Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling met

Titel: Bestands- en datasynchronisatie
Richting: 2de masterjaar in de informatica - databases Jaar: 2009

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

POLDERS, Jelle

Datum: 14.12.2009

Bestands- en datasynchronisatie

Jelle Polders

promotor :
dr. Stijn VANSUMMEREN

Eindverhandeling voorgedragen tot het bekomen van de graad
master in de informatica databases

universiteit hasselt

Dankwoord

Dit eindwerk vormt het sluitstuk van het eerste deel van mijn master opleiding in de informatica aan de transnationale Universiteit Limburg. Voor het tot stand komen van dit werk wil ik in de eerste plaats mijn promotor dr. Stijn Vansummeren bedanken voor de begeleiding, suggesties en de mogelijkheid om dit onderwerp te kunnen bestuderen. Verder wil ik ook mijn ouders bedanken voor hun steun doorheen mijn opleiding en de kans die ze mij gaven om te studeren. Mijn vrienden waarmee ik altijd het nodige plezier kon beleven om de moed erin te houden gedurende de jaren. En natuurlijk ook mijn medestudenten/vrienden aan wie ik de afgelopen vijf jaar zoveel mooie momenten te danken heb.

Jelle Polders,
Diepenbeek, augustus 2009.

Inhoudstafel

Lijst van figuren

Lijst van tabellen

Hoofdstuk 1

Inleiding

Waar we vroeger nog meerdere gebruikers per computer kenden, veranderen we geleidelijk aan in een maatschappij waar eenzelfde gebruiker meerdere computers heeft. Dat heeft tot gevolg dat onze belangrijkste data over meerdere computers verspreid en gerepliceerd wordt. Eén van de belangrijkste voordelen van zulke replica's is dat ze het verlies van data door systeemfalingen of gebruikersfouten helpen voorkomen.

Om er echter voor te zorgen dat we in alle replica's met de meest recente versie van de data werken, gebruiken we *synchronizers*. Synchronizers komen in verschillende softwaresystemen voor: gedistribueerde filesystemen, versiecontrolesystemen, databases, etc. Deze masterproef heeft als doelstelling het bestuderen, vergelijken en implementeren van de basisprincipes van recente *file en data synchonizers*. Intuïtief gezien zorgt een *file synchronizer* ervoor dat twee of meerdere gerepliceerde filesystemen gesynchroniseerd kunnen worden. Dit type synchronizer bekijkt de bestanden zelf echter als een zwarte doos en zal, in het geval waarbij eenzelfde bestand in twee verschillende replica's aangepast is, een foutboodschap geven. Een *data synchronizer*, daarentegen, synchronizeert niet op het niveau van filesystemen, maar op het niveau van de data in de bestanden zelf. Zulke synchronizers staan ons dan ook toe om eenzelfde bestand dat in twee verschillende replica's aangepast is te synchronizeren.

We zullen ons in deze masterproef focussen op wat Yasushi en Shapiro (Saito 05) *optimistische*, *multi-master*, *state-based* en *user-level* synchronizers noemen:

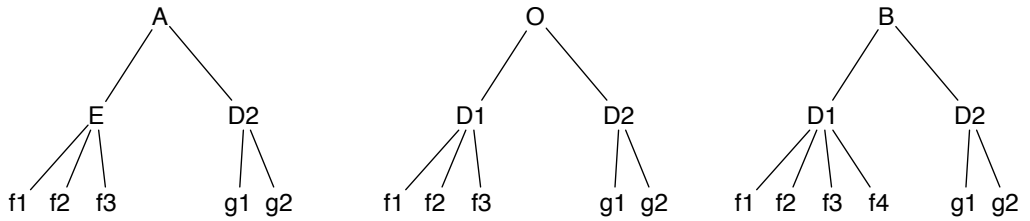
User-level versus operating-system level. *User-level synchronizers* maken, in tegenstelling tot de zogenaamde *operating-system level synchronizers*, geen deel uit van het besturingssysteem. Ze hebben dus geen specifieke besturingssyteem-afhankelijke vereisten en worden meestal expliciet door ge-

bruikers opgestart. Dat laatste staat in scherp contrast met bijvoorbeeld gedistribueerde filesystemen (welke tot de operating-system level synchronizers behoren) die onzichtbaar en transparant in de achtergrond werken. Omdat user-level synchronizers geen deel uitmaken van het besturingssysteem interageren ze met het filesysteem via dezelfde application programming interfaces (API's) als andere gebruikersprogramma's. Dat zorgt ervoor dat deze synchronizers gemakkelijker te bouwen en te ontwikkelen zijn dan synchronizers die wel deel uitmaken van het besturingssyteem. Dat ze relatief geïsoleerd zijn van andere systeemcomponenten vergemakkelijkt ook de specificatietaak. Een tweede voordeel is dat user-level tools beter aangepast zijn aan de heterogene instellingen die wij willen bereiken: het bouwen van een synchronizer die onafhankelijk is van het OS-platform brengt met zich mee dat de kernels van minstens twee besturingssystemen uitgebreid moeten worden.

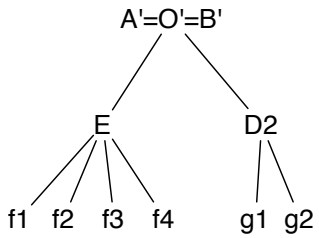
State-based versus operation-based. Belangrijk aan user-level synchronizers is dat er tijdens het synchroniseren niet geweten is welke operaties er sinds het laatste punt van synchronisatie uitgevoerd zijn op de replica's. De user-level synchronizer krijgt enkel de huidige toestand van de verschillende replica's gegeven en moet ze proberen te verzoenen op basis van informatie die de synchronizer zelf heeft bijgehouden over hun vorige toestand. Zulke synchronizers noemen we *state-based*. *Operation-based* synchronizers, daarentegen krijgen naast de huidige toestand ook de sequentie van alle operaties gegeven die sinds het laatste punt van synchronisatie op de replica's zijn uitgevoerd. Synchronizeren is dan een kwestie is van het samenvoegen van deze verschillende sequenties van operaties. Dat laat in het algemeen een fijnere manier van synchronizeren toe.

Beschouw bijvoorbeeld Figuur 1.1 waar drie filesystemen O, A en B afgebeeld worden. Interne knopen stellen folders voor, bladeren stellen bestanden voor. Hierbij is O het *archief* (de toestand op het laatste moment van synchronisatie) en zijn A en B twee aangepaste replica's die gesynchroniseerd moeten worden. In replica A is D_1 hernoemt naar E. In replica B is file f_4 toegevoegd aan D_1. Een state-based synchronizer kan niet weten of D_1 hernoemd of verwijderd is in A en zal het daarom conservatief als een verwijdering zien. Aangezien D_1 echter tegelijk aangepast is in B zal de syncrhonizer een conflict melden. Een operation-based synchronizer, daarentegen, weet dat het over een hernoeming gaat en kan rustig zowel D_1 naar E hernoemen, als f_4 aan E toevoegen. We krijgen dan bij een operation-based synchronizer de output zoals in Figuur 1.2.

Het grote nadeel van operation-based synchronizers is echter dat de se-



Figuur 1.1: Voorbeeld dat een conflict geeft bij een state-based synchronizer en niet bij een operation-based synchronizer.



Figuur 1.2: De output bij een operation-based synchronizer.

quentie van operaties uitgevoerd op de replica's op één of andere manier bijgehouden moet worden. De meeste software die we gebruiken om veel voorkomende gegevens (filesystemen, bookmarks, kalenders, tekstbestanden, XML-bestanden) aan te passen staan ons echter niet toe om deze operaties op te slaan. Bijgevolg moeten we vanuit praktische overwegingen wel voor state-based synchronizers kiezen.

Optimistisch versus pessimistisch. De traditionele replicatietechnieken, bijvoorbeeld gebruikt in gedistribueerde databasesystemen, proberen zogenaamde *single-copy consistentie* te onderhouden: wanneer er een aanpassing aan een replica gebeurt wordt de toegang tot de andere replica's geblokkeerd tot ze up-to-date gebracht zijn. Voor de gebruiker lijkt het dan alsof de replica's op elk moment gelijk zijn. We noemen deze techniek *pessimistisch*. *Optimistische* technieken, daarentegen, blokeren de toegang tot de data niet en gaan er van uit dat conflicten slechts zelden voorkomen.

Pessimistische technieken hebben drie grote nadelen:

1. Ze werken enkel wanneer alle replica's toegankelijk zijn. Wanneer één van de replica's ontoegankelijk is (bijvoorbeeld door een slechte netwerkverbinding of een afgekoppelde harde schijf) blijft de toegang tot de andere replica's geblokkeerd tot de verbinding hersteld is (Zhang 00; Dahlin 01)..

2. Ze schalen slecht naar een groot aantal replica's. Het is namelijk moeilijk een groot pessimistisch gerepliceerd systeem te bouwen dat frequente updates toelaat. De doorvoer en beschikbaarheid daalt immers wanneer het aantal replica's stijgt (Yu 00; Yu 01).

3. Sommige activiteiten vereisen inherent optimistische synchronisatietechnieken. Bij coöperatieve engineering of software development, bijvoorbeeld, is het vaak wenselijk dat mensen in relatieve isolatie kunnen werken. In dat geval is het beter toe te laten dat ze updates onafhankelijk uitvoeren en de occasionele conflicten opgelost worden nadat ze plaatsgevonden hebben dan de data te locken wanneer iemand ze aan het bewerken is (Jr. 88; Vesperman 07).

Optimistische synchronizers delen deze nadelen niet.Ten eerste is er een verbetering van de beschikbaarheid: zelfs wanneer netwerklinks en sites onbeschikbaar zijn kan er verder gewerkt worden. Ten tweede is optimistische replicatie goed schaleerbaar naar een groot aantal replica's omdat ze slechts kleine synchronisaties tussen sites nodig hebben. Ten derde laten ze sites en gebruikers toe om autonoom te blijven. Ten vierde laat optimistische replicatie ook asynchrone samenwerking tussen gebruikers toe, zoals in CVS (Concurrent Versions System).

Deze voordelen gaan echter ten koste van de consistentie van de gegevens . Beschouw bijvoorbeeld drie filesystemen A, B en C waarbij A en B gelijktijdig in hetzelfde deel op verschillende manier aangepast worden. In deze situatie krijgen we een conflict: het is wel mogelijk A en C of B en C te synchroniseren, maar A en B kunnen niet meer synchroniseerd worden. Bij een pessimistische replicatie, waar zulke gelijktijdige aanpassingen onmogelijk zijn, zou we dit probleem niet hebben.

Samenvattend kunnen we zeggen dat een pessimistisch algoritme wacht en een optimistisch algoritme speculeert. En aangezien in de praktijk (zeker in het geval van filesystemen) conflicten slechts zelden voorkomen, is optimistische replicatie dikwijls de juiste keuze.

Single-master versus multi-master We kunnen in optimistische replicatie een onderscheid maken tussen het aantal schrijvers. Langs de ene kant hebben we *single-master systemen* die één replica aanstellen als master. Hierbij onstaan alle updates bij de master en worden dan gepropageerd naar de andere replica's, of slaves. Deze worden ook *caching* systemen genoemd. Single-master systemen zijn eenvoudig maar hebben gelimiteerde beschikbaarheid, zeker wanneer de systemen veel updates meemaken.

Voorbeelden van single-master optimistische replicatiesystemen zijn onder ander de populaire versiecontrolesystemen CVS en SVN (Subversion).

Beide systemen laten toe dat gebruikers een groep gedeelde bestanden bewerken. Alle gebruikers hebben een private replica die ze autonoom kunnen aanpassen. Een centrale server houdt de "master" replica bij. Nadat een gebruiker zijn lokale kopie heeft aangepast kan hij zijn werk 'committen' en aan de centrale server vragen om te synchronizeren. Een commit slaagt onmiddelijk als geen enkele andere gebruiker intussen een aanpassing aan hetzelfde bestand heeft gemaakt. Als een andere gebruiker wel een verandering aan dezelfde file heeft gecommit, maar deze veranderingen niet overlappen, worden de bestanden automatisch gemerged. Anders zal de gebruiker ingelicht worden over het conflict welke hij of zij dan manueel moet oplossen.

Aan de andere kant hebben we *multimaster systemen* die updates onafhankelijk laten submitten op verschillende replica's en hen uitwisselen in de achtergrond. Multimaster systemen hebben een grotere beschikbaarheid dan single-master systemen (waneer de master onbereikbaar is, kunnen er immers geen gegevens uitgewisseld worden) maar ook een significant hogere complexiteit. Een ander probleem bij multimaster systemen is hun gelimiteerde schaalbaarheid ten gevolge van hun toegenomen aantal conflicten. De populaire gedistribueerde versiecontrolesystemen Git (Chacon 08) en Bazaar (Inc. 08) zijn voorbeelden van multimaster optimistische replicatiesystemen.

Conflictbehandeling We kunnen synchronizers ook nog opdelen naar de manier waarop ze conflicten behandelen. Conflicten komen bijvoorbeeld voor wanneer een file synchronizer de opdracht krijgt twee filesystemen A en B te synchroniseren, waarbij in beide filesystemen hetzelfde bestand op verschillende manier aangepast is; of wanneer een data synchronizer de opdracht krijgt twee replica's van eenzelfde bestand te synchronizeren waarin hetzelfde gedeelte van het bestand op verschillende manier aangepast is. In dat geval weet de synchroniser niet welke replica te kiezen, en krijgen we een conflict waarbij het bestand (of diens aangepaste gedeelte) in beide filesystemen ongesynchroniseerd blijft.

De beste aanpak om conflicten op te lossen is natuurlijk door er voor te zorgen dat ze niet kunnen voorkomen. Pessimistische algoritmen voorkomen conflicten door operaties te blokkeren of af te breken als dit nodig is. Single-master systemen voorkomen conflicten door updates enkel op één site te accepteren, maar toe te laten dat er gelezen kan worden van elke site. Deze aanpak heeft echter een kost, nl. de lagere beschikbaarheid zoals we eerder besproken hebben. De kans op conflicten kan ook gereduceerd worden door sneller te synchronizeren en/of door de te synchronizeren objecten op te splitsen in kleinere onafhankelijke eenheden.

Sommige systemen negeren conflicten door bijvoorbeeld een willekeurige replica te nemen en alle andere replica's hiermee te overschrijven. (In bovenstaand filesysteem kunnen we het conflicterende bestand in B bijvoorbeeld overschrijven met het conflicterende bestand in A.) Dit noemen we een *lost update*, wat uiteraard in de meeste gevallen niet de gewenste oplossing is.

Conflict resolutie is dikwijls erg applicatiespecifiek. De meeste systemen duiden een conflict gewoon aan en laten de gebruiker het probleem manueel oplossen. Andere systemen kunnen conflicten automatisch oplossen.

Wij kiezen er hiervoor om conflicten hoger hand, door middel van interactie met de gebruiker, te laten oplossen.

1.1 Overzicht

In het eerste deel van deze masterproef (Hoofdstuk 2) behandelen we filesynchronisatie. In het tweede deel (Hoofdstukken 3, 5 en 4) behandelen we datasynchronisatie. We starten hierbij in hoofdstuk 3 met het bespreken van diff3, dé standaard voor het mergen van ongecoördineerde veranderingen aan lijst-gestructueerde data zoals tekstbestanden. In Hoofdstuk 4 bespreken we hoe we ongeordende bomen kunnen synchronizeren. En afsluitend bekijken we in Hoofdstuk 5 hoe we veranderingen kunnen detecteren in geordende bomen. De implementatie is beschreven in 6.

Deel I

File synchronisatie

Hoofdstuk 2

Wat is een file synchronizer?

In dit hoofdstuk zullen we uitleggen wat een file synchronizer is.

2.1 Inleiding

Zoals eerder vermeld gaan we bij filesynchronisatie synchroniseren op het niveau van files, we gaan hierbij de data in de files niet synchroniseren. De files worden dus als een soort blackbox gezien, de enige vereiste is dat we files op gelijkheid kunnen testen. De synchronisatie gebeurt meestal op vraag van de gebruiker. Het algemene doel van een file synchronizer is om updates te detecteren en niet conflicterende updates te propageren. Dit lijkt eenvoudig maar een goede file synchronizer is niet eenvoudig te implementeren. Kleine misverstanden in de semantiek van filesysteem operaties kunnen ervoor zorgen dat data verloren kan gaan of overschreven kan worden, dit zijn uiteraard dingen die we willen vermijden. Bovendien is het concept van "user update" op zich al open voor verschillende interpretaties die leiden tot significante verschillen in de resultaten van de synchronisatie. Het probleem bij de documentatie geleverd bij file synchronizers is dat er ofwel totaal geen documentatie aanwezig is of dat de beschrijving in termen van low-level mechanismen is die niet matchen met de intuïtieve kijk van de gebruiker op het filesysteem. Dit maakt het moeilijker om een klare kijk te krijgen op wat de file synchronizer doet onder alle omstandigheden. Omdat een file synchronizer serieuze schade kan aanrichten bij ongewild of onverwacht gedrag willen we een bondig en rigoureus framework opzetten waarin de synchronisatie kan beschreven en behandeld worden en waarbij termen gebruikt worden die zowel gebruikers als implementeerders kunnen gebruiken.

Conceptueel gezien kunnen we de filesynchronisatie taak onderverdelen in twee conceptueel verschillende fasen: *update detectie*, m.a.w. het herkennen

waar updates in de replica's van het individuele filesysteem gemaakt zijn sinds het laatste punt van synchronisatie, en *verzoening*, het combineren van updates zodat de nieuwe toestand van elke replica gesynchroniseerd is.

De output van update detectie is een predicaat $dirty_S$ dat aangeeft welke updates er gemaakt zijn aan filesysteem S sinds het laatste punt van synchronisatie. Bij dit predicaat is het toegelaten om zogenaamde *veilige* fouten te maken. Met veilige fouten bedoelen we fouten waarbij er aangegeven wordt dat er updates hebben plaatsgevonden terwijl dit niet zo is. Het is echter niet toegelaten *onveilige* fouten te maken: fouten waarbij er updates niet gerapporteerd worden. De verzoener (Eng.: *reconciler*) zal deze predicaten gebruiken om te beslissen welke replica de meest up-to-date kopie van elke file of directory bevat. Het hele synchronisatieproces is voorgesteld in Figuur 2.1 Zoals we zien in de figuur starten beide replica's met dezelfde inhoud O. Op het moment waarop de synchronisatie gestart wordt zijn beide replica's in toestanden A en B, dit zijn de toestanden nadat de twee gebruikers updates hebben aangebracht aan het laatst gesynchroniseerde filesysteem O. Nadat de synchronisatie gestart is zal de update detector voor beide replica's respectievelijk de predicaten $dirty_A$ en $dirty_B$ berekenen. Deze geven aan of er al dan niet aanpassingen zijn aangebracht sinds de laatste synchronisatie. De verzoener zal deze predicaten gebruiken om nieuwe toestanden A' en B' te berekenen. Wanneer er in A en B geen conflicterende updates aanwezig waren zullen de nieuwe toestanden A' en B' hetzelfde zijn. De specificatie van de update detector is een relatie die moet van kracht zijn tussen O, A (of B) en $dirty_A$ (of $dirty_B$). Gelijkaardig is het gedrag van de verzoener gespecificeerd als een relatie tussen A, B, $dirty_A$, $dirty_B$, A' en B'.

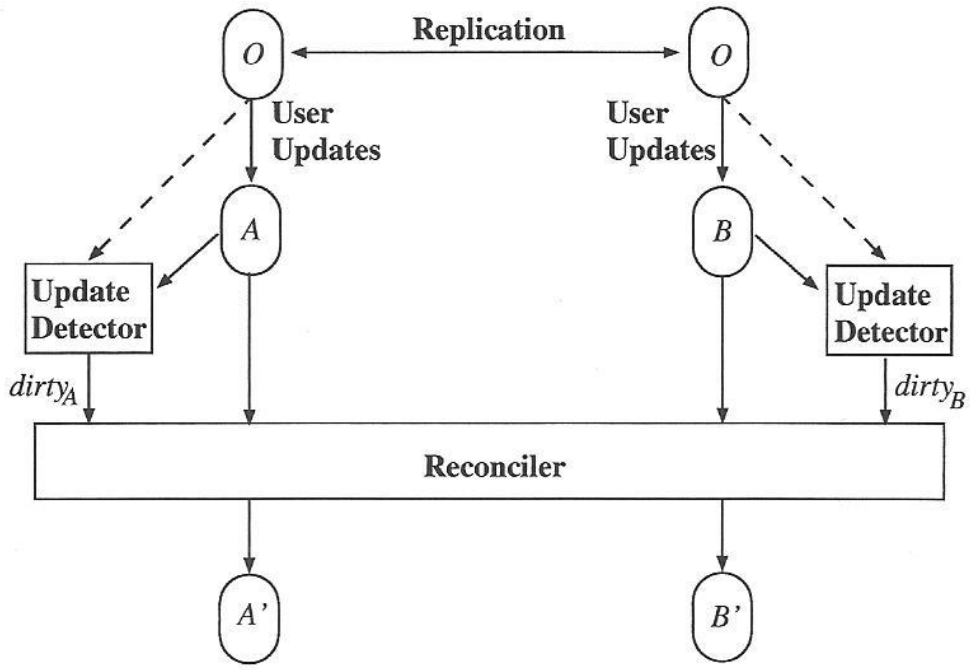
We zullen in dit hoofdstuk enkel filesynchronisatie bespreken maar sommige van de fundamentele resultaten die we hier opbouwen gelden voor zowel file als data synchronisatie.

Het hoofdstuk is als volgt opgebouwd: we starten in Sectie 2.2 met enkele basisdefinities te geven. In Sectie 2.3 bespreken we update detectie en in Sectie 2.4 doen we de verzoener uit de doeken. In Sectie 2.5 bespreken we enkele bestaande file synchronizers en afsluitend bespreken we in Sectie 2.6 enkele mogelijke uitbreidingen.

2.2 Basisdefinities

We moeten vooreerst een precieze manier hebben om over een filesysteem te praten. Dit hebben we nodig om op een rigoureuze manier te spreken over wat een synchronizer doet met het filesysteem dat het manipuleert.

Om de notatie in wat volgt eenvoudiger te maken maken we enkele ver-



Figuur 2.1: Het synchronisatieproces in het geval van twee replica's. We starten in beide replica's met dezelfde inhoud O. A en B zijn de toestanden nadat de twee gebruikers updates hebben aangebracht aan het laatst gesynchroniseerde filesysteem O. A' en B' zijn de toestanden nadat de verzoening heeft plaatsgevonden. Deze figuur is overgenomen uit (Balasubramaniam 98).

eenvoudigende veronderstellingen. Ten eerste veronderstellen we dat het filesysteem gedurende de synchronisatie enkel door de synchronizer zelf wordt aangepast. Dit betekent dat zover het de synchronizer betreft de filesystemen kunnen beschouwd worden als statische functies. Ten tweede veronderstellen we dat de filesystemen aan het einde van de vorige synchronisatie identiek waren. Ten derde beschouwen we enkel twee replica's. Ten slotte negeren we links (zowel harde als symbolische), file permissies, etc. In Sectie 2.6 lichten we toe wat er verandert als we sommige van deze restricties opheffen.

De metavariabelen x en y strekken over een verzameling $\mathcal{N}$ van filenamen. $\mathcal{P}$ is de verzameling van alle paden waarbij paden eindige sequenties zijn van namen gescheiden door een punt. Hierbij stellen de punten de slashes in Unix, de backslashes in Windows en de dubbelpunten in Mac OS voor. De metavariabelen p, q en r strekken zich over de paden. Het lege pad wordt hierbij genoteerd als ϵ. De concatenatie van twee paden p en q schrijven we als $p.q$. De lengte van het pad p duiden we aan als $|p|$, zo geldt bijvoorbeeld $|\epsilon| = 0$ en $|q.x| = |q| + 1$. Wanneer er geldt $p = q.r$ voor een r, is q een prefix van p en schrijven we $q \leq p$.

Wanneer f een functie is waarbij het domein een verzameling van paden is, schrijven we f/p voor de functie "f na p", gedefinieerd als $(f/p)(q) = f(p.q)$.

Aangezien we hier niet aan datasynchronisatie doen is het hier niet nodig te specificiëren wat files kunnen bevatten. Het enige dat we nodig hebben bij de verzameling van files $\mathcal{F}$ is dat het testen van gelijkheid ondersteunt wordt: gegeven de inhoud van twee files $f, g \in \mathcal{F}$ moeten we kunnen testen of $f = g$. Verder veronderstellen we dat elke knoop in het file systeem een klasse heeft, waarbij $\mathcal{S}$ staat voor de verzameling van alle klassen. Hierbij veronderstellen we klasse *DIR* voor een directory en klasse $FILE(f)$ voor een bestand met inhoud f. Definitie 2.1 geeft nu aan wat een filesysteem is.

Definitie 2.1 (Filesysteem)**.** *Een filesysteem is een totale functie S die paden p mapt op hun inhoud, wat een file f, een directory of niks kan zijn. We hebben dus het volgende:*

$$S : \mathcal{P} \rightarrow \mathcal{S} \cup \{\perp\} \tag{2.1}$$

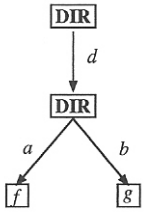
Filesystemen moeten voldoen aan volgende condities:

1. *Als $S(p.x) \neq \perp$[1] dan $S(p) = DIR$. Dit geeft aan dat enkel directories kinderen hebben.*

2. *Er een diepte n is zodat $S(q) = \perp$ voor elk pad met een lengte groter dan n. Dit geeft aan dat de diepte eindig is.*

Een voorbeeld van een filesysteem is weergegeven in Figuur 2.2. We

[1] Dit geeft aan dat $p.x$ een bestaande file of directory is.



Figuur 2.2: Een voorbeeld van een filesysteem met één subdirectory d met twee files a, met inhoud f, en b, met inhoud g. Deze figuur is overgenomen uit (Balasubramaniam 98).

hebben hier een root directory met één subdirectory d met twee files a, met inhoud f, en b, met inhoud g. Dit filesysteem kan voorgesteld worden door volgende functie:

$\{\epsilon \mapsto DIR, d \mapsto DIR, d.a \mapsto FILE(f), d.b \mapsto FILE(g)$, en $p \mapsto \perp$ voor alle andere paden $p\}$.

In wat volgt zullen we een afkorting gebruiken voor het weergeven van files, zo zullen we bijvoorbeeld $FILE(f)$ afkorten als f.

We schrijven *depth(S)* voor de lengte van het langste pad zodat $S(p) \neq \perp$. We schrijven $children_A(p)$ voor de verzameling namen die de directe kinderen van een pad p in een filesysteem A aangeven, dan is:

$$children_A(p) = \{q \mid q = p.x \text{ voor bepaalde } x \wedge A(q) \neq \perp\}.$$

We schrijven $children_{A,B}(p)$ voor $children_A(p) \cup children_B(p)$.

2.3 Update detectie

We hebben nu de nodige basisdefinities ingevoerd die het ons mogelijk maken de synchronisatie taak zelf te bespreken. In deze sectie zullen we het over update detectie hebben, het verzoenen laten we over aan Sectie 2.4. We zullen starten met de specificaties van de update detector kort samen te vatten, vervolgens zullen we verschillende implementatie strategieën bespreken die voldoen aan deze specificaties.

2.3.1 Specificaties

We voeren een voorwaarde toe aan het dirty predicaat die de specificatie van de verzoener eenvoudiger zal maken, namelijk dat het dirty predicaat

opwaarts gesloten is. Dit wil zeggen dat als er een pad is dat dirty gemarkeerd is al zijn voorouders ook dirty zijn.

Definitie 2.2. *Een dirtiness predicaat is een eindige verzameling $\phi \subseteq \mathcal{P}$ zodat $\forall q \in \phi$, $\forall p \leqslant q$ ook geldt $p \in \phi$. In het vervolg schrijven we $\phi(q)$ om aan te geven dat $q \in \phi$. Een onmiddellijke consequentie van deze definitie is dat als ϕ een dirtiness predicaat is, dan geldt er $p \leq q \wedge \neg\phi(p)$ impliceert $\neg\phi(q)$.*

Definitie 2.3. *Veronderstel dat O en S filesystemen zijn en dat $dirty_S$ een dirtiness predicaat is. Dan wordt er gezegd dat $dirty_S$ (veilig) de updates van O naar S berekent als $\neg dirty_S(p)$ $O/p = S/p$ impliceert, voor alle paden p.*

Een cruciale eigenschap van deze laatste definitie is dat als A, B en O filesystemen zijn en $dirty_A$ en $dirty_B$ de updates berekenen van O naar A en van O naar B, $\neg dirty_A(p)$ en $\neg dirty_B(p)$ samen impliceren dat $A/p = B/p$.

2.3.2 Implementatie strategieën

Er zijn verschillende manieren waarop update detectors die aan de bovengaande specificaties voldoen geïmplementeerd kunnen worden. We zullen in deze sectie er enkele beschouwen en hun voor- en nadelen weergeven. We spitsen ons in onze uitleg toe op het Unix filesysteem, maar de meest gebruikte strategieën die we beschrijven werken ook onder andere besturingssystemen.

Triviale update detector

Dit is de eenvoudigst mogelijke implementatie: we markeren elke file als dirty. De verzoener zal dus elke file (behalve degenen die identiek zijn in beide filesystemen) als een conflict beschouwen. Dit is in sommige situaties een acceptabele update detectie strategie. Het feit dat de verzoener de huidige inhoud van alle files in de twee filesystemen moet vergelijken is niet zo'n groot opgave als het filesysteem klein genoeg is en de link tussen beide snel genoeg is. Bovendien is het feit dat alle update's leiden tot conflicten niet een probleem in de praktijk als er slechts enkele van hen zijn. De hele file synchronizer ontaard in dat geval to een soort van recursieve remote diff.

Voor kleine filesystemen met een goede communicatielink is deze strategie goedkoop omwille van het feit dat we het dirty predicaat niet exact moeten berekenen. Het exact berekenen van dit predicaat zou duur zijn in het gebruik van diskruimte en nog belangrijker in de tijd die het nodig heeft om het verschil tussen de huidige inhoud en de opgeslagen kopies van het filesysteem te berekenen.

Exact update detector

Aan het andere uiteinde staat de update detector die het dirty predicaat exact berekent. Dit gebeurt bijvoorbeeld door een kopie bij te houden van het hele laatst gesynchroniseerde filesysteem en deze toestand te vergelijken met de huidige, m.a.w. de remote diff in het vorige geval met de twee locale diffs.

Zoals we reeds aangegeven hebben is het exact berekenen van het predicaat duur in zowel de gebruikte diskruimte als in de gebruikte tijd. Dit is echter wel een goede strategie in situaties waar de synchronizer off-line (bv. in het midden van de nacht) gerund wordt, of wanneer de link tussen de twee computers een erg lage bandwijdte heeft en de minimale communicatie die we nodig hebben voor valse conflicten dus een groot nadeel is.

Simpel modtime update detector

In een derde strategie gebruiken we de tijd waarop er een laatste aanpassing is gebeurt, dit resulteert in een veel goedkopere maar minder accurate oplossing. We slaan hierbij slechts één waarde op tussen de synchronisaties in elk replica, namelijk de tijd van de vorige synchronisatie (overeenstemmend met de lokale klok). Om updates te detecteren vergelijken we voor elke file de tijd waarop ze het laatst is aangepast met de tijd van de laatste synchronisatie. Is de tijd ouder dan die van de laatste synchronisatie dan is de file niet dirty, in het andere geval wel.

