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

Valósidejű kollaboráció integrálási lehetőségei

TDK DOLGOZAT

Készítette
Szűcs Péter Balázs

Konzulens
Dr. Asztalos Márk

2014. október 26.

Tartalomjegyzék

Kivonat

A kollaboratív szoftverek emberek egy csoportját segítik egy közös feladat elvégzésében, mely tipikusan magában foglalja a megosztott dokumentumok megjelenítését és szerkesztését. Bár a dokumentum szó szöveges tartalmat sejtethet, a kollaboratív szerkesztés témája ennél jóval többre is kiterjed. Szerkeszthetünk megosztva egy grafikus felületet, egy diagrammot, vagy épp egy CAD modellt. Egy valós idejű kollaboratív szerkesztő feladata, hogy lehetőséget biztosítson a megosztott dokumentum párhuzamos, több résztvevő általi olvasására és írására, minnél kisebb válaszidővel, nagy késleltetésű kommunikációs hálózatok felett, mint amilyen az internet, függetlenül a résztvevők földrajzi helyzetétől. A feladat számos kérdést vet fel a tervezéssel és megvalósítással kapcsolatban, a legnagyobb kihívást azonban az ilyen típusú rendszerek esetén a dokumentumok konzisztensen tartása jelenti. A téma immár több mint két évtizede kutatási terület számos eredménnyel és megvalósítással. A dolgozat célja, az eddig megismert módszerek áttekintése, valamint egy általános célú keretrendszer kidolgozása mely elősegíti a kollaborációs funckiók integrálását különböző alkalmazásokban.

Abstract

Collaborative software or groupware is a kind of software that is designed to support a group of people in achieving a common goal. This task typically includes viewing and editing of a shared media, which can be basically anything from a text document to a CAD model provided collaborative editing makes sense. A real-time collaborative editor allows its users who are connected by some communication network, like the internet, to view and edit this shared media in a parallel fashion regardless of their geographical locations. Consistency maintenance proved to be the most significant challenge in designing and implementing these kinds of systems. Due to this, collaborative editing has been research topic for over twenty years. The goal of our research is to create a general-purpose solution for real-time collaborative editing in applications, thus this paper aims to go through some of the important milestones of the past two decades, giving a brief introduction to some of the techniques used in present-day implementations, and also, to create a framework that supports the integration of collaborative features into existing applications.

1. fejezet

Bevezető

A kollaboratív szoftverek elsődleges célja, hogy felhasználók egy kisebb csoportja számára nyújtson lehetőséget valósidejű párhuzamos munkavégzésre az internet segítségével. Ezt úgy érik el, hogy a felhasználók számára lehetővé teszik valamilyen közös megosztott dokumentum párhuzamos szerkesztését. Ez a dokumentum lényegében bármi lehet, amit csak közösen szeretnénk szerkeszteni. A felhasználói kollaboráció témája már az 1980-as években is előkerült, azóta pedig rengeteg eredmény született a területen. Több módszert is ismerünk a probléma megoldására, illetve napjainkban sok szoftver teszi lehetővé a kollaborációt, felhasználás tekintetében azonban továbbra is számtalan lehetőség tárul elénk. Ilyen például a valósidejű kollaborációs funkciók integrálása, melyről ez a dolgozat szól.

1.1. Általános célú platformok

A kollaborációs szoftverek talán legszélesebb körben ismert példája a Google Docs[11], a Google web-alapú irodai csomagja, amely lehetővé teszi felhasználói számára a dokumentumok párhuzamos szerkesztését. A szolgáltatás gazdag felhasználói felülettel rendelkezik, mely minden alapvető funkciót biztosít, amire egy alapvető irodai csomagban szükségünk lehet. A felhasználók követhetik, hogy a többi felhasználó a dokumentum épp melyik részét szerkeszti, illetve ha a szerkesztőben épp azt a szakaszt nézzük amit más is szerkeszt, a változtatások azonnal látszanak (gépelésnél lényegében karakterleütésenként). A szerkesztő felület bármilyen változtatást jól kezel, az egyszerű gépeléstől a másolás/beillesztés műveleteken át a drag & drop műveletekig. Mint termék meglehetősen különböző, de a kollaborációs lehetőségeket tekintve hasonló eszközöket kínál a felhasználók számára a Microsoft által fejlesztett Office 365 [13].

Habár a Google Docs csomag és az Office 365 kimagasló példái a kollaborációs szoftvereknek, a téma nem korlátozódik kizárólag az irodai alkalmazások. A felhasználói együttműködés olyan feladatok esetén lehet kimondottan hasznos, amikor a kihívás egy kisebb csoport munkáját kívánja, valamint a feladat könnyen kisebb (egy ember által is elvégezhető) részfeladatokra osztható. Pédaként tekintsünk egy háromdimenziós szerkesztőt, ahol a feladatunk egy táj megkonstruálása egy házzal és egy híddal, a feladatot pedig három tervezőnek adjuk ki. Ha van előzetes koncepciójuk, az átlapolódás kicsi lesz, a feladat könnyen

részfeladatokra bontható, így a tervezők dolgozhatnak ugyan azon a munkameneten, továbbá a közösen elvégzett feladat kísérleti szempontból is érdekes lehet.

1.2. Kollaboráció a szoftverfejlesztésben

Személyes motivációm a szoftverfejlesztés területéről származik. Amikor egy kisebb csoport dolgozik lényegében ugyanazon a funkción, vagy különböző de erősen összefüggő funkciókon, a kódbázis hamar szerteágazhat. A probléma megoldására a verziókezelők hivatottak, mint például az SVN[17], a Git[18], vagy a Perforce[27]. Ez azonban folyamatos összefésülést kíván tőlünk. Ez akkor hátráltatja kimondottan a munkát, ha a fejlesztők különböző kísérleteket tesznek ugyanazon probléma megoldására. Bizonyos esetekben, a verziózás mellett a valósidejű kollaboráció egy kényelmesebb alternatívát kínálhat. Az ötletet a páros programozáshoz hasonlíthatnánk legegyszerűbben.

Az extrém programozás módszertanában[5] megjelenő páros programozás során két fejlesztő ugyanazzal a kódbázissal dolgozik egy eszköznél[26]. Egyikőjük kódot ír, a másik pedig megfigyel, ötleteket ad, segít, ellenőriz. A szerepeket pedig gyakran cserélik. Alkalmazása közben számomra a szerepváltás gyorsasága, illetve a hirtelen támadt ötletek kommunkálása jelentett gondot a megfigyelő szerepből. Ha például ellenőrizni akarunk egy másik kódfájlban lévő metódust, egy konfigurációs fájlt, anélkül, hogy szerepet (és helyet) cserélnénk, használhatjuk a saját eszközünket és ellenőrizhetjük az elképzelésünket, anélkül, hogy a másik felet félbe kellene szakítanunk, vagy várnunk kellene rá. Ha a munkamenet elején szinkronizáljuk a korábbi változtatásainkat, a munkamenet idejéig a kódbázisunk biztosítottan szinkronban van. A valósidejű kollaboráció egy hatékony kiegészítést jelenthet a páros programozáshoz, illetve fizikai korlátok esetén egy használható alternatívát kínál.

Rengeteg alkalmazást találunk ami megoldja ezt a problémát. Ilyen például a .Net fejlesztéshez használt VS Anywhere[1], ami Visual Studio-n belül teszi lehetővé kódfájlok párhuzamos szerkesztését. Hasonló alternatíva az Eclipse-hez használható DocShare[7], illetve Emacs-hez készített Rudel[9]. Azonban ha fejlesztőként keresünk egy keretrendszert, aminek segítségével kollaborációs funkciókat építhetünk egy alkalmazásba, jóval nehezebb dolgunk akad. Erre kínál megoldást például a beVeeWee fejlesztőkészlet, ami peer-to-peer módon tesz lehetővé felhasználói kollaborációt elsősorban .Net és Silverlight alkalmazásokhoz[4].

A dolgozat célja egy általános célú keretrendszer kidolgozása, mely ezt a feladatot oldja meg, valamint ehhez egy C# nyelvű implementáció elkészítése. A keretrendszer a Mandrake nevet kapta. A Mandrake egy központosított architektúrával működő keretrendszer, ami lényegében egy telepíthető webszolgáltatás formájában szinkronizációs szolgáltatást nyújt az alkalmazásoknak, a kommunikáció pedig egy minimális interfészen keresztül történik.

1.3. A dolgozat tartalmi áttekintése

A kollaborációs szoftverek tervezésekor a legnagyobb kihívást rendszerint a dokumentumpéldányok közötti szinkronizáció és a konzisztens dokumentumállapot fenntartása jelenti.

Ennek miként je több évtizede kutatási terület, épp ezért következő fejezet ezért a terület eddigi eredményeit mutatja be. Az ezt követő fejezetben egy keretrendszer vázát láthatjuk, ami felhasználói kollaboráció integrálását segíti elő alkalmazásokba. Megvizsgáljuk a választott megoldás előnyeit és hátrányait, utóbbiak esetleges megoldási lehetőségeit, valamint egy példán keresztül a keretrendszer felhasználási lehetőségeit. Ezután az utolsó fejezetben egy összefoglalást láthatunk a korábbiakról, valamint néhány elképzelést a jövőbeli fejlesztés lehetséges irányaira vonatkozóan.

2. fejezet

Szinkronizálási stratégiák

A fejezet célja a területen született eredmények és meglévő módszerek elméleti áttekintése. Először távolabbról közelítjük meg a feladatot, majd két eltérő megoldást vizsgálunk meg közelebbről ezek előnyeivel és hátrányaival. Elsőként egy összehasonlítás alapú módszerrel ismerkedünk meg, melynek neve differencia alapú szinkronizáció. Ezután egy művelet transzformáció nevű eseményalapú megközelítést vizsgálunk meg, végigtekintve a módszer fejlődésének fontos mérföldköveit. Először a módszer születésekor használt algoritmust vizsgáljuk meg, majd két kiforrottabb megközelítést láthatunk, amik eltérő módon orvosolják az előbbi problémáit.

2.1. Problémafelvetés

Lássuk tehát valamivel formálisabban mi is a megoldandó probléma. Rendelkezésünkre áll valamilyen megosztott média, ezt nevezzük dokumentumnak. Dokumentum lényegében bármi lehet amit szerkeszteni szeretnénk. Több felhasználónk van, mindegyik saját dokumentumpéldánnyal. Lehetővé szeretnénk tenni, hogy a felhasználóink változtatásai a lokális példányukon megjelenjen a többi felhasználó példányán. Ehhez valamilyen szinkronizációs eljárásra van szükségünk, ami összefésüli a különböző felhasználóktól érkező változtatásokat úgy, hogy az eredmény minden felhasználó számára konzisztens legyen. Nézzük hogyan érhetjük ezt el.

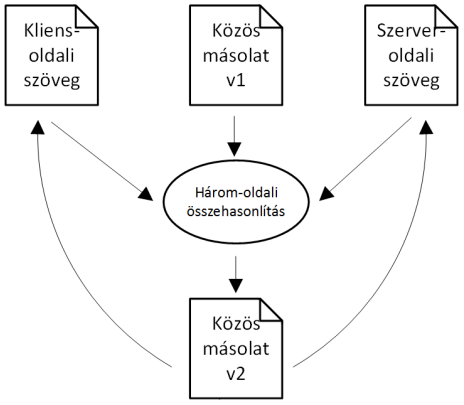
A dokumentumok szinkronizálására három alapvető módszer a pesszimista, a szerkesztés alapú és három-oldali összehasonlítás (three-way merge) alapú módszerek [10].

