transnationale Universiteit Limburg
School voor Informatietechnologie
$\sim$
Universiteit Hasselt

Semantische eigenschappen van XML-schematalen

Thesis voorgedragen tot het behalen van de graad van licentiaat in de informatica / doctorandus in de kennistechnologie, afstudeervariant databases

door

Kris Gabriels

Promotor: Prof. dr. Frank Neven

Januari 2006

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Hoofdstuk 1

Inleiding

XML (*Extensible Markup Language*) is de nieuwe standaard voor het uitwisselen van informatie over het internet [BPSM$^+$04]. Het enorme succes van dit dataformaat is vooral te danken aan haar eenvoud en flexibiliteit. In vele gevallen is het echter nuttig om aan de hand van een schema beperkingen op te leggen aan de structuur en/of inhoud van een XML-document. Voorbeelden van schematalen zijn DTD (*Document Type Definition*) [BPSM$^+$04], XML Schema [SMT05] en RELAX NG [CM01a]. In deze thesis onderzoeken we de expressieve kracht van deze schematalen en gaan we na op welke manier validatie en typing mogelijk zijn.

Om de expressieve kracht van schematalen na te gaan definiëren we eerst een aantal abstacte schematalen, namelijk DTD's, EDTD's, single-type EDTD's en restrained-competition EDTD's [MNSB05]. EDTD's komen overeen met boomautomaten en definiëren dus de klasse van reguliere boomtalen. Single-type EDTD's zijn EDTD's waaraan een extra beperking is opgelegd en definiëren dus slechts een deelverzameling van de reguliere boomtalen. Hetzelfde geldt voor restrained-competition EDTD's. Terwijl deze laatste drie schematalen verschillende types kunnen definiëren voor eenzelfde elementnaam, is dit niet mogelijk bij DTD's. Het type van een element hangt enkel af van zijn elementnaam (en niet van de context van het element), waardoor DTD's de locale boomtalen definiëren.
We gaan na in welke mate het beperken van de expressiviteit van schematalen invloed heeft op het efficiënt valideren en typen van XML-documenten. Bij het processen van XML-data op een streaming wijze is het bovendien gewenst om een element uniek te kunnen typen zodra men zijn begintag tegenkomt. Indien dit mogelijk is spreekt men van one-pass preorder typing. We tonen aan dat zowel single-type EDTD's als restrained-competition EDTD's one-pass preorder typing toelaten. Dit komt omdat het type van een element hier enkel afhangt van de preceding van een element. EDTD's waarop geen extra restricties zijn gedefinieerd laten geen one-pass preorder typing toe. Merk op dat validatie en typing vrij sterk samenhangen. Omdat het type

van een element niet expliciet wordt weergegeven in een XML-document, is het immers de taak van de validator om dit af te leiden.
Omdat RELAX NG geen essentiële beperkingen oplegt aan het content model van een element, komt de expressieve kracht van RELAX NG overeen met die van EDTD's. One-pass preorder typing in niet mogelijk. De XML Schema specificatie legt met de EDC-constraint wel een extra beperking op. Hierdoor is het onmogelijk om in eenzelfde content model twee verschillende types te associëren met eenzelfde elementnaam. Door deze beperking komt XML Schema overeen met single-type EDTD's en is one-pass preorder typing wel mogelijk.
Er zijn efficiënte algoritmes bekend om na te gaan of een EDTD tegelijk een DTD is of voldoet aan de single-type of restrained competition eigenschap (recognitie). Nagaan of een gegeven EDTD een equivalente DTD, single-type EDTD of restrained competition EDTD heeft (simplificatie), is EXPTIME-compleet. Het simplificatiealgoritme construeert uit een gegeven EDTD D bijvoorbeeld een single-type EDTD D' en gaat vervolgens na of D en D' equivalent zijn. Indien ja, dan is D' de single-type EDTD die dezelfde taal als D definieert. Indien nee, dan is de taal van D niet definieerbaar aan de hand van een single-type EDTD.
Als onderdeel van deze thesis hebben we een implementatie gemaakt van het simplificatiealgoritme met betrekking tot single-type EDTD's. De bestaande schemaconverter Trang [Cla03c] zet schema's in RELAX NG om naar XML Schema. Hierbij wordt echter geen rekening gehouden met de extra beperking die door de EDC-constraint aan XML Schema wordt opgelegd. Het resultaat is dat de resulterende XML Schema's niet altijd correct zijn. Indien het resulterende schema niet voldoet aan de EDC-constraint, gaan we na of er een equivalent schema bestaat dat hier wel aan voldoet. Indien een dergelijk schema bestaat vormt dit de output van de schemaconversie en hebben we dus een correct XML Schema. Indien het niet bestaat wordt een foutmelding gegeven. Ook al gaat het hier om een exponentieel algoritme, toch is het door het implementeren van een aantal optimalisaties in de praktijk in vele gevallen uitvoerbaar.