Bij Unix, Windows en Mac OS is er echter een probleem, wanneer we een file hernoemen wordt zijn modtime niet aangepast maar wel die van zijn bovenliggende map. Namen zijn hier namelijk een eigenschap van directories en niet van files. Dit geeft bijvoorbeeld problemen wanneer we twee files hebben, a en b, waarbij a niet aangepast is sinds de laatste synchronisatie. Stel we verwijderen b en hernoemen a naar b, in dit geval zal het pad b niet als dirty gemarkeerd worden waardoor a in de andere replica verwijderd en b ongemoeid gelaten wordt.

Kijken naar de modtime van de directory biedt hier ook geen oplossing. We weten niet welke verandering de modtime van de directory heeft aangepast en we zouden bijgevolg alle files in de directory als dirty moeten markeren. We zouden dus veel te veel files als dirty markeren, en in het geval dat de modtime van alle directories is aangepast komen we zelf in het geval van de triviale update detector.

Modtime-inode update detector

Een betere strategie voor update detectie onder Unix is gebaseerd op modtimes en inode nummers. We onthouden hierbij niet enkel de tijd van de laatste synchronisatie, maar ook het inode nummer van elke replica op het tijdstip van de laatste synchronisatie. De update detector markeert een pad nu als dirty als aan één van volgende voorwaarden voldaan is:

1. De inode nummer is niet dezelfde als de opgeslagen inode nummer.
2. De modtime is recenter dan het tijdstip van de laatste synchronisatie.

Het is hier niet nodig om naar de modtime te kijken van de omhullende directories.

Wanneer bijvoorbeeld b vervangen wordt door a zoals in de vorige sectie beschreven dan zal de nieuwe b een ander inode nummer hebben dan de oorspronkelijke b. Zowel a als b zullen dus als dirty gemarkeerd wordt, dit leidt tot (een correcte) verwijdering en invoeging in de andere replica.

De auteurs van (Balasubramaniam 98) hebben ook nog geëxperimenteerd met een derde variant, waar inode nummers enkel opgeslagen worden voor de directories en niet voor elke file. Dit gebruikt veel minder opslagruimte dan wanneer we voor elke file de inode nummer opslaan, maar is minder accuraat. Volgens hun bevindingen geeft het gebruik van de inode nummer van alle files in alle gevallen een beter resultaat.

On-line update detector

Een laatste strategie die we bespreken bekijkt de volledige trace van acties die de gebruiker uitvoert op het filesysteem. Deze strategie is moeilijk te implementeren onder Unix maar eenvoudiger te implementeren onder andere besturingssystemen zoals Windows. De detector zal een file als dirty markeren als de gebruiker er iets heeft aan aangepast zelf al is het netto-effect zero. Dit geeft dus niet hetzelfde resultaat als de exact update detector maar leunt er wel dicht bij aan en is een stuk goedkoper om uit te voeren dan de exact update detector. Deze strategie veronderstelt de mogelijkheid om arbitraire gebruikersacties op te sporen die het filesysteem aanpassen en is daardoor de voorkeurstrategie van gedistribueerde filesystemen van alle aard zoals Coda (Kistler 96) en (Kumar 94), Ficus (Reiher 94) en (Page 98), Bayou (Terry 95) en (Petersen 97) en LittleWorks (Huston 93).

2.4 Verzoening

Nu dat we de update detector uit de doeken hebben gedaan kunnen we overgaan naar de belangrijkste component van de synchronizer: de verzoener. Ook hier zullen we weer starten met de specificaties van de verzoener samen te vatten. Daarna zullen we het recursieve verzoeningsalgoritme geven en zullen we aantonen dat (a) het algoritme aan de gegeven vereisten voldoet, en (b) de specificaties het gedrag van het algoritme volledig vastleggen, m.a.w. dat elk synchronisatie algoritme dat aan de specificaties voldoet zich gelijkaardig moet gedragen.

2.4.1 Specificaties

In Sectie 2.3 hebben we gezien dat een pad geüpdatet is in A wanneer zijn waarde in A verschillend is van zijn originele waarde op het tijdstip van de laatste synchronisatie. We zeggen nu dat twee updates in *conflict* zijn wanneer de inhoud van A en B, die het resultaat zijn van de updates, verschillend zijn. Wanneer de nieuwe inhoud van A en B overeenkomt zijn de updates niet in conflict.

Stel dat A en B de huidige toestanden zijn van de twee filesystemen en dat we de dirtiness predicaten $dirty_A$ en $dirty_B$ hebben berekend. Wanneer we nu de verzoener uitvoeren met deze twee toestanden als input krijgen we twee nieuwe toestanden C en D. De verzoener moet het volgende gedrag vertonen:

1. proprageer alle niet conflicterende updates,
2. indien er een conflict is, doe niks.

Een actuele en goede synchronizer zal bij een conflict uiteraard een betere oplossing proberen te bieden dan gewoon niks te doen. We kunnen bijvoorbeeld enkele extra heuristieken toepassen gebaseerd op het type van files dat betrokken zijn in het conflict, of we kunnen de gebruiker om advies vragen, of we kunnen manuele aanpassingen doen. Deze acties kunnen echter gemakkelijk aan ons model toegevoegd worden door ze te zien alsof ze uitgevoerd zijn voor de synchronizer zijn echte werk begon.

Bij het verzoenen van twee replica's kunnen we volgende vier gevallen onderscheiden:

Definitie 2.4. *1. Als A en B overeenstemmen in p moeten we niks doen en $C(p)$ en $D(p)$ zijn in dit geval hetzelfde als $A(p)$ en $B(p)$.*

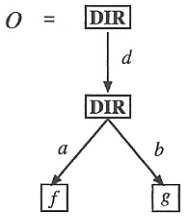
2. *Als $\neg dirty_A(p)$ dan is (aangezien $dirty_A$ opwaarts gesloten is) de hele subtree die zijn root in p heeft onveranderd in A en alle updates in de overeenstemmende subtree in B moeten geproprageerd worden in beide kanten. Dit betekent dat zowel C/p (de subtree met root in p in C) als D/p gelijk moeten zijn aan B/p.*

3. *Als $\neg dirty_B(p)$ dan verkrijgen we analoog het volgende: $C/p = D/p = A/p$.*

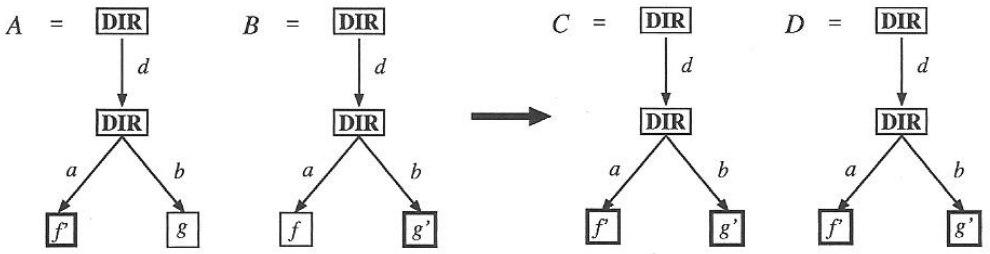
4. *Als $dirty_A(p)$ en $dirty_B(p)$ en $A(p) \neq B(p)$ dan zitten we in de conflicterende situatie en verkrijgen we het volgende: $C/p = A/p$ en $D/p = B/p$.*

We zullen deze gevallen a.d.h.v. enkele voorbeelden verduidelijken. We starten met een origineel filesysteem O weergegeven in Figuur 2.3. We gaan er in de voorbeelden van uit dat we een exact update detector gebruiken.

In een eerste voorbeeld passen we de inhoud van $d.a$ in A en $d.b$ in B aan, zodat $dirty_A$ *true* is voor de paden $d.a$, d en ϵ en *false* voor $d.b$, $dirty_B$ is *true* voor $d.b$, d en ϵ en *false* voor $d.a$. Het predicaat $dirty_B(d.a)$ is *false* waardoor de aanpassingen uitgevoerd in replica B geproprageerd worden in beide replica's. Hetzelfde verhaal geldt voor de aanpassingen aan $d.b$ in filesysteem B aangezien $\neg dirty_A(d.B)$. We verkrijgen dus het resultaat weergegeven in Figuur 2.4.

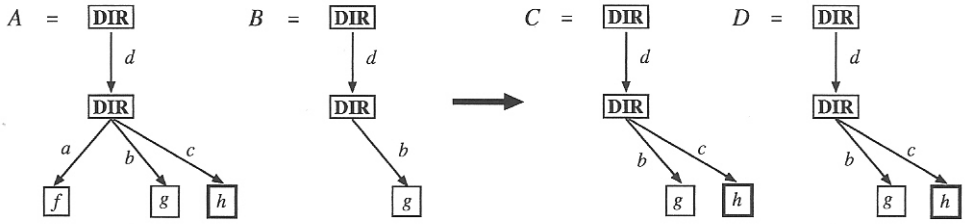


Figuur 2.3: Het origineel filesysteem O. Deze figuur is overgenomen uit (Balasubramaniam 98).



Figuur 2.4: Voorbeeld 1 van verzoening waarbij $d.a$ in A en $d.b$ in B zijn aangepast. Beide aanpassingen worden geproprageerd in de andere replica. Deze figuur is overgenomen uit (Balasubramaniam 98)

In een tweede voorbeeld, weergegeven in Figuur 2.5 voegen we in filesysteem A een nieuw pad $d.c$ toe. Dit is een voorbeeld van *insert/delete ambiguïteit.*

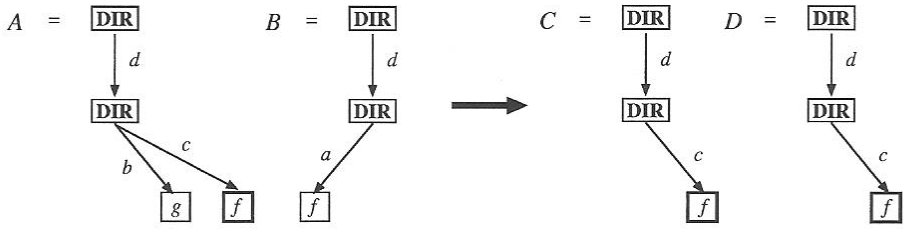


Figuur 2.5: Voorbeeld 2 van verzoening waarbij er in A een nieuw pad c wordt toegevoegd en er in B pad a wordt verwijdert. Beide aanpassingen worden geproprageerd in de andere replica. Deze figuur is overgenomen uit (Balasubramaniam 98).

Dit zorgt ervoor dat $dirty_A$ *true* is voor de paden $d.c$, d en ϵ en *false* voor de andere paden. In replica B verwijderen we pad $d.a$, hierdoor wordt $dirty_A$ *true* voor de paden $d.a$, d en ϵ en *false* voor het pad $d.b$. Dit is een klassiek voorbeeld van een insert/delete ambiguïteit. Wanneer de verzoener enkel de toestanden A en B kan zien weet hij niet of c toegevoegd is in A of verwijderd is in B en oorspronkelijk aanwezig was in beide replica's. En of a nieuw is in A of verwijdert is in B. Het dirty predicaat lost de ambiguïteit hier op: c is enkel dirty in A, terwijl a enkel dirty is in B. Dit leert ons dat a verwijderd is in B en c toegevoegd is in A. Wanneer we geen exacte update detector maar een minder accurate update detector hadden gebruikt was c mogelijk ook dirty in B en a eveneens dirty in A. Het effect zou dan zijn dat er een conflict zou gerapporteerd worden aan de verzoener en er geen veranderingen

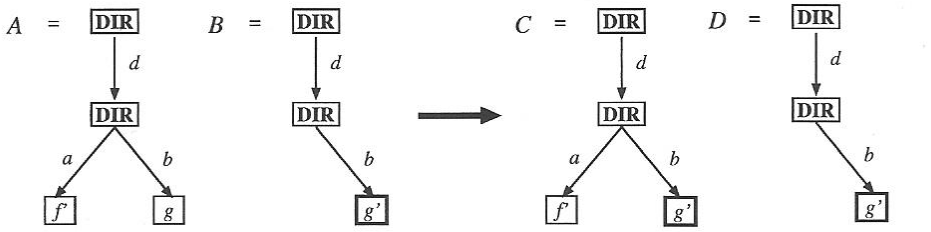
in het filesysteem zouden worden aangebracht. De faling gebeurt hier dus op een veilige manier.

In een derde voorbeeld weergegeven in Figuur 2.6 wordt de file a in A hernoemd naar c. In B verwijderen we de file b. Dit zorgt ervoor dat in A de paden $d.a$, $d.c$, d, en ϵ als dirty worden gemarkeerd. In B worden de paden $d.b$, d en ϵ als dirty gemarkeerd. Dit resulteert in de nieuwe toestanden C en D zoals weergegeven in de figuur.



Figuur 2.6: Voorbeeld 3 van verzoening waarbij de file a in A hernoemd wordt naar c en in B de file b verwijdert wordt. Beide aanpassingen worden geproprageerd in de andere replica. Deze figuur is overgenomen uit (Balasubramaniam 98).

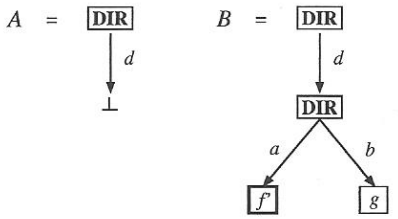
Beschouw het vierde voorbeeld weergegeven in Figuur 2.7 waar a is aangepast in A, a verwijderd is in B en b is aangepast in B. In A zijn hierdoor de paden $d.a$, d en ϵ als dirty gemarkeerd. In B zijn de paden $d.a$, $d.b$, d en ϵ als dirty gemarkeerd. Volgens het vierde geval van Definitie 2.4 moeten we $d.a$ onveranderd laten in C en D. We propragereren $d.b$ van B echter wel in A zoals weergegeven in de figuur. We bekomen dus een toestand waarin C en D verschillend zijn.



Figuur 2.7: Voorbeeld 4 van verzoening waarbij de file a aangepast wordt in A, a verwijdert wordt in B en b aangepast wordt in B. We bekomen na de verzoening twee inconsistente toestanden C en D. Deze figuur is overgenomen uit (Balasubramaniam 98).

Veronderstel nu als laatste en vijfde voorbeeld dat we, zoals weergegeven

in Figuur 2.8, in A subdirectory d volledig verwijderen en we in B pad $d.a$ aanpassen. Dan worden de paden $d.a$, $d.b$, d en ϵ dirty in A en $d.a$, d en ϵ dirty in B. Wat moet nu de inhoud van $D(d.b)$ zijn? Langs de ene kant moet doordat d dirty is in A en B en $A(d) \neq B(d)$ gelden dat $D/d = B/d$ en dus $D(d.b) = g$. Langs de andere kant moet doordat $d.b$ dirty is A en niet in B gelden dat $D/d.b = A/d.b$ en dus $D(d.b) = \perp$. Met de huidige specificatie kunnen we dit niet oplossen. Maar we geloven dat de correcte waarde voor $D(d.b)$ in dit geval g is.



Figuur 2.8: Voorbeeld 5 van verzoening waarbij in A de subdirectory d volledig verwijdert wordt en we in B pad $d.a$ aanpassen. Dit geeft ons na verzoening twee inconsistente toestanden C en D. Deze figuur is overgenomen uit (Balasubramaniam 98).

We gaan nu onze formele specificatie verfijnen zodat ook het laatste voorbeeld opgelost kan worden. Hiervoor voeren we in Definitie 2.5 eerst een nieuw begrip, een relevant pad, in.

Definitie 2.5. *Laat A en B twee filesystemen zijn. Een pad p is relevant in (A, B) indien ofwel $p = \epsilon$ ofwel $p = q.x$ voor bepaalde q en x, met $A(q) = B(q) = DIR$.*

Nu kunnen we onze formele specificatie verder verfijnen, dit is weergegeven in Definitie 2.6.

Definitie 2.6. *Er wordt gezegd dat het paar van nieuwe filesystemen (C, D) een synchronisatie is van een paar van originele filesystemen (A, B), met betrekking tot het dirtiness predicaat $dirty_A$ en $dirty_B$, als voor elk relevant pad p in (A, B) de volgende condities gelden:*

1. $A(p) = B(p) \Rightarrow C(p) = A(p) \wedge D(p) = B(p)$
2. $\neg dirty_A(p) \Rightarrow C/p = D/p = B/p$
3. $\neg dirty_B(p) \Rightarrow C/p = D/p = A/p$
4. $dirty_A(p) \wedge dirty_B(p) \wedge A(p) \neq B(p) \Rightarrow C/p = A/p \wedge D/p = B/p$

2.4.2 Algoritme

We hebben nu de specificaties van de verzoener besproken. We zullen in deze sectie een algoritme voor de verzoener bespreken dat aan deze eigenschappen voldoet. In de volgende sectie zullen we dan nagaan of het algoritme correct en compleet is. We zullen vooreerst een nieuwe definitie invoeren die we nodig zullen hebben in ons algoritme.

Definitie 2.7. *Laat S en T twee functies op paden zijn en laat p een pad zijn. Dan schrijven we $T[p \leftarrow S]$ voor de functie gevormd door het verplaatsen van de subtree in p in T met S, formeel gedefinieerd als volgt:*

$$T[p \leftarrow S](q) = \begin{cases} S(r) & \textit{indien } q = p.r \\ T(q) & \textit{anders} \end{cases}$$

Algoritme 1 krijgt een paar van filesystemen A en B en een pad p mee en geeft een paar van filesystemen C en D terug dat in de subtree met root p gesynchroniseerd is. 1

Algoritme 1 Verzoener

```
1: function RECON(A, B, p)
2:     if ¬dirty_A(p) ∧ ¬dirty_B(p) then
3:         return (A, B)
4:     else if A(p) = B(p) = DIR then
5:         (A_0, B_0) = (A, B)
6:         Laat nu {p_1, p_2, ..., p_n} = children_{A,B}(p) (in lexicografische volgorde)
7:         for all i such that 0 ≤ i < n do
8:             (A_{i+1}, B_{i+1}) = RECON(A_i, B_i, p_{i+1})
9:         end for
10:        return (A_n, B_n)
11:    else if ¬dirty_A(p) then
12:        return (A[p ← B/p], B)
13:    else if ¬dirty_B(p) then
14:        return (A, B[p ← A/p])
15:    else
16:        return (A, B)
17:    end if
18: end function
```

Een eenvoudige inductie op max(depth(A), depth(B)) - $|p|$ toont aan dat *recon* eindigt voor alle filesystemen A en B en paden p. We merken ook op

dat de updates aan de filesystemen A en B enkel uitgevoerd worden door de recursieve oproepen en Definitie 2.7. Dit garandeert dat $recon(A, B, p)$ onaangeroerd blijft in alle delen van A en B die buiten de subtree met root in p liggen.

2.4.3 Correctheid en compleetheid van het algoritme

We zullen nu aantonen dat Definitie 2.7 het gedrag van de verzoener volledig karakteriseert en dat ons verzoeningsalgoritme rekening houdend met de specificaties correct is.

Om aan te tonen dat ons verzoeningsalgoritme correct is zullen we eerst een verfijning toevoegen aan de originele vereisten die ons zullen toelaten onze aandacht te focussen op een specifieke regio van de twee filesystemen.

Definitie 2.8. *Er wordt gezegd dat een paar van nieuwe filesystemen (C, D) een synchronisatie is na p van originele filesystemen (A, B) als p een relevant pad is in (A, B) en de volgende condities voldaan zijn voor elk relevant pad $p.q$ in (A, B):*

$$A(p.q) = B(p.q) \Rightarrow C(p.q) = A(p.q) \wedge D(p.q) = B(p.q) \tag{2.2}$$

$$\neg dirty_A(p.q) \Rightarrow C/p.q = D/p.q = B/p.q \tag{2.3}$$

$$\neg dirty_B(p.q) \Rightarrow C/p.q = D/p.q = A/p.q \tag{2.4}$$

$$dirty_A(p.q) \wedge dirty_B(p.q) \wedge A(p.q) \neq B(p.q) \Rightarrow C/p.q = A/p.q \wedge D/p.q = B/p.q \tag{2.5}$$

We merken op dat Definitie 2.6 slechts een speciaal geval is van de bovenstaande waar $p = \epsilon$.

Definitie 2.9. *We schrijven $synch\,/_p(C, D, A, B)$ als*

1. *(C, D) is een synchronisatie van (A, B) na p, en*
2. *$p \not\leq q$ impliceert $C(q) = A(q) \wedge D(q) = B(q)$ voor alle paden q.*

De definitie van de verzoener legt het verzoenen uniek vast. Gegeven twee filesystemen die gesynchroniseerd zijn op een bepaald punt in het verleden is er maximaal één paar van nieuwe filesystemen dat voldoet aan de vereisten.

Stelling 2.1. *Laat A, B en O filesystemen zijn en veronderstel dat $dirty_A$ en $dirty_B$ de predicaten t.o.v. het originele filesysteem O zijn. Laat p een relevant pad zijn in (A, B). Als (C_1, D_1) en (C_2, D_2) beide synchronisaties zijn na p, dan $C_1/p = C_2/p$ en $D_1/p = D_2/p$.*

Bewijs. We stellen door inductie op max(depth(A), depth(B)) - $|p.q|$ dat

$$C_1/p.q = C_2/p.q \wedge D_1/p.q = D_2/p.q$$

voor elk relevant pad $p.q$ in (A, B).
Er zijn vier gevallen die we moeten beschouwen, afhankelijk van het feit of het pad $p.q$ dirty is of niet in A en B:

- Veronderstel dat er geldt $\neg dirty_A(p.q) \wedge \neg dirty_B(p.q)$.
  Aangezien het dirtiness predikaat opwaarts gesloten is geld er dus: $\neg dirty_A(p.q.r)$ and $\neg dirty_B(p.q.r)$ voor elke r.
  Dan geldt er omwille van Definitie 2.3 voor elke r $A(p.q.r) = B(p.q.r)$.
  Zodat we omwille van (2.2) krijgen:

  $$C_1(p.q.r) = A(p.q.r) = B(p.q.r) = D_1(p.q.r), \text{ en}$$
  $$C_2(p.q.r) = A(p.q.r) = B(p.q.r) = D_2(p.q.r).$$

  Dan geldt er: $C_1/p.q = C_2/p.q$ en $D_1/p.q = D_2/p.q$.

- Veronderstel $\neg\, dirty_A(p.q) \ \wedge \ dirty_B(p.q)$.
  Dan geldt er volgens (2.3) dat $C_1/p.q = D_1/p.q = B/p.q$ en $C_2/p.q = D_2/p.q = B/p.q$.
  Dus: $C_1/p.q = C_2/p.q$ en $D_1/p.q = D_2/p.q$.

- Veronderstel dat $dirty_A(p.q) \ \wedge \ \neg dirty_B(p.q)$.
  Dan geldt er omwille van (2.4) $C_1/p.q = D_1/p.q = A/p.q$ en $C_2/p.q = D_2/p.q = A/p.q$.
  Dus: $C_1/p.q = C_2/p.q$ en $D_1/p.q = D_2/p.q$.

- We veronderstellen als laatste dat $dirty_A(p.q) \wedge dirty_B(p.q)$.
  We kunnen nu twee gevallen onderscheiden afhankelijk van of $A(p.q) = B(p.q)$ geldt of niet.

  - Als $A(p.q) = B(p.q)$ dan krijgen we door (2.2):
    (a) $C_1(p.q) = A(p.q)$ en $D_1(p.q) = B(p.q)$
    (b) $C_2(p.q) = A(p.q)$ en $D_2(p.q) = B(p.q)$.

    Nu moeten we drie deelgevallen beschouwen:
    (i) $A(p.q) = B(p.q) \in \mathcal{F}$
    (ii) $A(p.q) = B(p.q) = \perp$
    (iii) $A(p.q) = B(p.q) = DIR$

    * In gevallen (i) en (ii) impliceren (a) en (b):

$C_1(p.q) = C_2(p.q)$ en $D_1(p.q) = D_2(p.q)$.

Bij definitie van een filesysteem geldt er voor elke $r \neq \epsilon$:

$C_1/p.q.r = C_2/p.q.r = D_1/p.q.r = D_2/p.q.r = \perp$.

Zodat we

$C_1/p.q = C_2/p.q$ en $D_1/p.q = D_2/p.q$

hebben. Hetgeen we moesten bewijzen.

* In geval (iii), krijgen we door de inductiehypothese toe te passen:

  $C_1/p.(q.x) = C_2/p.(q.x)$ en $D_1/p.(q.x) = D_2/p.(q.x)$

  voor alle x, daar $p.q.x$ een relevant pad is. Dit impliceert samen met (a) en (b) dat:

  $C_1/p.q = C_2/p.q$ en $D_1/p.q = D_2/p.q$.

  Hetgeen we moesten bewijzen.

– Als $A(p.q) \neq B(p.q)$ dan hebben we door (2.5):

$C_1/p.q = A/p.q$ en $D_1/p.q = B/p.q$
$C_2/p.q = A/p.q$ en $D_2/p.q = B/p.q$

wat impliceert dat $C_1/p.q = C_2/p.q$ en $D_1/p.q = D_2/p.q$, zoals gevraagd was.

□

De rest van deze sectie zullen we wijden aan het bewijs van de correctheid van het algoritme. Om dit te kunnen bewijzen zullen we eerst nog enkele eigenschappen moeten geven. We eisen:

- dat de volgorde waarin (niet overlappende) regio's van een filesysteem gesynchroniseerd zijn irrelevant is,
- dat het synchronizeren van een directory (een regio) het resultaat is van het synchronizeren van zijn inhoud (subregio's).

We beginnen de argumentatie met enkele nieuwe definities.

Definitie 2.10. *Twee paden p en q zijn onvergelijkbaar als geen van twee een prefix is van de andere, m.a.w. als $p \not\leq q \wedge q \not\leq p$.*

Hieruit volgt Stelling 2.2

Stelling 2.2. *Laat paden p en q onvergelijkbaar zijn. Dan geldt er als $q \not\leq r$ dat p en r onvergelijkbaar zijn.*

Bewijs. Dit volgt eenvoudig uit bovenstaande definitie. We nemen p en q, twee onvergelijkbare paden, en $q \leq r$. Als p en r niet onvergelijkbaar zijn dan geldt er bij definitie $p \leq r$ of $r \leq p$. In beide gevallen leiden we een contradictie af:

- Als $r \leq p$ dan $q \leq p$ wat een contradictie is met de veronderstelling dat p en q onvergelijkbaar zijn.
- Als $p \leq r$ dan geldt $p \leq q$ of $q \leq p$, wat een contradictie is met de veronderstelling.

□

Definitie 2.11. *Er wordt gezegd dat een verzameling van paden P paarsgewijs onvergelijkbaar is als voor elke p, $q \in P$ ofwel geldt $p = q$ ofwel p en q onvergelijkbaar zijn.*

Definitie 2.12. *Laat P de verzameling van paarsgewijze onvergelijkbare relevante paden zijn in (A, B). Er wordt gezegd dat het paar (C, D) een synchronisatie is na P van het originele filesysteem (A, B) als (C, D) een synchronisatie is van (A, B) na p voor elke $p \in P$.*

Definitie 2.13. *We schrijven synch $/_P(C, D, A, B)$ als:*

- *(C, D) een synchronisatie is van (A, B) na P,*
- *voor alle q, $\neg(\exists p \in P.p \leq q)$ impliceren $C(q) = A(q) \wedge D(q) = B(q)$.*

Lemma 2.1 (Monotoniciteit)**.** *Laat P een verzameling van paarsgewijze onvergelijkbare en relevante paden in (A, B) zijn. Laat q een relevant pad zijn in (A, B) zodat p en q onvergelijkbaar zijn voor elke $p \in P$. Indien zowel synch $/_P(C', D', A, B)$ als synch $/_q(C, D, C', D')$ dan ook synch $/_{P'}(C, D, A, B)$ met $P' = P \cup \{q\}$.*

We bewijzen dit in drie stappen:

1. Laat r een relevant pad zijn in (A, B) zodat $q \leq r$. Dan geldt er omwille van Stelling 2.2 voor elke r die aan het voorgaande voldoet dat $p \not\leq r$ voor elke $p \in P$. Zodat er door *synch* $/_P(C', D', A, B)$ geldt $C'(r) = A(r)$ en $D'(r) = B(r)$ en conditie (2.2) tot (2.5) van Definitie 2.8 waar zijn door *synch* $/_q(C, D, C', D')$. Dus kunnen we concluderen dat $(C,\ D)$ een synchronisatie is van (A, B) na q.

2. Laat r een relevant pad zijn in (A, B) zodat $p \not\leq r$ voor bepaalde $p \in P$. Dan geldt er omwille van Stelling 2.2 voor elke r die aan bovengaande voldoet het volgende: $q \not\leq r$. Zodat er door $synch\,/_q(C, D, C', D')$ geldt dat $C(r) = C'(r)$ en $D(r) = D'(r)$ en conditie (2.2) tot (2.5) van Definitie 2.8 waar zijn door $synch\,/_P(C, D, C', D')$. We kunnen dus concluderen dat (C, D) een synchronisatie is van (A, B) na P.

3. Laat r een pad zijn zodat $p' \not\leq r$ voor alle $p' \in P'$. Dan geldt er door $synch\,/_q(C, D, C', D')$ en $synch\,/_P(C, D, C', D')$ dat $C(r) = C'(r) = A(r)$ en $D(r) = D'(r) = B(r)$.

Door (1), (2) en (3) geldt er nu $synch\,/_{P'}(C, D, A, B)$.

Lemma 2.2 (Permutatie)**.** *Laat voor elke n $P_n = \{p_i \mid 1 \leq i \leq n\}$ een verzameling zijn van paarsgewijze onvergelijkbare relevante paden in (A, B). Als*

1. *$(A_0, B_0) = (A, B)$, en*
2. *$synch\,/_{p_{i+1}}(A_{i+1}, B_{i+1}, A_i, B_i)$ voor elke i waarvoor $0 \leq i < n$,*

dan $synch\,/_{P_n}(A_n, B_n, A, B)$.

Bewijs. Door inductie op n

- Voor het basisgeval waar $n = 0$ volgt er duidelijk uit de veronderstelling dat $synch\,/_{\{\}}(A_0, B_0, A, B)$.
- Veronderstel voor de inductiestap dat $n \geq 1$. Als P_n nu paarsgewijs onvergelijkbaar is dan is P_{n-1} dat zeker. Door de inductiehypothese hebben we dus $synch\,/_{p_{n-1}}(A_{n-1}, B_{n-1}, A, B)$. Bovendien geldt $synch\,/_{p_n}(A_n, B_n, A_{n-1}, B_{n-1})$. Dan krijgen we $synch\,/_{P_n}(A_n, B_n, A, B)$. door lemma 2.1.

□

Lemma 2.3 (Kinderen)**.** *Gegeven twee filesystemen A en B laat p een relevant pad zijn in (A, B) zodat $A(p) = B(p) = DIR$ en laat $P = children_{A,B}(p)$. Dan impliceert $synch\,/_P(C, D, A, B)$ dat $synch\,/_p(C, D, A, B)$.*

Bewijs. We veronderstellen:

$$synch/_P(C, D, A, B) \tag{2.6}$$

en tonen aan dat

1. (C, D) een synchronisatie is van (A, B) na p, en
2. voor alle q, $p \not\leq q$ impliceert $C(q) = A(q) \wedge D(q) = B(q)$.

Dan volgt $synch\,/_p(C, D, A, B)$ uit (1) en (2).