Legegyszerűbb mind közül a pesszimista megközelítés, vagy zárolás. Ekkor egy felhasználó szerkesztheti a megosztott dokumentumot, a többi felhasználó csak olvashatja. Ez a módszer nyilvánvalóan nem ad teret a közös szerkesztésnek. Ez némileg javítható, ha különböző szekciókat lehet zárolni dinamikusan, így a zárolás egysége a dokumentum helyett a szekció, ami már lehetőséget biztosít a dokumentum egy idejű közös szerkesztésére, a kitűzött célnak azonban továbbra sem felel meg.

A szerkesztés alapú módszer lényege, hogy minden felhasználói műveletet tükrözünk a többi felhasználónál, azaz a felhasználói műveletet végrehajtjuk a többi felhasználónál is. A módszer nehézsége abból fakad, hogy minden felhasználói eseményre helyesen kell rea-

gálnunk. Ahogy később látni fogjuk, már maga a gépelés helyes feldolgozása sem triviális egy elosztott környezetben, azonban a mai felhasználói felületek gazdag eszköztára (find & replace, autocorrect, drag & drop) további nehézségeket vethet fel. A módszer érzékeny a hibákra, hiszen az egyes szerkesztések megváltoztatják a többi szerkesztés helyét, így a hibák könnyen további hibákhoz vezethetnek. Hasonlóan érzékeny pont a csomagvesztés. A szerkesztés alapú rendszerek nem biztosítják természetükből fakadóan a dokumentumpéldányok megegyezőségét. Hátrányai és nehézségei ellenére azonban ez a megközelítés rendkívül elterjedt. Ezt az elképzelést használja többek közt a Google Docs[11], a Mockingbird[2] a CoWord[24], vagy a VS Anywhere[1].

Egy szintén elterjedt szinkronizálási statégia a három oldalú összehasonlítás (vagy three-way merge), amit jól ismerhetünk a különböző verziókezelő rendszerekből, mint az SVN vagy a GIT. Ennek folyamatát nagyvonalakban a 2.1 ábra szemlélteti.



2.1. ábra. *három-oldali összehasonlítás [10]*

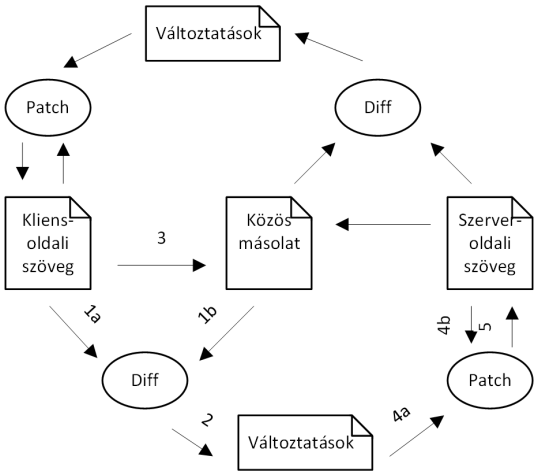
1. A kliens elküldi a dokumentum tartalmát a szervernek
2. A szerver összehasonlítja a szerveren jelenleg tárolt, a felhasználó által küldött, valamint a szerkesztés előtti példányokat, hogy kinyerje a felhasználó változtatásait, majd összefésüli a többi felhasználó változtatásaival.
3. A szerver egy új másolatot küld a kliensnek.

A szerveroldali három-oldali összehasonlítás nem skálázódik elég jól ahhoz, hogy valós idejű kollaboráció kivitelezhető legyen a segítségével, némi fejlesztéssel és továbbgondolással azonban, alkalmassá tehető egy ilyen szolgáltatás megvalósítására.

Lássuk tehát, hogyan tudjuk a fenti elképzeléseket felhasználni valósidejű kollaboráció megvalósításához. A következő alfejezetekben elsőként egy három-oldali összehasonlításon alapuló módszert fogunk látni, majd megvizsgáljuk hogyan tudunk konzisztens dokumentumállapotot biztosítani szerkesztésalapú módszerek segítségével.

2.2. Differencia alapú szinkronizáció

A differencia alapú szinkronizáció egy szimmetrikus módszer, mely lényegében egy végtelen ciklusban differenciát képez és összefésül. Ennek menetét a 2.2 ábra szemlélteti.



2.2. ábra. *differencia alapú szinkronizáció hálózat nélkül[10]*

A két szöveg bármikor módosulhat, célunk, hogy mindig közel megegyező állapotban tartsuk ezeket, valamint hogy szerkesztés mentes időszakban, a két dokumentum állapota megegyezzen.

1. A kliensoldali szöveg és a közös másolat alapján differenciát képzünk(diff).
2. Ez visszaadja a változtatásokat a kliens oldalon.
3. A kliensoldali szöveggel felülírjuk a közös másolatot. Fontos, hogy amivel felülírjuk, megegyezik az 1. lépés beli szöveggel.
4. Best-effort jelleggel elkészítünk a szerveroldali szöveg és a változtatások alapján egy javítást(patch).
5. Ezzel frissítjük a szerveroldali szöveget.

Ezek után a folyamat szimmetrikusan ismétlődik a másik oldalra is. Ez úttal a közös másolat megegyezik a kliensoldali szöveggel. A megoldást a patch fuzzy jellege teszi igazán előnyössé, vagyis az, hogy még akkor is jó eséllyel alkalmazható, ha közben a dokumentum-állapot változott. Így ha a kliens gépelt is a szinkronizáció ideje alatt, a szervertől érkező patch valószínűleg rendelkezik felismerhető kontextussal, hogy még mindig sikeresen alkalmazható legyen. Ha az összes patch sikertelen, akkor ezek a következő diff-ben törlendően jelennek meg és törlésre kerülnek a következő ciklusban. Nézzük meg az algoritmus működését egy konkrét példán keresztül[10].

1. A kezdeti állapot mindenhol "Macs had the original point and click UI"
2. A kliensoldali szöveget szerkesztjük a következőképp: "Macintoshes had the original point and click interface"
3. A közös másolatot frissítjük: "Macintoshes had the original point and click interface"
4. Ezalatt a szerveroldali szöveg megváltozott: "Smith & Wesson had the original point and click UI"

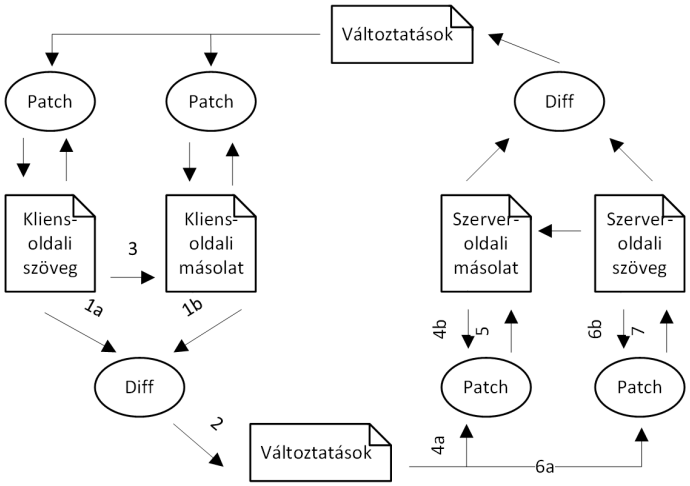
5. A 4. lépésben mindkét változtatást megpróbáljuk összefésülni a szerveroldali szöveggel. Az első esetében ez sikertelen, mivel a kontextus túl sokat változott, ahhoz hogy be tudjuk illeszteni az "intoshe"-t bárhova is, a másodikat azonban gond nélkül alkalmazni tudjuk.

6. Az 5. lépés eredményeképp kapott szerveroldali szöveg most "Smith & Wesson had the original point and click interface".

7. Az algoritmus most megismétlődik a másik irányba szimmetrikusan, melynek eredményeképp figyelmen kívül hagyjuk a "Macs" → "Macintoshes" változtatást és felülírjuk "Smith & Wesson"-al. A "UI" → "interface" változtatást érintetlenül hagyjuk.

2.2.1. Saját másolat használata

Az előzőekben leírt módszer a differencia alapú szinkronizáció legegyszerűbb módja, ami azonban nem működik egy kliens-szerver környezetben, hiszen a közös másolat nem biztosítható. A működés érdekében két félnek saját másolattal kell rendelkeznie. Ezt a kiegészítést szemlélteti a 2.3 ábra.



2.3. ábra. *Saját másolat használata [10]*

A szerveroldali szöveg és a kliensoldali másolat, valamint hasonlóan a kliensoldali szöveg és a szerveroldali másolat a szinkronizáció bármelyik felében meg kell hogy egyezzen.

2.2.2. Hatékonysági kérdések

A differencia alapú szinkronizáció legköltségesebb lépései a különbségképzés és a összefésülés műveletek, így a gyorsabb válaszidő és a nagyobb mértékű skálázhatóság érdekében fontos megvizsgálni, hogy tudunk-e javítani ezek hatékonyságán. Fontos szempont továbbá a pontosság, az ütközések súlyosságának csökkentése érdekében.

A differenciaképzés két eltérő szerepet tölt be a szinkronizációs ciklus során. Az egyik a szerveroldali másolat frissítése a kliensoldali szöveg és a kliensoldali másolat jelenlegi tartalmával, melynek eredményeképp a három szöveg azonos kell hogy legyen. Ez az egyszerűbb feladat, mely bármilyen eljárást használhatnánk, például csupán a változtatások

elküldését, vagy akár a teljes szöveg átküldését is. A második, több kihívást jelentő feladat, a szerveroldali szöveg frissítése a kliensoldali szövegen végzett módosításokkal. Mivel a szerveroldali szöveg időközben változhatott, így a kapott differenciának szemantikailag értelmesnek kell lennie.

A probléma szemléltetése édrekében nézzünk meg egy konkrét példát. A kliensoldali szöveg kiindulási állapota legyen "Ez egy török dal". Ha a "török" szót töröljük és kicseréljük "görög"-re, a változtatást felfoghatnánk úgy is, mintha kicseréltük volna a szó első és utolsó betűjét. Ez a minimális differencia. Ez azonban nem tükrözi a felhasználó szándékát, hiszen ő egy szót változtatott meg, nem két betűt, a középső rész azonossága csupán véletlen.

kliensoldali szöveg: Ez egy török dal

kliensoldali másolat: Ez egy görög dal

minimális differencia: Ez egy ~~t~~g<u>ö</u>rö~~k~~g<u>g</u> dal

szemantikus differencia: Ez egy ~~török~~<u>görög</u> dal

Ez a megkülönböztetés azért számít, mert ha például közben egy másik felhasználó megváltoztatja a szerveroldali szövöeget "török"-ről "tücsök"-re az eredmények vagy "görög"-nek kell lennie (kliens győz), vagy "tücsök"-nek (szerver győz), de semmiképpen sem "gücsök"-nek, ami ez esetben az összefésült eredmény. Így egy olyan algoritmust kell biztosítanunk, ami szemantikailag értelmes differenciát szolgáltat[10].