Deze thesis is als volgt opgebouwd. In hoofdstuk 2 bespreken we stringautomaten en boomautomaten omdat zij onmisbaar zijn voor een goed begrip van de reguliere boomtalen en omdat ze noodzakelijk zijn om bepaalde algoritmes op EDTD's te implementeren. In hoofdstuk 3 definiëren we een aantal abstracties van XML-schematalen, namelijk DTD, EDTD, single-type EDTD en restrained competition EDTD en brengen we deze in verband met one-pass preorder typing. In hoofdstuk 4 bespreken we het het recognitie- en simplificatieprobleem met betrekking tot EDTD's. Hoofdstuk 5 beschrijft de relatie tussen deze abstracte schematalen en in de praktijk gebruikte schematalen DTD, XML Schema en RELAX NG. Omdat RELAX NG als uitgangspunt dient voor de implementatie van het simplificatiealgoritme,

bevat hoofdstuk 6 een grondige bespreking van deze schemataal. Hoofdstuk 7 geeft een beschrijving van de gemaakte implementatie, waarbij de nadruk ligt op de gebruikte datastructuren en optimalisaties die het simplificatiealgoritme in de praktijk uitvoerbaar maken. Hoofdstuk 8 vormt een besluit.

Hoofdstuk 2

Automatentheorie

Omdat een XML-schema kan uitgedrukt worden als een boomautomaat (cf. hoofdstuk 3), verdiepen we ons in dit hoofdstuk in de automatentheorie. We kunnen echter operaties op boomautomaten, zoals unie en complement, en beslissingsproblemen, zoals het equivalentieprobleem, niet uitwerken alvorens de reguliere woordtalen te hebben besproken. Zij vormen immers de basis van de transitiefunctie bij boomautomaten. Daarom beschouwen we eerst een aantal formalismen die de reguliere woordtalen uitdrukken en vervolgens gaan we over tot een bespreking van boomautomaten. We bespreken hier enkel de operaties die daadwerkelijk gebruikt zijn in de implementatie die in hoofdstuk 7 beschreven wordt.

2.1 Reguliere expressies en eindige automaten

Gegeven een alfabet Σ, dan kunnen we over dit alfabet een verzameling woorden (of strings) definiëren. Een verzameling van woorden noemen we een (woord)taal. De verzameling van alle woorden over alfabet Σ, inclusief de lege string ϵ, noteren we met Σ^*. Deterministische eindige automaten (DFA's), niet-deterministische eindige automaten (NFA's) en reguliere expressies drukken dezelfde klasse van woordtalen uit, namelijk de reguliere woordtalen. Een taal die uitgedrukt is aan de hand van een DFA, een NFA of een reguliere expressie kan ook uitgedrukt worden in de twee andere vormen.

2.1.1 Definitie van DFA, NFA en reguliere expressie

Definitie 2.1 (DFA) *Een (deterministische) eindige automaat (DFA) is een 5-tupel $M = (Q, \Sigma, \delta, q_0, F)$ met Q een eindige verzameling toestanden, Σ een alfabet, $\delta : Q \times \Sigma \to Q$ de transitiefunctie, $q_0 \in Q$ de begintoestand en $F \subseteq S$ de verzameling eindtoestanden.*

Een string w wordt aanvaard door een DFA M als er een accepterende run van M op w bestaat. We formaliseren die concept als volgt. Neem een

DFA $M = (Q, \Sigma, \delta, q_0, F)$ en een string $w = w_1 w_2 \dots w_n$ van lengte n over het alfabet Σ. Dan wordt w door M aanvaard als er een sequentie van toestanden $r_0 r_1 \dots r_n$ uit Q bestaat zodat

- $r_0 = q_0$,
- $\delta(r_i, w_{i+1}) = r_{i+1}$ voor $i = 0, \dots, n-1$, en
- $r_n \in F$.

$r_0 r_1 \dots r_n$ noemen we de accepterende run van M op w. Omdat M deterministisch is, bestaat er hoogstens één accepterende run op een gegeven inputstring. De verzameling strings die door M aanvaard wordt, is de taal van M en wordt genoteerd als $\mathrm{L}(M)$.

Definitie 2.2 (NFA) *Een niet-deterministische eindige automaat (NFA) is een 5-tupel $N = (Q, \Sigma, \delta, q_0, F)$ met Q een eindige verzameling toestanden, Σ een alfabet, $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ de transitiefunctie, $q_0 \in Q$ de begintoestand en $F \subseteq Q$ de verzameling eindtoestanden.*

Hierbij is $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ en $\mathcal{P}(Q)$ is de machtsverzameling van Q (ook wel genoteerd als 2^Q). Een string w wordt aanvaard door een NFA N als er een accepterende run van N op w bestaat. We formaliseren die concept als volgt. Neem een NFA $N = (Q, \Sigma, \delta, q_0, F)$ en een string w over het alfabet Σ. Dan wordt w door N aanvaard als we w kunnen schrijven als $w = y_1 y_2 \dots y_n$ met elke $y_i \in \Sigma_\epsilon$ en als er een sequentie van toestanden $r_0 r_1 \dots r_n$ uit Q bestaat zodat

- $r_0 = q_0$,
- $r_{i+1} \in \delta(r_i, y_{i+1})$ voor $i = 0, \dots, n-1$, en
- $r_n \in F$.