- Om (2) aan te tonen merken we op dat $p \not\leq q$ impliceert dat $p.x \not\leq q$ voor alle x Dan hebben we door (2.6) $C(q) = A(q)$ en $D(q) = B(q)$
- Om (1) aan te tonen, tonen we aan dat condities (2.2) tot (2.5) van defintie 2.8 voldaan zijn voor elk relevant pad van de vorm $p.q$. Om deze condities aan te tonen beschouwen we twee subgevallen voor elke conditie, afhangend van het feit of $q = \epsilon$ of niet.
  - Veronderstel dat
    $$A(p.q) = B(p.q). \tag{2.7}$$
    * Veronderstel verder dat $q = \epsilon$.
      Sinds $p.x \not\leq p$ voor alle $p.x \in P$ hebben we $C(p) = A(p)$ en $D(p) = B(p)$ door (2.6), m.a.w. $C(p.q) = A(p.q)$ en $D(p.q) = B(p.q)$.
    * Veronderstel in plaats daarvan dat $q = x.r$ voor bepaalde x en r, en laat $p' = p.x$.
      Dan $p' \in P$ en zo is $p'.r$ relevant in (A, B). Ook volgt $A(p'.r) = B(p'.r)$ triviaal uit 2.7. Maar door 2.6 krijgen we $synch\,/_{p'}(C, D, A, B)$. Zodat we door (2.2) $C(p'.r) = A(p'.r)$ en $D(p'.r) = B(p'.r)$ krijgen, we krijgen m.a.w. $C(p.q) = A(p.q)$ en $D(p.q) = B(p.q)$.

    Dus voor $p.q$ geldt (2.2) voor elke q.
  - Veronderstel
    $$\neg dirty_A(p.q). \tag{2.8}$$
    * Veronderstel verder dat $q = \epsilon$.
      Sinds $p.x \not\leq p$ voor elke $p.x \in P$ hebben we door 2.6 $C(p) = A(p)$ en $D(p) = B(p)$. Maar door aanname geldt $A(p) = B(p) = DIR$. Zodat
      $$C(p) = D(p) = B(p). \tag{2.9}$$
      Aangezien elk dirtiness predicaat opwaarts gesloten is geldt er door aanname 2.8 $\neg dirty_A(p.x)$ voor alle $p.x$. Maar door 2.6 geldt er voor alle paden $p.x$ $synch\,/_{p.x}(C, D, A, B)$. Zodat we door (2.3) het volgende krijgen:
      $$C/p.x = D/p.x = B/p.x, \text{ voor alle paden } p.x \in P. \tag{2.10}$$

2.9 en 2.10 geeft ons nu: $C/p = D/p = B/p$, m.a.w. $C/p.q = D/p.q = B/p.q$

* Veronderstel nu het geval dat $q = x.r$ voor bepaalde x en r, en laat $p' = p.x$. Dan is $p' \in P$ en dus ook $p'.r$ relevant in (A, B). Dan volgt $dirty_A(p'.r)$ triviaal uit (2.8). Maar door (2.6) hebben we $synch\,/_{p'}(C,D,A,B)$. Zodat we door (2.3) $C/p'.x = D/p'.r = B/p'.r$ hebben, m.a.w. $C/p.q = D/p.q = B/p.q$.

Dus geldt (2.3) voor $p.q$ voor alle q.

– Veronderstel dat:

$$\neg dirty_B(p.q). \tag{2.11}$$

Dan geldt voor alle $p.q$ voor alle q (2.4), het argument hiervoor is gelijkaardig aan dat voor (2.3).

– Veronderstel:

$$dirty_A(p.q) \wedge dirty_B(p.q) \wedge A(p.q) \neq B(p.q) \tag{2.12}$$

* Veronderstel verder dat $q = \epsilon$.
Maar $A(p) \neq B(p)$ is in contradictie met de aanname $A(p) = B(p) = DIR$. Dat q niet kan leeg zijn.
* Veronderstel in plaats daarvan dat $q = x.r$ voor bepaalde x en r en laat $p' = p.x$.
Dan volgen $dirty_A(p'.r) \wedge dirty_B(p'.r) \wedge A(p'.r)$ triviaal uit de aanname. Maar er geldt door 2.6 dat $synch\,/_{p'}(C, D, A, B)$. Zodat we door 2.5 $C/p'.r = A/p'.r$ en $D(p'.r) = B(p'.r)$ hebben, m.a.w. $C/p.q = A/p.q$ en $D/p.q = B/p.q$.

Dus houdt (2.5) voor alle $p.q$ voor alle q.

□

We stellen nu dat het algoritme correct t.o.v. de vereiste specificaties.

Stelling 2.3. *Laat A, B en O filesystemen zijn en veronderstel dat $dirty_A$ en $dirty_B$ de updates voorstellen van O naar A respectievelijk B. Dan impliceert $recon(A, B, p) = (C, D)$ dat $synch\,/_p(C, D, A, B)$ voor elk relevant pad p in (A, B).*

Bewijs. Door inductie op (max(depth(A),depth(B))-$|p|$) toe toepassen zien we dat (C, D) een synchronisatie is van (A, B) na p. Dan krijgen we door Definitie 2.9 $synch\,/_P(C, D, A, B)$.

We moeten voor elk geval van het algoritme één geval beschouwen. In elk geval zullen we aantonen dat (C, D) een synchronisatie is van (A, B) na p.

- **Geval 1:** $\neg dirty_A(p) \wedge \neg dirty_B(p)$
  Dan is met algoritme *recon* $(C, D) = (A, B)$. We kunnen gemakkelijk verifiëren dat (2.2) tot (2.5) waar zijn:

  – Door Definitie 2.3 weten we dat $A(p.q) = B(p.q)$ voor alle q. De veronderstelling van gevallen (2.2), (2.3) en (2.4) in Definitie 2.8 gelden nu voor q, en de concluderende conclusies gelden triviaal.

  – De veronderstellingen van geval (2.5) gelden niet voor $p.q$ voor elke q.

  Dus (C, D) is een synchronisatie van (A, B) na p.

- **Geval 2:** $(dirty_A(p) \vee dirty_B(p)) \wedge A(p) = B(p) = DIR$
  Dan geldt bij algoritme *recon* $(C, D) = (A_n, B_n)$, waar als $p_1, p_2, \cdots, p_n$ de lexicografische opsomming is van $children_{A,B}(p)$ dan is (A_n, B_n) gedetermineerd door de volgende verzameling van vergelijkingen:

  $$(A_0, B_0) = (A, B)$$
  $$(A_{i+1}, B_{i+1}) = recon(A_i, B_i, p_{i+1})$$

  Voor elke $i < n$ krijgen we door inductie dat (A_{i+1}, B_{i+1}) een synchronisatie is van (A_i, B_i) na p_{i+1}. Zodat voor elke $i < n$ door Definitie 2.9 geldt $synch\,/_{p_{i+1}}(A_{i+1}, B_{i+1}, A_i, B_i)$. Dan krijgen we volgens lemma 2.2 $synch\,/_{children_{A,B}(p)}(A_n, B_n, A, B)$. Uiteindelijk krijgen we dan door Lemma 2.3 $synch\,/_p(A_n, B_n, A, B)$ zoals gevraagd.

- **Geval 3:** $((dirty_A(p) \vee dirty_B(p)) \wedge A(p) \neq B(p) \neq DIR) \wedge \neg dirty_A(p)$, m.a.w. $\neg dirty_A(p) \wedge dirty_B(p) \wedge (A(p) \neq DIR \vee B(p) \neq DIR)$
  Dan is met algoritme *recon* $(C, D) = (A[p \leftarrow B/p], B)$.
  We controleren de gevallen van Definitie 2.8 expliciet voor elk willekeurig relevant pad $p.q$. Er geldt duidelijk:

  $$D/p.q = B/p.q. \tag{2.13}$$

  En door Definitie 2.7 geldt er:

  $$C/p.q = B/p.q \tag{2.14}$$

  Aangezien elk dirtiness pedicaat opwaarts gesloten is en we $dirty_A$ hebben geldt er ook:

  $$\neg dirty_A(p.q). \tag{2.15}$$

  Gegeven de bovenstaande observaties tonen we aan dat condities (2.2) tot (2.5) van Definitie 2.8 gelden voor $p.q$:

  - Nu als (2.2) geldt voor $p.q$, m.a.w. als $A(p.q) = B(p.q)$ dan geldt door (2.14): $C(p.q) = A(p.q)$. Parallel met 2.13 verzekert dit dat conclusie (2.2) geldt voor $p.q$ zoals gevraagd.
  - Observatie (2.15) impliceert dat (2.3) geldt voor $p.q$. Maar observaties (2.13) en (2.14) geven $C/p.q = D/p.q = B/p.q$ zoals vereist.
  - Vervolgens veronderstellen we dat geval (2.4) geldt voor $p.q$. Dan geldt voor elke r met $q \leq r$ $\neg dirty_B(p.r)$. En bij Definitie 2.3 $A(p.r) = B(p.r)$, en zo is $A/p = B/p$ waar. Dit samen met (2.13) en (2.14) geeft ons $C/p.q = D/p.q = A/p.q$, zoals vereist.
  - Uiteindelijk, geval (2.5) geldt niet voor $p.q$ voor elke q, en aan (2.5) zelf is triviaal voldaan.

- **Geval 4:** $dirty_A\, p \wedge \neg dirty_B(p) \wedge (A(p) \neq DIR \vee B(p) \neq DIR)$
  Het bewijs hiervoor is symmetrisch aan het voorgaande geval.

- **Geval 5:** $dirty_A\, p \wedge dirty_B(p) \wedge (A(p) \neq DIR \vee B(p) \neq DIR)$
  Dan is met algoritme *recon* $(C, D) = (A, B)$. Als $q \neq \epsilon$ dan is $p.q$ geen relevant pad. Zodat het enige relevante pad dat beschouwd moet worden p zelf is en het is gemakkelijk te verifiëren dat condities (2.2) tot (2.5) gelden voor p:

  - Als (2.2) geldt voor p dan hebben we, aangezien $(C, D) = (A, B)$, $C(p) = A(p)$ en $D(p) = B(p)$, zoals vereist.
  - Zowel (2.3) als (2.4) gelden niet voor p.
  - Als (2.5) geldt voor p dan hebben we, aangezien $(C, D) = (A, B)$, $C/p = A/p$ en $D/p = B/p$, zoals vereist.

□

Stelling 2.4. *Rekeninghoudend met de vereiste specificaties is algoritme recon correct en compleet, m.a.w. als* $(C, D) = recon(A, B, \epsilon)$ *dan is* (C, D) *een unieke synchronisatie van* A *en* B*.*

Bewijs. Dit volgt uit Stellingen 2.1 en 2.3. □

2.5 File synchronizers in de praktijk

We zullen nu enkele voorbeelden geven van bestaande synchronizers en deze testen aan onze specificaties en voorbeeldjes uit Sectie 2.4. Het is niet nodig

van de file synchronizers ook nog op andere voorbeeldjes te testen, aangezien de voorbeeldjes alle mogelijke gevallen behandelen. Uiteraard zullen we niet alle synchronisers kunnen bespreken, we pikken er slechts enkele uit.

2.5.1 Unison

Unison, (Pierce 04) en (Pierce N), is een open-source file synchronizer voor Unix-achtigen en Windows. Unison is ontwikkeld in oCaml en valt onder de GPL-licentie. Het is ontwikkeld in het midden van de jaren 90 aan de engineering school van de universiteti van Pennsylvania. Unison was de eerste wijd gebruikte two-way filesynchronisatie tool en is gegroeid tot een volwassen tool met een vrij grote gebruikersgemeenschap en hoge ambities van portabiliteit, robuustheid en vloeiende operaties over verschillende besturingssystemen en filesysteem architecturen. Een ongebruikelijk kenmerk van Unison's geschiedenis is dat de ontwikkeling van het systeem parallel gebeurd is met een serieuze inspanning om zijn gedrag mathematisch te specificiëren. Unison is een state-based synchronizer.

We hebben Unison getest op onze voorbeeldje uit Sectie 2.4. Het gedrag van Unison was in onze test identiek aan onze specificaties in Sectie 2.4, dit was te verwachten aangezien Unison gebaseerd is op de specificaties beschreven in (Balasubramaniam 98) en ook wij ons gebaseerd hebben op deze paper voor onze specificaties op te stellen.

Bij de voorbeelden die leiden tot twee inconsistente kan de gebruiker d.m.v. het programma er handmatig wel voor zorgen dat de conflicten opgelost worden.

2.5.2 JFileSync

JFileSync is een open-source file synchronizer ontwikkeld onder de GPL-licentie. JFileSynch is ontwikkeld in Java en draait bijgevolg op elk besturingssyteem dat over een Java Virtual Machine beschikt.

Ook JFileSync (Heidrich 07) hebben we getest op onze voorbeeldjes uit Sectie 2.4. Voorbeeldjes één, twee en drie verlopen zoals beschreven in Sectie 2.4, de andere voorbeeldjes niet. In voorbeeldje vier krijgen beide replica's na synchronisatie de files a (met inhoud f') en b (met inhoud g'). En in voorbeeldje vijf krijgen beide replica's na synchronisatie de file a (met inhoud f'), de directory d wordt hierbij dus terug aangemaakt in de eerste replica. We zien dus dat JFileSync de gevallen waarbij we volgens onze specificatie twee inconsistente staten moeten krijgen verkeerd afhandelt, JFileSync zal de replica's altijd leiden tot twee consistente staten.

2.5.3 FullSync

Ook FullSync (Kopcsek 05) is een open-source file synchronizer ontwikkeld in Java ditmaal onder de GPL2-licentie. En bijgevolg ook een cross-platform synchronizer.

Ook FullSync hebben we getest op onze voorbeeldjes uit Sectie 2.4. Bij FullSync verloopt alleen het eerste voorbeeldje zoals beschreven. Bij het tweede voorbeeldje loopt het al mis, na synchronisatie bevatten beide replica's files a (met inhoud f), b (met inhoud g) en c (met inhoud h). FullSync heeft dus duidelijk een probleem met het propageren van de verwijdering van files, een bijkomende test, waarbij we in de ene replica een file verwijderen en in de andere replica niks wijzigen, door ons uitgevoerd heeft dit bevestigd. Bij voorbeeldje drie krijgen we in beide replica's de files a (met inhoud f), b (met inhoud g) en c (met inhoud f). Ook hier loopt het dus weer mis bij het verwijderen, net zoals bij het hernoemen (wat eigenlijk ook een verwijdering is). In voorbeeldje vier krijgen we twee replica's met elk files a (met inhoud f') en b (met inhoud g'). Ook hier zien we weer dat het verwijderen misloopt. Ook voorbeeldje vijf verloopt niet zoals door ons beschreven. Hierbij bevatten beide replica's na synchronisatie een file a (met inhoud f') en b (met inhoud g'). We zien bij de twee laatste voorbeeldjes wederom dat ook deze synchronizer beide replica's naar twee consistente staten brengt.

2.5.4 Microsoft Office Groove

Microsoft Office Groove (Microsoft 08) is een filesynchronisatie tool meegeleverd bij Microsoft Office.

Groove geeft enkel bij voorbeeldje vijf uit Sectie 2.4 een afwijkend gedrag. Hierbij krijgen we twee consistente staten waarbij C enkel de rootdirectory met een leeg pad d bevat, en D de rootdirectory bevat met subdirectory d, die enkel de file a (met inhoud f') bevat. Deze afwijking is waarschijnlijk te wijten aan een iets andere specificatie. Herinner dat we voor het probleem in voorbeeld vijf ons specificatie een beetje hebben moeten verfijnen, we hebben namelijk het begrip relevant pad ingevoerd. Waarschijnlijk werkt Groove niet met relevante paden, en hebben ze voor geval vier in Definitie 2.4 volgende regel: $C(p) = A(p) \wedge D(p) = B(p)$. M.a.w. C en D moeten enkel in p gelijk zijn aan respectievelijk A en B, en niet in de volledige deelboom in p zoals in onze specificatie.

Wanneer de replica's leiden tot twee inconsistente staten kunnen de conflicten handmatig opgelost worden met het programma.

2.5.5 Microsoft SyncToy

Microsoft SyncToy, (Microsoft 06) en (N.N. N), is een file synchronizer ontwikkeld in het .NET Framework en is vrij te verkrijgen.

De eerste drie voorbeeldjes uit Sectie 2.4 verlopen zoals beschreven. Het vierde voorbeeldje verloopt echter anders, hier bevatten beide replica's een file a (met inhoud f') en een file b (met inhoud g'). Bij het vijfde voorbeeldje krijgen we in beide replica's een subdirectory d met file a (met inhoud f'). We zien dus ook bij deze synchronizer dat het fout loopt wanneer we volgens onze specificatie twee inconsistente staten zouden moeten krijgen. SyncToy kiest er blijkbaar voor altijd tot twee consistente staten te leiden.

2.5.6 GoodSync

GoodSync (Systems 08) is een commerciële file synchronizer ontwikkeld door Siber Systems.

GoodSync geeft enkel bij voorbeeldje vijf uit Sectie 2.4 een afwijkend gedrag. GoodSync heeft exact hetzelfde gedrag als Microsoft Office Groove en gebruikt bijgevolg waarschijnlijk ook dezelfde specificatie.

Het is bij GoodSync mogelijk om d.m.v. van gebruikersinteractie de inconsistente staten in de laatste twee voorbeeldjes op te lossen.

2.5.7 Allway Sync

Allway Sync (Lab 08) is een synchronizer ontwikkeld door Usov Lab. De ontwikkeling is gestart in 2004 en loopt nog steeds door.

Ook deze synchronizer hebben we getest op onze voorbeeldjes uit Sectie 2.4. De eerste drie voorbeeldjes verlopen hierbij zoals door ons beschreven. Voorbeelden vier en vijf geven een afwijkend gedrag. Bij voorbeeldje vier krijgen we in elke replica een file a (met inhoud f') en een file b met inhoud (g'). In voorbeeldje vijf krijgen we in elke replica een directory met file a (met inhoud f'). We zien dus ook hier weer dat de synchronizer ervoor kiest altijd te evolueren naar twee consistente staten en de gevallen waarbij er volgens onze specificatie twee inconsistente staten zijn anders afhandelt dan door ons beschreven.

2.5.8 SyncBack

SyncBack (2brightsparks 08) is commerciële filesynchronizer ontwikkeld door 2brightsparks. De oudere versie is echter wel gratis te verkrijgen, dit is dan ook de versie die wij getest hebben.

Enkel het eerste voorbeeldje uit Sectie 2.4 verloopt zoals we beschreven hebben. De andere voorbeeldjes hebben een afwijkend gedrag. Bij voorbeeldje twee krijgen we in beide replica's files a (met inhoud f), b (met inhoud g) en c (met inhoud h). We zien dus dat het deleten van files niet correct verloopt. In voorbeeldje drie krijgen we in beide replica's files a (met inhoud f), b (met inhoud g) en c (met inhoud f). Ook hier verloopt het deleten niet correct, net als het hernoemen (wat eigenlijk een vorm van deleten is). Bij voorbeeldje vier krijgen we in beide replica's files a (met inhoud f') en b (met inhoud g'). Het deleten verloopt dus weer niet correct, bij het verzoenen wordt er geen rekening gehouden met het feit dat in replica b file a verwijdert is. Ook in voorbeeldje vijf hebben we problemen met het deleten, we krijgen hier na synchronisatie in beide replica's de files a (met inhoud f') en b (met inhoud g).

2.5.9 BestSync

BestSync (RiseFly 08) is een gratis verkrijgbare file synchronizer ontwikkeld door Risefly.

Alvorens we de synchronisatietaak konden starten moesten we eerst in de setup kiezen wat het programma moet doen wanneer een file in de ene replica verwijdert wordt. We hebben de keuze tussen: niks doen, het verwijderen van de resterende file in de andere replica, het kopiëren naar de andere replica en het verwijderen in de target folder wanneer het niet in de source folder zit. We kiezen voor de tweede keuze, het verwijderen van de resterende file in de andere replica. Bijgevolg krijgen we in de eerste drie voorbeeldjes uit Sectie 2.4 het resultaat zoals beschreven, maar door de keuze die we juist gemaakt hebben krijgen we in in de laatste twee voorbeeldjes niet het beschreven resultaat. Bij het vierde voorbeeldje krijgen we in beide replica's de file b (met inhoud g'). En in het laatste voorbeeldje wordt in de andere replica de subdirectory d ook volledig verwijdert.

2.5.10 Conclusie

We hebben gezien dat slechts één van de negen file synchronizers volledig aan onze specificatie voldoet, nl. Unison. Twee andere file synchronizers, Microsoft Office Groove en Goodsync, hebben slechts in één voorbeeldje een afwijkend gedrag, dit is waarschijlijk te wijten aan een afwijkende specificatie. We hebben ook gezien dat twee file synchronizers, FullSync en SyncBack, een probleem hadden bij het verwijderen en hernoemen van files. Ook bij BestSync gaf dit problemen, BestSync weet uit zijn eigen niet wat hij moet doet wanneer een file in een replica verwijderd is. De gebruiker moet tijdens

de setup van een nieuw project expliciet aangeven wat BestSync in dat geval moet doen. Verder leiden alle file synchronizers behalve Unison, Microsoft Office Groove en Goodsync de replica's automatisch naar twee consistente staten, wat volgens onze specificaties niet in alle gevallen correct is.

2.6 Uitbreidingen

Afsluitend schetsen we enkel uitbreidingen aan ons framework.

2.6.1 Gedeeltelijk succesvolle synchronisatie

Zoals we eerder hebben gezien zullen we bij conflicterende updates die niet door de gebruiker worden opgelost twee inconsistente filesystemen verkrijgen. Bijgevolg hebben we de volgende keer dat we willen synchroniseren niet één maar twee originele filesystemen. Het kan hierbij echter wel voorkomen dat delen van het filesysteem wel correct gesynchroniseerd zijn. Om dit geval af te kunnen handelen moeten we onze specificaties verfijnen, gelukkig kan dit in ons model eenvoudig.

In plaats van dat de replica's na de vorige synchronisatie één gemeenschappelijke toestand O hebben introduceren we in ons filesysteem een nieuw filesysteem Γ. Γ stelt de inhoud voor van elk pad p op het laatste tijdstip waarop p succesvol gesynchroniseerd was.

Enkel de betekenis van het *dirty* predicaat wordt in onze specificaties van de update detector nu lichtjes gewijzigd, $dirty_S(p)$ is *true* wanneer p in S verwijst naar iets verschillend van waarnaar het verwees na de laatste succesvolle synchronisatie.

De verzoener breiden we uit met een nieuw predicaat *succes*. Dit nieuwe predicaat geeft aan welke paden succesvol gesynchroniseerd zijn. Formeel zeggen we dat $(C, D, succes)$ een synchronisatie is van een paar van originele filesystemen (A, B), met respect voor de dirtiness predicaten $dirty_A$ en $dirty_B$ als voor elk relevant pad p in (A, B) het volgende geldt:

1. $A(p) = B(p) \Rightarrow C(p) = A(p) \wedge D(p) = B(p) \wedge succes(p)$
2. $\neg\, dirty_A(p) \Rightarrow C/p = D/p = B/p \wedge succes(p)$
3. $\neg\, dirty_B(p) \Rightarrow C/p = D/p = A/p \wedge succes(p)$
4. $dirty_A(p) \wedge dirty_B(p) \wedge A(p) \neq B(p) \Rightarrow C/p = A/p \wedge D/p = B/p \wedge \neg\, succes(p)$

Het *succes* predicaat wordt berekend om na elke synchronisatiestap de "laatste consistente toestand " te berekenen. Voor elk pad p geldt er:

$$\Gamma'(p) = \begin{cases} C(p) & \text{als } succes(p) \\ \Gamma(p) & \text{anders} \end{cases}$$

Het filesysteem Γ' wordt voor de volgende synchronisatieronde gebruikt als Γ door de update detector.

2.6.2 Meerdere replica's

Tot nu toe zijn we er in onze uitleg altijd vanuit gegaan dat er maar twee replica's zijn. In de praktijk komt het uiteraard regelmatig voor dat er meerdere replica's zijn. We zullen onze specificaties nu veralgemenen zodat er meer dan twee replica's kunnen gebruikt worden.

We zwakken eerst het feit dat dirtiness predicaten opwaarts gesloten zijn af. We schrijven dat $dirty\,@_S(p)$ betekent dat S exact op p geüpdatet is niet ergens onder p. We zeggen dat een predicaat *dirty* de updates berekent van O naar S als $\neg\, dirty\, @S(p)$ impliceert dat $O(p) = S(p)$ voor alle paden p.

Laat nu $Id = \{1, 2, \cdots, n\}$ de verzameling tags zijn die de n replica's voorstellen die we willen synchroniseren. Laat $\mathcal{F}_S = \{S_i | i \in Id\}$ de verzameling van originele replica's zijn die we willen synchroniseren. Voor elk pad p, laat $D_{p,S}$ de verzameling van identiteiten zijn van replica's die dirty zijn, m.a.w. $D_{p,S} = \{i |\, dirty\, @_{S_i}(p)\}$. Er wordt gezegd dat een verzameling van replica's $\mathcal{F}_R = \{R_i | i \in Id\}$ een synchronisatie van $\mathcal{F}_S$ is, met respect voor de dirtiness predicaten $dirty\, @_S$, als voor elk relevant pad p in $\mathcal{F}_S$ de volgende condities voldaan zijn:

$$\begin{aligned} \forall i, j \in Id.\ S_i(p) = S_j(p) &\Rightarrow \forall i \in Id.\ R_i(p) = S_i(p) \\ D_{p,S} \neq \emptyset \wedge \forall i, j \in D_{p,S}.\ S_i(p) = S_j(p) &\Rightarrow \forall i \in Id.\ R_i(p) = S_j(p)\ voor \\ &\quad bepaalde\ j \in D_{p,S} \\ \exists i, j \in D_{p,S}.\ S_i(p) \neq S_j(p) &\Rightarrow \forall i \in Id.\ R_i(p) = S_i(p). \end{aligned}$$

De verzoeningsstrategie van Coda (Kistler 96) en (Kumar 94) hangt af van een gelijkaardige vereiste als de bovenstaande.

Meer informatie over hoe meerdere replica's afgehandeld kunnen worden kan gevonden worden in (Greenwald 06).

2.6.3 Synchroniseren binnenin files

Uiteraard is er veel vraag naar synchronizers die bij een conflict ook binnenin de files synchroniseren. We spreken dan zoals reeds in de Inleiding aangehaald

van *datasynchronisatie*. Zoals we in deel II van deze masterproef zullen zien, zijn er verschillende soorten datasynchronizers elk geschikt voor bepaalde types bestanden. Nu is het mogelijk om, wanneer er zich een conflict voordoet omdat eenzelfde bestand in twee replica's is aangepast, te kijken over welk type bestand het gaat, en verder een geschikte datasynchronizer te gebruiken.

2.6.4 Aanpak van metadata

Met metadata bedoelen we de eigenschappen van files buiten hun gewone inhoud, zoals file permissies, gebruiker id's, groep id's en aanpassingstijden, als ook meer deskundige eigenschappen zoals "spelling" van filenamen op systemen (zoals Windows) waar er geen hoofdlettergevoeligheid is voor filenamen bij het lezen en schrijven van files, maar wel bij het tonen van een lijst met de inhoud van een directory. Er zijn nu twee aangewezen manieren om te denken over de metadata die geassocieerd is met een file of directory: we kunnen ofwel metadata simpelweg zien als deel van de inhoud van de files, of we kunnen metadata beschouwen als afgescheiden en onafhankelijk van de inhoud. Beide manieren leiden tot een verschillende kijk op hoe een synchronizer veranderingen moet propageren, in het bijzonder leidt het tot verschillende definities van conflicten. Als we er voor kiezen dat de metadata een deel van de inhoud is zal bijvoorbeeld het veranderen van een file's inhoud in de ene replica en het veranderen van de schrijfrechten in de andere replica leiden tot een conflict. Net zoals we een conflict hebben wanneer de inhoud in beide replica's aangepast wordt. Als de metadata echter volledig gescheiden is van de inhoud van de file dan verwachten we in het bovenstaande scenario dat zowel de verandering van de inhoud als de verandering van de permissies gepropageerd worden in de andere replica.

Sommige synchronizers, zoals Unison, kiezen voor de middenweg tussen deze twee aanpakken, ze combineren de virtuositeit van deze twee extreme kijken. Deze middenweg houdt het gedrag van het programma gemakkelijk te verstaan, terwijl het onechte conflicten voorkomt. We bekijken een file's inhoud en zijn metadata als een *atomaire* eenheid met als doel het propageren van veranderingen, maar niet als doel het definiëren van conflicten. We vereisen dus dat elke file zijn inhoud en al zijn metadata na synchronisatie gelijk is aan de toestand van één van de replica's voor de synchronisatie. Hoe dan ook als een deel van de metadata aangepast is op een identieke manier in beide replica's en de inhoud maar aangepast is in één replica, wordt er geen conlict geregistreerd en is het de synchronizer toegelaten de inhoud van de update te propageren.

De specificaties van Sectie 2.2 zijn gemakkelijk te verfijnen om dit gedrag toe te laten. We definiëren een atomair predicaat *atomic* op klasse $\mathcal{S}$.

Definitie 2.14. *Nemen we nu het geval dat we twee replica's A en B hebben die uit een filesysteem O voortkomen dan zeggen we dat (lokaal) atomiciteit gerespecteerd is als voor elk relevant pad p in A en elk relevant pad q in B geldt dat*

$$atomic(C) \Longrightarrow (C/p = A/p) \vee (C/p = B/p) \tag{2.16}$$
$$atomic(D) \Longrightarrow (D/q = B/q) \vee (D/q = A/q) \tag{2.17}$$

Merk op dat aan deze eigenschap altijd voldaan is als het atomiciteits predicaat constant false is. Zodat de originele specificatie inderdaad een speciaal geval is van de verfijnde.

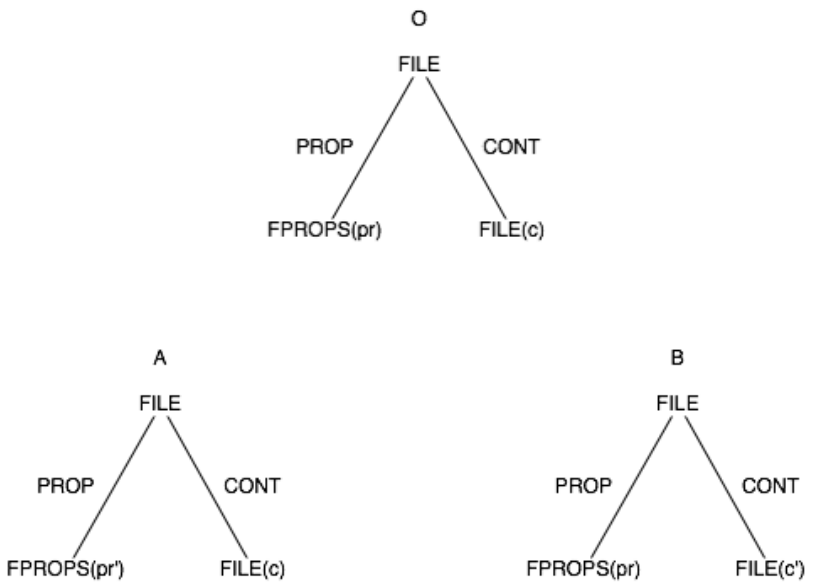
We concretiseren de verfijnde specificatie door een verzameling eigenschappen te introduceren van file/directory metadata. De verzameling (S) van klassen is gedefinieerd als volgt:

$$\begin{aligned} S = \quad & DIR, FILE \\ \cup \quad & FILECONTENT(c) \mid c \in contents \\ \cup \quad & FPROPS(pr) \mid pr \in properties \\ \cup \quad & DPROPS(pr) \mid pr \in properties \end{aligned} \tag{2.18}$$

We definiëren het predicaat *atomic* door *atomic*(FILE) = true en atomic(s) = false voor alle andere klassen s. Dit verzekert dat veranderingen aan de inhoud van een file en zijn geassocieerde metadata automatisch behandeld kunnen worden, terwijl veranderingen aan directory metadata onafhankelijk behandelt worden van veranderingen aan de directory zijn kinderen. Uiteindelijk definiëren we de verzameling van namen $\mathcal{N}$ als de verzameling filenamen plus twee speciale labels, CONTENTS en PROPS:

$$\mathcal{N} = names \cup \{CONTENTS, PROPS\}. \tag{2.19}$$

We geven nu een klein voorbeeldje waarbij de nieuwe specificatie verschilt van de oude. We beschouwen het originele filesysteem O en de hieruit afgeleide filesystemen A en B. In A is enkel de metadata aangepast van de file, waar in B enkel de inhoud van de file aangepast is. Dit is weergeven in Figuur 2.9 Als we nu de specificaties uit Definitie 2.6 zouden toepassen zouden zowel C en D bestaan uit een file met de inhoud uit B en de metadata uit A. Volgens de in deze sectie ingevoerde specificatie is dit echter niet correct aangezien de files als atomaire eenheden beschouwd worden en elke file zijn inhoud en al zijn metadata na synchronisatie gelijk moet zijn aan de toestand van één van de replica's voor de synchronisatie. Nu mag de file in D na synchronisatie niet volledig gelijk zijn aan die in A aangezien de verandering van de inhoud zo ongedaan gemaakt wordt. De file in C mag ook niet



Figuur 2.9: Voorbeeld van een origineel filesysteem O en twee daarvan afgeleide filesystemen A en B. In A is enkel de metadata van de file aangepast en in B enkel de inhoud. Volgens de in deze sectie gegeven specificatie leidt dit tot een conflict.

volledig gelijk zijn aan B aangezien dan de veranderingen in de metadata ongedaan gemaakt worden. We bekomen hier dus een conflictsituatie. Het voordeel van deze situatie is dan de gebruiker zelf kan kiezen wat er moet gebeuren. Zo is het mogelijk dat de inhoud in de file in B is aangepast met de metadata (met o.a. de leesrechten) van O in gedachten en het mogelijk niet gewenst is dat deze metadata zomaar zou veranderd worden zoals volgens 2.6 zou moeten gebeuren.