További problémát jelent a skálázhatóság. A jelenlegi leghatékonyabb algoritmusok lépésszáma $O(nd)$, ahol n a szöveg, d pedig a változtatások hossza[14], ami nem skálázódik jól nagy méretű dokumentumok, vagy jelentős méretű változtatások esetén. Sokat javíthatunk azonban a hatékonyságon, bizonyos Egyszerűsítések segítségével.

egyenlőség: Mivel a szinkronizációs ciklust valamilyen időközönként hajtjuk végre (nem pedig mondjuk karakterleütések után) szinkronizálás során a legtöbb differencia képzésnél két megegyező szöveget hasonlítunk össze, ezt a speciális esetet külön kezelve nyerhetünk a hatékonyságon.

közös előtag/utótag: Ha van közös rész a két összehasonlítandó szövegben, valószínű, hogy van megegyező szövegrész a változások előtt és/vagy után. Eltávolítva ezeket a közös elő-és utótagokat, az összehasonlítandó szövegrész jelentősen lecsökken. Például:

1. szöveg: A macska

2. szöveg: A fekete macska

Ebben az esetben a közös részek eltávolítása után az egyik az üres szövegre egyszerűsödik, a másik pedig a "fekete" szóra. A közös elő-és utótagokat lineáris időben felderíthetjük egy egyszerű ciklus segítségével.

egyszerű beszúrás/törlés: Miután a közös elő-és utótagokat eltávolítottuk, gyakori, hogy valamelyik szöveg üres. Ekkor attól függően, hogy melyik, tudhatjuk, hogy egyszerű beszúrásról, vagy törlésről van szó. Az előző példánkban, mivel az első szöveg üres, tudjuk, hogy a "fekete" szót szúrtuk be a szövegünkbe. Ilyen esetekben egy általános célú algoritmus futtatása fölösleges volna.

Ennek figyelembevételével, valamint további heurisztikák segítségével [10], a differencia alapú szinkronizáció egy jól skálázható és viszonylag könnyen megvalósítható, már jól ismert algoritmusokon alapuló megoldást biztosít valós idejű kollaboráció megvalósítására.

2.2.3. Alkalmazhatóság

Nézzük meg tehát, hogyan tudjuk alkalmazni a kitűzött cél megvalósítására. Ahogy láttuk, a differencia alapú szinkronizáció kiválóan alkalmazható szöveges dokumentumok esetén, illetve olyan esetekben, amikor a megosztott dokumentum hatékonyan reprezentálható szöveges formában. Ilyet számos helyen találunk. A grafikus felhasználói felületek tervezésére például a fejlesztői könrnyezetek jellemzően biztosítanak egy grafikus tervező felületet, az eredményt pedig valamilyen xml-szerű formátumban tárolják.

Mi történik azonban, ha visszatérünk a bevezetőben említett példához. Egy háromdimenziós modellező szoftverrel dolgozunk párhuzamosan és el akarunk forgatni egy objektumot. Ehhez adott háromszögek egy halmazát kell elforgatnunk egy megadott tengely körül. Habár el tudjuk képzelni a jelenet valamilyen szöveges reprezentációját (a mentett fájlból kiindulva például) és végezhetünk ezen összehasonlításokat és összefésüléseket, a megoldás nem tűnik sem szépnek, sem hatékonynak. Számos hasonló példát hozhatnánk fel, ami azt mutatja, hogy a módszer alkalmazhatóság szempontjából limitált, így egy általános célú keretrendszer támogatására nem alkalmas. Lássuk tehát, mit kínál számunkra a szerkesztésalapú megközelítés.

2.3. Művelet transzformáció

A művelet transzformáció alapötlete a következőképp szemléltethető. A felhasználói módosításokat (gépelés, törlés, beillesztés, stb.) *műveleteknek* tekintjük, melyek különböző *forrásokból* származhatnak. A fejezetben két művelettel foglalkozunk, a beillesztéssel (*Insert["karakterlánc", pozíció]*) és a törléssel (*Delete[karakterek száma, pozíció]*). Forrás alatt pedig lényegében egy szerkesztő felületet értünk, ami lehet például egy webes felület (Google Docs), vagy épp egy vastagkliens. Mivel az egyes műveletek a hálózati késleltetés miatt a különböző forrásoknál különböző időkben és sorrendekben hajtódnak végre, mielőtt egy műveletet végrehajtunk meg kell vizsgálnunk (minden forrásnál), hogy valamely korábban végrehajtott művelet megváltoztatta-e a dokumentumot úgy, hogy az új műveletünk már nem hajtható végre eredeti formájában. Ha igen, *transzformáljuk* úgy, hogy az új állapotban is helyes legyen a végrehajtás eredménye. Legyen O_a és O_b két művelet. Ekkor a transzformációra egy $T(O_a, O_b)$ füvvgényként tekinthetünk, ami meghatározza O_a O_b szerinti transzformáltját, O'_a-t. Például ha egy távoli forrástól az $O = Insert[\text{"}a\text{"}, 2]$

műveletet kapjuk, miközben töröltük az első karaktert a lokális másolatunkon, a kapott műveletet transzformálnunk kell az alábbiak szerint.

$$O' = T(Delete[1,1], Insert["a", 2]) = Insert["a", 1] \tag{2.1}$$

Ahogy a fejezet bevezető részéből kiderült, az egyidejű megosztott dokumentumszerkesztés egy elosztott környezetben számos problémát vet fel. Ilyen környezetben, elvárt alacsony válaszidővel, párhuzamos szerkesztés mellett, komoly kihívást jelent a konzisztencia fenntartása.

A csoportos, párhuzamos szerkesztést lehetővé tevő alkalmazások területén végzett kutatások az elmúlt évtizedekben egy újszerű technikát dolgoztak ki a konzisztencia megőrzésére. A módszer a művelet transzformáció (operational transformation) nevet kapta, elsőként pedig a GROVE (GRoutp Outline Viewer Editor) rendszerben alkalmazták 1989-ben[8]. Megszületése óta számos kutatócsoport bővítette ezt a technikát és alkotta meg saját implementációját. Ilyen többek között a REDUCE[19] és a Jupiter[15], melyekről később konkrétan is szó esik. A fejezet a téma három mérföldkövét tekinti át. Elsőként a GROVE rendszerben alkalmazott technikákat, majd egy, a GROVE által megoldatlan problémát ismerünk meg, végül két későbbi rendszert, melyek két eltérő megközelítéssel oldják meg a problémát.

2.3.1. Kauzalitás és függetlenség

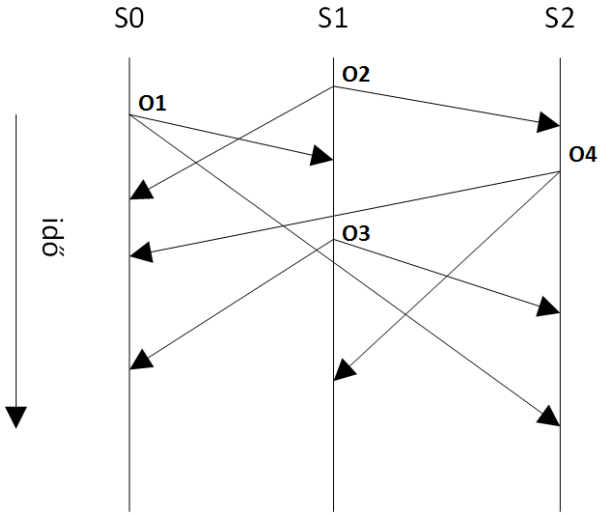
Ebben a szakaszban bemutatásra kerül néhány alapvető fogalom és elképzelés, melyek későbbi fejtegetéseink alapjául szolgálnak majd[12].

Kazuális rendezési reláció Legyen O_a és O_b két művelet az i és j forrásokból. Ekkor $O_a \rightarrow O_b$, azaz O_a kauzálisan megelőzi $O_b - t$, akkor és csak akkor igaz, ha (1) $i = j$ és O_a előbb jött létre mint O_b, vagy (2) $i \neq$j és O_a j forrásnál értelmezett verziója előbb hajtódott végre a j forrásnál, mint hogy O_b létrejött volna, vagy (3) létezik egy O_c művelet, melyre $O_a \rightarrow O_c$ és $O_c \rightarrow O_b$.

Függetlenség Legyen O_a és O_b két művelet. Ekkor $O_a \mid\mid O_b$, azaz a két művelet független, ha sem $O_a \rightarrow O_b$, sem pedig $O_b \rightarrow O_a$ nem teljesül.

Példaként tekintsük a 2.4 ábrán bemutatott forgatókönyet, három forrással (S_1, S_2 és S_3) és négy művelettel (O_1, O_2, O_3 és O_4). Feltesszük továbbá, hogy a művelet a lokális forrásnál azonnal végrehajtódik és csak ezután kerül továbbküldésre a többieknek. Az ábrán a nyilak reprezentálják a műveletek továbbítását. A függőleges vonalakra úgy tekinthetünk, mint a műveletek sorrendjére az adott forrásnál, így például S_1-nél először O_2 hajtódik végre és ezt követi O_1, O_3 és végül O_4

Az előző két definíció alapján tehát három egymástól függő műveletpárunk van, név szerint $O_1 \rightarrow O_3$, $O_2 \rightarrow O_3$, valamint $O_2 \rightarrow O_4$, mivel O_1 O_3 létrejötte előtt hajtódik végre, O_2 előbb jön létre mint O_3, végül pedig O_2 előbb kerül végrehajtásra, mint hogy



2.4. ábra. *Egy forgatókönyv [21]*

O_4 létrejönne. Ezen kívül három független műveletpárunk van, ezek pedig $O_1 \parallel O_2$, $O_1 \parallel O_4$, illetve $O_3 \parallel O_4$. Mivel O_1 és O_2 különböző forrásokból származnak, a rendezési reláció definíciójának (2)-es esetét kell tekintenünk. Mivel O_1 nem hajtódik végre előbb S_1-nél mint ahogy O_2 létrejön, $O_1 \rightarrow O_2$ nem teljesül, valamint mivel O_2 nem hajtódik végre S_0-nál O_1 létrejötte előtt, $O_2 \rightarrow O_1$ sem teljeseül, azaz a függetlenség fenti definíciója értelmében $O_1 \parallel O_2$. Ugyanezt a gondolatmenetet alkalmazhatjuk a másik két műveletpárra is.

2.3.2. A GROVE-féle megközelítés

A GROVE egy csoportos szövegszerkesztő, amit az MCC-nél[25] fejlesztettek ki azzal a céllal, hogy többen egyidejűleg szerkeszthessenek szöveges vázlatokat. [8]

A GROVE rendszer működése az előbb lefestett példánk alapján történt. Minden weboldalnak van egy saját másolata a dokumentumról, a helyi műveletek azonnal végrehajtódnak, majd továbbküldésre kerülnek.

A GROVE fejlesztésekor két konzisztenciával kapcsolatos problémát definiáltak ahhoz kötődően, hogy a műveleteket érkezési sorrendjükben, eredeti formájukban hajtjuk végre. Ezek a *divergencia*, valamint a *kauzalitás megsértése*. Példaként vegyük ismét a 2.4 ábrán felvázolt forgatókönyvunket. Ha a műveleteket érkezésük sorrendjében változtatás nélkül hajtjuk végre, az alábbi végrehajtási sorrendeket kapjuk: O_1, O_2, O_4 és O_3 az S_0 oldalon; O_2, O_1, O_3 és O_4 az S_1 oldalon; és O_2, O_4, O_3 és O_1 az S_2 oldalon. Ha a műveletek sorrendje nem felcserélhető, a végeredmény az egyes oldalakon nem megegyező ez a *divergencia* problémája. Továbbá, mivel a műveletek szinkronizálás nélkül jönnek létre és kerülnek továbbításra, fennálhat olyan eset, amikor az egyes műveletek természetes, kauzális sorrendjüktől eltérően érkeznek meg és hajtódnak végre. Ismét előző példánkra tekintve láthatjuk, hogy O_3 azután keletkezik, hogy O_1 megérkezne S_1-hez, így $O_3 \rightarrow O_1$. Azonban mivel S_2-höz O_3 előbb érkezik meg mint O_1, O_3 végrehajtása O_1 előtt nem definiált művelethez vezethet, mivel O_3 egy nemlétező kontextusra hivatkozik, amit O_1 állítana elő. Ez a probléma a *kauzalitás megsértése*. [21]

Konvergencia és sorrendiség

A fent említett két probléma alapján a GROVE-féle helyesség kritériumot két tulajdonság határozza meg.