$r_0 r_1 \dots r_n$ noemen we een accepterende run van N op w. Omdat N niet-deterministisch is, zijn meerdere runs op een gegeven inputstring mogelijk.

Definitie 2.3 (Reguliere expressie) *r is een reguliere expressie over een alfabet Σ indien r gelijk is aan*

- *a voor een $a \in \Sigma$,*
- *ϵ,*
- *$\emptyset$*
- *$(r_1 + r_2)$ met r_1 en r_2 reguliere expressies,*
- *$(r_1 r_2)$ met r_1 en r_2 reguliere expressies, of*

- *(r_1*) met r_1 een reguliere expressie.*

De reguliere expressies a, ϵ en $\emptyset$ definiëren respectievelijk de talen $\{a\}$, $\{\epsilon\}$ en de lege taal. De taal gedefinieerd door $(r_1 + r_2)$ en $(r_1 r_2)$ is de unie respectievelijk concatenatie van de talen gedefinieerd door r_1 en r_2. De taal gedefinieerd door $(r_1{}^*)$ is de ster-operatie toegepast op de taal gedefinieerd door r_1. Uit de laatste drie gevallen blijkt duidelijk dat de klasse van reguliere talen gesloten is onder unie, concatenatie en ster-operatie.

In [HU79], een standaardwerk met betrekking tot automatentheorie en berekenbaarheid, wordt een bewijs gegeven voor de equivalentie van DFA's, NFA's en reguliere expressies, en er worden tevens algoritmes voorgesteld om het ene formalisme om te zetten in een equivalent ander formalisme. Het algoritme om een reguliere expressie om te zetten in een equivalente NFA (polynomiaal) en het algoritme om een NFA om te zetten in een equivalente DFA (exponentieel), maken beide deel uit van de implementatie in hoofdstuk 7.[1]

2.1.2 Operaties op eindige automaten

De klasse van reguliere talen is gesloten onder unie, doorsnede, complement en substitutie. Een bewijs hiervan is te vinden in [HU79]. Hieruit kunnen we afleiden dat de klasse van reguliere talen ook gesloten is onder verschil. We beschrijven hier de algoritmes voor het berekenen van de unie, doorsnede, verschil en substitutie die gebruikt worden in de implementatie van hoofdstuk 7. Ze zijn allemaal van toepassing op DFA's. Voor het berekenen van de unie is er ook een algoritme op NFA's bekend, maar omdat we de resulterende automaat als input gaan gebruiken voor het berekenen van het verschil (cf. sectie 2.2.2), hebben we gekozen voor het algoritme op DFA's.

Unie
Neem twee DFA's $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ en $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. We construeren hieruit DFA $A = (Q, \Sigma, \delta, q_0, F)$ die $\mathrm{L}(A_1) \cup \mathrm{L}(A_2)$ herkent. Dan geldt:

- $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ en } r_2 \in Q_2\} = Q_1 \times Q_2$
- Voor elke toestand $(r_1, r_2) \in Q$ en elk symbool $a \in \Sigma$ definiëren we $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$.
- q_0 is de toestand (q_1, q_2)
- $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ of } r_2 \in F_2\} = (F_1 \times Q_2) \cup (Q_1 \times F_2)$

[1] Omdat we deze algoritmes niet zelf geïmplementeerd hebben - ze maken deel uit van een extern package JFLAP [Rod05] -, gaan we hier niet verder op in.

Doorsnede
Neem twee DFA's $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ en $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. We construeren hieruit DFA $A = (Q, \Sigma, \delta, q_0, F)$ die $\mathrm{L}(A_1) \cap \mathrm{L}(A_2)$ herkent. Dan geldt:

- $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ en } r_2 \in Q_2\} = Q_1 \times Q_2$
- Voor elke toestand $(r_1, r_2) \in Q$ en elk symbool $a \in \Sigma$ definiëren we $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$.
- q_0 is de toestand (q_1, q_2)
- $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ en } r_2 \in F_2\} = F_1 \times F_2)$

Verschil
Neem twee DFA's $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ en $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. We construeren hieruit DFA $A = (Q, \Sigma, \delta, q_0, F)$ die $\mathrm{L}(A_1) - \mathrm{L}(A_2)$ herkent. Dan geldt:

- $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ en } r_2 \in Q_2\} = Q_1 \times Q_2$
- Voor elke toestand $(r_1, r_2) \in Q$ en elk symbool $a \in \Sigma$ definiëren we $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$.
- q_0 is de toestand (q_1, q_2)
- $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ en } r_2 \notin F_2\}$

Substitutie
Neem DFA $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ en substitutie $f : \Sigma \to \Sigma'$. We construeren hieruit NFA $A = (Q, \Sigma, \delta, q_0, F)$ die $\mathrm{f}(\mathrm{L}(A_1))$ herkent. Dan geldt:

- $Q = Q_1$
- Voor elke transitie $\delta(q, \sigma) = q'$ en elke $\sigma' \in f(\sigma)$ wordt de transitie $\delta'(q, \sigma') = q'$ geconstrueerd.
- $q_0 = q_1$
- $F = F_1$

Omdat A niet noodzakelijk deterministisch is, wordt vervolgens het derminisatiealgoritme op A toegepast.