Deel II

Data synchronisatie

Hoofdstuk 3

DIFF3

In dit hoofdstuk bespreken we diff3, een datasynchronisatie algoritme dat gezien wordt als dé standaard voor het synchronizeren van ongecoördineerde veranderingen aan lijst-gestructueerde data zoals bijvoorbeeld tekstbestanden. Er wordt dieper ingegaan op enkele eigenschappen van het diff3 algoritme die onnatuurlijk aanvoelen. Het toepassen van het diff3 algoritme vergt de toepassing van een eerder ontwikkeld algoritme, dat ook uitgebreid uitgelegd wordt.

3.1 Inleiding

Diff3 is de best gekende tool voor synchronisatie van textuele data. Diff3 is in 1988 ontwikkeld door Randy Sith en is gepopulariseerd in versiecontrolesystemen zoals CVS en Subversion. Diff3 en zijn verwanten worden dagelijks door miljoenen gebruikers, o.a. softwareontwikkelaars gebruikt voor collaboratieve taken. Het basisidee van diff3 komt ook terug in verschillende tools voor het synchronizeren van bestandsstructuren zoals XML, zoals we zullen zien in Hoofdstuk 5.

Gezien de populariteit van diff3 is het verrassend dat de fundamentele eigenschappen van diff3 pas in 2007 in (Khanna 07) bestudeerd zijn. De inhoud van dat werk wordt in dit hoofdstuk beschreven.

In Sectie 3.2 tonen we een eerste keer de werking van het diff3 algoritme aan de hand van een voorbeeld. In 3.3 geven we het algoritme en enkele definities die we hiervoor nodig hebben. In Sectie 3.4 gaan we na of enkele eigenschappen in verband met het diff3 algoritme die intuïtief aanvoelen ook echt gelden. In sectie 3.5 vertellen we tenslotte wat een langst gemeenschappelijke subsequentie is en hoe we hieraan komen.

3.2 Diff3 by example

Diff3 kan bekeken worden als een algoritme dat als invoer tripels van de vorm (A,O,B) (met A, O en B lijsten) aanneemt en trippels (A', O', B') teruggeeft. Hierbij is O het archief (de laatste gemeenschappelijke versie) en A en B twee aangepaste versies van het archief.

Figuur 3.1(a) toont een voorbeeld. Hierbij zijn in A elementen 4,5 en 2,3 verwisseld en is 3 achter 5 geplaatst in B.

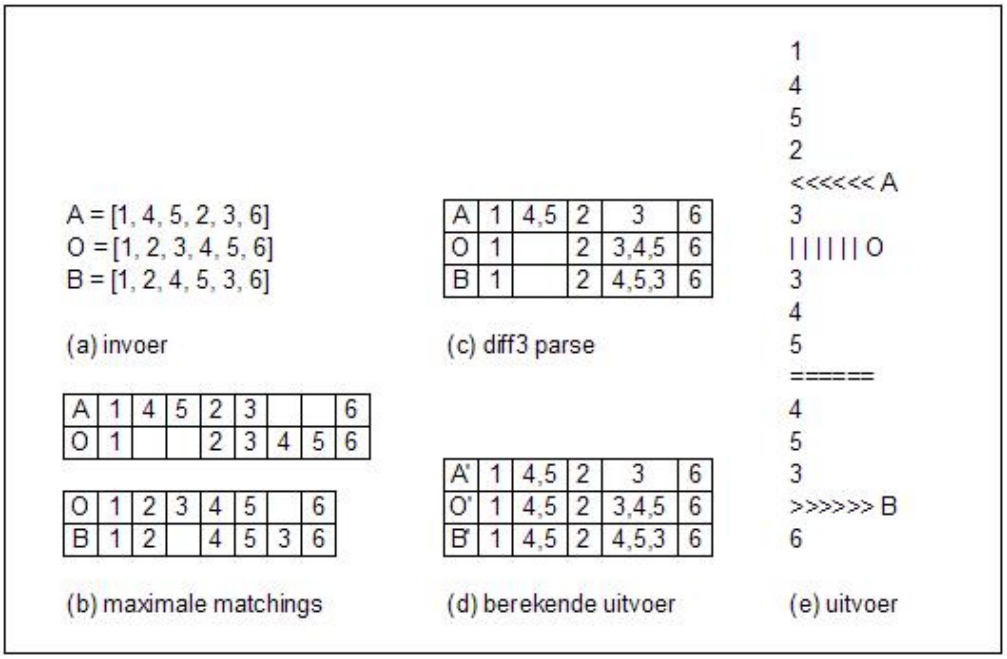
1. Op deze invoer start Diff3 dan met het aanroepen van de two-way vergelijkingstool diff om de *maximale matchings* (of langste gemeenschappelijke subsequenties) tussen O en A en tussen O en B te vinden zoals in Figuur 3.1(b). Intuïtief gezien aligneert zo'n maximale matching delen van O met overeenkomstige delen van A (respectievelijk B) op zo'n manier dat het aantal gedeelde elementen maximaal is.(Meer hierover in Sectie 3.5.)

2. Daarna splitst het algoritme de regio's in overlappende en verschillende regio's. Dit leidt tot de afwisselende sequenties van *stabiele* (alle replica's gelijk) en *onstabiele* (één of beide replica's aangepast) *blokken* getoond in Figuur 3.1(c). Hierbij zijn de eerste, derde en vijfde blok overlappende regio's. De tweede en vierde zijn onstabiele regio's waar A en/of B aangepast zijn.

3. Uiteindelijk onderzoekt het algoritme wat er veranderd is in elk blok en beslist het welke veranderingen er kunnen gepropageerd worden, zoals getoond in Figuur 3.1(d). In ons voorbeeld is het tweede blok enkel veranderd in A (door 4,5 te inserten). Deze verandering kan dus zonder problemen gepropageerd worden naar B. Het vierde blok is echter veranderd in zowel A en B. Bij dit blok kan er niks gepropageerd worden.

Figuur 3.1(e) geeft de ouput weer van het uitvoeren van de GNU diff3 tool met de flag -m, welke aangeeft dat we een gemergde versie willen hebben, en als parameters A, O en B. Voor de niet-conflicterende regio's is een enkele versie geprint, voor de conflicterende regio's wordt er weergegeven wat A, O en B bevatten.

In dit hoofdstuk wordt de uitvoer iets verfijnd, i.p.v. één versie die alle informatie bevat, wordt er van uitgegaan dat de uitvoer 3 versies bevat, A', B', O' zoals getoond in Figuur 3.1(d).



Figuur 3.1: Voorbeeld van de werking van diff3.

3.3 Het Diff3 algoritme

Alvorens het diff3 algoritme te kunnen geven, moeten we eerst nog enkele begrippen en notaties invoeren. Dit zal in deze sectie dan ook eerst gebeuren, waarna we het algoritme geven en verklaren.

3.3.1 Definities

We veronderstellen gegeven een verzameling $\mathcal{A}$ wiens elementen we *atomen* noemen. In de praktijk kunnen dit lijnen met tekst zijn (zoals in GNU diff3), of woorden, karakters, etc. We schrijven $\mathcal{A}^*$ voor de verzameling van lijsten met elementen in $\mathcal{A}$ en gebruiken variabelen J, K, L, O, A, B en C om elementen van $\mathcal{A}$ aan te duiden. Als L een lijst is dan duidt $|L|$ de lengte van L aan. Indien $k \in \{1, \ldots, |L|\}$, dan duidt $L[k]$ het k-de element aan van L aan. Een *span* in een lijst L is een paar van indices $[i..j]$ met $1 \leq i, j \leq |L|$. We schrijven $L[i..j]$ voor de lijst van elementen van L in locaties i t.e.m. j. Bijvoorbeeld, wanneer $L = 1, 4, 5, 2, 3, 6$ dan is $L[3..5] = 5, 2, 3$. $L[i..j]$ is de lege lijst wanneer $j < i$. De *lengte* van een span $[i..j]$ is $j - i + 1$ als $i \leq j$ en 0 als $i > j$.

Definitie 3.1 (Configuratie)**.** *Een configuratie is een triple (A, O, B)* $\in \mathcal{A}^* \times \mathcal{A}^* \times \mathcal{A}^*$*. We schrijven gewoonlijk configuraties in de meer suggestieve*

notatie ($A \leftarrow O \rightarrow B$) om te benadrukken dat O het archief is waaruit A en B afgeleid zijn.

In deze sectie is een synchronizer een functie die als input een configuratie ($A \leftarrow O \rightarrow B$) neemt en als output een configuratie ($A' \leftarrow O' \rightarrow B'$) teruggeeft. We zeggen dat ($A \leftarrow O \rightarrow B$) $\Rightarrow$ ($A' \leftarrow O' \rightarrow B'$) een *uitvoering* is van de synchronizer. Van een uitvoering ($A \leftarrow O \rightarrow B$) $\Rightarrow$ ($C \leftarrow C \rightarrow C$) waar de drie componenten van de output configuratie identiek zijn, wordt gezegd dat ze *conflictvrij* zijn. We schrijven in dat geval ($A \leftarrow O \rightarrow B$) $\Rightarrow$ C.

Zoals geïllustreerd in Sectie 3.2 is de eerste stap van diff3 het oproepen van een subroutine op (O,A) en (O,B) om een zogenaamde *niet-kruisende matching* M_A tussen de indices van O en A, en M_B tussen de indices O en B. Intuïtief gezien spreken we van een niet-kruisende matching wanneer we de lijsten onder elkaar kunnen plaatsen en de overenstemmende elementen met elkaar kunnen verbinden zonder kruisende bogen te verkrijgen.

Definitie 3.2. *Een niet-kruisende matching M_A tussen twee lijsten O en A is een booleaanse functie van paren van indices van O en A zodat er indien $M_A[i,j] = true$ geldt dat (a) $O[i] = A[j]$, (b) $M_A[i',j] = false$ en $M_A[i,j'] = false$ wanneer $i' \neq i$ en $j' \neq j$, en (c) $M_A[i',j'] = false$ wanneer er geldt $i' < i$ en $j' > j$ of $i' > i$ en $j' < j$. Een niet-kruisende matching tussen O en A is maximaal wanneer er geen andere niet-kruisende matching M'_A bestaat zodat $|\{(i,j)|M_A(i,j) = true\}| < |\{(i,j)|M'_A[i,j] = true\}|$.*

Voorwaarde (a) geeft aan dat het element met index i in O en index j in A (of B) dezelfde inhoud moeten hebben, anders kunnen we niet van een matching spreken. Voorwaarde (b) vertelt ons dat zowel i in O als j in A (of B) niet met andere elementen mogen matchen in de *maximale matching*. (Met andere woorden, M_A is de graaf van een partiële injectie tussen O en A.) Voorwaarde (c) eist dat er geen kruisende matchings voorkomen, dit komt voor wanneer A (of B) en O matchen in een element met een lagere index i' dan i en een index j' groter dan j of wanneer er geldt dat $i' > i$ en $j' < j$. In beide gevallen kruisen de bogen die i met j en i' met j' verbinden.(Met andere woorden M_A is de graaf van een strikt monotoon stijgende partiële functie van O naar A.) We behandelen dit algoritme momententeel als een black box, eenvoudig door te veronderstellen dat (a) het algoritme deterministisch is, en (b) het altijd *maximale* matchings voortbrengt. Later in dit hoofdstuk, in Sectie 3.5 zullen we dit algoritme nader bekijken. Voor de voorbeelden in de volgende sectie hebben de schrijvers van (Khanna 07) nagegaan dat de matchings die we gebruiken overeenstemmen met degene die gekozen zijn door GNU diff3.

Definitie 3.3 (Blok)**.** *Een blok (van A, O en B) is een triple* $H = ([a_i..a_j], [o_i..o_j], [b_i..b_j])$ *van een span in A, een span in O en een span in B zodat op zijn minst één van de drie niet leeg is. De grootte van een blok is de som van de lengten van de 3 spans. We schrijven* $A[H]$ *voor* $A[a_i..a_j] \in A^*$*, en gelijkaardig* $O[H] = O[o_i..o_j]$ *en* $B[H] = B[b_i..b_j]$*.*

Een *stabiele blok* is een blok waarbij de drie spans dezelfde lengte hebben en alle overeenstemmende elementen matchen in alle drie, bijvoorbeeld: een blok $([a..a+k-1], [o..o+k-1], [b..b+k-1])$ voor bepaalde $k > 0$, met $M_A[o+i, a+i] = M_B[o+i, b+i] = true$ voor elke $O \leq i < k$. Dit betekent dat een stabiele blok overeenstemt met een span in O die zowel in M_A en M_B matcht. Een *onstabiele blok* is een blok die niet stabiel is. Een onstabiele blok H is ingedeeld als volgt:

H is aangepast in A	als	$O[H] = B[H] \neq A[H]$
H is aangepast in B	als	$O[H] = A[H] \neq B[H]$
H is onecht in conflict	als	$O[H] \neq A[H] = B[H]$
H *is* (echt) in conflict	als	$O[H] \neq A[H] \neq B[H] \neq O[H]$

Een blok wordt in conflict genoemd als het echt of onecht in conflict is, een *niet-conflicterende blok* is dus een blok dat ofwel stabiel is ofwel enkel veranderd is in A of B. Gegeven een blok H definiëren we de output van H als volgt:

$$out(H) = \begin{cases} (A[H], O[H], B[H]) \text{ als } H \text{ stabiel of in een conflict is} \\ (A[H], A[H], A[H]) \text{ als } H \text{ veranderd is in A} \\ (B[H], B[H], B[H] \text{ als H veranderd is in } B) \end{cases}$$

3.3.2 Algoritme

Definitie 3.4 (Diff3 parse)**.** *Een diff3 parse van A, O en B t.o.v. de matchings* M_A *en* M_B *is een sequentie van stabiele en onstabiele blokken zodat, (I) wanneer* $M_A[o, a] = M_B[o, b] = true$*, komen de de indices a, o en b samen voor in één of andere stabiele blok, en (II) elke stabiele blok is zo groot als mogelijk.*

Merk op dat onder deze voorwaarden, de gegeven matchings M_A en M_B uniek de verdeling van de inputs in een alternerende sequentie van stabiele en onstabiele blokken bepalen. Het algoritme om deze blokken uit de matchings te berekenen gaat als volgt:

1. Initializeer $l_O = l_A = l_B = 0$.

2. Vind het kleinst natuurlijk getal $i \geq 1$ met $l_O + i < |O|$, $l_A + i < |A|$ en $l_B + i < |B|$ zodat ofwel $M_A[l_O+i, l_A+i] = \mathit{false}$ ofwel $M_B[l_O+i, l_B+i] = \mathit{false}$. Als er zo geen i is, ga dan naar stap 3 om als uitvoer een finale stabiele blok te geven.

   (a) Als $i = 1$, vind dan het kleinste natuurlijk getal met $o > l_O$ zodat er indices a, b bestaan met $M_A[o, a] = M_B[o, b] = \mathit{true}$. Als o niet bestaat, ga dan naar stap 3 om als uitvoer een finale onstabiele blok te geven. Anders, geef als uitvoer de (onstabiele) blok

   $$C = ([l_A + 1..a - 1], [l_O + 1..o - 1], [l_B + 1..b - 1]).$$

   Neem $l_O = o - 1$, $l_A = a - 1$, en $l_B = b - 1$ en herhaal stap 2.

   (b) If $i > 1$, geef volgende (stabiele) blok als uitvoer

   $$C = ([l_A + 1..l_A + i - 1], [l_O + 1..l_O + i - 1], [l_B + 1..l_B + i - 1]).$$

   Neem $l_O = l_O + i - 1$, $l_A = l_A + i - 1$, en $l_B = l_B + i - 1$, en herhaal stap 2.

3. Als ($l_O < |O|$ of $l_A < |A|$ of $l_B < |B|$), geef als uitvoer volgende finale blok

   $$C = ([l_A + 1..|A|], [l_O + 1..|O|], [l_B + 1..|B|]).$$

In de eerste stap van het algoritme initializeren we l_O, l_A en l_B. $l_O + 1$, $l_A + 1$ en $l_B + 1$ stellen de indices voor in respectievelijk de sequenties O, A en B die we gaan behandelen in de volgende stappen. We tellen één op bij l_O, l_A en l_B omdat de indices in de sequenties starten vanaf één.

In stap 2 zoeken we het kleinst natuurlijk getal i zodat, wanneer we i optellen bij de indices l_O, l_A en l_B, O en A en/of O en B niet matchen voor deze indices. M.a.w. we zoeken het eerste element dat niet in de matching van O en A en/of O en B startend vanaf de indices $l_O + 1$, $l_A + 1$ en $l_B + 1$ zit, dit bepaalt het einde van de overeenstemmende subsequenties. Als er zo geen i gevonden wordt, betekent dit dat de maximale matchings startend vanaf $l_O + 1$, $l_A + 1$ en $l_B + 1$ tot het einde van de sequentie lopen. We gaan dan naar stap drie die als uitvoer de juist gevonden subsequenties van A, B en C die matchen, geeft. Aangezien ze matchen noemen we ze stabiel. In het speciale geval waarbij l_O, l_A en l_B nul zijn, matchen de volledige sequenties.

In deelstappen (a) en (b), die horen bij stap twee, gaan we kijken welke waarde i heeft.

(a) In het geval $i = 1$, matchen ze zelfs niet in $l_O + 1$, $l_A + 1$ en $l_B + 1$. Dit maakt ook onmiddellijk duidelijk waarom de indices starten vanaf

één en niet vanaf nul, we moeten namelijk ook kunnen nagaan of de sequenties machten in het eerste element. We gaan nu op zoek naar het onstabiele blok dat start in $l_O + 1$, $l_A + 1$ en $l_B + 1$. Dit doen we door het kleinste natuurlijk getal o te zoeken dat groter is dan l_O en waarvoor er indices a en b bestaan zodat o in O met a in A en b in B matchen. Dit zijn dus de eerste indices van een nieuw stabiel blok. Als er zo geen o is, volgt er geen stabiel blok meer en geven we het onstabiele blok, startend vanaf respectievelijk, o, a en b en eindigend op het einde van de sequenties, als uitvoer. In het andere geval geven we als uitvoer het onstabiele blok startend op respectievelijk $l_O + 1$, $l_A + 1$ en $l_B + 1$ en eindigend op $o - 1$, $a - 1$ en $b - 1$. We nemen nu $l_O = o - 1$, $l_A = a - 1$ en $l_B = b - 1$, we herhalen nu stap twee.

(b) Wanneer $i > 1$ komen we in het geval (b), dit wil zeggen dat er wel degelijk een stabiele blok is startend vanaf $l_O + 1$, $l_A + 1$ en $l_B + 1$. We geven dan als uitvoer dit stabiele blok. Uiteraard eindigen deze op index + i - 1 aangezien ze niet meer stabiel zijn op index + i. We nemen nu $l_O = l_O + i - 1$, $l_A = l_A + i - 1$ en $l_B = l_B + i - 1$ en herhalen stap twee.

In stap drie geven we de finale matching als uitvoer. Dit doen we wel enkel wanneer $l_O < |O|$ of $l_A < |A|$ of $l_B < |B|$ geldt, deze voorwaarden garanderen dat minstens één van de drie spans niet leeg is en dat we dus over een blok mogen spreken.

Lemma 3.1. *Voor alle matchings M_A tussen O en A en alle matchings M_B tussen O en B heeft het bovenstaande diff3 algoritme een diff3 parse als output.*

Bewijs. Voor eigenschap (I) van Definitie 3.4 merken we op dat het begin van elke onstabiele blok geïdentificeerd is in stap 2(a). Het start op index $l_O + 1$ in O zodat ofwel $M_A[l_O + 1, l_A + 1] = \mathit{false}$ of $M_B[l_O + 1, l_B + 1] = \mathit{false}$. De onstabiele blok overspant de elementen $O[l_O + 1], \ldots, O[o - 1]$ in O waar $o > l_O$ de laagste index is met $M_A[o, a] = M_B[o, b] = \mathit{true}$ voor bepaalde a, b. Dus een onstabiele blok kan geen element bevatten in O die matcht in M_A en M_B.

Veronderstel nu dat aan eigenschap (II) niet voldaan in een parse output van het algoritme. Beschouw de eerste stabiele blok C die groter zou kunnen zijn. Als er een blok is die C voorafgaat, moet dit een onstabiele blok zijn, anders is C niet het eerste stabiele blok dat groter zou kunnen zijn. Door (I) weten we dat geen elementen in het onstabiele blok die C voorafgaat in C kunnen bevat zijn. Het is ook zo dat als C de output is in stap 2(b), het

eindigt in $A[l_A+i-1]$, $O[l_O+i-1]$, en $B[l_O+i, l_B+i]$ waar i voldoet aan de conditie dat ofwel $M_A[l_O+i, l_A+i] = \mathit{false}$ of $M_B[l_O+i, l_B+i] = \mathit{false}$. Het is duidelijk dat er niet meer elementen toegevoegd kunnen worden aan C. Gelijkaardig, als C de ouput is in stap 4, kan geen van A, O of B elementen bevatten die C bevat. Dus C moet maximaal zijn, wat een contradictie is. □

Om nu van de parse naar de uitvoer te gaan, berekenen we voor elk blok wat de uitvoer voor O', A' en B' moet zijn. Voor A' doen we dit bijvoorbeeld door voor elk blok de waarde berekend tijdens de parse van A te nemen of, indien de parse van A geen waarde in het beschouwde blok heeft, de waarde van de parse van B in dat blok. (Aangezien een blok drie waarden ofwel geen waarde bevat.) Hetzelfde doen we voor B'. Het berekenen van O' is gelijkaardig, ook hier nemen we in elk blok de waarde berekend voor O ofwel indien er zo geen waarde is, nemen we de enige waarde die er wel is voor dat blok.

3.4 Eigenschappen van Diff3

Diff3 is populair, maar er bestaan heel wat misverstanden over zijn werking. (Khanna 07) toonden aan dat er heel wat voor veel mensen intuïtieve eigenschappen zijn die niet blijken te gelden.

3.4.1 Lokaliteit

De meest belangrijke intuïtieve eigenschap bestaat eruit dat veel gebruikers van versiecontrolesystemen, zoals CVS, denken dat diff3 zonder problemen synchroniseert wanneer verschillende gebruikers lokaal in niet overlappende regio's aanpassingen maken. Veel gebruikers verwachten dat wanneer A en B enkel veranderd zijn in "niet-overlappende secties" synchronisatie een uniek, conflict-vrij resultaat geeft.

De onderzoekers van (Khanna 07) toonden echter aan dat dat niet het geval is. Om dit in te zien concentreren we ons op het geval waarin A enkel aangepast wordt aan het ene uiteinde en B enkel aan het andere uiteinde (zie Tabel 3.2).

Definieer een *tiling* τ voor een lijst O als een partitie van O in drie lijsten O_1, O_2 en O_3 zodat $O = O_1O_2O_3$. Een configuratie $(A \leftarrow O \rightarrow B)$ is τ-*respecting* als O_1 en O_3 elk veranderd zijn in ten hoogste één versie (A of B) en O_2 in geen van beide aangepast is. Als enkel één van O_1 of O_3 aangepast is, en deze aanpassingen dus in ofwel A ofwel B aangepast zijn of als O_1 en O_3

A	3,2,1	4,5,6	7,8,9
O	1,2,3	4,5,6	7,8,9
B	1,2,3	4,5,6	9,8,7

Figuur 3.2: Voorbeeld waarbij A enkel aangepast is aan het ene uiteinde en B aan het andere uiteinde.

in dezelfde versie zijn aangepast, zal het resultaat zoals verwacht conflictvrij zijn. Deze voorwaarden om τ-respecting te zijn garanderen dat A en B niet in dezelfde regio zijn aangepast en er scheidingsregio, O_2, is die in beide niet aangepast is. Het interessante geval is wanneer zowel A_1 als B_3 aangepast zijn.

We hebben nu gesproken over gescheiden delen O_1, O_2 en O_3, maar we hebben nog niet geformaliseerd wat "goed gescheiden" delen zijn en hoe de synchronizer deze herkent. Er zijn twee voor de handliggende manieren:

- vereis dat de aangepaste regio's gescheiden zijn door een *grote* onaangepaste regio, bv. dat O_2 langer moet zijn dan A_1, O_1, O_3 en B_3, of
- vereis dat de scheidingsregio *verschillend* is van alles dat op andere plaatsen voorkomt, bv. dat de string O_2 niet voorkomt in O_1, A_1, O_3 of B_3.

De meeste gebruikers van diff3 zullen waarschijnlijk veronderstellen (net zoals wij deden) dat één van deze voorwaarden voldoende is om een conflict-vrije synchronisatie te garanderen. We zullen nu een voorbeeldje geven waaruit blijkt dat deze veronderstelling in beide opzichten verkeerd is.

Laat $O_1 = \varnothing$, $O_2 = (1,2)^n$, en $O_3 = 1,2$ voor een bepaalde positief geheel getal n. In replica A is de O_1 component aangepast aan $A_1 = 1,2$ terwijl in replica B, de O_3 component aangepast is tot $B_3 = 3$. Beschouw de maximale matching M_A, weergegeven in Figuur 3.3 voor het paar (O,A) waar de 1,2 term in A_1 matcht tot de eerste 1,2 term in de O_2 component van O. Dan matcht de $(1,2)^{n-1}$ prefix in de O_2 component in A met de $(1,2)^{n-1}$ suffix in de O_2 component van O. Ten slotte matcht de laatste (1,2) term in de O_2 component van A met de O_3 component van O. De schrijvers van (Khanna 07) hebben nagegaan dat dit de maximale matching is tussen O en A die door diff3 gebruikt wordt. Voor het paar (O,B) is de enige maximum matching één waar zijn O_2 component matcht. Zoals getoond in Tabel 3.1 hebben we een "echt" conflict in deze uitvoering. Merk op dat het conflict onafhankelijk is van de waarde van de parameter n en dat het zelfs voorkomt wanneer de stabiele regio O_2 willekeurig groot is.

A	**1,2**	**$(1,2)^{n-1}$,1,2**	1,2
O		**1,2,$(1,2)^{n-1}$**	**1,2**

Figuur 3.3: De matching tussen O en A (in vet).

A	1,2,$(1,2)^{n-2}$	1,2,1,2
O	$(1,2)^{n-1}$	1,2
B	$(1,2)^{n-1}$	3
	stabiel	conflict

Tabel 3.1: Tegenvoorbeeld voor lokaliteit

We zouden nu heel pessimistisch kunnen zijn en ons af kunnen vragen of we wel ooit een conflictvrije synchronisatie hebben. Gelukkig is dit overdreven en kunnen we een eigenschap invoeren die ons garandeert dat we een conflictvrije synchronisatie verkrijgen. Noem een τ-respecting configuratie $(A \leftarrow O \rightarrow B)$ *veilig* als de O_2 component een element x bevat dat exact één keer voorkomt in O, A en B. Merk op dat er geen vereisten op de lengte van O_2 zijn, x moet er gewoon in voorkomen.

Stelling 3.1. *Elke veilige τ-respecting configuratie $(A \leftarrow O \rightarrow B)$ leidt tot een unieke conflictvrije synchronisatie.*

In de praktijk is meestal aan de veiligheidseigenschap voldaan: wanneer bijvoorbeeld de structuren die gesynchroniseerd moeten worden replica's zijn van een source code file, is het redelijk te verwachten dat O_2 bepaalde volledig unieke lijnen bevat, zoals een methode header of kenmerkende commentaar. We kunnen dus concluderen dat de intuïtieve verwachting van de gebruikers in de praktijk meestal ook klopt. Rest ons nog een bewijs te geven voor deze stelling. Om dit te kunnen moeten we echter eerst een eigenschap geven waarop het bewijs steunt.

Lemma 3.2. *Veronderstel, gegeven een configuratie $(A \leftarrow O \rightarrow B)$, een matching M_A tussen O en A, en een matching M_B tussen O en B. Als er een element x bestaat dat uniek voorkomt in elk van A, O en B, en als M_A en M_B het element x matchen, moet x bevat zijn in een stabiele blok in de diff3 parse die het resultaat is van M_A en M_B.*

Bewijs. Laat α_O, α_A en α_B respectievelijk de locaties van het element z in O, A en B aanduiden. We bewijzen de eigenschap door iteratief de blokken te beschouwen die de output zijn van het diff3 algoritme totdat x voor de eerste keer in een outputblok verschijnt. Laat l_O, l_A, en l_B de indices zijn

die de locaties van de elementen O, A en B aanduiden die behandeld worden door het algoritme. Bij aanname geldt: $l_O < \alpha_O$, $l_A < \alpha_A$ en $l_B < \alpha_B$.

Als het volgende blok dat als output geleverd wordt een onstabiele blok is zoals in stap 2(a), eindigt het blok juist voor de kleinste offset in O waarop er een element matcht in zowel M_A als M_B. Het is duidelijk dat de geüpdate indices l_O, l_A en l_B opnieuw moeten voldoen aan de eigenschap $l_O < \alpha_O$, $l_A < \alpha_A$, en $l_B < \alpha_B$ aangezien $M_A[\alpha_O, \alpha_A] = M_B[\alpha_O, \alpha_B] = \mathit{true}$. Aan de andere kant, als het volgende blok dat als output geleverd wordt een stabiele blok is zoals in stap 2(b), eindigt het blok juist voor de kleinste offset waarop er een element in O bestaat dat niet matcht in minstens één van M_A of M_B. Als de geüpdate indices nog altijd voldoen aan $l_O < \alpha_O$, $l_A < \alpha_A$ en $l_B < \alpha_B$ gaan we verder met het iteratieve proces, het handhaven van de invariant. In het andere geval moet het element z verschijnen in dit stabiele blok, wat de gewenste eigenschap bewijst. □

Met behulp van deze eigenschap kunnen we Stelling 3.1 bewijzen.