Konvergencia *Nyugalmi állapotban* (azaz amikor minden létrejött művelet minden oldalon végrehajtódott) az összes dokumentumpéldány megegyezik.

Precedencia Ha O_1 kauzálisan megelőzi O_2-t, akkor O_1 műveletnek minden oldalon O_2 előtt kell végrehajtódnia.

A GROVE alkotói egy olyan megoldást kerestek, ahol a műveletek végrehajtási sorrendjére az egyetlen megszorítás a műveletek közti kautális rendezési reláció. Ennek eredményeképp született meg a dOPT (distributed OPerational Transformation) algoritmus, melyet később konkrétabban is vázolok. A megoldásnak két lényeges komponense van. Az egyik egy időbélyegeket tároló lista, a precedencia biztosítására, a másik az előbb említett dOPT algoritmus, ami a konvergenciáért felelős.

A transzformációs tulajdonság

A konvergencia biztosítása érdekében a dOPT algoritmus a T transzformációs függvénytől a következő tulajdonságot követeli meg. Bármely két O_1 és O_2 független műveletre, legyen $O_1' = T(O_1, O_2)$ és $O_2' = T(O_2, O_1)$. Ezekre teljesülnie kell, hogy

$$O_1 \circ O_2' \equiv O_2 \circ O_1' \tag{2.2}$$

ahol a $\circ$ operátor két művelet egymás utáni végrehajtását jelenti, az $\equiv$ pedig, hogy a két műveletsor eredménye azonos dokumentumállapotot eredményez.

Az előbb definiált tulajdonságon felül a GROVE tervezői azt is felismerték, hogy vannak esetek, amikor a transzformációs függvénynek biztosítania kell, hogy az idempotens műveletek helyesen hajtódjanak végre. Példaként vegyünk két műveletet, O_1 és O_2, független törlés műveleteket, melyek ugyanarra a pozícióra hivatkoznak. T-nek ekkor biztosítania kell, hogy a végrehajtás sorrendjétől függetlenül csak egy karakter törlődhet. Ezt a kitételt a transzformációs tulajdonság előző definíciója nem ragadja meg.

A dOPT algoritmus

A dOPT egy listát tart nyilván az elvégzett műveletekről, ez a napló, vagy *Log*. A célunk tehát, hogy az érkező O művelethez meghatározzuk a transzformált O' műveletet, amit végrehajthatunk. Az eljárást az Algoritmus 1 vázolja.

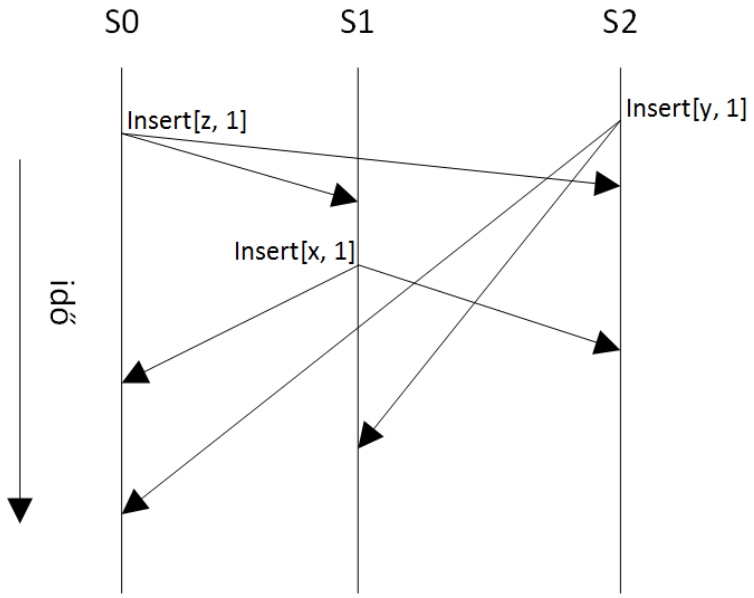
O' tehát kezdetben megegyezik a kapott művelettel, majd transzformáljuk az összes már végrehajtott művelet szerint, ami tőle független, majd a kapott O'-t végrehajtjuk és a napló végére fűzzük.

Algoritmus 1 - a dOPT algoritmus váza

$O' = O$
for $1 \leq i \leq n$ **do**
  if $Log[i]$ || O **then** $O' = T(O', Log[i])$
  end if
end for
Hajtsd végre O'-t
$Log = Log + O'$

A dOPT megoldatlan feladata

A következőkben egy olyan példát vizsgálunk meg, ahol a dOPT algoritmus nem tudja garantálni a konvergenciát. Ezt a forgatókönyvet illusztrálja a 2.5 ábra.



2.5. ábra. *A dOPT megoldatlan feladata [21]*

Feltesszük, hogy a GROVE transzformációs függvénye két azonos pozícióra hivatkozó beillesztés közül a magasabb prioritásút (magasabb forrásazonosítóval rendelkezőt) veszi figyelembe. Üres dokumentumból kiindulva S_3-on O_3 beilleszti "z"-t. Amikor O_1 megérkezik, beilleszti "x"-et "z" elé. Végül, mikor O_2 megérkezik, mivel O_2 || O_3 és O_2 || O_1, először transzformáljuk O_3 szerint, így lesz $O_2' = Insert[y, 2]$, mivel kisebb a prioritása mint O_3-nak. Ez után transzformáljuk O_1 szerint így lesz $O_2'' = Insert[y, 3]$. O_2'' végrehajtása után a dokumentum tartalma "xzy". S_1-en a végrehajtás menete és az eredmény az előzővel megegyező.S_2-n először O_2 beilleszti "y"-t a dokumentumba, majd mikor O_3 megérkezik transzformálni kell O_2 szerint, mivel O_2 || O_3, azonban O_3-on magasabb prioritása miatt nem változtat. Végrehajtása után a dokumentum tartalma "zy". Végül amikor O_1 megérkezik transzformálni kell O_2 szerint, hiszen O_2 || O_1. A transzformáció után $O_1' = Insert[x, 2]$, O_3 szerinti transzformációra pedig nincs szükség, hiszen $O_3 \rightarrow O_1$. Végrehajtás után a dokumentum tartalma "zxy", tehát a három állapot nem konzisztens.

A dOPT algoritmus tehát nem képes minden esetben biztosítani a konzisztenciát Bár

látszólag a prioritásra alkotott szabályunk okozhatja a problémát, különböző prioritás szabályok segítségével nem korrigálható az algoritmus hibája [21]. A következőkben két kifinomultabb megközelítést láthatunk a probléma megoldására.

2.3.3. REDUCE

A REDUCE követi a GROVE-ot abban a tekintetben, hogy egy teljesen elosztott környezettel dolgozik, ahol minden forrásnak saját másolata van a szerkesztendő dokumentumról. A REDUCE egy HB listát (History Buffer) használ az elvégzett műveletek tárolására, ami megegyezik a GROVE naplójával.

Szándék és konzisztencia

A REDUCE-féle megközelítésnél a divergencián és a kauzalitás megsértésén kívül felismertek egy további problémát is: a szándék sérülését.

Szemléltetésként vegyünk két független műveletet az első példánkból: O_1-et és O_2-t. S_0-n O_2-t egy olyan dokumentumon hajtjuk végre, aminek az állapota az azt megelőző O_1 hatására megváltozott. Így előfordulhat, hogy O_2 a megváltozott dokumentumban helytelen pozícióra hivatkozik, ami eltérő szerkesztést eredményez, mint ami O_2 *szándéka* volt, amit akkor érhetnénk el, ha ugyanazon a dokumentumállapoton hajtanánk végre, mint amiben maga is keletkezett.

Egy konkrétabb példán mindez a következőképp szemléltethető. Feltesszük, hogy kezdeti dokumentumunk az "ABCDE" karaktersorozatból áll. Legyen $O_1 = Insert[12, 2]$, ami a "12" karaktersorozatot illeszti be a kettes pozícióhoz, azaz "A" és "BCDE" közé. Legyen továbbá $O_2 = Delete[2, 3]$, azaz két karakter törlése a hármas pozíciótól kezdve, tehát a "CD" sorozat törlése. A végeredmény mindkét oldalon az "A12BE" karaktersorozat kell legyen. Azonban ha S_0-n a elsőként O_1 hajtódik végre és ezt követi O_2 végrehajtása, a végeredmény "A1CDE", ami nyilvánvalóan sérti mind O_1, mind pedig O_2 szándékát, hiszen a végeredményből hiányzik a beillesztendő "2", valamint jelen van a törlendő "CD". Egy sorrendezéssel kikényszeríthetnénk, hogy O_1-et és O_2-t minden oldalon azonos sorrendben hajtsunk végre, azonban az eredmény továbbra is inkonzisztens maradna a műveleteinkkel. Ez jól szemlélteti a fő különbséget a divergencia és a szándék megsértése közt. Az előbbit mindig meg tudjuk oldani valamilyen sorosítási módszerrel, utóbbit viszont nem[21].

Ennek megfelelően, a konzisztencia megőrzése érdekében a korábban említett *konvergencia* és *precedencia* mellett egy harmadik tulajdonságot kell biztosítanunk:

Szándék megőrzés Bármely O műveletre, O kimenetele megegyezik O *szándékával*, valamint O kimenetele nem változtatja meg a tőle független műveletek kimenetelét.

A precedencia megőrzése érdekében a REDUCE a GROVE-hoz hasonlóan egy állapotvektort használ. [19] A megkülönböztetett szándék és konvergencia betartása érdekében két módszert alkalmaz. Egy *undo/do/redo* sémát a konvergencia érdekében, valamint egy művelet transzformációs algoritmust a műveletek szándékának megőrzéséhez[21].

A konvergencia eléréséhez egy abszolút rendezési relációt is definiál a műveletek közt, melyet " $\Rightarrow$ " jelöl. [19] A műveletek tetszőleges sorrendben végrehajthatók, amíg a kauzalitás nem sérül. Ha egy új O művelet érkezik, visszavonjuk *(undo)* az összes HB-ben lévő műveletet melyet O *abszolút megelőz*, vérehajtjuk *(do)* O-t, majd újra végrehajtjuk *(redo)* a visszavont műveleteket.

Bemeneti és kimeneti feltételek

Mivel a REDUCE transzformációs függvényei nem felelősek a konvergenciáért, nem kell hogy rendelkezzenek a GROVE-ban definiált tulajdonságokkal. Amikor O_1-et transzformáljuk O_2 szerint, megköveteljük, hogy transzformáció utáni O_1' hatása egy olyan dokumentumon, ami tartalmazza O_2 eredményét, megegyezzen O_1 hatásával egy olyan dokumentumon, ami nem tartalmazza O_2 eredményét. Ezt a fajta transzformációt *magábafoglaló transzformációnak* hívják (IT - Inclusion Transformation), mivel a transzformáció eredményeképp O_1' magában foglalja O_2 hatását. Fontos észrevétel, hogy ez a transzformáció akkor helyes, ha O_1-et és O_2-t ugyan azon a dokumentum állapoton definiálták, így paramétereik összehasonlíthatóak és kiszámítható a helyes pozíció. E felismerés hiánya a korábban említett kirakós gyökere.

A szándék megőrzése érdekében a REDUCE bevezet egy új fajta transzformációt, a *kizáró transzformációt* (ET - Exclusion Transformation), ami O_1-et úgy transzformálja O_2 szerint, hogy O_2 eredményét kizárja. Ha visszatekintünk ismét első példánkra, láthatjuk, hogy bár O_1 és O_4 független műveletek, különböző dokumentumállapoton lettek definiálva. Így amikor O_4 megérkezik S_0-hoz, helytelen volna egyszerűen O_1 szerint transzformálnunk, helyette, az őt kauzálisan megelőző O_2 szerint kell végrehajtanunk rajta O_2 hatását kizáró transzformációt, hogy egy olyan O_4'-t kapjunk, ami így már ugyan arra a dokumentum állapotra vonatkozik, mint O_1, tehát végrehajthatjuk rajta a befoglaló transzformációt O_1 szerint.