Omdat het herhaaldelijk toepassen van bovenstaande algoritmes tot vrij grote automaten kan leiden, zijn we op zoek gegaan naar een algoritme dat een DFA is een soort minimale vorm brengt. In [HU79] wordt een algoritme besproken dat een gegeven DFA A omzet in een equivalente DFA A' met een

minimaal aantal toestanden. We stellen dit minimaliseringsalgoritme hier kort voor. We beginnen met een beschrijving van equivalente toestanden. Twee toestanden p en q van A zijn equivalent indien voor elke inputstring w geldt dat $\delta(p, w)$ een eindtoestand is als en slechts als $\delta(q, w)$ een eindtoestand is.[2] Op de naïeve manier kunnen we deze equivalenties onmogelijk in eindige tijd nagaan. Daarom sommen we alle koppels toestanden op die niet equivalent zijn, de overblijvende koppels zijn dan per definitie wel equivalent. Om na te gaan of de toestanden p en q niet equivalent zijn, beschouwen we alle koppels toestanden $r = \delta(p, a)$ en $s = \delta(q, a)$ voor elk alfabetsymbool a. Als we reeds hebben kunnen aantonen dat r en s niet equivalent zijn dankzij een string w, kunnen we hieruit afleiden dat er een string aw bestaat zodat ofwel $\delta(p, aw)$ ofwel $\delta(q, aw)$ een eindtoestand is, maar niet beide. Hierdoor weten we dus dat p en q niet equivalent zijn.
Om een DFA A om te zetten in een equivalente DFA A' met een minimaal aantal toestanden worden eerst de onbereikbare toestanden verwijderd. Vervolgens wordt de verzameling X van niet-equivalente koppels van toestanden iteratief geconstrueerd. Eerst worden aan X de koppels toestanden toegevoegd waarvan één toestand een eindtoestand is en de andere niet. Voor elk koppel (r, s) in X en elk alfabetsymbool a voegen we het koppel (p, q) aan X toe met $\delta(p, a) = r$ en $\delta(q, a) = s$ transities uit A. Omdat we te maken hebben met een DFA weten we dat deze twee toestanden p en q bestaan. Dit proces gaat door totdat er geen nieuwe koppels meer aan X kunnen toegevoegd worden. Alle koppels toestanden die niet in X voorkomen zijn equivalent. Om nu de equivalente DFA A' te maken behouden we van elk van deze equivalente koppels (x, y) enkel de toestand x, de toestand y en zijn bijhorende transities worden verwijderd. Merk op dat we bij het verwerken van de equivalente koppels toestanden moeten nagaan of beide toestanden nog daadwerkelijk bestaan. Indien een toestand wordt verwijderd, moeten we bijhouden met welke toestand hij is 'samengesmolten'.
Stel dat het alfabet Σ k symbolen bevat en de verzameling toestanden Q als grootte n heeft. De verzameling X kan maximaal n^2 koppels niet-equivalente toestanden bevatten. Per koppel uit X wordt nagegaan of er een symbool a bestaat waardoor een nieuw koppel aan X zou kunnen toegevoegd worden. De totale tijdscomplexiteit van het algoritme bedraagt dus $O(kn^2)$.

2.1.3 One-unambiguous reguliere expressies

In deze sectie beschrijven we aan de hand van voorbeelden enkele semantische eigenschappen van reguliere expressies.

Zij w een woord dat aanvaard wordt door een reguliere expressie r. Als w op verschillende manieren uit r kan afgeleid worden, is r *ambigu*. Als er slechts één afleiding van w uit r bestaat, is r *unambiguous*. Zij r' de

[2] De transitiefunctie δ is hier uitgebreid op strings.

reguliere expressie die uit r verkregen wordt door elk i-de voorkomen van alfabetsymbool a te vervangen door a_i. Neem bijvoorbeeld $r = (a+b)^*aa^*$, dan is $r' = (a_1 + b_1)^*a_2a_3^*$. Aan de hand van string aaa tonen we aan dat r ambigu is. String aaa kan op drie verschillende manieren afgeleid worden uit r, omdat zowel $a_1a_1a_2$, $a_1a_2a_3$ als $a_2a_3a_3$ aanvaard worden door r'. Elke ambigue reguliere expressie kan echter omgezet worden in een equivalente unambiguous reguliere expressie. De unambiguous reguliere expressie $(a + b)^*a$ definieert dezelfde taal als r.

