,

Programozási Nyelvek és Módszerek Tanszék

XII. Erdélyi Tudományos Diákköri Konferencia – Kolozsvár, 2009. május 15-17

1. Tartalom

2. Bevezető

A szoftver piacon egyre nagyobb és komplexebb alkalmazásokat adnak ki melyek megírásán akár több száz programozó is dolgozhat. A nagyobb terjedelmű alkalmazások fejlesztésekor a programozói cégek gyakran használnak verziókövető rendszereket, mert ezek tárolják a forrásállományok különböző verzióit és lehetővé teszik, hogy egy forrásállományon akár többen is dolgozzanak egyszerre. A fejlesztés során hibák jelenhetnek meg, ezek megszüntetése nehézkes lehet, mivel a hibát tartalmazó forrásállományt több programozó fejlesztheti. A dolgozatban egy olyan algoritmust mutatunk be amely egy adott verzió forrásállományának különböző sorai esetén meghatározza, hogy mely verzióban volt utoljára módosítva. Ezt az információt felhasználva megtudhatjuk a hibát okozó sorok szerzőjét, és a hibát az ő segítségével könnyebben orvosolhatjuk.

Az algoritmust olyan verziókövető rendszerekre fejlesztettük ki amelyek elérhetővé teszik a forrásállományok verziófáját és az állományok különböző verzióit. A ClearCase verziókövető rendszer biztosítani tudja ezeket az információkat.

A dokumentum végén bemutatjuk az algoritmus egy megvalósítását, melyet a ClearCase verziókövető rendszerre fejlesztettük ki, de az alkalmazás tervezése lehetővé teszi más verziókövető rendszerekre való kibővítését.

2.1 Verziókövetés, verziókövető rendszerek

Az információ vezérlés döntő szerepet játszik egy alkalmazás sikeres kifejlesztésében. A legtöbb projekt komplex rendszer, melynek elkészítése számos programozó együttműködését igényli. A programozók sikeres együttműködéséhez szinte elengedhetetlen a köztük levő információ csere vezérlése.

A szervezett szoftver fejlesztés különböző eszközök és módszerek alkalmazását foglalja magába. Ezen eszközök magja az, hogy a forráskódot tárolja és elérhetővé teszi a különböző programozók számára.

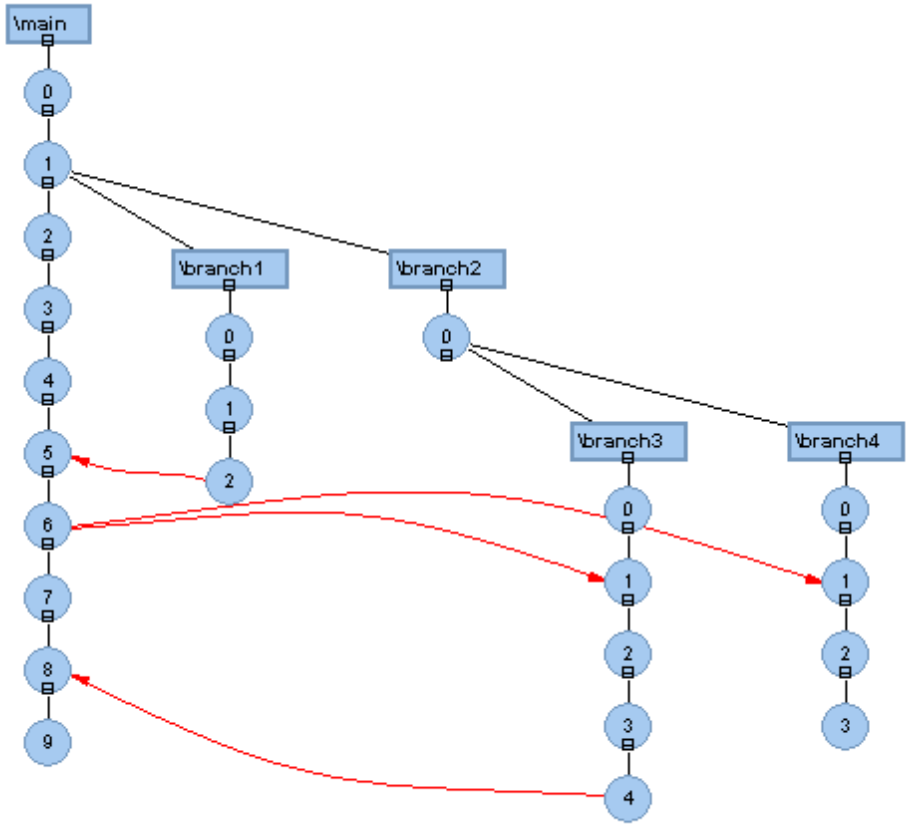
A szoftver fejlesztés során a programozók párhuzamosan dolgoznak különböző állomány halmazokon, amelyek, akár átfedhetik egymást. Belátható, hogy a programozók általi változtatások nyomkövetésére és kezelésére van szükség. Ezt hívjuk verziókövetésnek.