Bewijs. Veronderstel, zonder verlies van algemeenheid, dat O_1 aangepast is tot A_1 in A (bv. $A = A_1O_2O_3$) en dat O_3 aangepast is tot B_3 in B (bv. $B = O_1O_2B_3$). Beschouw elke maximale matching M_A tussen O en A. We beweren dat het element x moet gematcht zijn in $\mathrm{M_A}$. Veronderstel immers uit het ongerijmde van niet. Laat l het aantal elementen aanduiden dat gematcht zijn door M_A tussen de A_1 component van A en de O_1 component van O. Aangezien het element x niet matcht in M_A, is het aantal elementen dat matcht bij M_A begrensd door $l + (|O_2| + |O_3| - 1)$. Beschouw nu de matching M'_A die overeenstemt met M_A in de matching van elementen tussen A_1 en O_1 en ook de O_2 en O_3 componenten van A en O volledig matcht. Dan is $l + (|O_2| + |O_3|)$ het totaal aantal elementen dat matcht bij M'_A, wat een contradictie is met de veronderstelling dat M_A een maximale matching is. Dus moet x gematcht zijn in M_A. Bovendien is het zo dat, aangezien A en O identiek zijn na x, alle elementen in A na x moeten matchen met alle elementen in O na x om een maximale matching te zijn. Gelijkaardig moet M_B alle elementen tot x in B matchen met alle elementen tot x in O matchen.

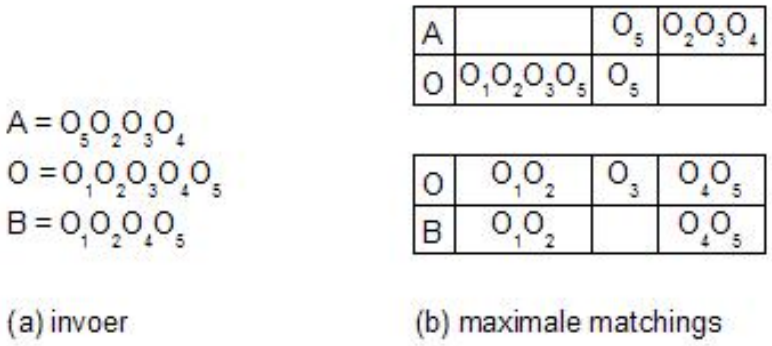
Volgens Lemma 3.2 moet x bevat zijn in een stabiele blok in diff3's output. Beschouw, om het bewijs te vervolledigen, om het even welke onstabiele blok H die als output geleverd wordt door het algoritme. Aangezien het uniek element x bevat is in een stabiele blok gaan ofwel alle elementen in de A, O en B componenten van de blok H x vooraf ofwel volgen ze allemaal na x. In het eerste geval kan H enkel aangepast zijn in A, aangezien M_B alle elementen tot x in B matcht met alle elementen tot x in O. Gelijkaardig

in het laatste geval kan H enkel aangepast zijn in B. Dus is elke onstabiele blok conflictvrij. Ten slotte, om te zien dat de resulterende output uniek is, moet men opmerken dat in elke parse, al de blokken ná x ofwel stabiel ofwel veranderd zijn in A en deze vóór x ofwel stabiel ofwel veranderd zijn in B. In de ouput zullen dus de elementen tot x genomen worden uit A en deze volgend op x uit B □

We hebben nu wel een eigenschap die aangeeft wanneer regio's goed gescheiden zijn, maar kunnen we deze eigenschap veralgemenen? Dat zou ze nog veel interessanter maken. Kan het bijvoorbeeld uitgebreid worden naar de situatie waarin elke gebruiker aanpassingen heeft gedaan in verschillende regio's van de lijst, en de regio's gescheiden zijn door een unieke elementen en geen enkele regio zowel in A als in B aangepast is? Meer concreet, laten we zeggen dat een een *veralgemeende tiling* τ een partitie is van O in $2k + 1$ niet-lege stukken met k een positief geheel getal zodat $k \neq 1$. We hebben dan delen $O_1, O_2, \ldots, O_{2k+1}$. We zeggen nu dat een configuratie $(A \leftarrow O \rightarrow B)$ τ-*respecting* is als elk deel O_{2i+1} voor $0 \leq i \leq k$ aangepast is in ten hoogste één van A en B, terwijl elk deel O_{2i} voor $O \leq i \leq k$ in geen van beide aangepast is. Er wordt dan gezegd dat een τ-respecting configuratie $(A \leftarrow O \rightarrow B)$ *veilig* is als elke O_{2i} component een element x_{2i} bevat dat exact één keer voorkomt in elk van O, A en B.

Het komt er op neer dat net zoals in eerdere voorbeelden O enkel aangepast mag worden in de oneven regio's en de even regio's, telkens met een uniek element, als scheidingsregio's dienen.

Jammer genoeg gaat deze veralgemening niet op. We zijn niet langer verzekerd van een conflictvrije synchronisatie. Beschouw bijvoorbeeld de uitbreiding waarbij $k = 2$, zodat $O = O_1O_2O_3O_4O_5$. Verder nemen we aan dat voor elke i, j met $1 \leq i < j \leq 5$, O_i en O_j disjunct zijn, dit betekent dat zij geen enkel element delen. Laat $A = A_1O_2O_3O_4A_5$ en $B = O_1O_2B_3O_4O_5$ zijn. En laat $A_1 = O_5$ en $A_5 = B_3 = \varnothing$. Als nu $|O_5| > |O|/2$, dan matcht de unieke maximale matching M_A tussen A en O de A_1 component in A met de O_5 component in O. Beschouw aan de andere kant de matching M_B tussen B en O die matcht in alle componenten behalve B_3 met O_3. Dit is weergegeven in Figuur 3.4. Het is gemakkelijk in te zien dat de eerste diff3 blok in conflict is. Wanneer we het algoritme uitvoeren komen we in stap 2(a) terecht aangezien de sequenties niet matchen in het eerste element, i is dus één. In stap 2(a) vinden we geen o die aan de voorwaarden voldoet aangezien de drie sequenties nergens matchen en dus komen we in stap 3 terecht en aangezien l_O, l_A en l_B nog steeds nul zijn krijgen we als uitvoer de onstabiele blok die de gehele sequenties bevat.



Figuur 3.4: Tegenvoorbeeld veralgemening tiling τ

3.4.2 Idempotentie

Lokaliteit is niet de enige eigenschap die gebruikers intuïtief aanvoelen en niet blijkt te kloppen. In de rest van deze sectie zullen we er nog enkele geven.

We beginnen met de eigenschap dat elke uitvoering van een synchronizer "zoveel als mogelijk" moet doen en een een stabiele staat moet bereiken. Dit betekent dat wanneer er onmiddelijk erna, zonder dat er intussen aanpassingen zijn gemaakt, opnieuw synchroniseerd wordt, er geen verdere aanpassingen gepropageerd worden en de staat dus gewoon behouden blijft. Dit kan formeel uitgedrukt worden als volgt:

Eigenschap 3.1. *Een synchronisatiealgoritme is* idempotent *als* $(A \leftarrow O \rightarrow B) \Rightarrow (A' \leftarrow O' \rightarrow B')$ *impliceert* $(A' \leftarrow O' \rightarrow B') \Rightarrow (A' \leftarrow O' \rightarrow B')$.

Stelling 3.2. *Diff3 is* niet *idempotent.*

Bewijs. Beschouw de uitvoering in het bovenste deel van Figuur 3.5, waar

$$([1,2,4,6,8] \leftarrow [1,2,3,4,5,5,5,6,7,8] \rightarrow [1,4,5,5,5,6,2,3,4,8]) \\ \Rightarrow ([1,2,4,6,8] \leftarrow [1,2,3,4,6,7,8] \rightarrow [1,4,6,2,3,4,8]).$$

Als we nu deze uitvoer opnieuw als invoer nemen, zoals weergegeven in Figuur 3.5. krijgen we het volgende:

$$([1,2,4,6,8] \leftarrow [1,2,3,4,6,7,8] \rightarrow [1,4,6,2,3,4,8]) \\ \Rightarrow ([1,4,6,2,4,6,8] \leftarrow [1,4,6,2,4,6,7,8] \rightarrow [1,4,6,2,4,8]).$$

Merk op dat diff3 in geen van beide gevallen een andere keuze heeft: elk van de input configuraties heeft juist één paar van maximale matchings. □

A	1	2	4		6		8
O	1	2,3	4	5,5,5	6	7	8
B	1		4	5,5,5	6	2,3,4	8
	stabiel	conflict	stabiel	veranderd in A	stabiel	conflict	stabiel

A	1		2		4	6	8
O	1		2	3	4	6,7	8
B	1	4,6	2	3	4		8
	stabiel	veranderd in B	stabiel	veranderd in A	stabiel	conflict	stabiel

Figuur 3.5: Tegenvoorbeeld idempotentie

3.4.3 Bijna succes bij gelijkaardige replica's

Zoals we eerder gezien hebben begint het diff3 algoritme met het vergelijken van O t.o.v. A en B afzonderlijk. In het algortime worden A en B nooit rechtstreeks vergeleken. Maar ondanks dat deze vergelijking niet rechtstreeks gebeurd, verwachten we intuïtief toch dat wanneer A en B gelijkaardig zijn, of in het extreme geval zelf hetzelfde zijn, de synchronisatie succesvol verloopt. Ongeacht het feit of ze al dan niet veel verschillen van O. Dit is spijtig genoeg niet het geval: het is perfect mogelijk dat de synchronisatie faalt wanneer A en B gelijkaardig zijn.

Voor elk paar van replica's A, B, laat $m(A, B)$ de lengte van een grootste gemeenschappelijke *subsequentie* van A en B waarbij een subsequentie een deel van de sequentie is waarin elementen weggelaten zijn, aanduiden. Bijvoorbeeld wanneer $A = 1, 2, 3, 4, 5, 6$ en $B = a, b, 1, 2, c, 5, d$ dan is $1, 2, 5$ de grootste gemeenschappelijke subsequentie van A en B. Bijgevolg is $m(A, B) = 3$.

Laat ϵ een functie zijn die natuurlijke getallen mapt op reële getallen tussen 0 en 1. Er wordt gezegd dat een paar van replica's A, B *ϵ-close* is als $m(A, B) > (1 - \epsilon(n))n$, waarbij $n = max\{|A|, |B|\}$. Dit wil zeggen dat de lengte van de maximale matching tussen A en B gedeeld door de lengte van de grootste sequentie (A of B) groter of gelijk moet zijn aan $1 - \epsilon(n)$. Dit dus de graad aan waarmee ze matchen. We kunnen nu formeel stabiliteits eigenschappen definiëren die de notie van "similariteit" met zich meebrengen.

Eigenschap 3.2. *Een synchronisatie algoritme garandeert* bijna succes op gelijkaardige replica's *als er een universele constante $c > 0$ bestaat zodat voor elke ϵ en elk ϵ-close paar (A,B) geldt dat als $(A \leftarrow O \rightarrow B) \Rightarrow (A' \leftarrow O' \rightarrow B')$, A' en B' $(c\epsilon)$-close zijn.*

Stelling 3.3. *Diff3 garandeert* geen *bijna succes op gelijkaardige replica's.*

Bewijs. Beschouw de volgende invoer configuratie (dit is een veralgemening

van de invoer in Sectie 3.2):

$$A[i] = \begin{cases} O[1] & i = 1 \\ O[\frac{n}{2} + i - 1] & 1 < i \leq \frac{n}{2} \\ O[i - \frac{n}{2} + 1] & \frac{n}{2} < i < n \\ O[n] & i = n \end{cases}$$

$$O[i] = \Big\{ i \quad 1 \leq i \leq n$$

$$B[i] = \begin{cases} O[i] & 1 \leq i \leq 2 \\ O[\frac{n}{2} + i - 2] & 2 < i < \frac{n}{2} + 2 \\ O[3] & i = \frac{n}{2} + 2 \\ O[\frac{n}{2} + i + 1 - n] & \frac{n}{2} + 2 < i < n \\ O[n] & i = n \end{cases}$$

Dit geeft:

$$(A \leftarrow O \rightarrow B) = \begin{pmatrix} [1, \frac{n}{2} + 1, \ldots, n-1, 2, \ldots, \frac{n}{2}, n] \\ \uparrow \\ [1, \ldots, n] \\ \downarrow \\ [1, 2, \frac{n}{2} + 1, \ldots, n-1, 3, \ldots, \frac{n}{2}, n] \end{pmatrix}$$

Merk op dat het paar (A, B) $\frac{1}{n}$-close is, aangezien hun grootste gemeenschappelijke subsequentie van lengte $n-1$ is. De unieke maximale subsequentie van O en A is $[1, 2, \ldots, \frac{n}{2}, n]$, tussen O en B is het $[1, 2, \frac{n}{2}+1, \ldots, n-1, n]$. Dit leidt tot drie stabiele diff3 blokken en twee onstabiele blokken, zoals getoond in Tabel 3.2. Alhoewel de tweede van deze onstabiele blokken in conflict is, is de eerste enkel in A aangepast. De output van deze blok propageert dus $[\frac{n}{2} + 1, \ldots, n-1]$ naar O en B, wat de volgende output oplevert:

$$(A' \leftarrow O' \rightarrow B') = \begin{pmatrix} [1, \frac{n}{2} + 1, \ldots, n-1, 2, \ldots, \frac{n}{2}, n] \\ \uparrow \\ [1, \frac{n}{2} + 1, \ldots, n-1, 2, \ldots, n] \\ \downarrow \\ [1, \frac{n}{2} + 1, \ldots, n-1, 2, \frac{n}{2} + 1, \ldots, n-1, 3, \ldots, \frac{n}{2}, n] \end{pmatrix}$$

In de uiteindelijk verzoende staat, zijn A' en B' enkel alleen ongeveer $\frac{1}{3}$-close ($m(A', B') = n$, terwijl $\max\{|A'|, |B'|\}$ ongeveer $\frac{3n}{2}$ is), en er bestaat dus geen c zodat zij $\frac{c}{n}$-close is voor elke positieve n. □

A	1	$\frac{n}{2}+1,\ldots,n-1$	2	$3,\ldots,\frac{n}{2}$	n
O	1		2	$3,\ldots,n-1$	n
B	1		2	$\frac{n}{2}+1, n-1, 3,\ldots,\frac{n}{2}$	n
	stabiel	veranderd in A	stabiel	conflict	stabiel

Tabel 3.2: Tegenvoorbeeld verschillende eigenschappen

3.4.4 Stabiliteit

Een andere intuïtieve eigenschap die ons aanvaardbaar lijkt, is dat elke twee uitvoeringen waarbij de inputs gelijk zijn de outputs ook gelijk moeten zijn.

Eigenschap 3.3. *Een synchronisatie algoritme is stabiel als er een universele constante $c > 0$ bestaat zodat, voor elke drie paren (O_1, O_2), (A_1, A_2) en (B_1, B_2), met elk paar ϵ-close, er geldt dat als $(A_1 \leftarrow O_1 \rightarrow B_1) \Rightarrow (A'_1 \leftarrow O'_1 \rightarrow B'_1)$ en $(A_2 \leftarrow O_2 \rightarrow B_2) \Rightarrow (A'_2 \leftarrow O'_2 \rightarrow B'_2)$ elk paar van replica's (O'_1,O'_2), (A'_1,A'_2) en (B'_1,B'_2) $c\epsilon$-close is.*

Stelling 3.4. *Diff3 is* niet *stabiel, zelfs niet voor niet-conflicterende uitvoeringen.*

Bewijs. Beschouw de volgende uitvoering

$$([X,Y,X] \leftarrow [X,Y,0,Y,X] \rightarrow [Y,X,0,Y]) \Rightarrow [Y,X,0]$$

$$([X,Y,X] \leftarrow [X,Y,0,Y,X] \rightarrow [0,Y,X,Y]) \Rightarrow [0,X,Y],$$

waar $X = [1,\ldots,\frac{n}{2}]$ en $Y = [\frac{n}{2}+1,\ldots,n]$. Het is gemakkelijk in te zien dat de corresponderende paren in de twee input configuraties alle $\frac{2}{3n}$-close zijn terwijl de ouput slecht ongeveer $\frac{1}{2}$-close is. □

3.5 Langst gemeenschappelijke subsequentie

In Sectie 3.3 hebben we beloofd dat we dieper in zouden gaan op het algoritme dat maximale matchings berekent. Deze verwachting lossen we nu in. We zullen aantonen dat het vinden van een maximale matching, dus een maximale gemeenschappelijke sequentie van A en B mogelijk is in een lage tijds- en ruimtecomplexiteit. Dit zullen we doen door aan te tonen dat het probleem identiek is aan een ander problemen, waarvoor we een efficiënt algoritme. Het algoritme dat wij geven is een aangepaste versie van het algoritme dat geïntroduceerd is in (Myers 86) en is eenvoudig en gebaseerd op een intuïtief edit graph formalisme.

In tegenstelling tot bij andere algoritmen wordt het "greedy" design paradigma toegepast. En we zullen het verband met het single-source shortest path probleem blootleggen. Het algoritme kan verfijnd worden zodat het slechts lineaire ruimte gebruikt, en in (Myers 86) wordt aangetoond dat het verwachte tijdsgedrag $O(N+D^2)$ is. Volgens hetzelfde werk is een worst-case variatie van $O(NlgN + D^2)$ tijd mogelijk.

Alvorens het algoritme te geven in Sectie 3.5.2 zullen we eerst edit graphs bespreken.

3.5.1 Definitie

Veronderstel gegeven twee lijsten $A = a_1a_2 \ldots a_N$ en $B = b_1b_2 \ldots b_M$ van lengte N en M respectievelijk. Om de maximale matching tussen A en B te kunnen berekenen voeren we volgend begrip in:

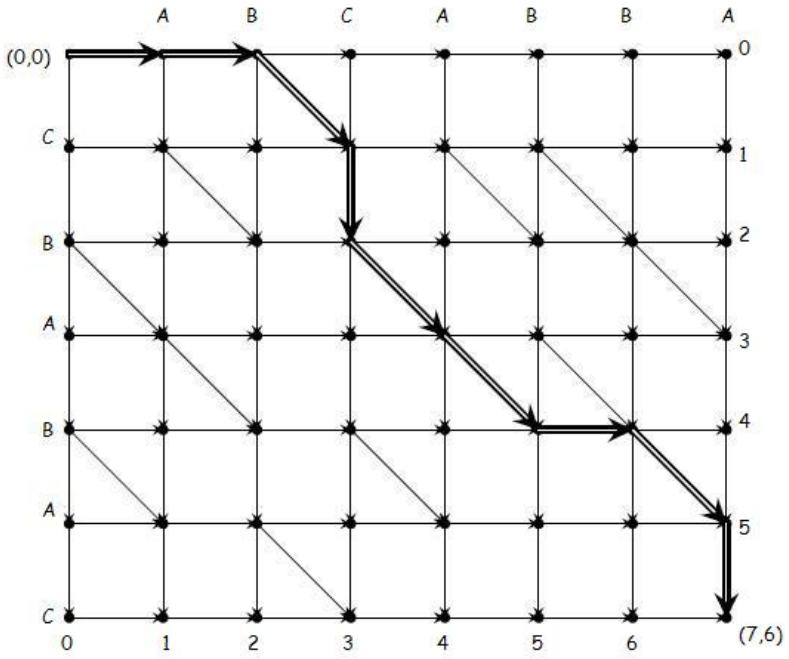
Definitie 3.5. *De edit graph voor A en B is de gerichte acyclische graaf met als knopen de paren (x,y) met* $0 \leq x \leq M$ *en* $0 \leq y \leq M$*. Vanuit elke (x,y) is er een horizontale pijl naar (*$x + 1$*,y) en een verticale pijl naar (x,*$y + 1$*). Voorts is er een diagonale pijl naar (*$x + 1$*,*$y + 1$*) indien* $a_{x+1} = b_{y+1}$*. Het totale aantal diagonale punten duiden we aan met R.*

In Figuur 3.6 zien we een voorbeeld van een edit graph voor $A = abcabba$ en $B = cbabac$. De punten (x,y) met $a_x = b_y$ noemen we *match punten*. Intuïtief gezien zijn dit de punten die in een niet-kruisende matching kunnen voorkomen. Merk op dat elk pad in de edit graph van (0,0) tot (N,M) aanleiding geeft tot een niet-kruisende matching tussen A en B, en vice versa. Inderdaad:

- Beschouw bijvoorbeeld het pad dat in Figuur 3.6 in $\Rightarrow$ pijlen aangegeven is. Een horizontale pijl duidt op een element in A dat niet in B gematcht kan worden. Een verticale pijl duidt op een element in B dat niet in A gematcht kan worden. Een diagonale pijl duidt op een match punt.We kunnen nu uit dit pad een niet-kruisende matching als volgt berekenen. Neem de sequentie punten $(x_1, y_1), \ldots, (x_L, y_L)$ in het pad die we bezoeken na het volgen van een diagonale pijl. In figuur 3.6 hebben we bijvoorbeeld

  $$(3, 1)(4, 3)(5, 4)(7, 5) \tag{3.1}$$

  omdat $(x_1, y_1), \ldots, (x_Ly_L)$ in de volgorde staan waarin ze in het pad bezocht worden hebben we $x_i < x_{i+1} en y_i < y_i + 1$ voor $1 \leq i \leq L$.



Figuur 3.6: Voorbeeld van een edit graph.

Aldus is M_A met

$$M_A[x,y] == \begin{cases} \mathit{true} & \text{indien } (x_i,y_i) = (x,y) \text{ voor een } 1 \leq i \leq L \\ \mathit{false} & \text{anders} \end{cases} \tag{3.2}$$

een niet kruisende matching:

1. duidelijk is $a_x = b_y$ wanneer $M_A(x,y) = \mathit{true}$
2. aangezien de punten $(x_1,y_1),\ldots,(x_L,y_L)$ noch een x-coördinaat noch een y-coördinaat delen is $M_A[x',y] = \mathit{false}$ en $M_A[x,y'] = \mathit{false}$ en $x' \leq x$ en $y' \leq y$
3. omdat $(x_1,y_1),\ldots,(x_L,y_L)$ in de volgorde staat waarin ze in het pad bezocht worden, is $M_A[x',y'] = \mathit{false}$ wanneer $x' < x$ en $y' > y$ of $x > x'$ en $y' < y$

- Omgekeerd, gegeven een niet-kruisende matching M_A tussen A en B in de edit graph kunnen we een pad van $(0,0)$ tot (N,M) construeren als volgt. Zoek het "kleinste" punt (x,y) met $M_A[x,y] = \mathit{true}$ volgens de lexicografische orde

$$(a,b) < (a',b') \text{ indien } a < a' \text{ of } (a = a' \text{ en } b < b') \tag{3.3}$$

Dan is (x,y) een matchpunt aangezien $a_x = b_y$. Construeer dus een pad voor $(0,0)$ tot (x,y) door enkel horizontale en verticale pijlen te volgen tot $(x-1,y-1)$, en dan de diagonale pijl van $(x-1,y-1)$ tot (x,y). Neem dan het volgende kleinste punt (x',y') door horizontale en verticale pijlen te nemen, en vervolgens de diagonaal tot (x',y'). We blijven dit herhalen tot we alle punten in $\mathrm{M_A}$gehad hebben. Uiteindelijk gaan we van dat laatste punt tot (N,M).

Bovenstaande overeenkomst laat dus zien dat het vinden van een maximale matching tussen A en B overeenkomt met het vinden van een pad van (0,0) naar (N,M) in de edit graph dat een maximaal aantal diagonale pijlen volgt (of equivalent, een minimaal aantal horizontale en verticale pijlen.) Wanneer we aan elke pijl een gewicht geven (0 voor de diagonale pijlen en 1 voor de andere) dan is het vinden van de maximale matching dus equivalent met het vinden van een pad van (0,0) naar (N,M) met de minste kost in de edit graph, en is het dus een instantie van het single source shortest pad probleem. In de volgende sectie geven we een algoritme dat dit probleem zeer efficiënt oplost voor de speciale klasse van edit graphs die we slechts hoeven te beschouwen.

Voor de volledigheid willen we echter nog opmerken dat dit probleem ook equivalent is met het zoeken van een langste gemeenschappelijke subsequentie en met het zoeken van een minimaal edit script, twee problemen die vaak voorkomen in de informatica. We lichten ze even toe.

Definitie 3.6 (Subsequence)**.** *Een subsequentie van een lijst A is eender welke lijst bekomen door elementen in A te verwijderen. Een common subsequence van twee lijsten A en B is een lijst die zowel een sequentie van A als van B is.*

Het mag duidelijk zijn dat de sequentie matchpunten bezocht in een pad van (0,0) naar (N,M) in de edit graph aanleiding geeft tot een common subsequence, en omgekeerd.

Het vinden van een maximale common subsequence is dus equivalent met het vinden van een pad van (0,0) naar (N,M) met minimale kost.

Definitie 3.7 (Edit script)**.** *Een* edit script voor A en B *is een verzameling van invoeg- en verwijder-commando's die A in B transformeren. Hierbij verwijderdt het delete commando "xD" symbool a_x van A. Het insert commando "x I $b_1, b_2, \ldots, b_t$ voegt de symbolen $b_1, b_2, \ldots, b_t$ toe direct na a_x. De commando's moeten gezien worden of ze allemaal op hetzelfde moment uitgevoerd worden. De* lengte *van een edit script is het aantal invoeg en verwijder commando's. Een edit script voor A en B is* minimaal *indien er geen ander edit script bestaat dat een kleinere lengte heeft.*

Elk pad van (0,0) naar (N,M) komt overeen met een edit script en omgekeerd. De horizontale pijlen komen overeen met delete commando's, de verticale met insertions. Opnieuw is het vinden van een minimaal edit script dus equivalent met het vinden van een pad met minimale kost.

3.5.2 Algoritme

We gaan nu een algoritme geven voor het vinden van een pad van $(0, 0)$ naar (N, M) met het minste aantal horizontale en verticale bogen en dus ook voor het vinden van een maximale matching. Alvorens dit algoritme te geven voeren we nog enkele begrippen in.

Definitie 3.8. *We zeggen een pad een D-pad is als het in start $(0, 0)$ en exact D niet-diagonale bogen heeft.*

Hieruit volgt logisch dat een 0-pad enkel uit diagonale bogen bestaat. Wanneer we nu inductie toepassen kunnen we zeggen dat een D-pad uit een

$(D-1)$-pad bestaat gevolgd door een niet-diagonale boog en een mogelijk lege sequentie van diagonale bogen die we een *snake* noemen.

De bogen van de edit graph wordt nu genummerd zodat diagonaal k bestaat uit de punten (x, y) voor de welke $x - y = k$.(Zie Figuur 3.7.)

Met deze definitie worden de diagonalen genummerd van $-M$ tot N. Merk op dat een verticale (horizontale) boog met startpunt op diagonaal k een eindpunt heeft op diagonaal $k-1$ (respectievelijk k+1).

Lemma 3.3. *Een D-pad moet eindigen op een diagonaal $k \in \{-D, -D+2, \ldots, D-2, D\}$.*

Bewijs. Een 0-pad bestaat enkel uit diagonale bogen en start op diagonaal nul, bijgevolg eindigt het op diagonaal nul. Veronderstel inductief dat een D-pad moet eindigen op diagonaal k in $\{-D, -D+2, \ldots, D-2, D\}$. Elk $(D+1)$-pad bestaat uit een prefix D-pad eindigend op diagonaal k, een niet-diagonale boog eindigend op diagonaal $k+1$ of $k-1$, en een snake die ook moet eindigen op diagonaal $k+1$ of $k-1$. Er volgt dan dat elk $(D+1)$-pad moet einidgen op diagonaal $\{-D \pm 1, -(D+2) \pm 1, \ldots, (D-2) \pm 1, D \pm 1\} = \{-D-1, -D+1, \ldots, D-1, D+1\}$. Hiermee is het lemma bewezen. □

Het lemma zegt bijgevolg ook dan een D-pad eindigt op een oneven diagonaal wanneer D oneven is en op een even diagonaal wanneer D even is.

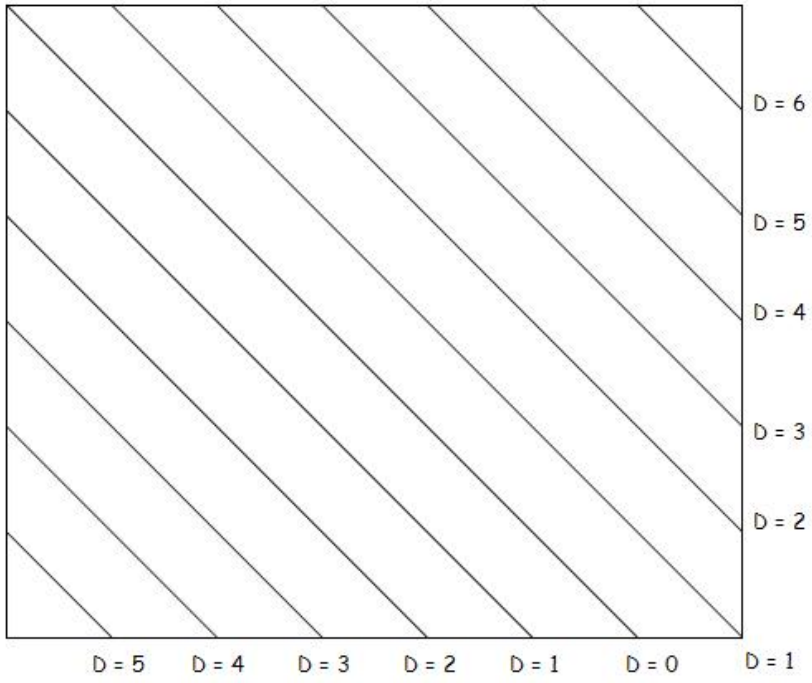
Definitie 3.9. *Een D-pad is* verst reikend *in diagonaal k als en enkel als het één van de D-paden is die eindigen op diagonaal k en het de grootst mogelijk rij(kolom) nummer heeft van al deze paden.*

Intuïtief betekent dit dat van alle D-paden dit pad het verst van het startpunt $(0, 0)$ verwijderd is. Het volgende lemma geeft een inductieve karakterisatie van verst reikende D-paden en omvat een greedy principe om ze te berekenen: verst reikende D-paden worden bereikt door greedily verst reikende (D-1) paden uit te breiden.

Lemma 3.4. *Een verst reikend 0-pad eindigt op (x, x), waarbij*

$$x = \min\{z - 1 \mid a_z \neq b_z \text{ of } z > M \text{ of } z > N\}.$$

Een verst reikend D-pad op een diagonaal k kan zonder verlies van algemeenheid ontleed worden in een $(D-1)$-pad op diagonaal $k-1$, gevolgd door een horizontale boog, gevolgd door de langst mogelijk snake of het kan ontleed worden in een verstreikend (D-1)-pad op diagonaal $k+1$, gevolgd door een verticale boog, gevolgd door de langst mogelijke snake.



Figuur 3.7: Voorbeeld van een edit graph met diagonalen.