Ahhoz, hogy meg tudjuk fogni a kapcsolatot a műveletek közt, be kell vezetnünk egy új fogalmat.

Kontextus Egy dokumentum állapot kontextusa azoknak a műveleteknek a sora, melyeket végrehajtottunk a kiinduló állapottól kezdve, hogy a jelenlegi dokumentum állapothoz jussunk. Ha adott O művelet, akkor O *definíciós kontextusa (DC(O))* annak a dokumentum állapotnak a kontextusa, amiben O létrejött. Hasonlóképp O *végrehajtási kontextusa (EC(O))* annak a dokumentum állapotnak a kontextusa, amin O-t végre kell hajtanunk.

Ennek birtokában pedig egy fontos kijelentést tehetünk. Egy művelet szándéka akkor őrizhető meg, ha definíciós kontextusa megegyezik a végrehajtási kontextusával, azaz $DC(O) = EC(O)$. Ennek kikényszerítéséhez a REDUCE a korábban említett két primitív transzformációt használja: $IT(O_1, O_2)$, valamint $ET(O_1, O_2)$. A szakasz címében szereplő elő és utófeltételek bevezetéséhez további két fogalomra van még szükségünk.

kontextus-ekvivalens reláció O_1 és O_2 műveletek kontextus-ekvivalensek $(O_1 \sqcup O_2) \Leftrightarrow DC(O_1) = DC(O_2)$.

kontextus-megelőző reláció O_1 megelőzi O_2-t, azaz $O_1 \rightarrow O_2 \Leftrightarrow DC(O_2) = DC(O_1) + [O_1]$.

Ezeket felhasználva a két transzformációs függvényt az alábbiak szerint specifikálhatjuk.

IT$(O_1, O_2) : O_1'$

Bemeneti feltétel: $O_1 \sqcup O_2$

Kimeneti feltétel: $O_2 \rightarrow O_1'$ és O_1' hatása $DC(O_1')$-ben megegyezik O_1 hatásával $DC(O_1)$-ben

ET$(O_1, O_2) : O_1'$

Bemeneti feltétel: $O_1 \rightarrow O_2$

Kimeneti feltétel: $O_2 \sqcup O_1'$ és O_1' hatása $DC(O_1')$-ben megegyezik O_1 hatásával $DC(O_1)$-ben

A fenti feltételeket kielégítő karakterláncokkal operáló IT és ET függvényekre ad példát a [20] és [22].

A GOT algoritmus és a feladat megoldása

Egy adott O művelet és $EC(O)$ végrehajtási kontextus esetén a GOT (General Operational Transformation) algoritmus az előbb definiált IT/ET függvények segítségével alakítja át O-t úgy EO-vá (O végrehajtási formája) , hogy teljesüljön a $DC(EO) = EC(O)$ egyenlőség. Az algoritmus az alábbi három esetet különbözteti meg. Példánkban feltételezzük, hogy $EC(O) = HB = [EO_1, EO_2, EO_3]$.

1) Az $EC(O)$-ban szereplő összes művelet kauzálisan megelőzi O-t. Ekkor $DC(O) = EC(O)$, tehát $EO = O$.

2) $EC(O)$-ban az O-t kauzálisan megelőző műveletek az O-tól független műveletek előtt vannak felsorolva, tehát például $EO_1 \rightarrow O$, $EO_2 \parallel O$ és $EO_3 \parallel O$. Ekkor transzformálhatjuk O-t a független műveletek szerint sorban, így $DC(EO) = EC(O)$.

3) $EC(O)$-ban legalább egy O-t kauzálisan megelőző művelet a tőle függetlenek után helyezkedik el (ez az az eset, ahol az eredeti dOPT algoritmus elbukott). Mivel ekkor a példánk szerint $EO_1 \rightarrow O$, $EO_2 \parallel O$ és $EO_3 \rightarrow O$ áll fenn, $DC(O) = [EO_1, EO_3']$, ahol EO_3' EO_3 akkori formája, mikor O létrejött. Ha most O-t bármelyik művelet szerint transzformálnánk, megsértenénk az előbb bevezett IT, illetve ET függvények be-és kimeneti feltételeit.

Ahhoz, hogy az algoritmust formálisabban is megadhassuk, szükség van néhány jelölés bevezetésére. A műveletek egy L listájában $L[i, j]$ azt a listát jelöli, amit L-ből kapunk ha vesszük az EO_i és EO_j közti elemeket, valamint L^{-1} jelöli L megfordítását. Továbbá $LIT(O, L)/LET(O, L)$ jelöli az IT/ET transzformációs függvény alkalmazását L listában

Algoritmus 2 - GOT(O, L): EO

O: végrehajtandó művelet
L: a műveletek $[EO_1, ..., EO_m]$ listája $EC(O)$-ban
EO: O végrehajtási formája

$V :=$ jobbról az első műveletet $L[1, m]$-ben, amire $EO_k \mid\mid O$.
if $\nexists V$ **then**
  return O
end if

$W := O$-t kauzálisan megelőző műveletet $L[k+1, m]$-ben.
if $\nexists W$ **then**
  return $EO := LIT(O, L[k, m])$
end if

$L_1 := [EO_{c_1}, ..., EO_{c_r}]$ azon műveletek listája, melyek kauzálisan megelőzik O-t $L[k, m]$-ben.
$L'_1 := [EO'_{c_1}, ..., EO'_{c_r}]$ az alábbiak szerint
$EO'_{c_1} := LET(EO_{c_1}, L[k, c_1 - 1]^{-1})$

for $2 \leq i \leq r$ **do**
  $O_t := LET(EO_{c_i}, L[k, c_i - 1]^{-1})$
  $EO'_{c_i} := LIT(O_t, [EO_{c_1}, ..., EO_{c_i}])$
end for

$O' := LET(O, L'^{-1}_1)$
return $EO := LIT(O', L[k, m])$

szereplő műveletek szerint sorban O-ra. Ezek után az algoritmus a következőképp írható le.

Megmutatható, hogy a korábban definiált be-és kimeneti feltételek teljesülését, a fentebb leírt algoritmus garantálja, erre azonban a dolgozatban nem térünk ki.

Az itt szerzett ismeretek alapján, vegyük most újra szemügyre korábbi megoldatlan feladatunkat. Feltesszük, hogy $O_3 \Rightarrow O_1 \Rightarrow O_2$ teljesül, továbbá azt, hogy az $IT(O_a, O_b)$ transzformációs függvény az alábbi csúsztatási szabályt alkalmazza: ha mind O_a, mind pedig O_b ugyan arra a pozícióra hivatkozó beszúrás műveletek, akkor O_a csúszik. Ez konzisztens a GROVE-nál bemutatott prioritás szabállyal. [21]

Az alkalmazott transzformációk és a végső dokumentumállapot ("xzy") megegyeznek S_1-en és S_3-on, ahogy a GROVE esetében is láttuk, S_2 esetén azonban eltér a működés. Elsőként O_2 illeszti be "y"-t. Ez után érkezik O_3. Ekkor vissza kell vonnunk O_2-t, mivel $O_3 \Rightarrow O_2$ teljesül. Ez után végrehajtjuk O_3-at eredeti formájában, majd O_2-t transzformáljuk O_3 szerint ($IT(O_2, O_3)$, a GOT algoritmus 2-es esete szerint, mivel $O_3 \mid\mid O_2$), így $O'_2 = Insert[y, 2]$ a shiftelési szabályunk értelmében. O'_2 és O_3 végrehajtása után a dokumentum állapota "zy". Mikor végül O_1 megérkezik, vissza kell vonnunk O'_2-t, mivel $O_1 \Rightarrow O'_2$. Ez után O_1-et eredeti formájában végrehajtjuk ($O_3 \rightarrow O_1$) és O'_2-t transzformál-

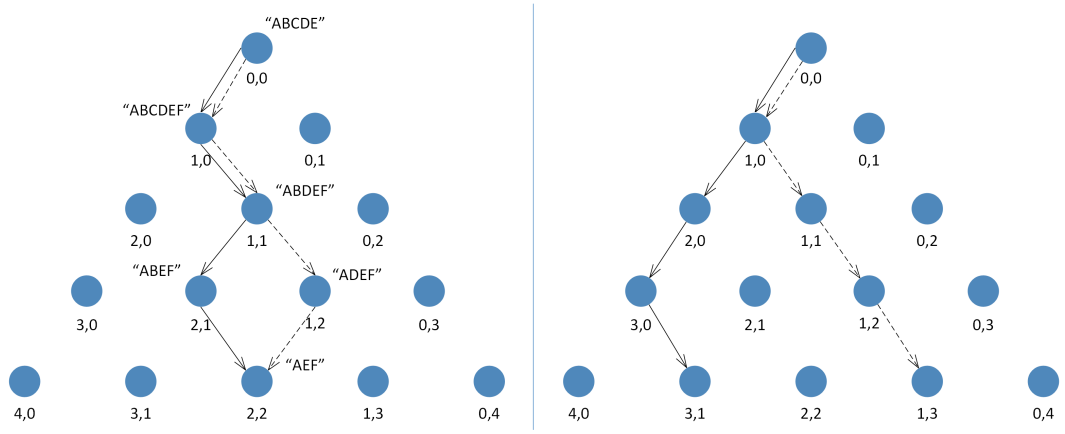
juk O_1 szerint (a 2-es eset szerint, mivel $O_2 \parallel O_1$), így $O_2'' = Insert[y, 3]$. O_2'' végrehajtása után a dokumentum állapota S_2-nél "xzy", tehát az eredmény egybeesik a többi forrásával, valamint a műveletek szándékával. Bár nem használtunk kizáró transzformációt a megoldás során, konplexebb példák esetén, mint amilyen a 2.4 ábrán is látható, ez a fajta transzformáció is szükséges.

A GOT algoritmus tehát korrigálja a dOPT hibáját a szándék, a kontextus és a két fajta transzformáció bevezetésével, sikeresen megtartva az eredeti elosztott környezetet. A következő alfejezetben egy másfajta megközelítést láthatunk.

2.3.4. Jupiter

A Jupiter[15] egy kollaborációs rendszer, amit a Xerox PARC[16] fejlesztett ki. A rendszer egy dOPT-ból származó hasonlóan optimista algoritmust használ a konzisztencia fenntartására, felépítése azonban eltér az eddig látottaktól. A Jupiter esetében, bár mindenki saját másolattal rendelkezik a dokumentumról, a felhasználókat egy központi szerver szolgálja ki, illetve ő tartja karban az egyes objektumok állapotát. Amikor a felhasználó változtat a dokumentumon, művelet azonnal végrehajtódik a saját másolatán (a gyors válaszidő érdekében), majd továbbításra kerül a szerver felé. A szerver szükség esetén transzformálja a kapott műveletet, végrehajtja a központi példányon, majd továbbküldi a többi kliensnek. A kliensek a kapott műveletet ismét transzformálják ha szükséges, majd végrehajtják a lokális másolatukon [15].

A Jupiter fő érdekessége az eddig látott GROVE-hoz és REDUCE-hoz képest, hogy bár az algoritmusa a dOPT-ból származik, nem jelent gondot a konzisztencia megtartása. Ennek két oka van. A korábbiaktól eltérően egy központosított topológiával állunk szemben, azaz a kliensek csak a szerverrel kommunikálnak, ami a korábbiakhoz képest lényeges egyszerűsítés[6]. Tobábbá, a korábbi műveletek tárolására nem egy egyszerű listát használunk, hanem egy kétdimenziós állapottér gráfot. Ezt az 2.6 ábra szemlélteti.