Een unambiguous reguliere expressie heeft de eigenschap dat elk symbool van een woord dat aanvaard wordt door de reguliere expressie aan precies één symbool van deze reguliere expressie gelinkt kan worden. Bij een *one-unambiguous* reguliere expressie kan men, als men een woord van links naar rechts doorloopt, elk symbool van dit woord onmiddelijk mappen op het overeenkomstige symbool uit de reguliere expressie, zonder eerst vooruit te kijken naar de volgende symbolen van dit woord. De reguliere expressie $(a + b)^*a$ is unambiguous, maar niet one-unambiguous. We weten immers niet of het eerste symbool van het woord aa gemapt wordt op de eerste of tweede a uit de reguliere expressie, zonder eerst te kijken naar het volgende symbool in het woord. We geven een formele definitie van one-unambiguous reguliere expressies.

Definitie 2.4 *Een reguliere expressie r is one-unambiguous als en slechts als er geen twee woorden wa_iv en wa_jv' in $L(r')$ zitten met $i \neq j$.*

De reguliere expressie $(a+b)^*a$ is inderdaad niet one-unambiguous. De taal $\mathrm{L}((a_1 + b_1)^*a_2)$ bevat de woorden $b_1a_1a_2$ en b_1a_2 met $w = b_1$, $i = 1$, $j = 2$, $v = a_2$ en $v' = \epsilon$. Er bestaat echter een equivalente reguliere expressie die wel one-unambiguous is, namelijk $b^*a(b^*a)^*$. We merken nog op dat niet elke reguliere taal kan beschreven worden door een one-unambiguous reguliere expressie [BKW98].

2.2 Boomautomaten

Een boomautomaat definieert een verzameling bomen. Vooraleer over te gaan tot een definitie en bespreking van boomautomaten, formaliseren we het begrip boom.

De verzameling bomen over een alfabet Σ, genoteerd als $\mathcal{T}_\Sigma$, wordt inductief als volgt gedefinieerd:

(i) elke $\sigma \in \Sigma$ maakt deel uit van $\mathcal{T}_\Sigma$;

(ii) als $\sigma \in \Sigma$ en $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$ met $n \geq 0$, dan maakt ook $\sigma(t_1, \ldots, t_n)$ deel uit van $\mathcal{T}_\Sigma$.

Omdat er geen limiet staat op het aantal kinderen van een knoop, worden deze bomen *unranked* genoemd. De verzameling knopen van een boom $t \in \mathcal{T}_\Sigma$, genoteerd als $\mathrm{Dom}(t)$, is de deelverzameling van $\mathbb{N}^*$ die als volgt gedefinieerd wordt: als $t \in \mathcal{T}_\Sigma$ met $\sigma \in \Sigma$, $n \geq 0$ en $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$, dan is $\mathrm{Dom}(t) = \{\epsilon\} \cup \{iu \mid i \in \{1, \ldots, n\}, u \in \mathrm{Dom}(t_i)\}$. ϵ is dus de wortel en vj is het j-de kind van v. Het label van een knoop u in t wordt genoteerd als $\mathrm{lab}^t(u)$.

2.2.1 Definitie

Definitie 2.5 *Een (niet-deterministische) boomautomaat (NTA) is een tupel $A = (\Sigma, Q, \delta, F)$ met Σ een alfabet, Q een eindige verzameling toestanden, $F \subseteq Q$ de verzameling eindtoestanden en $\delta : Q \times \Sigma \to Reg(Q)$ de transitiefunctie waarbij $Reg(Q)$ een reguliere woordtaal over Q is.*

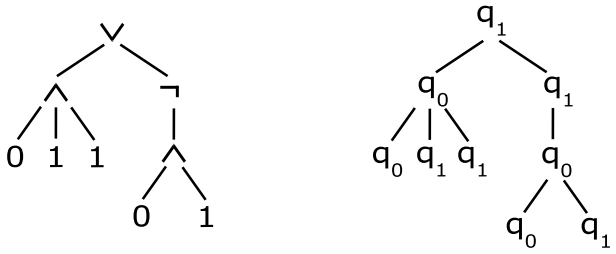
De reguliere woordtaal van de transitiefunctie kan op verschillende manieren worden uitgedrukt. De klasse van boomautomaten waarbij de reguliere woordtaal wordt uitgedrukt als reguliere expressie noemen we NTA(REG). Indien deze wordt uitgedrukt door een niet-deterministische stringautomaat spreken we van NTA(NFA).

Een boom t wordt aanvaard door een NTA A als er een accepterende run van A op t bestaat. We formaliseren die concept als volgt. Een run van A op t is de labelingsfunctie $\lambda : \mathrm{Dom}(t) \to Q$ zodat voor elke $v \in \mathrm{Dom}(t)$ met n kinderen geldt dat $\lambda(v1) \ldots \lambda(vn) \in \delta(\lambda(v), \mathrm{lab}^t(v))$. Een run is accepterend als en slechts als $\lambda(\epsilon) \in F$. De verzameling bomen die door A aanvaard wordt, ook de taal van A genoemd, noteren we als $\mathrm{L}(A)$. Een verzameling bomen is *regulier* indien er een boomautomaat bestaat die ze aanvaardt.