A verziókövetés tehát elérhetővé teszi hogy, ki a felelős egy adott változásért, úgyszintén lehetőséget teremt arra is, hogy megkapjuk a forrásállományok egy órával, egy héttel, vagy egy évvel ezelőtti verzióit. A verziókövetés kapcsán lehetővé válik, az is, hogy a sok fejlesztő hozzájárulását egyesítsük.

A verziókövető rendszereken (alkalmazásokon) keresztül valósul meg a tulajdonképpeni verziókövetés. A legelterjedtebb ilyen rendszerek: ClearCase, Concurrent Version System (CVS), Git és SubVersion (SVN) .

2.2 Alapok a ClearCase verziókövető rendszerben

A ClearCase verziókövető rendszer tárolja az állományok összes **verzióját**. Egy állomány verziói egy hierarchikus **verziófába** vannak logikailag rendszerezve. Egyes állományok verziófája állhat egyetlen ágból(főágból), de a fejlesztők mellék ágakat hozhatnak létre a különböző feladatok elszigetelésére. Lehetőség van két verzió **egyesítésére** is, amely során egy verzióból sorokat viszünk át egy másik verzióba.



2.1. ábra. ClearCase verziófa. A négyzetek tartalmazzák az ágak neveit, a körök jelképezik a verziókat. Baloldalt látható a főág, melyből több mellékágat hoztunk létre. A nyilak az egyesítéseket jelölik.

3. Algoritmus leírása

Az algoritmus feladata, hogy egy verziókövető rendszerben fejlesztett forrásállomány kiválasztott verziójának, megadja a soreredetét. Egy verzió soreredetének nevezzük azt a listát amely a verzió állományának minden soráról tartalmazza, hogy mely verzióból származik.

Az algoritmus során a teljes verziófából kinyerjük azokat a csúcsokat amelyek az algoritmus szempontjából szükségesek és ezekből felépítünk egy egyszerűsített gráfot. Majd ezt bejárjuk a 0. csúcstól indulva és kiértékeljük a csúcsok soreredetét A bejárás során eljutunk a kiválasztott csúcshoz és az előtte levő csúcsok soreredetéből kiértékelhetjük az ő soreredetét is.

A következő Blame algoritmus bemenetei: VT az állomány verziófája és kv a kiválasztott verzió, kimenete a megadott csúcsra kiértékelt soreredet. A csúcsok soreredeteinek tárolására egy soreredetek nevű listát használunk, amely minden csúcsra tartalmazza a csúcs soreredetét. Egy gráf csúcs-halmazára a $V[G_e]$ jelölést használjuk.

Algoritmus Blame(VT, kv, soreredet)

1. Egyszerűsített_Gráf(VT , kv , G_e)
2. **minden** $v \in V\left[G_e\right]$ csúcsra **végezd el**
3. soreredetek[v] = null
4. **vége(minden)**
5. v = 0. csúcs
6. Bejár(v, soreredetek)
7. soreredet ← soreredetek[kv]
8. **vége (algoritmus)**

A Blame algoritmus első sorában meghívjuk az Egyszerűsített_Gráf algoritmust, amely kinyeri a verziófából azokat a csúcsokat amelyekre kimondottan szükségünk van a kiválasztott verzió soreredetének kiértékeléséhez és visszaadja az ezekből álló G_e egyszerűsített gráfot. Ezen algoritmust az 1. alfejezetben részletezzük. A 2-4. sorokban inicializáljuk soreredetek listát. Az 5. sorban kiválasztjuk a kezdeti csúcsot amelyre meghívjuk a rekurzív Bejár algoritmust, amely bejárja a gráfot és feltölti a soreredetek listát mindenik csúcs soreredetével. A 7. sorban a kimeneti soreredetet beállítjuk a kiválasztott csúcs soreredetére.