2.6. ábra. *A Jupiter állapottér gráfja*

Az egyes csomópontok állapotokat jelölnek, amik címkékkel vannak ellátva. A címkén két szám látható. Az első a kliens által feldolgozott üzeneteket jelzi, a második a szerver által feldolgozott üzeneteket. Az üzenetekben, a művelet mellet az aktuális címkét is el-

küldjük. A folytonos nyilak a kliens, a szaggatott nyilak a szerver útját jelölik. Amíg a két útvonal megegyezik, nincs szükség transzformációra. Ez lényegében azt jeletni, hogy épp ugyan azokat az üzeneteket dolgozzák fel. Ha az útvonalak eltérnek, konfliktusról beszélünk. Ilyenkor különböző üzeneteket dolgoznak fel, azaz tulajdonképp a szerver épp egy másik kliens üzeneteit dolgozza fel. Ilyenkor transzformáció szükséges. Az ábrán ez az (1,1) állapot után következik be. Ekkor kliens és a szerver a másik féltől kapott üzenet alapján előállítja azt a műveletet, ami a (2,2)-be viszi őket. Azaz kiszámolják $T(O_a, O_b) = O'_a$-t, illetve $T(O_b, O_a) = O'_b$-t, majd ezt alkalmazva (2,2)-be jutnak, ami ismét konzisztens állapot.

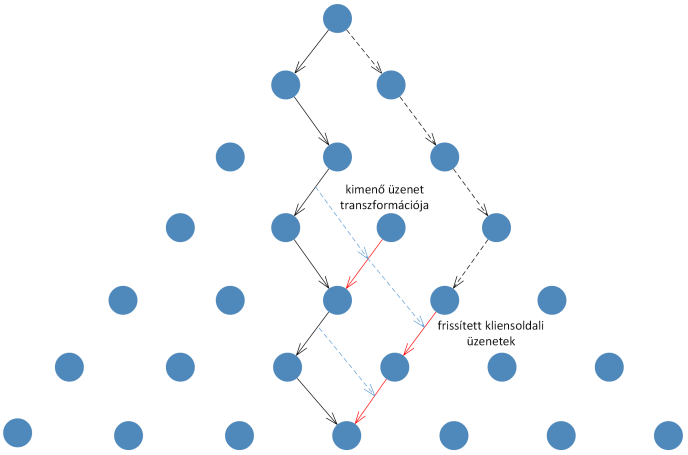
Ennél valamivel bonyolultabb az eset, amikor nem csak egy az utak nem csak egy lépésben térnek el. Ezt az esetet a 2.6 ábra jobb oldalán láthatjuk. Ilyenkor meg kell vizsgálnunk, hogy a két út hol tért el, amit az üzenetben kapott címke mutat meg. Az ennél korábbi műveleteket elhagyhatjuk. A megmaradt üzenetek szerint transzformálva és végrehajtva az érkező műveletet egy lépéssel közelebb kerülünk egy újabb közös állapothoz.

A módszer leganagyobb hátránya ebben formában, hogy a szerver minden klienshez fenntart egy állapottérgráfot, valamint a hálózati forgalom a sok üzenet miatt indokolatlanul nagy lehet. Ezeken később a Google javított a Wave tervezésekor[23].

Google Wave

A Google Wave egy Google által fejlesztett keretrendszer online kollaborációhoz, aminek fejlesztését végül leállították, majd Apache Wave-ként élt tovább[3]. Kommunikáció és szinkronizáció szempontjából, Wave alapjául a Jupiter rendszer szolgált, bizonyos pontokon azonban eltértek az eredeti elképzeléstől. A legfontosabb változtatás talán, hogy a kliensek csak nyugta után küldhetnek új üzenetet. Nyugtát akkor kapnak, ha az előző üzenetüket a szerver feldolgozta és elküldte a többi kliensnek. Az eközben keletkező üzeneteket a kliens tárolja, majd egy üzenetben küldi el. Ez két rendkívül fontos dolgot tesz lehetővé. Egyrészt, a szervernek többé nem kell fenntartania minden klienshez egy állapottér gráfot, elég a munkamenethez egyet, hiszen saját állapota az eddig összesen feldolgozott üzenetek számára egyszerűsödik, a kliens állapotát pedig az üzenetből megtudhatja. Ennek köszönhetően a szerveloldali logika jelentősen egyszerűsödik. Továbbá a kliens a kimenő várakozó listájában lévő üzeneteket minden bejövő szervertől kapott üzenet segítségével frissíthet. Ez azt jelenti, hogy optimális esetben a nyugta megérkezésekor olyan műveleteket küld a szervernek, amik a szerver útjába esnek az állapottér gráfon, azaz nincs szükség transzformációra. Erre a 2.7 mutat példát.

Ez további terhet vesz le a szerverről, ami kimondottan előnyös tekintve, hogy a kommunikáció szempontjából a szerver egyszeres hibapont. A nyugták egy további szerencsés mellékhatása, hogy a hálózati forgalom jelentősen csökken, mivel a műveletek tömösítve kerülnek továbbításra.



2.7. ábra. *Frissítés a szerver útvonala alapján [23]*

2.4. Összegzés

A fejezet során végigtekintettük a szinkronizáció alapötleteit, majd közelebbről is megvizsgáltunk egy három-oldali összehasonlításon alapuló módszert, valamint egy szerkesztésalapú módszert. Előbiről kiderült, hogy olyan területeken alkalmazható hatékonyan, ahol a megosztott média könnyen reprezentálható szöveges formában, ezért nem alkalmas egy általános célú keretrendszer támogatásához így áttértünk egy szerkesztésalapú módszerre a művelettranszformációra. Ahogy láthattuk, a módszer bár jóval általánosabb, jóval nehezebb a konzisztens dokumentumállapot biztosítása. A feladatra három megoldást tekintettünk át. A REDUCE az eredeti, elosztott környezetbeli problémát oldotta meg a szándék és a kontextus bevezetésével, a Jupiter pedig egy központosított kommunikációra szűkített, valamivel egyszerűbb feladatot oldott meg kétdimenziós állapotreprezentáció segítségével. A megoldás meglehetősen hatékonynak bizonyult, később a Google Wave alapjául szolgált.

A következő fejezetben az itt szerzett ismeretek birtokában felvázoljuk, a bevezetőben megfogalmazott feladat egy lehetséges megoldását.

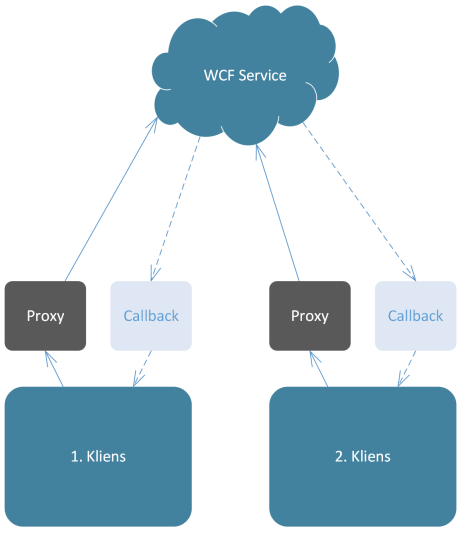
3. fejezet

A Mandrake keretrendszer

A célom egy olyan általános célú keretrendszer kialakítása fejlesztők számára, ami lehetővé teszi bármilyen kollaborációs funkció integrálását meglévő alkalmazásokba. Ha ismét visszatérünk a bevezetőben említett háromdimenziós szerkesztő példájához, egy olyan eszközt szeretnénk adni a fejlesztő kezébe, ami számára átlátszóvá teszi a kommunikációt és szinkronizációt, az ezért felelős komponensekkel csupán egy minimális interfészen keresztül kell kommunikálnia.

3.1. Architektúra

A megvalósításkor választásom a Google Wave által használt felépítésre és szinkronizációra esett. Ennek egyik oka, hogy egyszerű transzformációt tesz lehetővé, ahogy azt az előző fejezetben láthattuk. Másik oka pedig, hogy a központosított felépítés lehetővé teszi számunkra, hogy a szinkronizációra egy telepíthető és központilag menedzselhető szolgáltatásként tekintsünk. Ez az üzemeltetés szempontjából kimondottan kellemes lehetőség. A felépítést a 3.1 ábra szemlélteti.



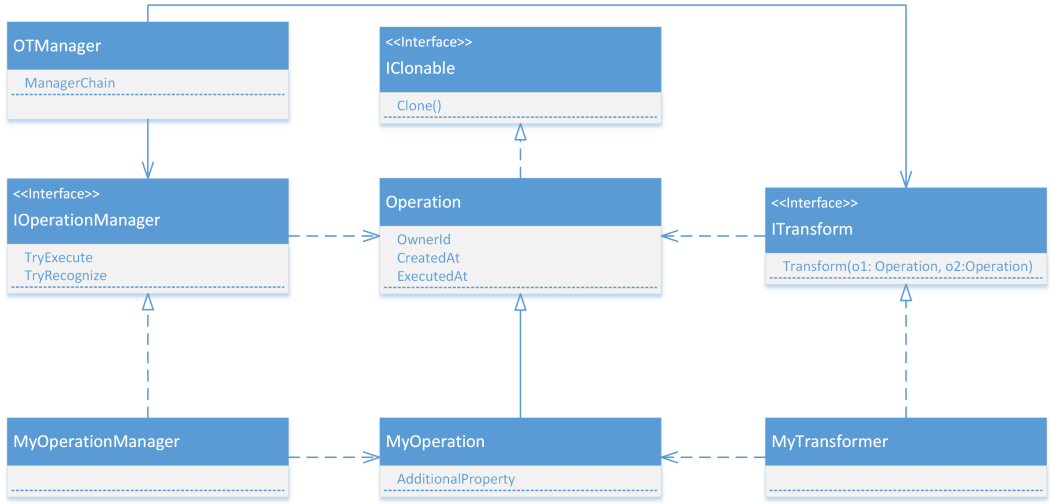
3.1. ábra. *Felépítés*

Az elképzelés tehát az, hogy a Jupiternél látott szervert egy webszolgáltatás valósítja

meg, amihez kliensek kapcsolódhatnak. A szerver vezényli a kommunikációt és szinkronizációt a kliensek között, a működéshez pedig három lényeges dologra van szükség. Alkalmazásspecifikus műveletekre, komponensekre amik képesek felismerni és továbbítani ezeket a műveleteket, valamint komponensekre, amik képesek végrehajtani ezeket a műveleteket. A szolgáltatás ezekkel a komponensekkel egy minimális interfészen keresztül kommunikál. Ezeket a komponenseket a fejlesztő biztosítja. Ha ezek rendelkezésre állnak, a szolgáltatást telepíthetjük és konfigurálhatjuk.

3.2. A keretrendszer felhasználási lehetőségei

Vegyük az előbb leírtakat közelebbről is szemügyre. Tekintsük a 3.2 ábrán vázolt osztálydiagrammot, mely vázolja a fejlesztő számára fontos komponenseket.