Voorbeeld 2.1

Als voorbeeld beschouwen we de boomvoorstelling van booleaanse formules, waarbij de interne knopen gelabeld zijn met $\vee$, $\wedge$ en $\neg$, en de bladeren met 0 en 1. We stellen een boomautomaat $A = (\Sigma, Q, \delta, F)$ voor die precies die bomen aanvaardt waarvan de booleaanse formules tot *waar* geëvalueerd worden. Dan is het alfabet $\Sigma = \{0, 1, \vee, \wedge, \neg\}$, de verzameling toestanden $Q = \{q_0, q_1\}$, de verzameling eindtoestanden $F = \{q_1\}$ en de transitiefunctie wordt voorgesteld door de volgende regels:

$$
\begin{array}{ll}
\delta(q_0, 0) = \epsilon & \delta(q_1, 0) = \emptyset \\
\delta(q_0, 1) = \emptyset & \delta(q_1, 1) = \epsilon \\
\delta(q_0, \vee) = q_0 {q_0}^* & \delta(q_1, \vee) = (q_0 + q_1)^* q_1 (q_0 + q_1)^* \\
\delta(q_0, \wedge) = (q_0 + q_1)^* q_0 (q_0 + q_1)^* & \delta(q_1, \wedge) = q_1 {q_1}^* \\
\delta(q_0, \neg) = q_1 & \delta(q_1, \neg) = q_0
\end{array}
$$



Figuur 2.1: Een boom en zijn accepterende run

Figuur 2.1 toont een boom die aanvaard wordt door A en zijn accepterende run.

Merk op dat een boomautomaat A een boom t bottom-up evalueert. Eerst worden de bladeren van t in overeenstemming met de transitiefunctie met een toestand gelabeld en vervolgens werkt men naar boven tot de wortel is bereikt, indien deze tenminste bereikbaar is (zie volgende sectie). Als de wortel gelabeld kan worden met een eindtoestand, wordt de boom aanvaard.

2.2.2 Volledigheid en determinisme

Volledigheid

Een boomautomaat (Σ, Q, δ, F) is *volledig* indien voor elk woord $w \in Q^*$ en elk symbool $\sigma \in \Sigma$ er een toestand $q \in Q$ bestaat zodat $w \in \delta(q, \sigma)$. Een volledige boomautomaat heeft dus de eigenschap dat er voor elke boom minstens één run bestaat.

We stellen hier een algoritme voor om een boomautomaat van de vorm NTA(NFA) volledig te maken. Eerst wordt aan de verzameling toestanden Q een 'vuilbak'-toestand q_{trash} toegevoegd. Neem nu NFA *all* die alle woorden over het alfabet van toestanden Q aanvaardt. Voor elk symbool $\sigma \in \Sigma$ wordt $\delta(q_{trash}, \sigma) = all - \bigcup_{i \in Q} \delta(q_i, \sigma)$ aan de transitiefunctie δ toegevoegd.

Voorbeeld 2.2

We maken de boomautomaat $A = (\Sigma, Q, \delta, F)$ uit voorbeeld 2.1 volledig door eerst aan Q de toestand q_{trash} toe te voegen. Zij R de verzameling strings die gedefinieerd wordt door $(q_0 q_0{}^*) + ((q_0 + q_1)^* q_1 (q_0 + q_1)^*)$ en zij S de verzameling strings die gedefineerd wordt door $(q_1 q_1{}^*) + ((q_0 + q_1)^* q_0 (q_0 + q_1)^*)$. Dan ziet de transitiefunctie δ er als volgt uit:

$$\begin{array}{ll}
\delta(q_0, 0) = \epsilon & \delta(q_1, 0) = \emptyset \\
\delta(q_0, 1) = \emptyset & \delta(q_1, 1) = \epsilon \\
\delta(q_0, \vee) = q_0 {q_0}^* & \delta(q_1, \vee) = (q_0 + q_1)^* q_1 (q_0 + q_1)^* \\
\delta(q_0, \wedge) = (q_0 + q_1)^* q_0 (q_0 + q_1)^* & \delta(q_1, \wedge) = q_1 {q_1}^* \\
\delta(q_0, \neg) = q_1 & \delta(q_1, \neg) = q_0
\end{array}$$

$$\begin{array}{l}
\delta(q_{trash}, 0) = Q^* - \{\epsilon\} \\
\delta(q_{trash}, 1) = Q^* - \{\epsilon\} \\
\delta(q_{trash}, \vee) = Q^* - R \\
\delta(q_{trash}, \wedge) = Q^* - S \\
\delta(q_{trash}, \neg) = Q^* - \{q_0, q_1\}
\end{array}$$

Om de notatie eenvoudig te houden, zien we af van een schrijfwijze aan de hand van eindige automaten.