A következő Bejár algoritmus egy mélységi bejáró rekurzív algoritmus. Minden meglátogatott csúcsra kiértékeljük a soreredetet, majd tovább lépünk azokra a csúcsokra amelyekbe van kimenő él. Azon csúcsok halmazát amelyekből van bejövő él a v csúcsba B[v] -vel míg azon csúcsok halmazát melyekbe van kimenő él v csúcsból K[v] -vel jelöltük.

Algoritmus Bejár(v, soreredetek)

```
1.  Kiértékel(v, soreredet)
2.  soreredetek[v] = soreredet
3.  minden  k∈K[v] végezd el
4.    ha soreredetek[k] = null akkor
5.          kiértékelhet = igaz
6.          minden  bv∈B[k] végezd el
7.                ha  soreredetek[bv] = null akkor
8.                      kiértékelhet = hamis
9.                vége(ha)
10.         vége(minden)
11.         ha kiértékelhet akkor
12.               Bejár(k, soreredetek)
13.         vége(ha)
14.   vége(ha)
15. vége(minden)
16. vége(algoritmus)
```

Az algoritmus első sorában meghívjuk a Kiértékel algoritmust amely a bemeneti paraméterként megadott csúcsra kiszámolja a kimenti paraméterként megadott soreredetet. Ezen algoritmust a 2. alfejezetben részletezzük. A meghatározott soreredetet a 2. sorban átadjuk a soreredetek listának. A 3-15. sorokban megvizsgálunk minden k csúcsot amelybe kivezető él van és ha még nem jártunk ott és ha kiértékelhető a 12. sor rekurzív algoritmushívásával meglátogatjuk őket. Onnan tudhatjuk, hogy egy k csúcsban nem jártunk, ha még a soreredete nincs meg a soreredetek listában. Ezt ellenőrizzük a 4. sorban. Egy k csúcsot akkor értékelhetünk ki ha az összes olyan csúcs soreredete megvan amelyből van él a k csúcsba. Ezt az 5-10. sorokban ellenőrizzük.