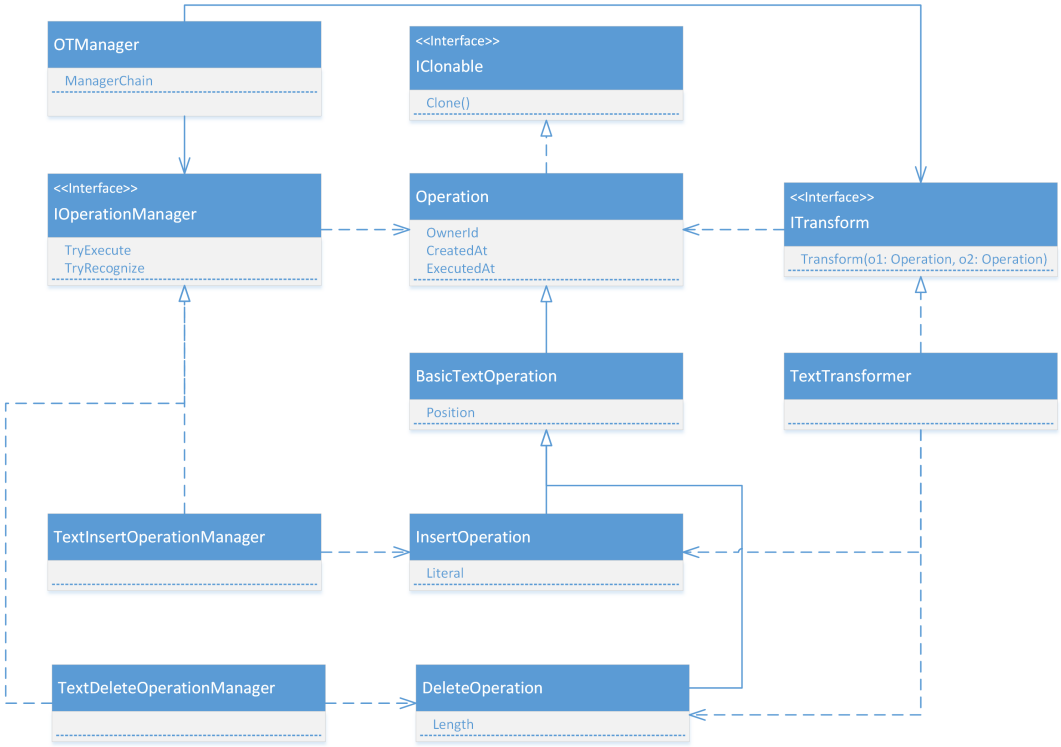


3.2. ábra. *Főbb komponensek*

Ahhoz hogy használjuk a keretrendszert, az első dolgunk, az alkalmazásspecifikus műveletek megtervezése. Ezek az Operation osztályból származnak. Ez után szükségünk van olyan komponensekre, amik felismerik ezeket a felhasználói események alapján, valamint képesek ezeket végrehajtani ehhez az IOperationManager interfészt kell megvalósítanunk. A szinkronizációhoz egy további komponens típusra van szükségünk, ami tudja, hogy a műveleteinket hogyan kell transzformálni. Ehhez az ITransform interfészt kell megvalósítsuk. Így rendelkezésünkre állnak a műveletek és a használatukhoz szükséges komponensek. Miután ezzel megvagyunk, telepítjük a szolgáltatást és függőség-injektálás segítségével átadjuk számára a transzformációhoz szükséges komponenseket. Kliensünkben ezután szükségünk van egy kliensoldali menedszer objektumra(egy OTManager példányra), aminek átadjuk a szolgáltatás referenciát valamint a transzformációért és végrehajtásért, illetve felismerésért felelős komponenseket, valamint gondoskodunk róla, hogy a megfelelő felhasználói események eljussanak hozzá.

Nézzük meg ezt közelebbről is egy szövegszerkesztő példáján keresztül. A szövegszerkesztő alkalmazásom egyszerű szövegmanipulációra képes (beszúrás, törlés, másolás, beillesztés), szeretném azonban ezt az alkalmazást kollaborációs funkcióval felruházni, hogy a

szöveg több felhasználó által párhuzamosan szerkeszthető legyen. Ez esetben a 3.2 osztálydiagramm 3.3 ábra szerint alakul.



3.3. ábra. *A keretrendszer felhasználása szövegszerkesztéshez*

A feladat szerint tehát két művelet típusra van szükségünk, ezek az eddigi példákban megjelenő beszúrás(Insert[x, y]), valamint törlés(Delete[u, v]) műveletek. Ennek megfelelően két konkrét művelet leírónk lesz, az egyik a beszúrást leíró InsertOperation, a másik a Törlést leíró DeleteOperation.

Most, hogy le tudjuk írni a szükséges műveleteket, szükségünk van olyan komponensekre, amelyek képesek felismerni a megfelelő műveletet. Ennek megvalósítása a TextInsertOperationManager és a TextDeleteOperationManager osztályok, melyek felüldefiniálják a TryExecute és TryRecognize műveleteket. Ehhez a felhasználó által kiváltott eseményeket és esetlegesen a konkrét felhasználói felületen megtalálható elemeket használják fel. Mivel itt függünk a konkrét felületi elemek implementációjától, ennek kifejtésére itt nem kerül sor.

Amikor az OTManager megkapja az általunk érdekesnek vélt felhasználói eseményt, végigiterál az IOperationManager listáján és meghívja a menedzser osztályok TryRecognize metódusát. Amennyiben valamelyik felismeri az eseményt mint valid műveletet, úgy visszatér a konkrét műveletet leíró objektummal, azaz ha felismeri, hogy töröltünk egy részt a szövegből, a TextDeleteOperationManager egy DeleteOperation példánnyal tér vissza. Ebben az esetben, az iteráció megáll, a kapott művelet pedig a kliens kimenő listájába kerül. Felmerülhet a kérdés, hogy mi történik, ha több elem is felismeri a kapott eseményt. Ekkor az első művelete jut érvényre, tehát az OTManager ManagerChain listája prioritásos sorrendben kell tartalmazza a menedzser példányokat. Mindazonáltal ha egy felhasználói eseményből több fajta műveletre is következtethetünk, akkor valószínűleg tervezési hibával

állunk szemben. A végrehajtás ezzel analóg módon történik. Az OTManager a kapott művelettel végigiterál az elemeken és meghívja a metódust, amíg valamelyik nem jelzi, hogy képes végrehajtani. Most tehát vannak műveletleíróink és komponenseink, amik képesek felismerni és végrehajtani azokat. A helyes működéshez egy olyan elemre van szükségünk, ami képes a műveleteket transzformálni (ezt az elemet mind a két oldalon használjuk majd). Példánk szerint ehhez négy műveletre van szükség, hiszen két féle műveletünk van (beszúrás és törlés), valamint mindegyik transzformációja értelmezhető mindegyik szerint. Ezek C# nyelvű implementációját láthatjuk az alábbiakban (mindig a második paramétert transzformáljuk az első szerint).

Ha két beszúrás műveletünk van O_2-t jobbra toljuk el, amennyiben O_1 egy kisebb pozícióra hivatkozik, valamint ha ugyanarra a pozícióra hivatkoznak, de O_1 előbb keletkezett.

```
private void Transform(InsertOperation o1, InsertOperation o2)
{
    if (o1.Position < o2.Position
       || (o1.Position == o2.Position && o1.CreatedAt < o2.CreatedAt))
    {
        o2.Position += o1.Length;
    }
}
```

Ha egy törlés műveletet transzformálunk egy beszúrás művelet szerint, a törlés ablakát jobbra toljuk, ha az a beszúrás utáni pozícióra vonatkozik, valamint ha az ablakon belülre szúrunk be, a beszúrást elvesztjük.

```
private void Transform(InsertOperation o1, DeleteOperation o2)
{
    if (o1.Position < o2.StartPosition
       || (o1.Position == o2.Position && o1.CreatedAt < o2.CreatedAt))
    {
        o2.StartPosition += o1.Literal.Length;
        o2.EndPosition += o1.Literal.Length;
    }

    else if (o1.Position >= o2.StartPosition)
        o2.EndPosition += o1.Length;
}
```

Az előző eset fordítottja esetében a beszúrást balra toljuk, ha az a törlés ablaka utáni pozícióra hivatkozik, valamint a beszúrni kívánt karakterláncot töröljük, ha az ablakon belüli pozícióra hivatkozik, ezzel biztosítva a transzformációs tulajdonságot.

```
private void Transform(DeleteOperation o1, InsertOperation o2)
{
    if (o2.Position >= o1.EndPosition) o2.Position -= o1.Length;
```

```
    else if (o2.Position >= o1.StartPosition) o2.Literal = "";
}
```

A legbonyolultabb esettel akkor állunk szemben, ha a törés műveletet egy másik törlés szerint transzformáljuk. Ekkor öt esetet különböztethetünk meg. Ha az ablakok közt nincs átfedés, a jobboldalit eltoljuk balra a másik törlés hosszával (1. eset). Ha a törlési ablakok egymásba ágyazódnak, a belső törlést elvetjük, vagy csökkentjük a külső ablakot (2-3. esetek). Ha átlapolódnak az ablak megfelelő végét toljuk a megfelelő irányba (4-5. esetek).

```
private void Transform(DeleteOperation oa, DeleteOperation ob)
{
    if (oa.EndPosition < ob.StartPosition)
    {
        ob.StartPosition -= oa.Length;
        ob.EndPosition -= oa.Length;
    }
    else if (ob.StartPosition >= oa.StartPosition
            && oa.EndPosition >= ob.StartPosition)
        ob.EndPosition = ob.StartPosition;

    else if (oa.StartPosition >= ob.StartPosition
             && ob.EndPosition >= oa.EndPosition)
        ob.EndPosition -= oa.Length;

    else if (ob.StartPosition < oa.EndPosition)
        ob.StartPosition += oa.EndPosition - ob.StartPosition;

    else if (oa.StartPosition < ob.EndPosition)
        ob.EndPosition -= ob.EndPosition - oa.StartPosition;
}
```

A metódusokat megvalósító komponenst ezután átadhatjuk a keretrendszernek. Bejövő művelet esetén az OTManager meghívja a komponensünk Transform metódusát, ami a fenti műveletek segítségével előállítja a transzformált műveletet. Miután minden szükséges művelet szerint transzformálta, az OTManager átadja a végrehajtásra alkalmas műveletet a megfelelő menedzser példánynak.

Még egy dolgot kell ellenőriznünk, mielőtt a komponenst átadjuk a keretrendszernek, ez pedig a transzformációs tulajdonság, azaz biztosítanunk kell, hogy ha a szerver és a kliens ugyanazt a műveletpárt transzformálja, akkor egy ugyanabba az állapotba jutnak (feltéve, hogy csak egy lépésben térnek el). Az implementációnk szerint ez azt jelenti, hogy Transform(o1, o2) és Transform(o2, o1) ugyanarra az eredményre kell vezessen. Az első esetben ez könnyen látható, hiszen a fordított esetben o2-t egyszerűen érintetlenül hagyjuk, azaz az egyik oldalon eltoljuk a beillesztést a másik oldalon egyszerűen elé illesztünk be. A második és harmadik esetben a transzformációs tulajdonságot az biztosítja, hogy a törlés

ablakába eső beszúrást mindenképp eldobjuk, valamint a törlés ablakának mozgatása az előző esettel analóg módon történik. Végül pedig lássuk a két törlés művelet esetét. Az első esetben a két törlési ablak elkülönül, ilyenkor a jobbra esőt csúsztatjuk balra (1. eset). Ha egymásbaágyazódnak (2-3. esetek) a belső törlést eldobjuk, vagy a külső ablakot igazítjuk, akármelyik a transzformálandó művelet, egyenlő számú karaktert törlünk a kisebbik kezdőpozíciótól, tehát a transzformációs tulajdonság továbbra sem sérül. Átalapolódás esetén ismét két szimmetrikus eset (4-5. esetek) garantálják, hogy a transzformálandó törlés ablakát a megfelelő irányból csökkentsük.