Determinisme

Een boomautomaat (Σ, Q, δ, F) is *deterministisch* indien voor elk symbool $\sigma \in \Sigma$ en elk woord $w \in Q^*$ er hoogstens één toestand $q \in Q$ bestaat zodat $w \in \delta(q, \sigma)$. Dit komt overeen met de eis dat voor elk paar verschillende toestanden q_1 en q_2 en voor elk symbool σ geldt dat $\delta(q_1, \sigma) \cap \delta(q_2, \sigma)$ leeg is.

Voorbeeld 2.3

De automaat $A = (\Sigma, Q, \delta, F)$ met $\Sigma = \{a\}$, $Q = \{p, q, r\}$, $F = \{r\}$ en als transities

$$\begin{array}{l}
\delta(p, a) = (pp)^* \\
\delta(q, a) = pp(pp)^* \\
\delta(r, a) = qq(qq)^*
\end{array}$$

aanvaardt bomen waarvan alle knopen met a gelabeld zijn. Bovendien moet elke interne knoop een even aantal kinderen hebben en elk pad van wortel tot blad moet minstens 2 lang zijn. A is niet deterministisch omdat $\delta(p, a) \cap \delta(q, a) \neq \emptyset$.

Om een boomautomaat om te vormen tot een equivalente deterministische boomautomaat wordt gebruik gemaakt van de subset-constructie [GV04]. Het idee hierachter is alle mogelijke runs van de niet-deterministische automaat tegelijk te simuleren. Zij $A = (\Sigma, Q, \delta, F)$ een niet-deterministische boomautomaat. We construeren hieruit een equivalente deterministische boomautomaat $A' = (\Sigma, Q', \delta', F')$ waarbij

- Q' uit alle mogelijke deelverzamelingen van Q bestaat,
- F' die deelverzamelingen van Q zijn die één of meerdere toestanden uit F bevatten, en

- voor elke toestand $R \in Q'$, elk woord $R_1 \dots R_n \in Q'^*$ en elk symbool $\sigma \in \Sigma$ de transitie $\delta'(R, \sigma)$ zo te definiëren dat

$$\begin{gathered} R_1 \dots R_n \in \delta'(R, \sigma) \\ \Updownarrow \\ R = \{q \in Q \mid \exists q_1 \in R_1, \dots, q_n \in R_n : q_1 \dots q_n \in \delta(q, \sigma)\}. \end{gathered}$$

Om aan te tonen dat deze woordtaal over Q' regulier is, definiëren we de substitutie $f : Q \to Q'$ zodat $f(q) = \{S \mid S \subseteq Q \text{ en } q \in S\}$. Het is nu gemakkelijk in te zien dat

$$\delta'(R, \sigma) = \bigcap_{q \in R} f(\delta(q, \sigma)) - \bigcup_{q \in (Q-R)} f(\delta(q, \sigma)).$$

Aangezien de reguliere woordtalen gesloten zijn onder unie, doorsnede, verschil en substitutie is $\delta'(R, \sigma)$ dus een reguliere taal. Deze transitiefunctie is gemakkelijk te implementeren als A van de vorm NTA(NFA) is: de algoritmes voor het berekenen van de unie, doorsnede, verschil en substitutie zijn welbekend voor eindige automaten.

De deterministische boomautomaat A' is exponentieel groter dan zijn niet-deterministische tegenhanger A en de constructie van A' uit A gebeurt in $O(2^n)$ tijd met n het aantal toestanden van A.

Voorbeeld 2.4

We illustreren deze constructie op de niet-deterministische boomautomaat A uit voorbeeld 2.3.[3] Uit A construeren we $A' = (\Sigma, Q', \delta', F')$ waarbij $Q' = \{\{p\}, \{q\}, \{r\}, \{p, q\}, \{p, r\}, \{q, r\}, \{p, q, r\}\}$ en $F' = \{\{r\}, \{p, r\}, \{q, r\}, \{p, q, r\}\}$. De transitiefunctie δ' bevat exponentieel meer regels dan de originele δ, meer bepaald $2^n \times k$ met n het aantal toestanden in A en k het aantal alfabetsymbolen. De berekenig van de transitiefunctie gebeurt door bewerkingen met NFA's en de substitutie f is

$$\begin{aligned} f(p) &= \{\{p\}, \{p, q\}, \{p, r\}, \{p, q, r\}\}, \\ f(q) &= \{\{q\}, \{p, q\}, \{q, r\}, \{p, q, r\}\}, \\ f(r) &= \{\{r\}, \{p, r\}, \{q, r\}, \{p, q, r\}\}. \end{aligned}$$

De uiteindelijke transitiefunctie wordt gegeven door onderstaande regels. Lange reguliere expressies zijn over meerdere regels gesplitst.