3.1 Meghatározó verziók kinyerése a verziófából

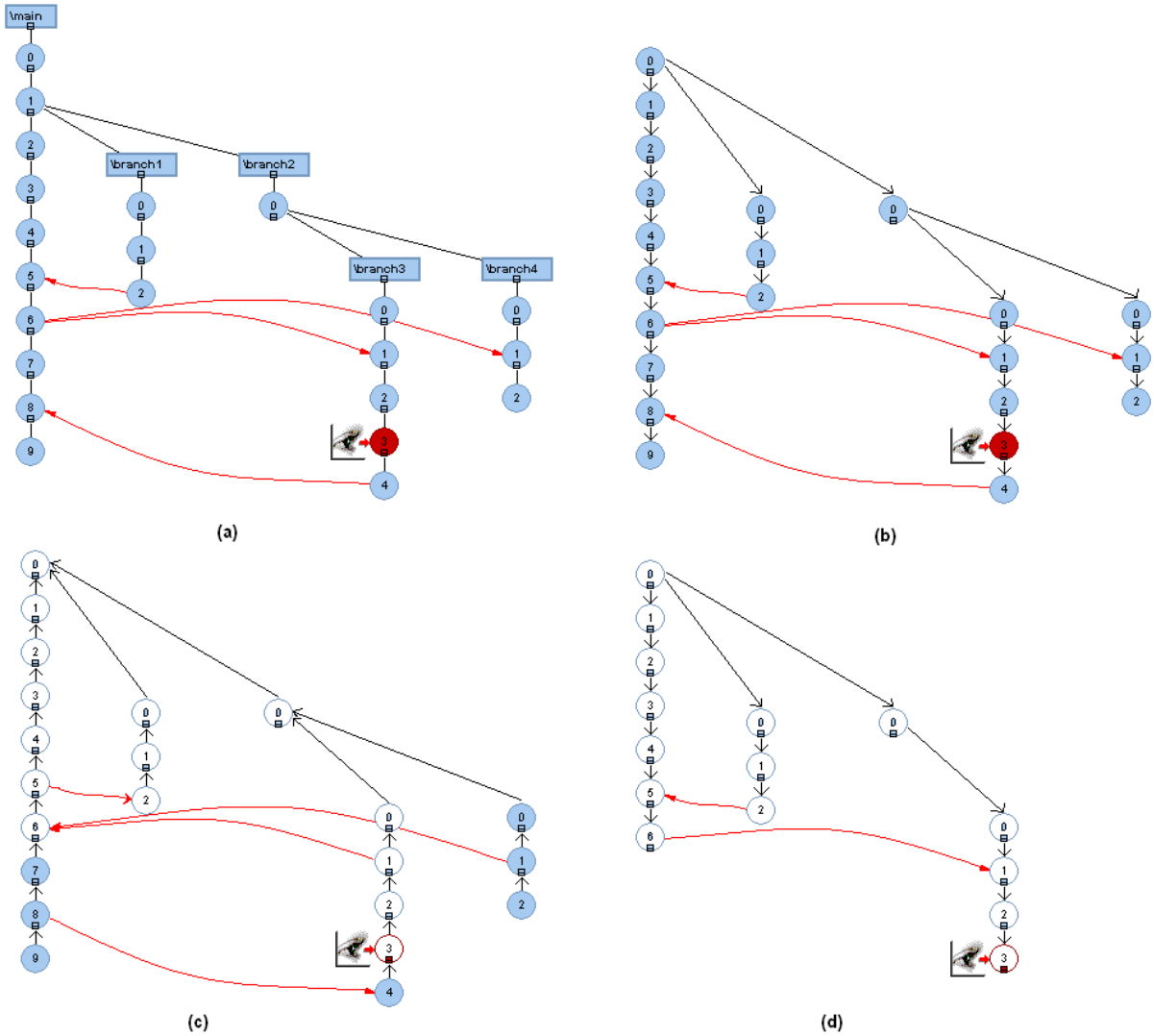
Ahhoz, hogy kiértékeljük a sorok származását a kívánt állományára, sok esetben nincs szükségünk mindenik verzióra, hanem csak azokra amelyekből egyáltalán származhatnak sorok.

Azon verziókból származhatnak sorok amelyek meghatározó ágon vannak. Egy ág akkor meghatározó ha az elemzendő verzió az ágon található, erről az ágról a 0.-tól az elemzendő verzióig érdekelnek a verziók. Egy ág még akkor meghatározó, ha egy verziójából meghatározó ág származik le, erről az ágról a 0.-tól az elágazó verzióig érdekelnek a verziók. Azok az ágak is meghatározó ágak amelyekből egyesítés van más meghatározó ágba, az ilyen ágakról a 0.-tól az egyesítő verzióig érdekelnek a verziók.

Az állomány verziófáját egy $G=(V,E)$ irányított gráffá alakítjuk át. A gráf csúcs-halmazát, melyet $V[G]$ -vel jelölünk, az állomány verziói fogják alkotni, míg a gráf él-halmazát, melyet $E[G]$ -vel jelölünk, a verziók közötti kapcsolatok fogják alkotni, fentről lefele irányítva és az egyesítő nyilaknak megegyező irányítással. A gráf csúcsait ellátjuk egy egyedi kulccsal amely jellemezze azt, hogy a verziófa mely verziójának felel meg.

A verziófából átalakított $G=(V,E)$ gráfunkból létrehozunk egy $G_e=(V_e,E_e)$ egyszerűsített gráfot amelybe csak azok a csúcsok kerülnek be amelyekből sorok származhatnak. Olyan módon határozhatjuk meg ezen csúcsokat, hogy megadjuk azokat a csúcsokat amelyekből elérhető az elemzendő csúcs. Egy s csúcsból elérhető a v csúcs, ha a s-ból eljuthatunk v-be a csúcsok közötti éleken keresztül. Belátható, hogy csak meghatározó ágakon levő csúcsokból juthatunk el az elemzendő csúcsunkhoz.