3.3. Nehézségek

A szinkronizáció módjának választott művelettranszformáció egy kellően általános eszközkészletet szolgáltat a probléma megoldására, a Jupiter és Google Wave rendszerektől örökölt felépítés pedig egy kellemes megoldást kínál a műveletek kezelésére, és a konzisztens dokumentumállapot fenttartására. Most azonban lássuk, hogy milyen főbb nehézségekkel és hátrányokkal járnak ezek a választások, illetve hogy hogyan segíthetünk ezeken.

3.3.1. Központosított felépítés

A legszembeötlőbb hátrány talán, hogy a Jupiter rendszertől örökölt szerver a kommunikáció és szinkronizáció szempontjából egyszeres hibapont, ezt vállaltuk amikor feladtuk az elosztottságot az egyszerűbb transzformációért cserébe.

Bár ez jelentős veszteségnek tűnhet, elosztottságot könnyen vihetünk a rendszerbe egy magasabb absztrakciós szinten. Feloszthatjuk például a munkameneteket nagyobb logikai egységek között. Munkamenetek két halmazát kiszolgálhat például két különböző kiszolgáló, ami a felhasználó számára átlátszó. Megvalósítható valamilyen failover eljárás, ami hiba esetén átadja az általa kiszolgált munkameneteket egy másik arra alkalmas kiszolgálónak, ami a felhasználónak vélhetően csak kisebb várakozást és kellemetlenséget okoz.

3.3.2. Egymást kizáró műveletek

Egy másik nehézséget jelentenek az egymást kizáró műveletek. Eddig végig olyan műveletekről volt szó, amik megjelenhetnek egymással egyidőben, ezért volt szükség transzformációkra. Lehetnek azonban olyan műveleteink, amik nem értelmezhetőek együtt, vagy épp nem kívánatos az együttes végrehajtásuk. Kanyarodjunk vissza ismét a bevezetőben említett példánkhoz. Ha mozgatok egy objektumot a térben, nem szeretném ha egy másik felhasználó is képes lenne mozgatni, vagy forgatni. Ilyenkor azt szeretnénk, hogy végül az a műveletsor jusson érvényre, ami előbb elkezdődött. Nem valószínű azonban, hogy gondot jelent, ha a másik felhasználó megváltoztatja a mozgatott objektum színét. Mivel lefektettük, hogy gyors válaszidő érdekében a lokális műveleteket azonnal végrehajtjuk, azt az elképzelést, hogy a szervertől engedélyt kérünk nem alkalmazhatjuk. Ennek továbbá van egy jóval praktikusabb oka is. Ahhoz hogy egy felhasználói felületen végzett művelet hatását megvizsgáljuk mielőtt érvényre jut, majd eldöntsük, hogy hagyjuk-e érvényre jutni, azt kívánná, hogy felülírjuk a kollaboráció során használt felületi elemek viselkedését, azaz

ahhoz hogy egy általánoscélú keretrendszert használni tudjunk teljes mértékben át kell szabnunk az alkalmazásunkat. Így ezt az elképzelést biztosan elvethetjük.

Egy megoldási elképzelés, hogy bizonyos műveletekben függőségeket definiálunk az őket kizáró művelettípusokon, a transzformáció során pedig figyeljük az ilyen függőségeket és a később érkező művelet transzformáltjaként a művelet inverzét állítjuk elő. Előző példánkban ez azt jelenti, hogy ha mozgatás közben kapunk egy forgatás műveletet, akkor egyszerűen nem hajtjuk végre, a másik felhasználó pedig a mozgatás művelet mellett, végrehajtja saját forgatásának inverzét. Felmerülhet azonban, hogy mi a közben létrejött műveletek sorsa, illetve hogy mi a teendő, ha például a folyamatosság élménye érdekében a forgatás több részletben érkezett.

Egy másik megoldás lehet, ha bevezetünk egy speciális műveletfajtát a zárolás kifejezésére és az ilyen műveletekkel speciálisan járunk el a transzformáció során. Ez az előzőnél elegánsabb megoldásnak tűnik, mivel nem kell esetlegesen egy sokszorosan összefüggő függőségi gráffal dolgoznunk, azonban a megvalósítással kapcsolatban hasonló kérdés merül fel. Hogyan kezeljük, a zárolási kérelem előtt érkező műveletekkel. Ehhez segítséget nyújthat a [**?**], ennek menedzselése azonban további átgondolást igényel.

4. fejezet

Összefoglalás

Az előző fejezetben láthattunk tehát egy keretrendszert, ami lehetővé teszi kollaborációs funkciók alkalmazásokba való integrálását. A konzisztens állapot megtartása érdekében a művelet transzformációt választottam, mivel ez egy kellően általános eszköztárat biztosít különböző területekről származó feladatok megragadására. A szinkronizációt egy Google Wave-hez hasonló módszer biztosítja, ami lehetővé teszi számunkra, hogy a kommunikációt és szinkronizációt menedzselő szerver részre egy telepíthető és leírókon keresztül konfigurálható webszolgáltatásként tekintsünk, mely kimondottan előnyös lehet üzemeltetés szempontjából például egy cégen belüli belső használat során. Ahhoz, hogy ezt fejlesztőként fel tudjuk használni az alkalmazásunkban csupán saját műveletekre van szükségünk, valamint komponensekre, amelyek képesek ezeket felismerni és végrehajtani. Bár a megoldás hordoz magával nehézségeket, kommunikáció és szinkronizáció egyszerűsége és a megoldás általánossága kárpótol ezekért.

4.1. Továbbfejlesztési lehetőségek

Tekintve, hogy egy általános célú megoldást keresünk, nem meglepő, hogy mind fejlesztőként, mind felhasználóként könnyű elképzelni további lehetőségeket irányokat a fejlesztéssel kapcsolatban. Az egyik érdekes irány a fejlesztési folyamat bizonyos fokú formalizálása és ezáltal gördülékenyebbé tétele.

4.1.1. Transzformációs szabályok formális leírása

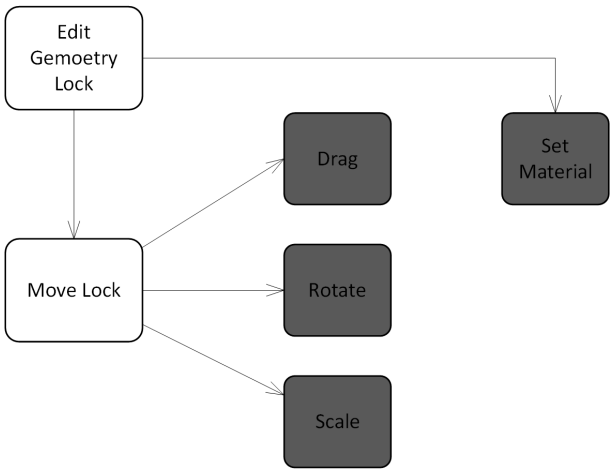
Bár a műveletek felismeréséért felelős komponensek implementálását nem kerülhetjük el, a műveletek leírásához és a transzformációhoz nem kell mindenképp egy fejlesztőt alkalmaznunk, vagy legalábbis nem kellene feltétlen kódot írnunk. A transzformációs szabályokat például leírhatjuk formálisabb módon, majd generálhatunk ebből kódot. Bár a fejlesztő bevonása nyilvánvalóan elkerülhetetlen ha a generált kódot használni is akarjuk, a szabályok egy formális leírása azonban segíti a könnyebb kommunikációt, illetve lehetővé teszi, hogy egy magasabb absztrakciós szinten dolgozzunk és ne kelljen a megoldás részleteivel foglalkoznunk. Ez azonban lényegében egy saját fordítót kíván, aminek megalkotása nehéz feladat. Egy jóval egyszerűbb eset, ha eszközt biztosítunk a műveletek tulajdonságainak

leírására.

4.1.2. Egymást kizáró műveletek formális megadása

Az korábban említett egymást kizáró műveletek problémájánál, érezhetjük, hogy bármelyik fentebb javasolt megoldást választjuk, vagy akár valamilyen további elképzelés szerinti rendszert dolgozunk ki, ahhoz hogy az ilyen műveletek esetén is könnyebbséget jelenetene, ha a szükséges kódot generálhatnánk, a szabályokat pedig egy formálisabb módon adhatnánk meg.

Ha például zárolási műveleteket választunk, könnyű elképzelni egy grafikus eszközt, ahol a függőségeket egy gráffal reprezentálhatjuk. Példaként kanyarodjunk vissza a háromdimenziós szerkesztéshez és tekintsük a 4.1 ábrát.



4.1. ábra. *Főbb komponensek*

A zárolási műveleteket tehát leírhatjuk például úgy, hogy irányított élek segítségével megadjuk, hogy milyen műveletek későbbi tejlesülését szeretnénk kizárni. Az ábrán tehát két zárunk van. Az egyik a mozgatást, forgatást és méretezést tiltja meg, de engedi például az anyagtulajdonságok finomhangolását, míg a másik esetben ha az objektum geometriáját szerkesztjük, minden más szerkesztési lehetőséget zárolunk az objektumon.

Ilyen és hasonló eszközök segítségével a fejlesztési folyamat jóval gördülékenyebbé tehető, mivel a nagyobb hangsúly kerül az algoritmusok megalkotására és kényszerek megadására, mint ezek konkrét implementációjára, ami esetenként magában foglalhatja egy munka többszöri elvégzését. Ezen felül több lehetőségünk nyílik bevonni a megrendelőt, illetve a terület szakértőit is fejlesztésbe. Ez azonban a jövő kutatási területe.

Irodalomjegyzék

[1] Vs anywhere for visual studio. `https://vsanywhere.com/web/`.

[2] Mockingbird. `https://gomockingbird.com/about/`, 2012.

[3] Apache. Apache wave. `http://incubator.apache.org/wave/`.

[4] BeWeeVee. Beweevee. `http://www.beweevee.com/`, 2014.

[5] chromatic. *Extreme programming pocket guide - team-based software development.* O'Reilly, 2003.

[6] Gordon V. Cormack. A calculus for concurrent update (abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 269–, New York, NY, USA, 1995. ACM.

[7] Eclipsepedia. Ecf/docshare plugin. `https://wiki.eclipse.org/ECF/DocShare_Plugin`, 2014.

[8] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989.

[9] Emacs. Rudel. `http://www.emacswiki.org/emacs/Rudel`, 2014.

[10] Neil Fraser. Differential synchronization. In *DocEng'09, Proceedings of the 2009 ACM Symposium on Document Engineering*, pages 13–20, 2 Penn Plaza, Suite 701, New York, New York 10121-0701, 2009.

[11] Google. Google docs. `http://www.google.hu/intx/com/work/apps/business/products/docs/`.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[13] Microsoft. Office 365. `http://office.microsoft.com/`.

[14] Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[15] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the*

8th Annual ACM Symposium on User Interface and Software Technology, UIST '95, pages 111–120, New York, NY, USA, 1995. ACM.

[16] PARC. Parc a xerox company. `https://www.parc.com/`, 2014.

[17] Michael Pilato. *Version Control With Subversion.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[18] Henry Van Styn. Git. *Linux J.*, 2011(208), August 2011.

[19] C. Sun, Y. Yang, Y. Zhang, and D. Chen. A consistency model and supporting schemes for real-time cooperative editing systems, 1996.

[20] Chengzheng Sun, David Chen, and Xiaohua Jia. Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems, 1998.

[21] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

[22] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

[23] David Wang, Alex Mah, and Soren Lassen. Google wave operational transformation. `http://www.waveprotocol.org/whitepapers/operational-transform`, 2010.

[24] Wikipedia. Coword — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/CoWord`, 2013.

[25] Wikipedia. Microelectronics and computer technology corporation — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Microelectronics_and_Computer_Technology_Corporation`, 2013.

[26] Wikipedia. Pair programming — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Pair_programming`, 2014.

[27] Laura Wingerd. *Practical perforce - channeling the flow of change in software development collaboration.* O'Reilly, 2005.