Bewijs. De basis voor 0-paden is duidelijk. Zoals eerder gezegd bestaat een D-pad uit een $(D-1)$-pad, een niet-diagonale boog, en een snake. Als het D-pad eindigt op diagonaal k volgt er dat het $(D-1)$-pad moet eindigen op diagonaal $k \pm 1$ afhankelijk van of een verticale of horizontale boog de snake voorafgaat. De uiteindelijke snake moet maximaal zijn, aangezien het D-pad niet verst reikend is als de snake uitgebreid kan worden. Veronderstel dat het $(D-1)$-pad niet verst reikend is in zijn diagonaal. Maar dan kan een verst reikend $(D-1)$-pad geconnecteerd worden de laaste snake van het D-pad d.m.v. een passende beweging. Dus het D-pad kan altijd ontleed worden zoals gewenst. □

Gegeven de eindpunten van het verst reikende pad $(D-1)$-paden in diagonalen $k+1$ en $k-1$, zeg (x', y') en (x'', y'') respectievelijk, geeft Lemma 3.4 een procedure om de eindpunten van het verst reikend D-pad in diagonaal k te berekenen. Neem namelijk de verst reikenden van $(x', y'+1)$ en $(x''+1, y'')$ in diagonaal k en volg dan diagonale bogen totdat het niet langer mogelijk is om dit te doen of totdat de grenzen van de edit graph bereikt zijn. Dit vereist het berekenen van de eindpunten van D-paden in de relevante $D+1$-diagonalen voor opeenvolgende toenemende waardee van D totdat het verst reikend pad in diagonaal $N-M$ (N, M) bereikt.

Algoritme 2 geeft nu een algoritme voor het construeren van een maximale niet-kruisende matching. We hebben hierbij een constante MAX waarvoor wij de waarde $M + N$ nemen. Deze constante geeft aan hoelang het edit scrip maximaal mag zijn. Het is mogelijk deze waarde te verlagen, maar in dat geval hebben we geen garantie meer dat we een edit script vinden die sequentie A naar B omzet. Het element $V_D[k]$ bevat y-waarde van het verst reikend D-pad waarbij $x = k$.

De procedure LCS neemt $O((M + N)D_{\text{LCS}})$ tijd in beslag en $O((M + N)^2)$ ruimte, met D_{LCS} de lengte van de maximale niet-kruisende matching (berekend in lijn 20). Lijnen 1, 2, 4, 5 worden in constante tijd uitgevoerd. De binneste lus (Lijn 8) wordt wordt maximaal $(D_{\text{LCS}} + 1)(D_{\text{LCS}} + 2)/2$ keer herhaald omdat de buitenste lus (Lijn 7) $D_{\text{LCS}} + 1$ keer herhaald wordt en tijdens de k-de iteratie de binneste lus maximum k keer herhaald wordt. Alle lijnen binnen de binnenste lus nemen constante tijd behalve de while lus en de aanroep van de LijstPad functie. De while-lus neemt maximaal $O(M+N)$ tijd in beslag per iteratie van de buitenste lus. Ook de procedure LijstPad neemt $O(M + N)$ tijd in beslag, omdat deze maximaal $M + N$ keer een constante operatie uitvoert. LijstPad wordt ook slechts één maal tijdens de volledige uitvoer van LCS opgeroepen. Dit zorgt er voor dat de tijdscomplexiteit van LCS $O((M + N)D_{\text{LCS}})$ bedraagt. Voor de ruimtecomplexiteit is de array doorslaggevend, dit zorgt voor een $O((M + N)^2)$ ruimtecomplexiteit.

Zoals eerder aangehaald kan het LCS/SES probleem gezien worden als een geval van het single-shortest pad probleem op een gewoon edit graph. Dit suggereert dan een efficiënt algoritme kan verkregen worden door Dijkstra's algoritme (Dijkstra 59) te specializeren. In (Myers 86) wordt zo een specialisatie besproken. Volgens deze bron heeft deze specialisatie ook tijdscomplexiteit $O(ND)$. Dat algoritme vraagt echter een erg complexere implementatie dan Algoritme 2.

Algoritme 2 Algoritme

```
1: constant MAX = M + N
2: D_LCS integer
3: var V_{-1}, V_0, ..., V_MAX :array[-MAX ... MAX] of integer
4: function LCS
5:     V_{-1}[1] ← 0
6:     V_0[1] ← 0
7:     for all D met 0 ≤ D < MAX do
8:         for k ← -D to D in steps of 2 do
9:             if k = -D or k ≠ and V_{D-1}[k-1] < V_{D-1}[k+1] then
10:                x ← V_{D-1}[k+1]
11:            else
12:                x ← V_{D-1}[k-1] + 1
13:            end if
14:            y ← x - k
15:            while x < N and y < M and a_{x+1} = b_{y+1} do
16:                (x, y) ← (x + 1, y + 1)
17:                V_D[k] ← x
18:            end while
19:            if x ≥ N and y ≥ M then
20:                D_LCS := D
21:                L:= BerekenPad(D, k)
22:                return matching gebaseerd op L
23:            end if
24:        end for
25:    end for
26: end function

27: function BerekenPad(d,k)
28:    if d = 0 then
29:        return de lijst het pad van (0, 0) tot (V_0[0], V_0[0])
30:    else if V_d[k] einde van maximale snake is die volgt op verticale boog
       vanaf V_{d-1}[k + 1] then
31:        Lijst := BerekenPad(d - 1, k + 1)
32:        Voeg de verticale boog, gevolgd door de snake die eindigt in V_d[k]
       achteraan de lijst toe
33:        return Lijst
34:    else if V_d[k] het einde is van een maximale snake is die volgt op een
       horizontale boog vanaf V_{d-1}[k - 1] then
35:        Lijst := BerekenPad(d - 1, k - 1)
36:        Voeg de horizontale boog, gevolgd door de snake die eidigt in V_d[k]
       achteraan de lijst toe
37:        return Lijst
38:    end if
39: end function
```

Hoofdstuk 4

Synchronizeren van ongeordende bomen met keys

In Hoofdstuk 3 hebben we gezien hoe we lijst-gebaseerde gegevens zoals tekstbestanden kunnen synchronizeren. In dit hoofdstuk bespreken we het synchronizeren van ongeordende, boomgestructureerde gegevens.

4.1 Inleiding

Aangezien elk bestand als een lijst bekeken kan worden, kan diff3 (besproken in Hoofdstuk 3) in principe gebruikt worden om eender welke type bestanden te synchronizeren. Vaak heeft de data die we willen synchronizeren echter een natuurlijke, hiërarchische boomstructuur. Denk bijvoorbeeld aan de gegevens opgeslagen in een XML-document, maar ook aan gegevens opgeslagen in digitale adresboeken, kalenders, etc.

Voor zulke gegevens synchronizeert diff3 vaak te grof. Beschouw bijvoorbeeld de XML documenten O, A, en B getoond in Figuur 4.1 waarbij O het originele document voorstelt en A en B aangepaste replica's zijn. In A zijn de lijnen met de naam en beschrijving omgewisseld. Diff3 zal dit zien als een veranderde lijn en ten onrechte een verandering melden. De derde versie bevat een extra woord "nummer" in de naam. Diff3 zal ook dit beschouwen als een veranderde lijn. Als gevolg hiervan zal diff3 als output de conflicterende configuratie ($A \leftarrow O \rightarrow B$) teruggeven. Dit probleem hebben we niet wanneer we een synchronisatiealgoritme gebruiken dat weet heeft van de boomstructuur van de gegevens. Dan kan er bijvoorbeeld gesynchroniseerd worden door in B de volgorde van naam en beschrijving om te wisselen, en in A de naam aan te passen.

In dit hoofdstuk bespreken we methodes om *ongeordende* bomen te syn-

O	<voorbeeld id="1"> <naam>Voobeeld één</naam> <beschrijving>Dit is een voorbeeld</beschrijving> </voorbeeld>
A	<voorbeeld id="1"> <beschrijving>Dit is een voorbeeld</beschrijving> <naam>Voobeeld één</naam> </voorbeeld>
B	<voorbeeld id="1"> <naam>Voobeeld nummer één</naam> <beschrijving>Dit is een voorbeeld</beschrijving> </voorbeeld>

Figuur 4.1: Difference voorbeeld, drie versies van XML-bomen.

chronizeren. Merk op dat XML-bomen zoals hierboven in principe geordend zijn (de volgorde tussen de elementen is bijvoorbeeld van belang bij XHTML documenten). Voor de opslag van veel gegevens (kalenders, bookmarks, contacten, ...) is deze volgorde, echter niet van belang. Voor zulke XML-documenten zijn de technieken van dit hoofdstuk ook van toepassing.

4.2 Synchronizeren van deterministische bomen

In dit hoofdstuk bedoelen we met een *boom* een ongeordende boom waarin elke knoop x een label $lab(x)$ draagt (bijvoorbeeld "voorbeeld" en "naam" in Figuur 4.1). De bladeren in de boom hebben bovendien ook nog een inhoud $content(x)$. (In Figuur 4.1 zouden "naam" en "beschrijving" bladeren zijn met inhoud respectievelijk "Voorbeeld één" en "Dit is een voorbeeld").

Eigenlijk hebben we in Hoofdstuk 2 al een algoritme gezien dat een beperkte klasse van ongeordende bomen, de zogenaamde *deterministische bomen*, kan synchronizeren.

Definitie 4.1. *Een boom T is* deterministisch *indien elke knoop x hoogstens één kind met hetzelfde label heeft. Formeel: voor alle kinderen y en z van x in T moet $y = z$ wanneer $lab(y) = lab(z)$.*

De bomen in Figuur 1.1 zijn bijvoorbeeld deterministich. Merk op dat we elke deterministische boom als een filesysteem kunnen bekijken (interne knopen zijn directories en bladeren zijn files). Als gevolg daarvan kunnen we deterministische bomen op een gelijkaardige manier als filesystemen synchronizeren. Algoritme 3 is een aanpassing van Algoritme 1 dat drie bomen A, O,

en B aanneemt en ofwel drie gesynchroniseerde bomen teruggeeft, ofwel een conflict meldt. De update-detectie en verzoeningsfase zijn hier met elkaar verweven. In het begin wordt het algoritme aangeroepen met de wortels van de drie bomen.

4.3 Synchronizeren van niet-deterministische bomen

In de praktijk zijn bomen echter niet altijd deterministisch. Beschouw bijvoorbeeld de auteur-databank getoond in Figuur 4.2. Deze is duidelijk niet-deterministisch. Om zulke bomen te kunnen synchronizeren zullen we *keys* gebruiken.

Keys zijn in het algemeen fundamenteel voor datamodellen en conceptueel design. In relationele databases hebben we bijvoorbeeld keys nodig om tuples te kunnen identificeren. Op precies dezelfde manier zullen we keys gebruiken om knopen in verschillende niet-deterministische bomen te kunnen identificeren.

Er zijn verschillende vormen van keyspecificaties voor boom-gestructureerde gegevens terug te vinden in zowel de XML standaard als de XML Schema standaard. We volgen in dit hoofdstuk de auteurs van (Buneman 01; Buneman 02) en beschouwen volgende hiërarchische *keys*. Deze keys geven een eenvoudige methode om *relative keys* te definiëren: keys die slechts binnen een bepaalde context van toepassing zijn. In de volgende sectie brengen we de essentiële concepten van keys aan (Buneman 01).

4.3.1 Hiërarchische keys

Definitie 4.2 (Padexpressies.)**.** *Een* padexpressie *is een (mogelijk lege) sequentie van knooplabels. We schrijven ϵ voor de lege sequentie en $P.Q$ voor de concatenatie van padexpressies P en Q.*

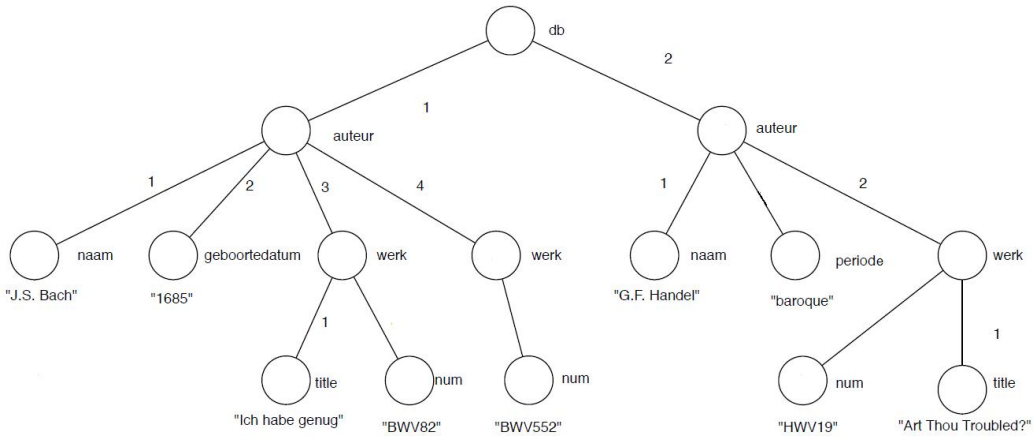
Gegeven een boom D selecteert een padexpressie P alle knopen x zodat het pad van x tot de wortel in D precies P is. (Het lege pad selecteert de wortel zelf). We schrijven $[\![P]\!]$ om de verzameling knopen in D aan te duiden die geselecteerd worden door P. We schrijven $n[\![P]\!]$ om de verzameling van knopen bereikt door te starten in knoop n en pad P te volgen, aan te duiden.

Om keys te kunnen definiëren is het eerst nodig om zogenaamde *waarde-gelijkheid* tussen twee knopen te definiëren.

Definitie 4.3 (Waarde-gelijkheid)**.** *De* waarde *van een blad x is het koppel $(lab(x), content(x))$. De* waarde *van een interne knoop y is het koppel*

Algoritme 3 Synchronizeren van deterministische bomen A, O, en B

function DETSYNC(knoop a in A, knoop o in O, knoop b in B)
  if $lab(a) = lab(o) = lab(b)$ **then**
    if a, o, en b zijn alle drie interne knopen **then**
      if er een a' kind van a is dat overeenkomt met een o' kind van o en aangepast is, en geen overeenstemmend element heeft in b **then**
        return CONFLICT
      end if
      if er een b' kind van b is dat overeenkomt met een o' kind van o en aangepast is, en geen overeenstemmend element heeft in a **then**
        return CONFLICT
      end if
      Voer DETSYNC recursief uit op elk tripel (a', o', b') met a' kind van a, o' kind van o, b' kind van b en $lab(a') = lab(o') = lab(b')$.
      Zij I_A de kindknopen a' van a wiens label niet in O voorkomen
      Zij I_B de kindknopen b' van b wiens label niet in O voorkomen
      Voer DETSYNC recursief uit op elk tripel (a', a', b') met $a' \in I_A$ en $b' \in I_B$ en $lab(a') = lab(b')$.
      Voor elke $a' \in I_A$ waarvoor we nog geen recursieve oproep hebben gedaan voegen we de deelboom met wortel a' in A als kind aan zowel o als b toe.
      Voor elke $b' \in I_B$ waarvoor we nog geen recursieve oproep hebben gedaan voegen we de deelboom met wortel b' in B als kind aan zowel o als a toe als o en a verschillend zijn, anders enkel aan a.
    else if a, o, en b zijn alle drie bladeren **then**
      if $content(a) = content(o)$ **then**
        stel $content(a)$ en $content(o)$ gelijk aan $content(b)$
      else if $content(b) = content(o)$ **then**
        stel $content(b)$ en $content(o)$ gelijk aan $content(a)$
      else if (**then**$content(a) = content(b)$
        stel $content$(o) gelijk aan $content$(a)
      else
        return CONFLICT
      end if
    else
      return CONFLICT
    end if
  else
    return CONFLICT
  end if
end function



Figuur 4.2: Een voorbeeld auteur-databank.

$(lab(y), \{w(z) \mid z \textit{ kind van } x\})$ *waar* $w(z)$ *de waarde van* z *aanduidt. Twee knopen* x *en* y *zijn* waarde-gelijk*, aangeduid met* $x =_w y$ *indien hun waardes gelijk zijn.*

Wanneer we een key definiëren, specificiëren we twee dingen: de verzameling op dewelke we een key definiëren (in relationele databases is dit een relatie) en de "attributen" welke samen unieke elementen identificeren in de verzameling. Dit is de motivatie voor onze centrale definitie van *keys*:

Definitie 4.4 (Key)**.** *Een key is koppel* $(Q, \{P_1, \ldots, P_k\})$ *met* $Q, P_1, \ldots, P_k$ *alle padexpressies.*

Het idee is dat een padexpressie Q een verzameling van knopen identificeert, welke we de *doelverzameling* (Eng.: target set) noemen waarop de key constraints moeten gelden. De padexpressie Q zelf noemen we het *doelpad* (Eng.: target path) en de verzameling $\{P_1, \cdots, P_n\}$ de *key paden.* Merk op dat er voor elke knoop $n \in [\![Q]\!]$ een verzameling van knopen $n[\![P_i]\!]$ is die bereikt worden door P_i vanuit n te volgen. Om aan de key te voldoen moet $n[\![P_i]\!]$ een singleton zijn, voor elke i. De keypaden beperken de doelverzameling als volgt: neem elk paar knopen $n_1, n_2 \in [\![Q]\!]$ en beschouw het paar van knopen bereikt door het volgen van het keypad P_i vanuit n_1 en n_2, $(n_1[\![P_i]\!], n_2[\![P_i]\!])$. Als voor elke $1 \leq i \leq k$ de knopen in $n_1[\![P_i]\!]$ en $n_2[\![P_i]\!]$ waarde-gelijk zijn, dan moeten n_1 en n_2 dezelfde knoop zijn. Beschouw bijvoorbeeld de volgende key definitie:

$$(\text{persoon.emp}, \{\text{naam.voornaam}, \text{naam.achternaam}\}).$$

Het doelpad persoon.emp identificeert een verzameling van knopen in de boom. Dit is de doelverzameling. Elk van deze knopen definieert een deelboom met een werknemer-label in de wortel. Binnen zo'n subtree moeten we precies één knoop met keypad naam.achternaam en één knoop met keypad naam.voornaam vinden. Elk paar verschillende knopen n_1, n_2 in de doelverzameling moet dan waarde-verschillen in ofwel de knopen bereikbaar via keypad naam.voornaam ofwel de knopen bereikbaar via naam.achternaam.

Een ander voorbeeld, merk op dat de boom in Figuur 4.2 voldoet aan de key (auteur, {naam}): er zijn twee knopen aan het einde van het doelpad auteur. Voor elke knoop is er een knoop in de verzameling van knopen bereikt door het keypad naam "J.S.Bach" en "G.F.Handel" te volgen. Deze knopen zijn niet waarde-gelijk. De boom voldoet echter niet aan de key (auteur,{geboortejaar}) aangezien geboortejaar enkel aanwezig is bij de eerste auteur en afwezig is bij de tweede auteur.

We gaan nu de formele semantiek van een key geven. Voor redenen die we later zullen geven, is het handig een key te definiëren ten opzichte van een gegeven knoop in de boom in plaats van ten opzichte van de wortelknoop.

Definitie 4.5 (Key)**.** *Een knoop n in een boom D voldoet aan een key $(Q, \{P_1, \ldots, P_k\})$ als*

1. *voor elke $n' \in n[\![Q]\!]$ en voor elke $1 \leq i \leq k$ de verzameling $n[\![P_i]\!]$ een singelton is;*

2. *voor elke n_1, n_2 in $n[\![Q]\!]$, indien voor elke $1 \leq i \leq k$ de knopen in $n_1[\![P_i]\!]$ en $n_2[\![P_i]\!]$ gelijk zijn, dan is $n_1 = n_2$.*

Merk hierbij op dat beide vormen van gelijkheid gebruikt zijn, zowel waarde-gelijkheid ($=_w$) als knoopgelijkheid ($=$).

We zeggen dat een boom voldoet aan een key indien diens wortel voldoet aan de key. Beschouw bijvoorbeeld de key $(A, \{B\})$. Volgende boom (voorgesteld als een XML-document) voldoet niet aan deze key. Het voldoet echter wel aan $(A, \{C\})$

```
<db><A><B>1</B><C>1</C></A> <A><B>1</B><C><2></C></A></db>
```

Enkele voorbeelden van keys, uitgedrukt ten opzichte van de root van een boom:

- (db.persoon, {id}): Elke persoon in de databank is uniek geïdentificeerd door de waarden van zijn id's.

- (persoon, $\{\epsilon\}$): Elke twee persoon knopen onmiddelijk onder de wortel hebben verschillende waarden (ϵ is het lege pad).

- (werknemer, {}): Een lege key. Dit wil zeggen dat er precies één werknemer-knoop is onmiddellijk onder de wortel.

De keys die we tot nog toe besproken hebben zijn *absolute* keys, die steeds ten opzichte van de wortel moeten gelden. Vaak willen we echter aangeven dat een key slechts binnen een bepaalde context moet gelden. Dit wordt mogelijk gemaakt met behulp van *relative keys*.

Definitie 4.6. *Een* relative key *is een koppel* $(Q, (Q', S))$ *met* Q *een pad-expressie en* (Q', S) *een key. We noemen* Q *het* contextpad. *Een document voldoet aan een relative key* $(Q, (Q', S)$ *als voor alle knopen* n *in* $[\![Q]\!]$, n *voldoet aan de key* (Q', S).

Met andere woorden, (Q, K) is een relative key als K een key is voor elke subboom met wortel in $[\![Q]\!]$. Enkele voorbeelden:

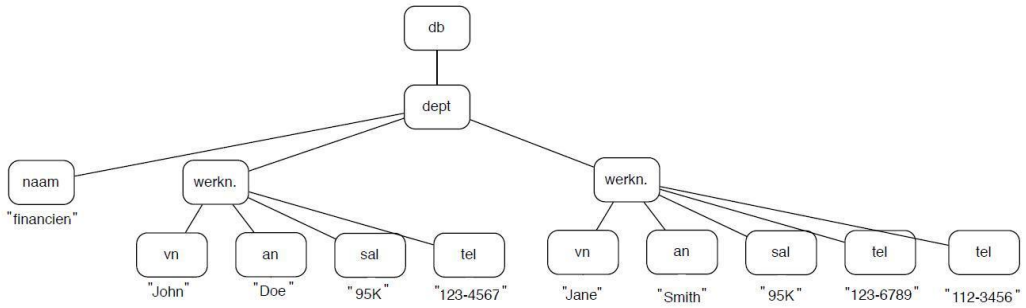
- (bib.boek.hoofdstuk, (vers,{number})). Een versnummer identificeert uniek een vers in een hoofdstuk.
- (bib.boek, (hoofdstuk, {nummer})). Hoofdstuknummers identificeren uniek een hoofdstuk in een boek.
- (bib, (boek, {naam}. Namen identificeren uniek boeken binnen bibliotheken.

Merk op dat in een relative key $(Q, (Q', S))$, Q start van de wortel en Q' start in een knoop in $[\![Q]\!]$. Het is om deze reden dat we de semantiek van keys ten opzichte van willekeurige knopen gedefinieerd hebben in Definitie 4.5.

4.3.2 Keys gebruiken om bomen deterministisch te maken

In deze sectie bespreken we hoe, gegeven een verzameling van relative keys Σ en een niet-deterministische boom D, elke gekeyde knoop in D van zijn keywaarde voorzien kan worden. (We noemen een knoop n *gekeyed* indien $n \in [\![Q.Q']\!]$ voor een relative key $(Q, (Q', \{P_1, \ldots, P_k\}) \in \Sigma)$.

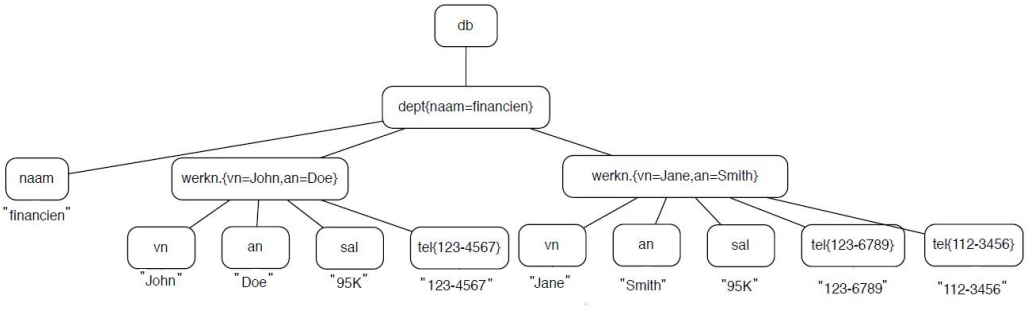
Beschouw de boom in Figuur 4.3 en veronderstel dat de naam van een departement gebruikt kan worden als key voor het departement zelf en dat binnen elk departement elke werknemer uniek geïdentificeerd kan worden door zijn voor- en achternaam. Merk dus op dat het mogelijk is dat twee werknemers dezelfde voor- en achternaam hebben zolang ze maar tot verschillende departementen behoren.



Figuur 4.3: Een boom zonder key-informatie.

Elke werknemer heeft ook maximum één salariswaarde en één of meerdere telefoonnummers. Dan toont Figuur 4.4 dezelfde boom geannoteerd aan de knopen. Werknemer-knopen zijn bijvoorbeeld geannoteerd met voor- en achternaamwaarden en telknopen zijn geannoteerd met hun subwaarde. Merk op dat wanneer we deze annotaties tot de labels van de knopen rekenen, we een *deterministische* boom bekomen. Bijgevolg kunnen we, wanneer we voldoende relative keys Σ kennen waaraan de bomen voldoen, niet-deterministische bomen synchronizeren door (1) de beschikbare relative key-informatie te gebruiken om de bomen te transformeren tot deterministische bomen en (2) vervolgens algoritme 3 te gebruiken. Dit veronderstelt natuurlijk wel dat de bomen "fully keyed" zijn ten opzichte van Σ: elke kind-knoop heeft ofwel een uniek label of is gekeyed ten opzichte van Σ. Voor veel boomgestructureerde data zoals bookmarks, kalenders, en zelfs XML-documenten kan men zonder veel moeite een verzameling relative keys vinden ten op zichte waarvan ze fully keyed zijn (Buneman 02).

Het algoritme dat we zullen gebruiken om de knopen met hun keywaarden te annoteren komt uit (Buneman 02).



Figuur 4.4: Een Boom met key-informatie.

Het annoteringsalgoritme Het annoteringsalgoritme doorloopt de boom in preorde en kijkt op elk moment naar de knopen die gekeyed moeten worden en naar knopen die *keypadwaarden* bevatten. Gegeven een key $(Q,(Q',\{P_1,\ldots,P_k\}))$ in Σ, is een knoop die op een pad $Q.Q'$ ligt een knoop die gekeyed moet worden en een knoop die op een pad $Q.Q'.P_i$ ligt (met $i \in [1,k]$) een knoop die een keypadwaarde bevat. Merk op dat wanneer we de boom in preorde doorlopen, we op het moment dat we de knoop die gekeyed moet worden (bijvoorbeeld knoop n via het pad $Q.Q'$) "buitengaan" (na het bezoeken van zijn kinderen), de knopen die keypadwaarden van n bevatten (de knopen via het pad $Q.Q'.P_i$) reeds bezocht zijn. Wanneer we een knoop ontmoeten die een keypadwaarde bevat, onthouden we de hele deelboom van de knoop en bewaren we de de waarde van de deelboom in de knoop.

Het algoritme hieronder beschrijft het annoteren van keyed knopen met hun keywaarde. We veronderstellen dat de boom D voldoet aan de key constraints opgelegd door de verzameling relative Σ, welke bestaat uit q keys

$$(C_1,(S_1,\{P_{11},\ldots,P_{1m_1}\})),\ldots,(C_q,(S_q,\{P_{q1},\ldots,P_{qm_q}\})).$$

Met $C_i.S_i$ duiden we de sequentie van label namen in C_i gevolgd door S_i van de i-de key aan en met $p(M)$ duiden we de sequentie van labels op een stack M aan. Het algoritme gaat als volgt:

1. Initialiseer een lege hoofdstack M.

2. Doorloop de knopen van de boom D in preorde.

   (a) Plaats het knooplabel op alle actieve stacks bij het binnengaan van een knoop.

   Als $p(M) = C_i.S_i$ voor bepaalde i, dan
   initialiseer een lege stack M_{ij} voor elk keypad, $P_{ij}, j \in [1,m_i]$.
   Voor elke stack M_{ij} overeenstemmend met keypad P_{ij} doe
   Als $p(M_{ij})$ gelijk is aan de sequentie van labels in keypad P_{ij}, dan
   start het onthouden van P^v_{ij}, de waarde van de huidige knoop. $(**)$

   (b) Voor het verlaten van een knoop,

   Als P^v_{ij} op dit moment onthouden wordt en P^v_{ij} is gelijk aan de sequentie van labels in keypad P_{ij}, dan
   stop het onthouden van P^v_{ij} $(**)$

verwijder P_{ij}^v en stack M_{ij}.
Als $p(M) = C_i.S_i$ voor bepaalde i, dan
annoteer de knoop met keypadwaarden $P_{i1}^v, \ldots, P_{im_i}^v$.
Pop alle actieve stacks.

De hoofdstack M wordt gebruikt om het pad van de wortel tot de huidige knoop bij te houden. Er zijn twee hoofdzaken die uitgevoerd worden tijdens het doorlopen van de boom in preorde. Wanneer een knoop eerst bezocht wordt, plaatsen we de naam van de knoop op alle actieve stacks en checken we of het pad van de huidige knoop gelijk is aan de concatenatie van labels in $C_i.S_i$ voor een een key i. De check tegen $C_i.S_i$ vertelt ons of de knoop die gekeyed wordt, overeenstemt met key i. Als dit zo is creëren we een lege stack voor elk pad in key i. Wanneer we een knoop bereiken op het einde van een keypad van i starten we het onthouden van de waarde (of deelboom) wanneer we de deelboom van deze knoop doorkruisen. Wanneer we een knoop opnieuw bezoeken na diens kinderen geprocessed te hebben, checken we of het onthouden van de waarde voltooid is. Als dit zo is, stoppen we met het onthouden van de waarde en verwijderen we het keypad van de stack. Als de huidige knoop die we willen verlaten een keyed knoop is, annoteren we de huidige knoop met al de waarden van zijn keypaden. Merk op dat alle relevante keypaden van een keyed knoop bezocht zijn voor de keyed knoop verlaten wordt. Hierdoor hebben we de garantie dat we alle waarden van de keypaden heben wanneer we een keyed knoop verlaten. Merk verder op dat er maximum $\sum_{i=1}^{q} m_i$ keypadstacks actief zijn op elk moment. De reden hiervoor is dat een keyed knoop van een pad l afgesloten moet worden voor een andere keyed knoop van pad l kan bezocht worden. De grootte van elke stack is maximum h waar h de hoogte van de boom is.

Analyse We tonen nu aan dat de uitvoeringstijd van het algoritme gedomineerd wordt door de grootte van de boom en de keyspecificatie. We analyseren eerst de stukken gemarkeerd met (**). In het extreme geval komt het keypad van een knoop voor binnen het keypad van een andere knoop. Deze situatie doet zich voor wanneer we overlappende keypaden hebben, meer bepaald wanneer we keys (A,(B,{$C.D$})) en ($A.B$, (C,{$D.E$})) hebben. De keypadwaarden onder het pad $D.E$ van knoop C komen voor binnen de keypadwaarde onder het pad $C.D$ van knoop B. Een naïeve implementatie die P_{ij}^v (een keypadwaarde) kopieert, kan de grootte van de boom verhogen met een factor N waar N het aantal knopen in de boom is. De totale grootte van alle keypadwaarden is $O(N^2)$ aangezien de totale grootte van alle buitenste keypadwaarden, tweede buitenste keypadwaarden en zo verder maximaal

respectievelijk $N, N-1, \ldots, 1$ is. (Om onze analyse eenvoudig te houden, veronderstellen we dat P_{ij}^v geïmplementeerd is als een pointer naar de knoop die de keypadwaarde bijhoudt).