Algoritmus szempontjából könnyebb ha megfordítjuk a problémát és ahelyett, hogy a gráfban azokat a csúcsokat keresnénk, amelyekből elérhető az elemzendő csúcs, megfordítjuk a gráf éleinek irányítását, s az így megadott $G'=(V,E')$ gráfban, mélységi keresés stratégiával keressük meg azokat a csúcsokat, amelyek elérhetők az elemzendő csúcsból. A mélységi keresés során az utoljára elért, v csúcsból kivezető, még meg nem vizsgált éleket derítjük fel. Ha a v-hez tartozó összes élt megvizsgáltuk, akkor a keresés „visszalép”, és megvizsgálja annak a csúcsnak a kivezető éleit, amelyből v-t elértük. Ezt addig folytatja amíg el nem éri az összes csúcsot, amely elérhető az eredeti kezdő csúcsból. Az összes elért csúcs lesz a $G_e=(V_e,E_e)$ egyszerűsített gráf csúcs-halmaza, és a köztük levő eredeti irányítású élek fogják alkotni az egyszerűsített gráf él-halmazát.



3.1. ábra. A branch3/3 verzióra a meghatározó verziók kinyerése a verziófából. (a) Az állomány verziófája. (b) Az állomány verziófája átalakítva irányított gráffá. (c) A gráf elérhető (fehér), és elérhetetlen csúcsai. (d) A meghatározó verziók melyeket az egyszerűsített gráf ad meg. Úgy hozzuk létre, hogy az eredeti gráfból(b) kizárjuk az elérhetetlen csúcsokat(c).

A következő Egyszerűsített_Gráf algoritmus bemenetként megkapja az állomány verziófáját és a kiválasztott verziót, amelyből meghatározza a kimeneti $G_e=(V_e,E_e)$ egyszerűsített gráfot. Hogy egy csúcsot elértünk vagy sem a bejárva logikai tömbben tároljuk. $K[v]$ azon csúcsok halmazát adja meg amely csúcsokba van kivezető él v-ből.

Algoritmus Egyszerűsített_Gráf(VT , kv , G_e):

1. G ← VT -nek megfelelő irányított gráf
2. $G' \leftarrow G$
3. **minden** $e \in E[G']$ élre **végezd el**
4. e irányítását megváltoztatjuk
5. **vége(minden)**
6. **minden** $v \in V[G']$ csúcsra **végezd el**
7. bejárva[v] ← hamis
8. **vége(minden)**
9. Mk-Bejár(kv, bejárva)
10. V_e ← bejárt csúcsok
11. E_e ← bejárt csúcsok közti élek
12. $G_e \leftarrow (V_e, E_e)$
13. **vége(algoritmus)**

Algoritmus Mk-Bejár(v, bejárva):

1. bejárva[v] ← igaz
2. **minden** $w \in K[v]$ **végezd el**
3. **ha** bejárva[w] = hamis **akkor**
4. Mk-Bejár(w)
5. **vége(ha)**
6. **vége(minden)**
7. **vége(algoritmus)**

Az Egyszerűsített_Gráf algoritmus első sorában beállítjuk G-t a verziófának megfelelő gráfnak. A 2-5. sorokban megfordítjuk a gráf éleinek irányítását így hozva létre egy új G' gráfot. A 6-8. sorokban minden csúcs bejárva értékét hamisra állítjuk. A 9. sorban meghívjuk az Mk-Bejár algoritmust amely a kiválasztott csúcsból elérhető csúcsok bejárva értékét igazra állítja. A 10-12. sorokban felépítjük az egyszerűsített gráfot amely a bejárt csúcsokból és az ezek közti élekből fog állni.

Mk-Bejár(v) hívásakor a v csúcs még nem volt bejárva. Az első sorban v bejárva értékét igazra állítjuk. A 2-4. sorokban megvizsgáljuk v összes w szomszédját, és rekurzív módon bejárjuk a w alatti részt ha w még bejáratlan.

3.2 Soreredet kiértékelése egy adott csúcsra

Egy adott csúcsra kiértékelhetjük a soreredetet ha az összesre melyből él fut be már kiértékeltük.

Úgy határozzuk meg a soreredetet, hogy a csúcsot összehasonlítjuk az összes olyan csúccsal amelyből él fut be. Az összehasonlítás során a befutó élen levő csúcs soreredetét és a két csúcs közötti különbségeket felhasználva felépítjük az összehasonlításból származó soreredetet. Ha a csúcsba csak egy él fut be, akkor, ezen az élen végzett hasonlításból származó soreredet lesz a csúcs soreredete. Ha viszont több csúcsból fut be él, akkor az aktuális csúcs soreredetét, ezen csúcsok összehasonlításából származó soreredetek összefésüléséből kapjuk.

A kezdeti (0.) csúcsba egyetlen csúcsból sincs él, tehát nyilvánvaló, hogy ezen csúcs állományának egyetlen sora sem származhat más csúcs állományából. Tehát abban az esetben ha a kezdeti csúcs soreredetét kell kiértékeljük egyszerűen feltöltjük a soreredet listát annyi, a csúcs egyedi kulcsát tartalmazó, sorral ahány sor van a csúcs állományában.

Algoritmus Kiértékel(v, soreredet):

```
1.  ha v = 0. csúcs akkor
2.          állomány ← v csúcsnak megfelelő állomány
3.          minden i=0,i<állomány sorainak száma végezd el
4.                  soreredet[i] ← v egyedi kulcsa
5.          vége(minden)
6.  különben
7.          soreredet ← null
8.          minden bv∈B[v] csúcsra végezd el
9.                  részeredmény ← Összehasonlít(bv ,v)
10.                 soreredet ← Összefésül(soreredet, részeredmény, v)
11.         vége(minden)
12. vége(ha)
13. vége(algoritmus)
```

Kiértékel bemeneti paramétere egy v csúcs, amelyre az algoritmus meghatározza a soreredetet. Az algoritmus első négy sorában ellenőrizzük, hogy a bemenetként megkapott csúcs nem-e a kezdeti csúcs, mert ebben az esetben nincs több dolgunk minthogy feltöltjük a soreredet listát annyi sorral ahány sor van a csúcs állományában. Ha a bementi csúcs nem a kezdeti csúcs, akkor a 7. sorban a soreredet listát üresre állítjuk, majd a 8-10. sorokban minden olyan csúccsal amelyből él fut be összehasonlítjuk és az ezekből származó soreredeteket összefésüljük. Az Összehasonlít és Összefésül algoritmusokat a következő alfejezetekben tárgyaljuk.

3.3 Összehasonlítás

Két csúcs összehasonlítása során kiértékeljük a hasonlításból származó soreredetet arra a csúcsra amelyre még nincs kiszámolva a soreredet, felhasználva a másik csúcs soreredetét és a két csúcs közötti sor különbségeket. Legyen vs azon csúcs amelyre a soreredet ki van számolva, v pedig a másik csúcs. Az algoritmus során összehasonlítjuk a két csúcsnak megfelelő állományt, ha a két állomány megegyezik vagyis nincs egyetlen különböző sor sem, akkor a hasonlításból származó soreredet a vs csúcs soreredete lesz. Ha léteznek különbségek a két állomány között akkor a vs soreredet lista másolatán végrehajtjuk azokat a módosításokat amelyek a két állomány között történtek. Ha a v csúcs állományában új sor jelenik meg a vs csúcs állományához képest, akkor nyilvánvaló hogy ezen sor a v csúcsból ered. Ha a v csúcs állományában bizonyos sorok hiányoznak a vs csúcs állományához képest, akkor ezen törölt sorok soreredetét nem kell tovább követni, tehát a soreredet listából is töröljük őket. Ha bizonyos sorok módosultak azt a különbséget úgy értelmezzük az állományok összehasonlításakor mint sorok törlése és új sorok beszúrása, tehát ezt az esetet nem kell külön tárgyalnunk az algoritmus szempontjából.