Het algoritme scant D één keer, waarbij er in 2(a) en 2(b) acties ondernomen worden voor elke knoop. In 2(a) veronderstellen we dat de tijd voor elke padvergelijking proportioneel is ten opzichte van de lengte van het pad, een naïeve implementatie voor deze stap neemt $qh+h\sum_{i=1}^{q} m_i$ tijd. De check van de voorwaarde voor $p(M) = C_i.S_i$ voor een i neemt qh tijd in beslag en de check of een stack M_{ij} overeenstemt met keypad P_{ij} neemt $h\sum_{i=1}^{q} m_i$ tijd in beslag. Er zijn mogelijk efficiëntere alternatieven voor de implementatie van de check $p(M) = C_i.S_i$, zie bijvoorbeeld (Altinel 00; Diao 02). Aangezien we enkel de pointer naar de keypadwaarde onthouden wanneer noodzakelijk neem stap 2(b) maximaal $1 + h$ tijd in beslag met h de tijd nodig om te checken of $p(M_{ij}$ gelijk is aan P_{ij}. Het checkt ook of het huidige pad gelijk is aan $C_i.S_i$ voor bepaalde i en voert overeenstemmend annotaties uit. Dit neem maximaal $qh + max_i\, m_i$ tijd in beslag waar qh de tijd is nodig om te checken of het huidige pad gelijk is aan $C_i.S_i$ voor bepaalde i en $max\, m_i$ de tijd is nodig om een knoop te annoteren met al zijn keypadwaarden (of pointers). Het poppen van alle actieve stacks neemt ook maximaal $\sum_{i=1}^{q} m_i$ tijd in beslag. Dit maakt de totale tijd voor iedere knoop $qh + h\sum_{i=1}^{q} m_i + 1 + h + qh + max_i\, m_i + \sum_{i=1}^{q} m_i = O(h(\sum_{i=1}^{q} m_i + q))$. Als D N knopen heeft is de totale tijd $O(Nh(\sum_{i=1}^{q} m_i + q))$.

4.4 Archiveren van bomen

Een leuk neveneffect van de aanwezigheid van relative keys is dat we ze ook kunnen gebruiken om verschillende versies van eenzelfde boom compact in één groot archief op te slaan. Dat is vaak nuttig om later een specifieke versie te kunnen ophalen. Versiecontrolesystemen zoals CVS en subversion maken ook zulke archieven aan, maar die zijn gebaseerd op edit-scripts (zie Sectie 3.5 voor de definitie van een edit script.) Buneman et al. (Buneman 02) hebben echter aangetoond dat voor boomgestructureerde gegevens onderstaande aanpak efficiënter is.

Timestamps Om aan te geven welke knopen in welke versie voorkomen zullen we *timestamps* gebruiken. Een timestamp is een verzameling van versienummers. Zo duidt de verzameling $\{1, 3, 6\}$ aan dat de betreffende knoop aanwezig was in versies 1, 3, en 6. Voor het gemak gebruiken we intervals om een aaneenliggende sequentie van versienummers aan te duiden. Zo duidt het interval [1-3,5,7-9] dezelfde verzameling aan als $\{1, 2, 3, 5, 7, 8, 9\}$. Het ar-

chief is gewoon een boom waarin knopen geannoteerd zijn met timestamps. Kinderen erven de timestamps van hun vader. Bijvoorbeeld, in Figuur 4.5 zijn vier versies van een boom te zien.

Het corresponderende archief (met ook de geannoteerde keys) opgesteld na deze versies is te zien in Figuur 4.6. Hier zien we bijvoorbeeld dat de linkse departementknoop de timestamp van zijn ouder overerft. De [2-4] timestamp bij werknemerknoop "Jane Smith" geeft aan dat de knoop bestaat in versies twee tot en met vier. Het archief heeft een speciale "virtuele" wortel. Wanneer in versie 5 de boom bijvoorbeeld leeg zou worden, dan draag deze wortel de timestamp [1-5] en diens kind slechts de timestamp [1-4]. Merk op dat het, gegeven de timestamps, triviaal is om de originele boom van een bepaalde versie te herconstrueren.

Veronderstellingen Het archiveringsalgoritme veronderstelt dat de verzameling relative keys Σ waaraan de verschillende versies van de boom moeten voldoen, zelf aan bepaalde eigenschappen voldoet. Vooreerst moet Σ *transitief* en *insertion friendly* zijn. Intuïtief gezien betekent dit dat de ouders van gekeyde knopen zelf gekeyed zijn. Immers om een werknemerknoop op het pad db.dept.werknemer in een nieuwe versie correct te kunnen identificeren met een knoop in het archief van Figuur 4.6, moeten we eerst de overeenkomstige db- en dept-knopen kunnen identificeren. Dat zullen we enkel kunnen indien die knopen zelf ook gekeyed zijn. We geven hieronder de formele definities van transitiviteit en insertion-friendliness.

Ten tweede moet de verzameling Σ de gekeyde knopen tot op zekere diepte "bedekken". Om deze notie formeel te maken voeren we volgende definitie in.

Definitie 4.7. *Beschouw de verzameling van paden*

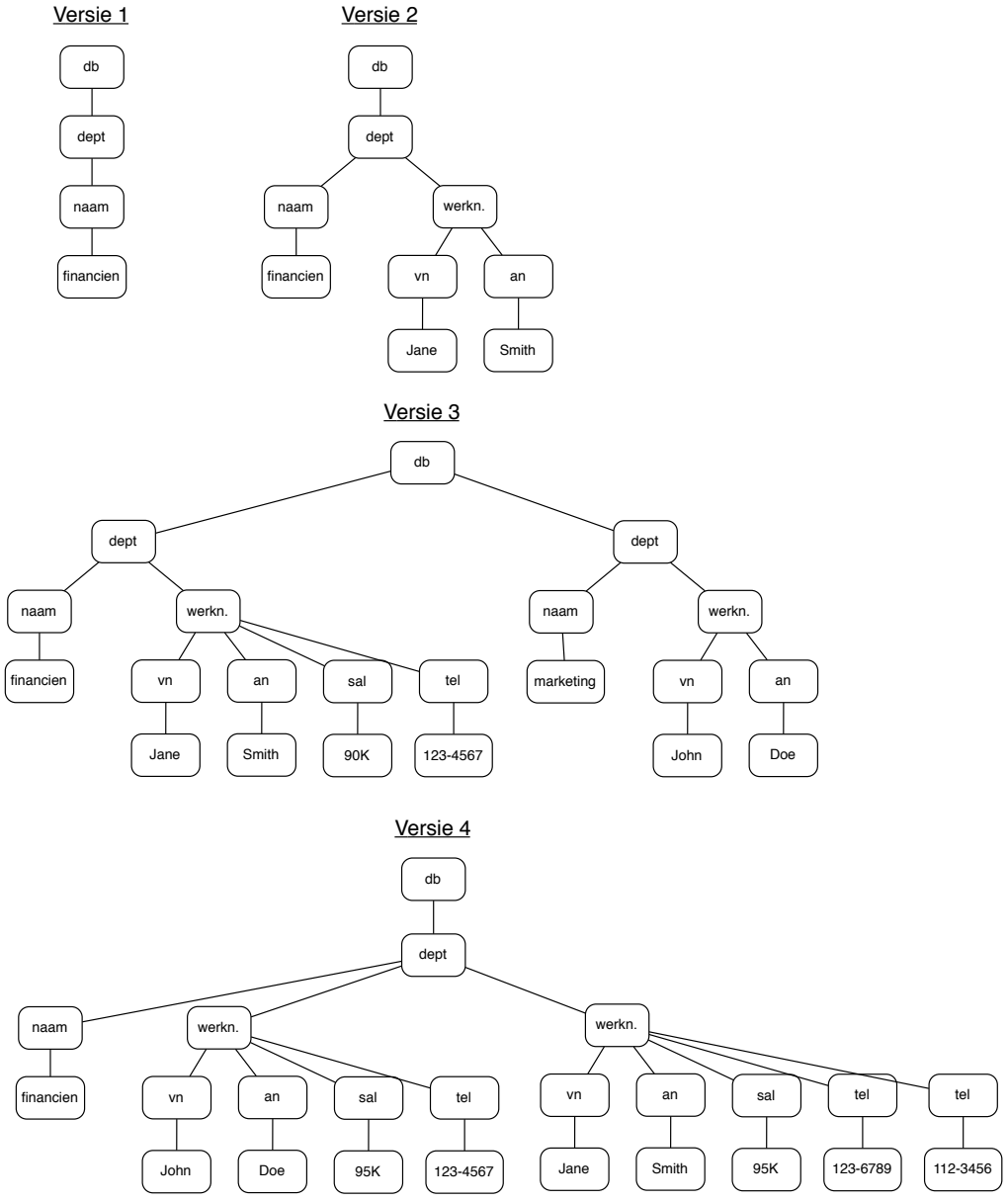
$$\{C.S \mid (C, (S, \{P_1, \ldots, P_k) \in \Sigma\}.$$

Een pad in deze verzameling is een grenspad *als het geen strikte prefix is van een ander pad in de verzameling. Een knoop in D is een* grensknoop *als de sequentie van labels van de wortel tot die knoop gelijk is aan een grenspad.*

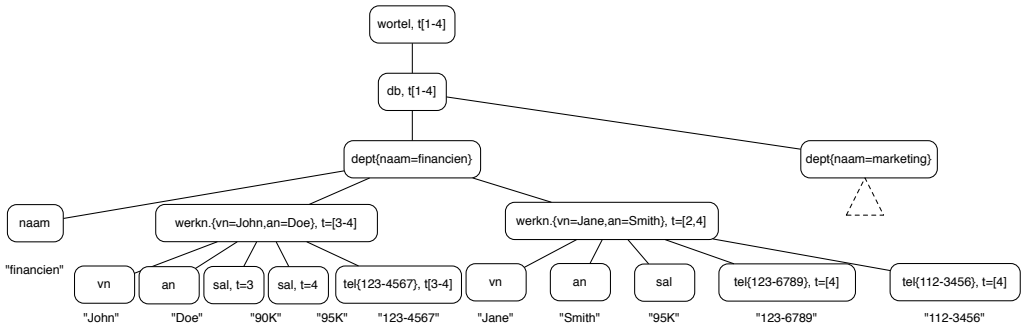
Met andere woorden een grensknoop is de diepst mogelijk gekeyde knoop.

Definitie 4.8. *Formeel gezien* bedekt Σ de gekeyde knopen *indien in elk geldig document, elke knoop die niet onder een grensknoop voorkomt zelf gekeyed is.*

In Figuur 4.6 bijvoorbeeld moet elk kind van de werknemerknopen gekeyed zijn aangezien werknemerknopen zelf geen grensknopen zijn.



Figuur 4.5: Sequentie van versies.



Figuur 4.6: Voorbeeld van een archief met timestamps.

Transitiviteit De bedoeling van relative keys is bepaalde componenten van een boom uniek te identificeren. Het is duidelijk dat een relative key zoals (bib.boek.hoofdstuk, (vers, {nummer})) alleen niet voldoende is om een bepaald vers in de Bijbel te identificeren. Intuïtief gezien echter, geloven we dat, als we een boeknaam, een hoofdstuknummer en een versnummer geven, we uniek een vers geidentificeerd hebben. Het is deze intuïtie die door transitiviteit geformalizeerd wordt.

Beschouw twee relative keys. We zeggen dat $(Q_1, (Q'_1, S_1))$ *onmiddellijk* $(Q_2, (Q'_2, S_2))$ *voorafgaat* als $Q_2 = Q_1.Q'_1$. We definiëren de *voorafgaanrelatie* als de transitieve sluiting van de onmiddellijke voorafgaanrelatie.

Definitie 4.9. *Een verzameling* Σ *van relative keys is* transitief *als voor elke relative key* $(Q_1,(Q'_1,S_1)) \in \Sigma$ *er een key* $(\epsilon, (Q'_2,S_2)) \in \Sigma$ *bestaat welke* $(Q_1,(Q'_1,S_1))$ *voorafgaat.*

Bijvoorbeeld, volgende verzameling relatieve keys is transitief:

$$(\epsilon, (\text{bib.boek}, \{\text{name}\})), (\text{bib.boek}, (\text{hoofdstuk}, \{\text{nummer}\}))$$

Volgende verzameling van keys is echter niet transitief:

$$(\epsilon, (\text{bib.boek}, \{\text{name}\})), (\text{bib.boek.hoofdstuk}, (\text{vers}, \{\text{nummer}\}))$$

Merk op dat elke transitieve verzameling minstens een absolute key van de vorm ϵ, $(Q'_2,S_2))$ moet bevatten.

Insertion friendliness Beschouw de volgende (transitieve) verzameling relatieve keys:

$$(\epsilon, (\text{universiteit}, \{\text{name}\})), (\text{universiteit}, (\text{dept.emplyee}, \{\text{emp-id}\}))$$

Om een werknemerknoop te *identificeren* in deze database, moeten we enkel een universiteitsnaam en een werknemer-id binnen deze universiteit specificeren. Echter, om een nieuwe werknemer toe te kunnen voegen aan de database moeten we duidelijk een department voor de werknemer specifiëren. Ondanks het feit dat de keyspecificatie transitief is, is er geen manier om een department te identificeren en is er dus ook geen manier om een werknemer toe te voegen. Dit geeft aanleiding tot onze uiteindelijke definitie van *insertion friendliness* als hieronder getoond: met insertion friendly keys kan men altijd ondubbelzinnig een knoop tussenvoegen in het "keyed" deel van de boom door te specifiëren waar de knoop, gebruikmakend van keys, moet tussengevoegd worden.

Definitie 4.10 (Insertion friendly keys)**.** *Een verzameling Σ van relative keys is insertion friendly als ze transitief en er voor elke $(Q_1,(Q_2.n,S_1))$ in Σ en n een label er een relative key $(Q'_1,(Q_2.n,S_1)) \in \Sigma$ bestaat met $\|Q'_2\| > 0$ en $Q_1.Q_2 = Q'_1.Q'_2$.*

Informeel geeft deze definitie ons de eigenschap dat elke knoop met een prefix langs het pad $Q_1.Q_2$ kan geïdentificeerd worden door bepaalde keys. De toevoeging van de volgende key maakt het voorgaande voorbeeld insertion friendly. In het bijzonder kunnen we nu, om een werknemer toe te voegen, specifiëren tot welk departement hij/zij behoort:

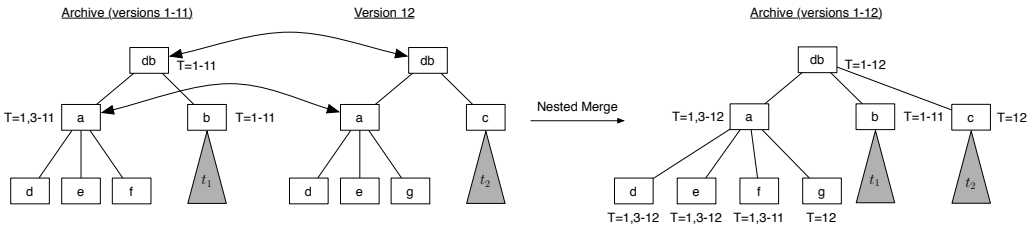
$$(\text{universiteit}, (\text{departement}, \{\text{dept-naam}\}))$$

Het archiveringsalgoritme Het archiveringsalgoritme is, net als het synchronisatiealgoritme, gebaseerd op het genest mergen van bomen. We beschrijven eerst het idee achter deze Nested Merge alvorens we het algoritme in detail beschrijven. Het idee is om recursief knopen in D (de nieuwe versie) met knopen in A (het archief) te mergen die dezelfde keywaarde hebben startend van in de wortel. Wanneer een knoop y van D merged met een knoop x van A wordt de timestamp van x uitgebreid met i, het nieuwe versienummer. De deelbomen van knopen x en y zijn dan recursief met elkaar gemerged. Knopen in D die geen corresponderende knoop in A hebben, worden simpelweg toegevoegd aan A met het nieuwe versienummer als timestamp. De timestamps van knopen in A die niet langer bestaan in de nieuwe versie D worden afgesloten, ze bevatten dus niet het versienummer i.

In Figuur 4.7 wordt het basisidee van Nested Merge geïllustreerd voor versie $i = 12$. Een pijl tussen een knoop in het archief en een knoop in versie 12 geeft aan dat ze overeenstemmend zijn, dus dat ze dezelfde keywaarde hebben. Overeenstemmende knopen worden gemerged in het nieuw archief

en hun timestamps worden uitgebreid met het laatste versienummer, namelijk 12. Aangezien de b-knoop niet langer bestaat in Versie 12, wordt zijn timestamp in het archief niet uitgebreid met het laatste versienummer. Aan de andere kant wordt, aangezien c voor het eerst bestaat in Versie 12, een nieuwe knoop c gecreëerd in het archief met timestamp t = [12]. Nested Merge detecteert dat de inhoud van de a-knoop in het archief en dat van versie 12 niet waarde-gelijk zijn. Hierdoor worden ze in het nieuwe archief afzonderlijk opgeslagen, ondanks dat we weten dat het eigenlijk over dezelfde knoop gaat.

Laat ons nu het algoritme formeel bespreken. We veronderstellen dat A en D reeds geannoteerd zijn met keys zoals besproken in Sectie 4.3. Zij i de kleinste integer groter dan alle versienummers vermeld in het archief. We veronderstellen dat D voldoet aan de verzameling relative keys Σ. We veronderstellen tevens dat het archief A een alleenstaande wortelknoop r_a bevat wanneer er geen enkele versie aanwezig is. (Het archief is dus nooit leeg.) Voor elke knoop x in A, laat time(x) de timestamp geannoteerd in knoop x aanduiden. Indien knoop x nog geen timestamp heeft toegewezen gekregen zeggen we dat time(x) niet bestaat. De timestamp van de wortel bestaat altijd: vóór de eerste versie gemerged wordt in A is time(r_A) de lege verzameling. Laat lab(x) in deze sectie het volledige label van knoop x aanduiden, zijn keywaarde bevattend maar zonder zijn timestamp annotaties. Als voorbeeld, het label van de werkemerknoop van John Doe in Figuur 4.6 is werknemer{fn=john,ln=Doe}. In het algemeen heeft een label van een knoop de vorm $l(p_1 = v_1, \ldots, p_k = v_k)$. De labels van twee knopen $l(p_1 = v_1, \ldots, p_k = v_k)$ en $l'(p_1 = v'_1, \ldots, p_k = v'_k)$ zijn gelijk als de overeenstemmende labels identiek zijn ($l = l'$) en hun keywaarden hetzelfde zijn ($v_1 =_w v'_1, \ldots, v_k =_w v'_k$). We voegen aan D een nieuwe wortel r_D toe wiens label identiek is aan $lab(r_A)$. De oude wortel van D wordt een kind van r_D. Laat kinderen(x) de verzameling van kinderknopen van x aanduiden. In het begin roepen we Nested Merge algoritme, getoond in Algoritme 4, aan met de volgende argumenten: NestedMerge($r_A, r_D, \{\}$). Het laatste argument bevat



Figuur 4.7: Illustratie van Nested Merge.

de overgeërfde timestamp, initieel leeg.

Algoritme 4 bepaalt eerst de huidige timestamp. Het is i toegevoegd aan time(x) als time(x) bestaat. Anders is het de van de ouder overgeërfde timestamp. Merk op dat time(r_A) altijd bestaat. Het is een eigenschap van het algoritme dat lab(x) = lab(y) als NestedMerge(x,y,T) aangeroepen wordt. Vervolgens gaat het algoritme checken of y een grensknoop is.

Grensknopen worden speciaal behandeld omdat er geen keyed knopen onder grensknopen zijn. Dus de recursieve merging van identieke knopen is niet meer van toepassing voor de afstammelingen van grensknopen. Nu zijn er twee mogelijkheden:

1. Ofwel is x een blad. Aangezien x zelf een grensknoop is, wordt diens waarde bepaald door diens naam en content. Het is een eigenschap van het algoritme dat lab(x) = lab(y) als NestedMerge(x,y,T) aangeroepen wordt, moet y dan zelf ook wel een blad met dezelfde waarde zijn. In dat geval gebeurt er dus niets, behalve dat lijnen 2 t.e.m. 4 de timestamp van x verhoogd hebben.

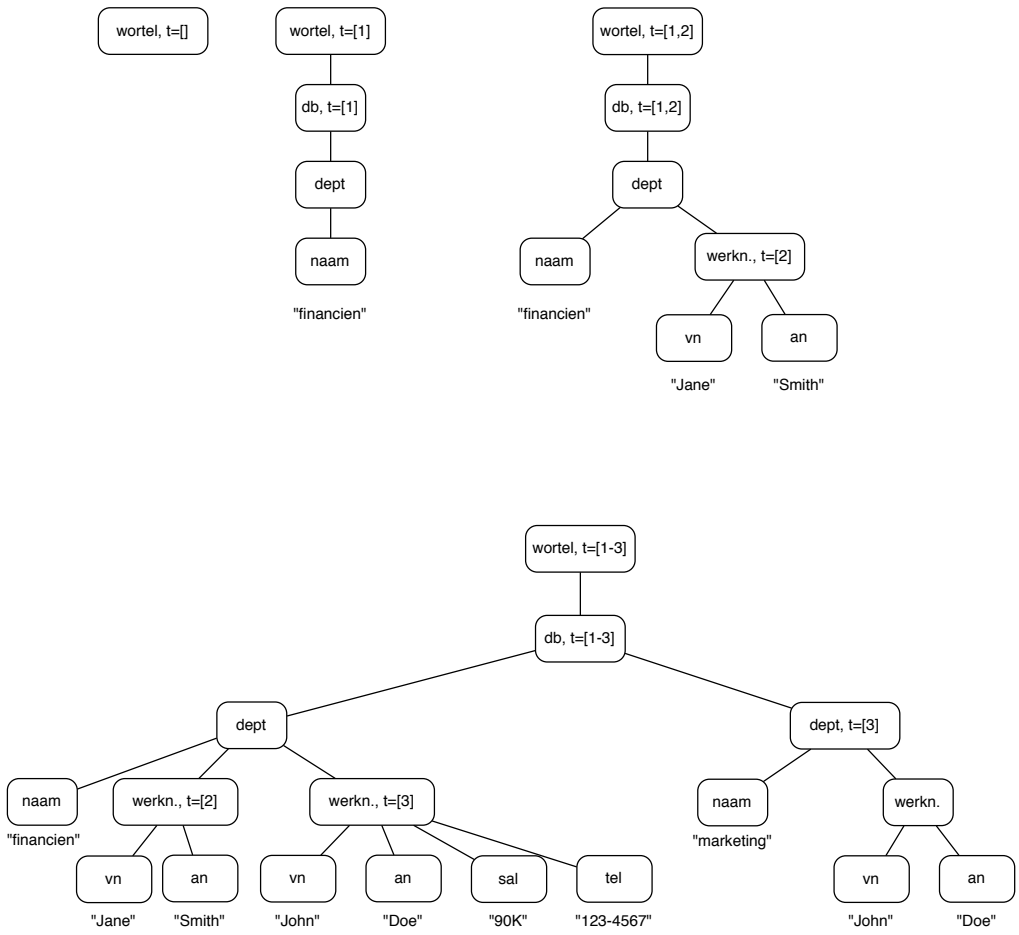
2. Ofwel is x een interne knoop. De kinderknopen van elke grensknoop hebben de eigenschap dat ze ofwel allemaal knopen met een toegewezen timestamp zijn ofwel geen van hen een timestamp toegewezen heeft. (We spreken van *timestampknopen* in wat volgt om knopen met een toegewezen timestamp aan te duiden.) De overgang van geen timestamp kinderknopen naar alle kinderknopen zijn timestampknopen gebeurt wanneer de nieuw versie y zó is dat de waarde van y verschilt van de waarde van x in het archief. In dat geval geven we aan elke kind van x dat geen oveereenkomstig kind van y heeft de timestamp $T - \{i\}$. Aan alle andere kinderen van x geven we de timestamp T. Alle kinderen van y die zonder overeenkomstige kind van x voegen we toe als kind van x met timestamp $\{i\}$.

   Indien alle kinderen van x wel timestampknopen zijn, dan wordt aan de timestamp van die kinderen die een overeenkomstig kind in y hebben i toegevoegd. Alle kinderen van y die zonder overeenkomstige kind van x voegen we toe als kind van x met timestamp $\{i\}$.

Als y geen grensknoop is, "verdelen" we de knopen in kinderen(x) en kinderen(y) in drie verzamelingen. (1) De eerste verzameling, XY bevat paren van knopen in kinderen(x) en kinderen(y) respectievelijk met gelijke keywaarden. Merk op dat door de eigenschap van keys, voor elke knoop in kinderen(x) er maximaal één knoop in kinderen(y) kan gevonden worden met gelijke waarde. (2) De tweede en derde verzamelingen zijn X' en Y'.

Algoritme 4 Archiveringsalgoritme

1: **procedure** NESTEDMERGE(x, y, T)
2: **if** time(x) bestaat **then**
3: voeg i toe aan time(x),
4: stel T gelijk aan time(x).
5: **end if**
6: **if** y een grensknoop is **then**
7: **if** $x \neq_w y$ **then**
8: **if** geen van kinderen(x) is een timestampknoop **then**
9: geef elk kind van x de timestamp $T - \{i\}$
10: **end if**
11: **for all** y' in kinderen(y) **do**
12: **if** er bestaat x' in kinderen(x) zodat $x' =_w y'$ **then**
13: voeg i toe aan time(x')
14: **else**
15: voeg y' toe als kind aan x met timestamp $\{i\}$
16: **end if**
17: **end for**
18: **else**
19: **if** alle knopen in kinderen(x) zijn timestampknopen **then**
20: voeg i toe aan time(x')
21: **end if**
22: **end if**
23: **else**
24: Laat $XY = \{(x', y') | x' \in$ kinderen(x), $y' \in$ kinderen(y), $lab(x') = lab(y')\}$.
25: Laat X' de rest van de knopen in kinderen(x) die niet voorkomen in XY aanduiden en Y' de rest van de knopen in kinderen(y) die niet voorkomen in XY aanduiden.
26: **for all** $(x', y') \in XY$ **do**
27: (a) NestedMerge(x', y', T)
28: **end for**
29: **for all** $x' \in X'$ **do**
30: (b) **if** time(x') nog niet bestaat dan stel time(x') $= T - \{i\}$.
31: **end for**
32: **for all** $y' \in Y'$ **do**
33: (c) Stel time(y') $= \{i\}$ en voeg y' toe als kindknoop van x.
34: **end for**
35: **end if**
36: **end procedure**



Figuur 4.8: Status van het archief als versies 1-3 gemerged zijn. Keys zijn hier achterwege gelaten.

De verzamelingen X' (respectievelijk Y') bestaan uit knopen in kinderen(x) (respectievelijk kinderen(y)) waarvoor er geen knopen in kinderen(y) (respectievelijk kinderen(x) bestaat met een gelijke keywaarde.

Nested Merge wordt recursief opgeroepen op paren van knopen in XY, terwijl ze de huidige timestamp T overerfen. Om te verzekeren dat knopen van X' niet langer bestaan op tijdstip i, annoteert timestamp T (zonder i) knopen van X' als ze nog niet bevat zijn in timestamp annotaties die vroeger eindigen dan i. Deelbomen geworteld in knopen Y' zijn toegevoegd aan deelbomen van x en ze zijn geannoteerd met timestamp i aangezien ze pas beginnen te bestaan op tijdstip i.

Figuur 4.8 samen met Figuur 4.6 tonen de evolutie van het archief als versie één tot vier aan 4.5 toegevoegd worden.

Hoofdstuk 5

Detecteren van veranderingen in geordende bomen

5.1 Inleiding

Alhoewel we de technieken uit Hoofdstuk 4.3 kunnen gebruiken om *ongeordende* bomen te synchronizeren is het voor bepaalde soorten gegevens (bijvoorbeeld (X)HTML, Docbook, ...) van belang om ook *geordende* bomen te kunnen synchronizeren. Het feit dat de kinderen van een knoop geordend zijn in zulke bomen bemoeilijkt de synchronisatietaak echter significant. Lindholm bijvoorbeeld (Lindholm 04; Ranta 01) beschrijft twee complexe synchronisatie-algoritmes.

Deze algoritmes werken essentieel op een gelijkaardige manier als Diff3: om drie geordende bomen A, O en B te synchronizeren (met O de versie bij laatste synchronisatie en A en B de aangepaste replica's) wordt er eerst een *matching* tussen O en A en een matching tussen O en B berekend. Aan de hand van deze matching worden A, O, en B dan met elkaar verzoend.

In deze context is een matching als volgt gedefinieerd.

Definitie 5.1. *Een* boommatching *tussen twee geordende bomen O en A is een binaire relatie M op de knopen van O en A zodat voor elke x in O er hoogstens één y is in B met $(x, y) \in M$.*

Met andere woorden, M is een partiële injectie van O naar A. Wanneer $(x, y) \in M$, dan zeggen we dat x met y matcht. Intuïtief gezien betekent dit dat x en y "dezelfde" knoop zijn. De knopen van O die niet voorkomen in M zijn dus verwijderd in A; de knopen van A die niet in M voorkomen zijn toegevoegd.

In dit hoofdstuk beschrijven we twee algoritmes om matchings voor geordende bomen op te stellen: LaDiff (Chawathe 96a) en XyDiff (Cobena 01).

LaDiff (Chawathe 96a) is ontworpen voor geordende bomen in het algemeen; XyDiff is specifiek ontworpen voor XML-documenten.

5.2 LaDiff

LaDiff focust zich op toepassingen waarbij er geen keys of object-id's zijn die kunnen gebruikt worden om knopen in de ene versie met de andere versie te matchen. We gebruiken de term *keyless* voor zulke data.

Wanneer we in deze sectie over een boom spreken bedoelen we een geordende boom waarvan alle knopen x een label $l(x)$ en een waarde $v(x)$ hebben.

Bij het vergelijken van keyless data is er vaak meer dan één manier om objecten te matchen. Om dit probleem op te lossen hebben we nood aan matching criteria waaraan een matching moet voldoen om goed of geschikt te zijn. Over het algemeen hangen deze matching criteria af van het beschouwde domein. In (Chawathe 96a) gebruiken ze de volgende criteria.

5.2.1 Matching criteria voor keyless data

Het eerste matching criterium zorgt ervoor dat knopen die te verschillend zijn niet met elkaar gematcht worden.

Matching criterium 1 Zij T_1 en T_2 twee bomen waarin alle knopen zowel een label als een waarde hebben. Dan mag voor bladeren $x \in T_1$ en $y \in T_2$, (x,y) enkel in een matching zijn als $l(x) = l(y)$ en $\mathit{compare}(v(x), v(y)) \leq f$ voor een bepaalde parameter f zodat $0 \leq f \leq 1$.

Hierbij stelt $l(x)$ het label van x voor en $v(x)$ de waarde van x, *compare* is de functie die de update van een blad berekent. Deze functie vraagt twee argumenten, de oude en nieuwe waarde, en geeft een waarde terug in het domein $[0, 2]$. Wanneer de twee waarden erg gelijkend zijn wordt een kost kleiner dan één teruggegeven. Wanneer de waarden erg verschillend zijn, wordt een kost groter dan één teruggeven.

We willen ook interne knopen die niet veel gemeenschappelijk hebben uitsluiten. Hierbij is het aantal gemeenschappelijke afstammelingen een meer natuurlijke notie van de waarde (welke dikwijls nul is). Laat ons zeggen dat een interne knoop x een knoop y bevat als y een blad afstammend van x is, en laat $|x|$ het aantal bladeren van x aangeven. Het volgende criterium zegt dat interne knopen x en y enkel mogen matchen als ten minste een zeker percentage van hun bladeren matchen.

Matching criterium 2 Beschouw een matching M met $(x, y) \in M$, waar x een interne knoop is in T_1 en y een interne knoop is in T_2. Definieer $common(x, y) = \{(w, z) \in M | x \text{ bevat } w, \text{ en } y \text{ bevat } z\}$. Dan moet $l(x) = l(y)$ en $\frac{|\,common(x,y)|}{\max(|x|,|y|)} > t$ voor een bepaalde t die voldoet aan $\frac{1}{2} \leq t \leq 1$

5.2.2 Bijkomende veronderstellingen

We zijn op zoek naar de "beste" matching M die aan bovenstaande matching criteria voldoet zodat er geen matching M' bestaat die ook voldoet en die meer knopen matcht. Jammer genoeg, de enige gekende manier om beste matchings zoals hierboven gedefinieerd te vinden is een algoritme in tijd $O(n^2)$ loopt (Zhang 95). Wanneer we echter veronderstellen dat de bomen aan de volgende twee eigenschappen voldoen, dan kunnen we deze beste matchings sneller vinden.

Veronderstelling 1 De verzameling van labels die in de bomen mogen voorkomen is gekend, en er bestaat een orde $<_l$ op deze labels zodat een knoop met label l_1 enkel kan voorkomen als afstammeling van een knoop met label l_2 als $l_1 <_l l_2$.

De volgende veronderstelling geeft aan dat de *compare* functie een goede functie is om onderscheid te kunnen maken tussen bladeren. De veronderstelling zegt dat voor elk blad s in het oude document er maximum één blad s' is in het nieuwe document die *close* is met s en vice versa. Beschouw bijvoorbeeld een world-wide web filmdatabase die films, acteurs, producers, etc. bevat. Een boom die dit voorstelt kan bijvoorbeeld filmtitels als bladeren hebben. Deze veronderstelling zegt dat wanneer we nu twee momentopnamen van deze data vergelijken een filmtitel in de ene momentopname slechts gelijkaardig mag zijn (Eng.:*closely resemble*) aan één filmtitel in de andere momentopname.

Veronderstelling 2 Voor elk blad $x \in T_1$, is er maximum één blad $y \in T_2$ zodat, $compare(v(x), v(y)) \leq 1$. Gelijkaardig is er voor elk blad $y \in T_2$ maximum één blad $x \in T_1$, zodat $compare(v(x), v(y)) \leq 1$.

Deze veronderstelling geldt uiteraard niet altijd. Wanneer we bijvoorbeeld wetteksten beschouwen waarbij de teksten erg gelijkend zijn op elkaar, zal dit niet gelden. Dit vormt niet automatisch een probleem. Wanneer aan de veronderstelling voldaan is, geeft het algoritme dat we zodadelijk gaan

beschrijven de optimale matching. Wanneer er niet aan voldaan is, krijgen we een suboptimale, maar nog steeds correcte oplossing.

Gegeven onze veronderstellingen kunnen we aantonen dat de unieke maximale matching ook de beste matching is. Stelling 5.1 zegt dit. Het bewijs kan teuggevonden worden in (Chawathe 96b).

Stelling 5.1 (Unieke maximale matching)**.** *Als T_1 en T_2 bomen zijn die voldoen aan veronderstellingen 1 en 2, dan is er een maximale matching M van de knopen van T_1 en T_2 die voldoet aan matching criteria 1 en 2. Meer zelfs, M is ook de unieke beste matching die voldoet aan de matching criteria.*

5.2.3 Algoritme

De veronderstellingen uit de vorige sectie staan ons toe een simpel algoritme te ontwikkelen om de best matching te berekenen. Dit algoritme wordt *Match* genoemd en wordt voorgesteld in (Chawathe 96b) en loopt in kwadratische tijd. In (Chawathe 96a) hebben ze echter een algoritme gegeven dat sneller is, *FastMatch* genoemd. Het is het *FastMatch*-algoritme dat wij zullen bespreken.

Het *FastMatch*-algoritme gebruikt een functie *equal* om knopen te vergelijken. Voor bladeren geldt $equal(x,y)$ is true als en slechts als $l(x) = l(y)$ en $compare(v(x), v(y)) \leq f$, met f een parameter tussen 0 en 1. Voor interne knopen $equal(x,y)$ is true als enkel als $l(x) = l(y)$ en $\frac{|\,common(x,y)|}{\max(|x|,|y|)} > t$ met $t > \frac{1}{2}$ een parameter. Laat $chain_{T_1}(l)$ de lijst van knopen in T_1 met label l aanduiden, bekomen door een left-to-right preorder traversal van T_1. Het algoritme gaat nu als volgt:

1. $M \leftarrow \phi$
2. Voor elke blad l doe
   (a) $S_1 \leftarrow chain_{T_1}(l)$.
   (b) $S_2 \leftarrow chain_{T_2}(l)$.
   (c) $lcs \leftarrow LCS(S_1, S_2, equal)$.
   (d) Voor elk paar knopen $(x, y) \in lcs$, voeg (x, y) toe aan M.
   (e) Voor elke ongematchte knoop $x \in S_1$, als er een ongematchte knopen $y \in S_2$ is zodat $equal(x, y)$ dan
      - Voeg (x, y) toe aan M.
      - Markeer x en y als matched.
3. Herhaal stap $2(a) - 2(e)$ voor elke interne knoop label l.

Hierbij is *LCS* de longest common subsequence routine besproken in Hoofdstuk 3, waarbij we *equal* in plaats van label gelijkheid gebruiken om knopen te vergelijken. Knopen die nog steeds niet gematcht zijn na het aanroepen van LCS worden behandeld in lineaire tijd.

In (Chawathe 96b) wordt er aangetoond dat de uitvoeringstijd van *FastMatch* propertioneel is t.o.v. $(ne + e^2)c + 2 \ln e$ met n het totaal aantal bladeren; c de gemiddelde kost voor het vergelijken van twee bladeren (gebruikmakend van de *compare* functie); l het aantal interne knoop labels; en e de grootte van de uiteindelijke matching.

Zoals gezegd hebben we enkel als aan Veronderstelling 2 is voldaan de garantie dat het algoritme de beste oplossing geeft. Wanneer er niet aan deze veronderstelling voldaan is, kan het zijn dat het algoritme slechts een suboptimale oplossing geeft. We beschrijven nu een postprocessing stap die in veel gevallen de suboptimale oplossing naar een optimale converteert. Top down beschouwen we elke boomknoop x op zijn beurt. Laat y de partner van x overeenstemmend met de huidige matching zijn. Voor elk kind c van x die gematcht is met een knoop c' zodat *parent*$(c') \neq y$ checken we of we c kunnen matchen met een kind c'' van y, zodat *compare*$(c, c'') \leq f$, waar f de parameter is gebruikt in Matching Criterium 1. Als dit zo is, veranderen we de huidige matching zodat c c'' matcht. Deze postprocessing fase verwijdert een aantal suboptimale oplossingen geïntroduceerd als Veronderstelling 2 niet geldt voor alle knopen.

Zelfs met deze postprocessing stap is het nog steeds mogelijk dat we een suboptimale oplossing hebben. FastMatch begint met het matchen van bladeren en gaat dan over naar hogere levels in de boom op een bottom-up manier. Met deze benadering, kan een mismatch op een lager level gepropageerd worden en een mismatch op één of meerdere hogere levels veroorzaken. De postprocessing stap corrigeert alle mismatches behalve die van lagere levels naar hogere levels zijn gepropageerd.

5.3 XyDiff

XyDiff (Cobena 01) is geschikt voor het vinden van matchings tussen XML-bomen. Dit algoritme vindt matchings tussen grote gemeenschappelijke deelbomen van twee documenten en propageert deze matchings. Het wordt het *BULD Diff algoritme* genoemd. Hierbij staat BULD voor Bottom-Up,Lazy-Down propagatie. Dit omdat matchings bottum-up gepropageerd worden en meestal lazy neerwaarts. Deze benadering wordt boven andere benaderingen verkozen omdat het toelaat de matching in lineaire tijd te berekenen. Eerst geven we de intuïtie en daarna de meer gedetailleerde beschrijving.

5.3.1 Intuïtie

We zullen het algoritme eerst illustreren aan de hand van een voorbeeld zodat de intuïtie duidelijk wordt. We nemen twee versies van een document V_1 en V_2, waarbij V_2 de laatste nieuwe versie is.

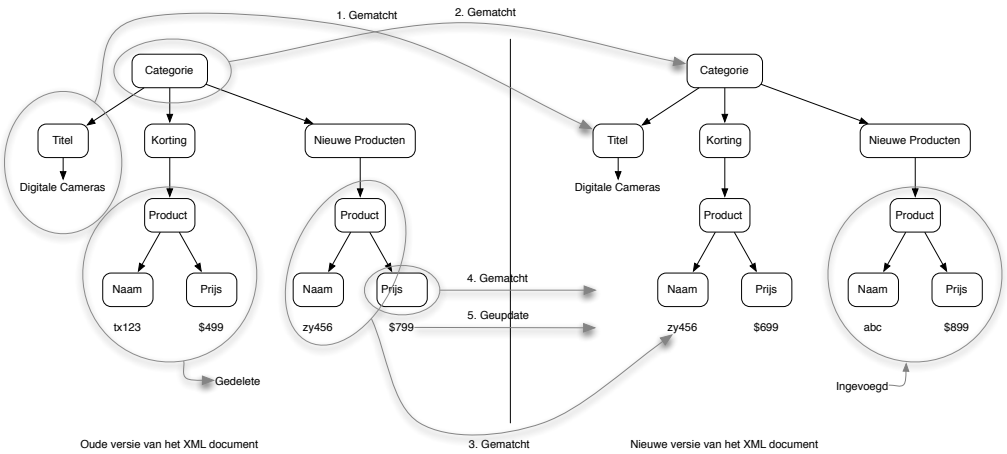
Het algoritme probeert steeds de grootst mogelijke identieke delen in beide versies met elkaar te matchen. Om dat te kunnen doen, beginnen we met in een associatieve array voor elke subtree van het oude document V_1 een unieke signatuur (hashwaarde) te berekenen. Als er ID-attributen gedefinieerd zijn in de DTD van de XML documenten dan matchen we de overeenstemmende knopen met gelijke ID waarden, en propageren we deze matchings in een eenvoudige bottum-up en top-down stap.

Vervolgens beschouwen we elke deelboom in V_2, beginnend met de grootste, en gaan we kijken of hij identiek is aan één van de bomen waar we een hashwaarde aan hebben toegekend. Als dit het geval is matchen we beide deelbomen. We matchen dus elke knoop in de deelboom in V_1 met de overeenstemmende knoop van de deelboom in V_2. In Figuur 5.1 bijvoorbeeld vinden we geen identieke deelboom voor de deelboom met wortel Categorie, maar de deelboom hiervan met wortel Titel kan wel gematcht worden.

Vervolgens proberen we om de ouders van reeds gematchte deelbomen te matchen. We doen dit wel enkel als ze dezelfde labels hebben. Het is duidelijk dat er op deze manier een risico onstaat om verkeerde matches te detecteren. We proberen dit te voorkomen door de propagatie van de bottum-up matching te controleren op basis van de lengte van het pad naar de voorouder en het "gewicht" van de matchende deelbomen. Zo zal bij een grote deelboom de propagatie doorgaan tot aan de wortel. Dit is logisch aangezien een grote deelboom voor een groot stuk de boom in de ouderknoop bepaald en er dus ook voor zorgt dat ze minstens voor een groot stuk gelijkend zijn. Bij een kleine deelboom is het dan weer mogelijk dat zelfs zijn ouder niet gematcht wordt.

Het feit dat ouders op deze manier gematcht worden helpt ons nieuwe matchings te vinden tussen de afstammelingen aangezien paren van zulke deelbomen als goede matchkandidaten kunnen beschouwd worden. De matching van grote identieke deelbomen kan dus helpen om zusterdeelbomen te matchen die nauwelijks verschillen. In het voorbeeld in Figuur 5.1 zien we dat de deelboom *Naam/zy456* matcht. De ouder *Product* matcht hierdoor ook. Aangezien de ouders gematcht worden, worden de deelbomen in de *Prijs* uiteindelijk ook gematcht, alhoewel ze verschillend zijn. Dit laat toe te detecteren dat de prijs geüpdatet is.

Wanneer beide ouders een enkel kind hebben met een gegeven label propageren we de match onmiddellijk. In het andere geval propageren we de



Figuur 5.1: Matching subtrees

matching niet onmiddellijk (lazy down). Verdere matchings (van kleinere deelbomen) resulteren uiteindellijk in matchings met een kleine kost.

De trage benedenwaardse propagatie die de schrijvers van (Cobena 01) ingevoerd hebben is een belangrijk verschil t.o.v. voorgaand werk over dit topic. Merk op dat als twee gematchte knopen m en m' kinderen hebben met hetzelfde label l, we m keer m' paren moeten beschouwen. Dit uitvoeren zou kwadratische tijd nodig hebben.

Zoals gezegd start het algoritme met de grootste deelboom in V_2. De eerste matchings zijn dan duidelijk, omdat het erg onwaarschijnlijk is dat er meer dan één grote deelboom in V_1 is met dezelfde hashwaarde. Echter, wanneer het algoritme uitgevoerd wordt op de volgende kleinere deelbomen dan zijn er vaak meerdere deelbomen in V_1 met dezelfde hashwaarde. We zeggen dan dat dat deze deelbomen kandidaten zijn om de beschouwde deelboom in V_2 te beschouwen. Op dit punt gebruiken we de voorafgaande matches om de beste kandidaat te kiezen, door te kijken welke kandidaat het dichtst aansluit met de bestaande matches. Het is typisch zo dat als de ouder van één van de kandidaten reeds matcht met de ouder van de beschouwde knoop, dat dit dan de beste kandidaat is.

Na dit deel van het algoritme hebbben we alle knopen van V_2 beschouwd en misschien gematcht. Er zijn twee redenen waarom een knoop geen matching kan hebben: ofwel omdat ze nieuwe data bevat die toegevoegd is aan het document, of omdat we een matching gemist hebben. De reden waarom we een matching kunnen missen is omdat, op het tijdstip dat de knoop beschouwd werd, er niet voldoende informatie was of er geen reden was om een matching toe te laten met één van de kandidaten. Maar gebaseerd op de nu

beschikbare kennis kunnen we nu een optimalisatiepas doen om de verworpen knopen opnieuw te beschouwen.

Bijvoorbeeld, in Figuur 5.1 zijn de korting knopen nog niet gematcht omdat de inhoud van hun deelbomen volledig veranderd is. Maar in de optimalisatiefase kunnen we detecteren dat dit de enige deelboom is van de knoop Categorie met dit label, dus we kunnen deze matchen.

Eens dat er geen matchings meer kunnen plaatsvinden, kunnen we besluiten dat de niet-gematchte knopen in V_2 (V_1) toegevoegd zijn in V_2 (V_1). In Figuur 5.1 kunnen bijvoorbeeld de deelbomen voor de producten $tx123$ en *abc* niet gematcht worden en zij kunnen dus beschouwd worden als respectievelijk verwijderde en toegevoegde data. Tenslotte moeten we nog voor elke matching node beslissen of de knopen op de juiste plaats zitten, of ze verplaatst zijn. Dit is ook nog een intensieve taak.

5.3.2 Gedetailleerde beschrijving

We kunnen het algoritme formeel in vijf fasen indelen. We zullen nu elke fase apart bespreken.

Fase 1 Gebruik ID-attribute informatie. In één doorloping van elke boom registreren we knopen die uniek geïdentificeerd zijn met hun ID-attribuut, gedefinieerd in de DTD van de documenten. Het bestaan van een ID-attribuut voor een gegeven knoop geeft een unieke voorwaarde om de knoop te matchen. Zijn matching moet namelijk dezelfde ID-waarde hebben. Als zo een paar van knopen gevonden is in het andere document worden ze gematcht. Andere knopen met ID-atttributen kunnen niet gematcht worden, zelfs niet in latere fasen. Vervolgens propageren we de gevonden matchings in een eenvoudige bottom-up en top-down doorloping. Merk op dat als ID-attributen frequent gebruikt worden in de documenten, de meeste matchingbeslissingen in deze fase kunnen gemaakt worden.

Fase 2 Bereken signaturen en rangschik deelbomen op hun gewicht. In een bottom-up pass van elke boom berekenen we de signatuur van elke knoop van de oude en nieuwe documenten. De signatuur is een hashwaarde berekend gebruikmakend van de inhoud van de knoop en de signatuur van diens kinderen. Het stelt dus uniek de inhoud voor van de gehele deelboom geworteld op de beschouwde knoop. Simultaan berekenen we een *gewicht* voor elke knoop. Dit gewicht is simpelweg de lengte van de inhoud voor tekstknopen en de som van de gewichten van de kinderen voor elementknopen.

We construeren vervolgens een priority queue die deelbomen van het nieuwe document bevat. De deelbomen worden voorgesteld door hun wortels, en ze worden in de priority queue gerangschikt naar hun gewicht. De priority queue voorziet ons van de volgende meest zware deelboom voor welke we een matching willen vinden. Wanneer verschillende knopen hetzelfde gewicht hebben, wordt de eerste deelboom van de queue gekozen. Bij de start bevat de queue enkel de wortel van het gehele document.

Fase 3 Probeer een matching te vinden startend vanaf de zwaarste knoop. We verwijderen de zwaarste deelboom van de queue, een knoop in het nieuwe document, en construeren een lijst van kandidaatknopen in het oude document die dezelfde signatuur hebben. Uit deze nemen we de beste kandidaat (zie later), en we matchen beide. Als er geen matching is en de knoop is een element, worden zijn kinderen toegevoegd aan de knoop.

Als er meerdere kandidaten zijn, is de beste kandidaat diegene wiens ouders matchen met de ouders van de beschouwde knoop, als er zo één is. Als we geen geschikte kandidaat vinden kijken we een niveau hoger. Het aantal niveau's die dat we accepteren om te beschouwen hangt af van het gewicht van de knoop.

Wanneer een kandidaat geaccepteerd wordt matchen we het paar van deelbomen en hun voorouders zo lang ze hetzelfde label hebben. Het aantal voorouders die we matchen hangt af van het gewicht van de knoop.

Fase 4 Optimalisatie: Gebruik structuur om matchings te propageren. We doorlopen de boom bottum-up en daarna top-down en proberen de knopen van het oude en nieuwe document te matchen zodat hun ouders gematcht zijn en ze hetzelfde label hebben.

In (Cobena 01) wordt er aangetoond dat de worst-case kost $0(n * (\log(n))$ is.

Hoofdstuk 6

Implementatie

In dit hoofdstuk bespreken we kort de implementatie en zeggen we hoe het programma werkt.

6.1 Packages

De broncode van de implementatie is onderverdeel in drie packages. We zullen ze hier achtereenvolgens bespreken.

6.1.1 datastructure

Het package datastructure bevat de benodigde datastructuren die niet standaard in Java aanwezig zijn en waarvoor we zelf een implementatie voorzien hebben. Het bevat in essentie twee structuren, een key structuur waarmee we de keys, die we gebruiken om bomen deterministisch te maken, kunnen voorstellen. Deze structure bevat enkel de klasse Key.java De tweede structuur is de node structuur, hiermee stellen we de knopen voor. Zij bestaat uit een interface Node.java, een abstracte klasse AbstractNode.java en twee implementaties FileSystemNode.java en XMLNode.java.

6.1.2 algorithm

Het tweede package is algorithm, dit package bevat de noodzakelijke algoritmen. Zij bestaat uit zeven klassen, BuildEditTree.java, FileSystemTree.java, XMLTree.java, KeyAnnotater.java, MergeFileSystem.java en MergeXML.java. Met FileSystemTree.java worden de bestandsstructuren omgezet naar een boom. Wanneer hierbij XML-documenten worden tegengekomen wordt XMLTree.java gebruikt om deze te converteren naar een boom. Met BuildEditTree.java wordt de boom opgebouwd die we nodig hebben in Algoritme 1.

Dit algoitme zelf is in MerrgeFileSystem geïmplementeerd, welke dus twee file systemen merged. Het mergen van de XML-files gebeurd met Algoritme 3 in MergeXML.java. KeyAnnotater.java tenslotte bevat het annotatiealgoritme.

6.1.3 gui

Het gui package bevat slechts één klasse (JSynchronizerApplication.java), deze implementeert het volledige gui-gedeelte en is aangemaakt met de GUI-builder van Netbeans.

6.2 XML

Om XML-documenten te lezen en schrijven wordt gebruik gemaakt van een bestaande library, namelijk JDOM 1.1.1 ((Hunter 09)). JDOM is een java representatie van een XML-document, JDOM geeft een manier om op een eenvoudige en effeciënte wijze dat XML-document te lezen, manipuleren en schrijven. Het is een alternatief voor DOM (Hégaret 05) en SAX (Megginson 04), hoewel het vlekkeloss integreert met beide.

6.3 Handleiding

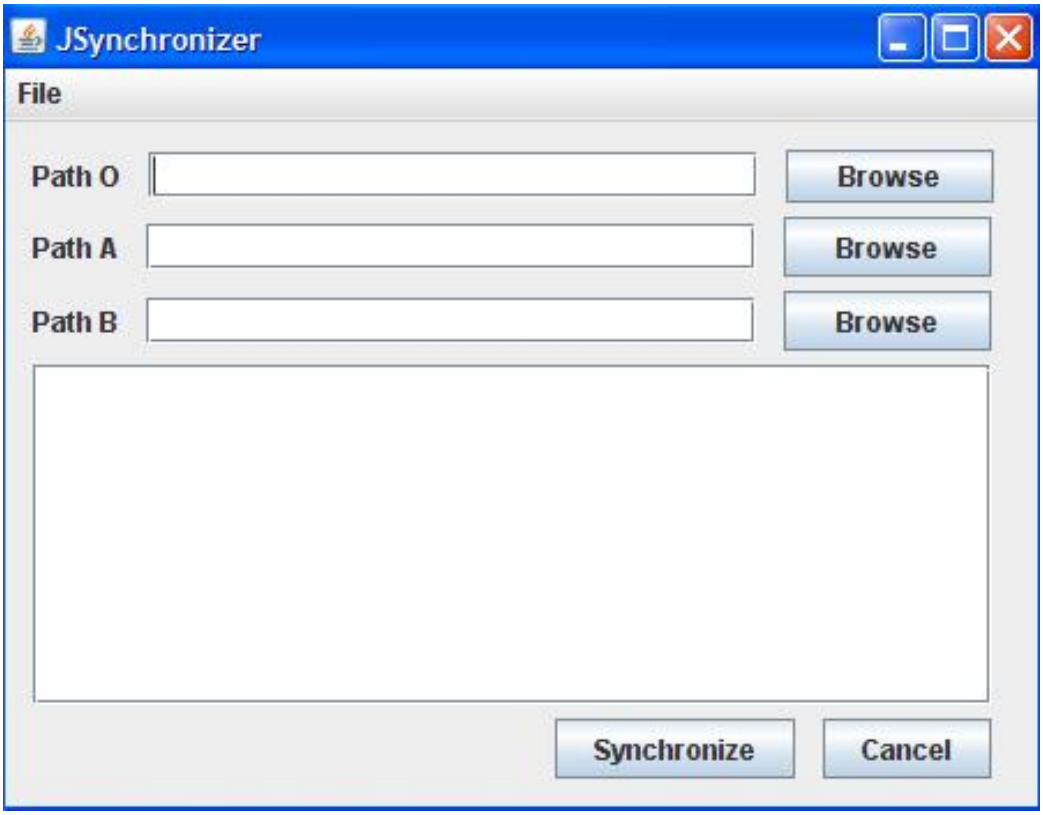
Het programma wordt getoond in Figuur 6.1, het is eenvoudig en intuïtief te bedienen. Met de drie browse-knoppen kunnen de drie te mergen structuren geladen worden. Uiteraard moeten ze alle drie eenzelfde file- of directory-naam krijgen, maar dan met een verschillend pad. Het synchronizeren zelf gebeurd met de synchronize-knop, in het tekstvak wordt dan aangegeven of het synchronizeren succesvol was. De wijzigingen worden automatisch in de filestructuren zelf aangebracht. Let er dus op dat na het synchronizeren uw files, indien nodig, overschreven zullen worden.

De keys nodig bij het synchroniseren worden ingeladen vanuit een file. De keys moeten in de file gedefinieerd worden zoals beschreven in Hoofdstuk 4 en dit per extensie. Voor elke key moet er een nieuwe lijn begonnen worden. Bijvoorbeeld zo:

(db.dept,(werkn.,{vn,an}))

(db,(dept,naam))

...



Figuur 6.1: De graphical user interface.

Door onderscheid te maken in de extensies kunnen we aangeven welke keys voor welke xml-files gelden. In het .conf bestand dat in de directory van de jar-file moet staan geven we welke file met keys bij welke extensie hoort. Dit doen we bijvoorbeeld als volgt:

(voorbeeld, vb.jkey)

...

Ook hier moet telkens een nieuwe lijn begonnen worden. Het is erg belangrijk dat al deze gegevens in het juiste formaat staan, anders zal het programma niet correct werken.

Bibliografie

[2brightsparks 08] 2brightsparks. 2brightsparks. `http://www.2brightsparks.com`, 2008.

[Altinel 00] Mehmet Altinel. *Efficient Filtering of XML Documents for Selective Dissemination of Information.* pages 53–64, 2000.

[Balasubramaniam 98] S. Balasubramaniam & Benjamin C. Pierce. *What is a file synchronizer?* In MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking, pages 98–108, New York, NY, USA, 1998. ACM.

[Buneman 01] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara & Wang chiew Tan. *Keys for XML.* In In WWW'10, pages 201–210, 2001.

[Buneman 02] Peter Buneman, Sanjeev Khanna, Keishi Tajima & Wang chiew Tan. *Archiving Scientific Data.* In In ACM SIGMOD, pages 1–12, 2002.

[Chacon 08] Scott Chacon. git the fast version control system. `http://git-scm.com/`, 2008.

[Chawathe 96a] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-molina & Jennifer Widom. *Change Detection in Hierarchically Structured Information.* In In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 493–504, 1996.

[Chawathe 96b] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-molina & Jennifer Widom. *Change Detection*

in Hierarchically Structured Information. In In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 493–504, 1996.

[Cobena 01] Gregory Cobena, Serge Abiteboul & Amelie Marian. *Detecting Changes in XML Documents.* In In ICDE, pages 41–52, 2001.

[Dahlin 01] Mike Dahlin, Bharat Chandra, Lei Gao & Amol Nayate. *End-to-end WAN Service Availability.* In In Proc. 3rd USITS, pages 97–108, 2001.

[Diao 02] Yanlei Diao, Peter Fischer, Michael J. Franklin & Raymond To. *Yfilter: Efficient and scalable filtering of XML documents.* In In ICDE, pages 341–342, 2002.

[Dijkstra 59] Edsger W. Dijkstra. *A note on two problems in connexion with graphs.* Numerische Mathematik, vol. 1, pages 269–271, 1959.

[Greenwald 06] Michael B. Greenwald, Sanjeev Khanna, Keshav Kunal, Benjamin C. Pierce & Alan Schmitt. *Agreeing to Agree: Conflict Resolution for Optimistically Replicated Data.* In DISC, pages 269–283, 2006.

[Hégaret 05] Philippe Le Hégaret. Document object model (dom). `http://www.w3.org/DOM/`, 2005.

[Heidrich 07] Jens Heidrich. Jfilesync - java file synchronization. `http://jfilesync.sourceforge.net`, 2007.

[Hunter 09] Jason Hunter & Brett McLaughlin. Jdom. `http://www.jdom.org`, 2009.

[Huston 93] L. B. Huston & P. Honeyman. *Disconnected Operation for AFS.* In Proceedings of the USENIX Mobile and Location-Independent Computing Symposium, pages 1–10, Cambridge, MA, 2–3 1993.

[Inc. 08] Canonical Inc. Bazaar. `http://bazaar-vcs.org/`, 2008.

[Jr. 88] Leonard Kawell Jr., Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, & Irene Greif. *Replicated*

Document Management in a Group Communication System., 1988.

[Khanna 07] Sanjeev Khanna, Keshav Kunal & Benjamin C. Pierce. *A Formal Investigation of Diff3.* In Foundations of Software Technology and Theoretical Computer Science (FSTTCS), December 2007.

[Kistler 96] James Jay Kistler. *Disconnected Operation in a Distributed File System.* Doctoraatsthesis, Mellon University, 1996.

[Kopcsek 05] Jan Kopcsek. Fullsycn - publishing, backup, synchronisation. `http://fullsync.sourceforge.net`, 2005.

[Kumar 94] Puneet Kumar. *Mitigating the effects of Optimistic Replication in a Distributed File System.* Doctoraatsthesis, Mellon University, December 1994.

[Lab 08] Usov Lab. Allway sync. `http://allwaysync.com`, 2008.

[Lindholm 04] Tancred Lindholm. *A three-way merge for XML documents.* In DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering, pages 1–10, New York, NY, USA, 2004. ACM.

[Megginson 04] David Megginson. Sax. `http://www.saxproject.org/`, 2004.

[Microsoft 06] Microsoft. Synctoy. `http://www.microsoft.com/windowsxp/using/digitalphotography/prophoto/synctoy.mspx`, 2006.

[Microsoft 08] Microsoft. Microsoft office groove 2007. `http://office.microsoft.com/nl-nl/groove/FX100487641043.aspx`, 2008.

[Myers 86] Eugene W. Myers. *An O(ND) difference algorithm and its variations.* Algorithmica, vol. 1, pages 251–266, 1986.

[N.N. N] N.N. Synctoy. `http://en.wikipedia.org/wiki/SyncToy`, N.N.

[Page 98] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning & G. J. Popek. *Perspectives on Optimistically Replicated Peer-to-Peer Filing.* Software—Practice and Experience, vol. 28, no. 2, pages 155–180, February 1998.

[Petersen 97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer & Alan J. Demers. *Flexible update propagation for weakly consistent replication.* In SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles, pages 288–301, New York, NY, USA, 1997. ACM.

[Pierce 04] Benjamin C. Pierce & Jerome Vouillon. *What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer*, 2004.

[Pierce N] Benjamin C. Pierce. Unison file synchronizer. `http://www.cis.upenn.edu/~bcpierce/unison`, N.N.

[Ranta 01] Mervi Ranta & Tancred Lindholm. *A 3-way Merging Algorithm for Synchronizing Ordered Trees - the 3DM merging and differencing tool for XML*, 2001.

[Reiher 94] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner & Gerald J. Popek. *Resolving File Conflicts in the Ficus File System.* In USENIX Conference Proceedings, pages 183–195, Boston, MA, June 1994. USENIX.

[RiseFly 08] RiseFly. Bestsync2008. `http://www.risefly.com/foldersynceng.htm`, 2008.

[Saito 05] Yasushi Saito & Marc Shapiro. *Optimistic replication.* ACM Comput. Surv., vol. 37, no. 1, pages 42–81, March 2005.

[Systems 08] Siber Systems. Goodsync. `http://www.goodsync.com`, 2008.

[Terry 95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer & Carl H. Hauser. *Managing Update Conflicts in Bayou, a*

Weakly Connected Replicated Storage System. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado, 1995.

[Vesperman 07] Jennifer Vesperman. *Essential CVS*. 2007.

[Yu 00] Haifeng Yu & Amin Vahdat. *Design and Evaluation of a Continuous Consistency Model for Replicated Services*, 2000.

[Yu 01] Haifeng Yu & Amin Vahdat. *The Costs and Limits of Availability for Replicated Services*, 2001.

[Zhang 95] K. Zhang. Personal communication. May 1995.

[Zhang 00] Yin Zhang, Vern Paxson & Scott Shenker. *The Stationarity of Internet Path Properties: Routing, Loss, and Throughput*. Rapport technique, In ACIRI Technical Report, 2000.