Az Összehasonlít algoritmus bemenetei: vs azon csúcs amelyre már ki van számolva a soreredet, v pedig az a csúcs amellyel össze akarjuk hasonítani vs -t, hogy kiértékeljük az összehasonlításból származó soreredetet. Kimeneti paraméter a hasonlításból származó soreredet. Egy különbséget

Algoritmus Összehasonlít(vs, v, soreredet):

1. soreredet ← soreredetek[vs];
2. $állomány_{vs}$ ← vs csúcsnak megfelelő állomány
3. $állomány_{v}$ ← v csúcsnak megfelelő állomány
4. különbségek ← diff($állomány_{vs}$, $állomány_{v}$)
5. **minden** $k \in különbségek$ különbségre **végezd el**
6. **ha** k.típus = beszúrás **akkor**
7. beszúr soreredet –be v egyedi kulcsát k.kezdőindex -től k.sorokszáma -ig
8. **vége(ha)**
9. **ha** k.típus = törlés **akkor**
10. töröl soreredet -ből k.kezdőindex -től k.sorokszáma -ig
11. **vége(ha)**
12. **vége(minden)**
13. **vége(algoritmus)**

Az első sorban lemásoljuk a vs csúcs soreredetét. A 2-3. sorokban meghatározzuk a két csúcshoz tartozó állományt. A 4.sorban összehasonlítjuk soronként a két állományt megkapva a köztük levő különbségeket. Az 5-12 sorokban a különbségeknek megfelelően módosítjuk a soreredet listát.

3.4 Összefésülés

Két soreredet lista összefésülésére akkor van szükség amikor egy adott csúcsba több más csúcsból van él. Ilyenkor ahhoz, hogy az adott csúcs soreredet listáját meg tudjuk határozni össze kell hasonlítani mindenik csúccsal és az összehasonlításból származó eredményeket kell végül összefésülni.

Összefésülésekor párhuzamosan végigmegyünk a két soreredet lista sorain, és kiválasztjuk, hogy melyik kerüljön az összefésült listába. Ha a két párhuzamos sor megegyezik, akkor az összefésült soreredetbe bekerül ez a sor. Különben, ha a két sor közül az egyik, azon csúcs egyedi kulcsát tartalmazza amelyre aktuálisan fésüljük össze a soreredetet, akkor az eredménybe a másik sor kerül, ha a két sor különböző és egyik sem az aktuális csúcs egyedi kulcsa, akkor nem lehet konkrétan tudni, hogy melyik csúcsból származik a sor, ilyen esetben javasolt, hogy a két sor közül az kerüljön az összefésült soreredetbe, amely a v ágáról származó csúcs egyedi kulcsát tartalmazza.

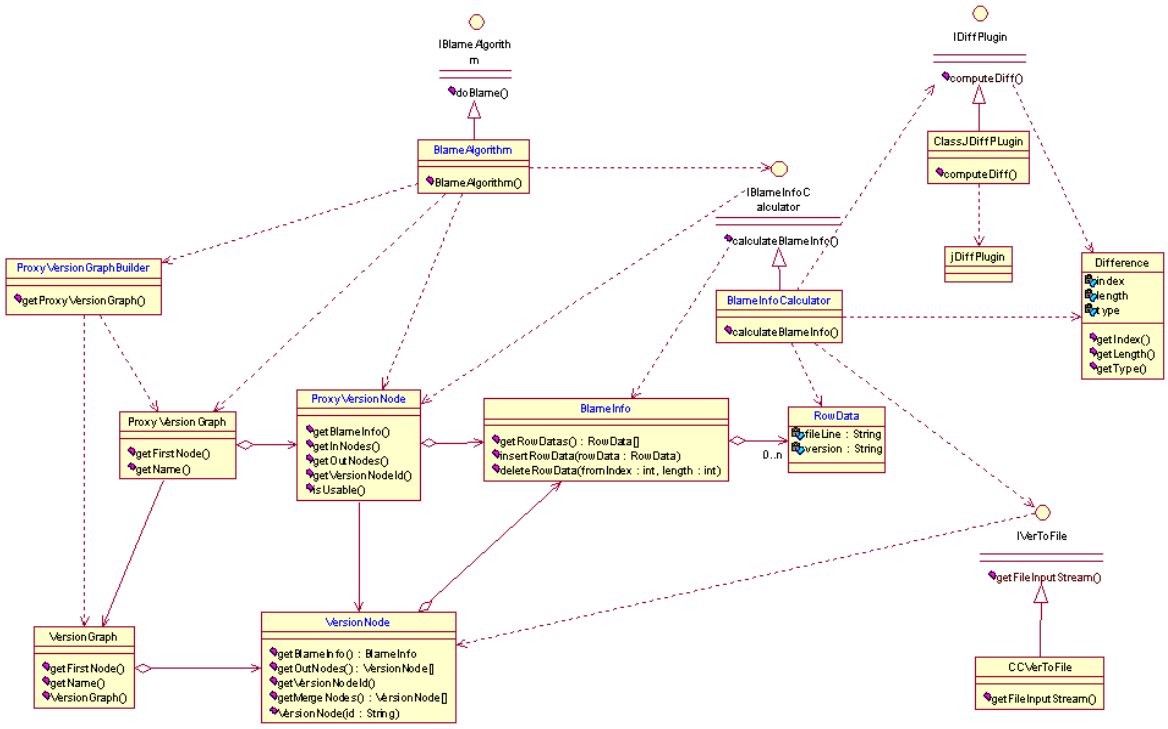
Algoritmus Összefésül(soreredet1, soreredet2, v, soreredet)

```
1.  ha soreredet1 = null akkor
2.      soreredet ← soreredet2
3.  különben
4.      minden i=0, i< soreredet1 sorainak száma végezd el
5.          ha soreredet1[i] = soreredet2[i] akkor
6.              soreredet[i] ← soreredet1[i]
7.          különben
8.              ha soreredet1[i] = v egyedi kulcsa akkor
9.                  soreredet[i] ← soreredet2[i]
10.             különben
11.                 ha soreredet2[i] = v egyedi kulcsa akkor
12.                     soreredet[i] ← soreredet1[i]
13.                 különben
14.                     soreredet[i] ← azon csúcs egyedi kulcsa amelyik v ágán van
15.                 vége(ha)
16.             vége(ha)
17.         vége(ha)
18.     vége(minden)
19. vége(ha)
20. vége(algoritmus)
```

4. Az algoritmus alkalmazása

Ebben a fejezetben az algoritmus egy megvalósítása van bemutatva, melyet a ClearCase verziókövető rendszerre fejlesztettük ki, de az alkalmazás tervezése lehetővé teszi más verziókövető rendszerekre való kibővítését.

Az algoritmusra vonatkozó részt az 5.1 ábrán látható osztálydiagram szemlélteti.



5.1 ábra. Az algoritmus alkalmazását szemléltető osztálydiagram.

A BlameAlgorithm osztály az algoritmus váza, a doBlame metódus hívásakor létrehozunk egy ProxyVersionGraphBuilder objektumot amely a paraméterként megkapott VersionGraph -ra visszaadja a ProxyVersionGrap -ot. Tehát ezen osztály feladta a meghatározó verziók kinyerése a verziófából. Miután megkapjuk az egyszerűsített gráfot, melyet a ProxyVersionGraph objektum tárol, elkezdjük bejárni és az elért csúcsokra meghívjuk a BlameInfoCalculator objektum calculateBlameInfo metódusát amely a soreredet kiértékelését valósítja meg. A csúcshoz tartozó állomány megadásáért a CCVerToFile objektum a felelős. Az állományok összehasonlítására egy jDiffPlugin nevű beépülőt használtunk melyet a ClassJDiffPLugin osztályon keresztül tettünk elérhetővé. Az állományok közti különbségek modellezésére a Difference osztályt hoztuk létre. A bevezetett interfészek lehetővé teszik, az egyes részek lecserélhetőségét.

5. Irodalomjegyzék

[1] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato: *Version Control* with Subversion *For* Subversion *1.5*, O'Reilly Media, USA, 2004

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: *Algoritmusok*, Műszaki Könyvkiadó, Budapest, 1997

[3] Wikipédia: Http*://en.wikipedia.org/wiki/Rational_ClearCase*

[4] William Nagel: Subversion *Version Control: Using The* Subversion *Version Control System in Development Projects*, Courier , Stoughton, Massachusetts, 2005