[3] Dezelfde automaat wordt ook als voorbeeld gebruikt in [GV04]. De transitie $\delta'(\{p, q\}, a) = \{p\}\{p\}(\{p\}\{p\})^*$ is echter niet correct, en hebben we dus aangepast. Het voorbeeld is getest aan de hand van de implementatie die in hoofdstuk 7 beschreven is.

$$
\begin{array}{rcl}
\delta'(\{p\},a) & = & \epsilon \\
\delta'(\{q\},a) & = & \emptyset \\
\delta'(\{r\},a) & = & \emptyset \\
\delta'(\{p,q\},a) & = & (\{\{p\}+\{p,q\}+\{p,q,r\}\}(\{p\}+\{p,q\}+\{p,q,r\}))^* \\
 & & (\{p\}(\{p\}+\{p,q\}+\{p,q,r\})+(\{p\}+\{p,q\}+\{p,q,r\})\{p\}) \\
 & & ((\{p\}+\{p,q\}+\{p,q,r\})(\{p\}+\{p,q\}+\{p,q,r\}))^* \\
\delta'(\{p,r\},a) & = & \emptyset \\
\delta'(\{q,r\},a) & = & \emptyset \\
\delta'(\{p,q,r\},a) & = & (\{p,q\}+\{p,q,r\})(\{p,q\}+\{p,q,r\}) \\
 & & ((\{p,q\}+\{p,q,r\})(\{p,q\}+\{p,q,r\}))^*
\end{array}
$$

Deze transities kunnen als volgt geïnterpreteerd worden. Tijdens een run van A' op een boom t wordt aan alle bladeren van t de toestand $\{p\}$ toegekend. Aan een knoop wordt de toestand $\{p,q\}$ toegekend als minstens één van de kinderen toestand $\{p\}$ heeft, d.w.z. een knoop v krijgt toestand $\{p,q\}$ als er een pad van v tot de bladeren onder v bestaat dat precies lengte 1 heeft. Een knoop v krijgt toestand $\{p,q,r\}$ als alle kinderen van v toestand $\{p,q\}$ of $\{p,q,r\}$ hebben, wat betekent dat elk pad van v tot de bladeren onder v minstens lengte 2 heeft.

2.2.3 Doorsnede

Zij $A_1 = (\Sigma, Q_1, \delta_1, F_1)$ en $A_2 = (\Sigma, Q_2, \delta_2, F_2)$ twee boomautomaten. We construeren $A = (\Sigma, Q, \delta, F)$ die $\mathrm{L}(A_1) \cap \mathrm{L}(A_2)$ herkent door tegelijk A_1 en A_2 te simuleren [GV04]. Hiertoe nemen we $Q = Q_1 \times Q_2$, $F = F_1 \times F_2$ en voor elke $\sigma \in \Sigma$ en $(q_1, q_2) \in Q$ wordt $\delta((q_1, q_2), \sigma)$ zo gedefinieerd dat

$$\delta((q_1,q_2),\sigma) = \{(r_1,s_1)\dots(r_n,s_n) \mid r_1\dots r_n \in \delta_1(q_1,\sigma) \text{ en } s_1\dots s_n \in \delta_2(q_2,\sigma)\}.$$

We tonen nu aan dat deze woordtaal over $Q_1 \times Q_2$ regulier is. Neem hiervoor de substituties $f_1 : Q_1 \to Q_1 \times Q_2$ en $f_2 : Q_2 \to Q_1 \times Q_2$ zodat $f_1(r) = \{(r,s) \mid s \in Q_2\}$ en $f_2(s) = \{(r,s) \mid r \in Q_1\}$. Dan geldt

$$
\begin{array}{rcl}
\delta((q_1,q_2),\sigma) & = & \{(r_1,s_1)\dots(r_n,s_n) \mid r_1\dots r_n \in \delta_1(q_1,\sigma) \text{ en } s_1\dots s_n \in {Q_2}^*\} \\
 & & \cap\, \{(r_1,s_1)\dots(r_n,s_n) \mid r_1\dots r_n \in {Q_1}^* \text{ en } s_1\dots s_n \in \delta_2(q_2,\sigma)\} \\
 & = & \{(r_1,s_1)\dots(r_n,s_n) \in f_1(r_1\dots r_n) \mid r_1\dots r_n \in \delta_1(q_1,\sigma)\} \\
 & & \cap\, \{(r_1,s_1)\dots(r_n,s_n) \in f_2(s_1\dots s_n) \mid s_1\dots s_n \in \delta_2(q_2,\sigma)\} \\
 & = & f_1(\delta_1(q_1,\sigma)) \cap f_2(\delta_2(q_2,\sigma)).
\end{array}
$$

Aangezien de reguliere woordtalen gesloten zijn onder doorsnede en substitutie is $\delta((q_1,q_2),\sigma)$ een reguliere taal. Deze transitiefunctie is gemakkelijk te implementeren als A van de vorm NTA(NFA) is: de algoritmes voor het berekenen van de doorsnede en substitutie zijn welbekend voor eindige automaten.

Nog een woordje over de complexiteit van deze constructie. Als de boomautomaten A_1 en A_2 respectievelijk m en n toestanden hebben en gedefinieerd zijn over een alfabet met k symbolen, dan heeft de boomautomaat A die $\mathrm{L}(A_1) \cap \mathrm{L}(A_2)$ herkent $m \times n$ toestanden en $m \times n \times k$ transities.

Voorbeeld 2.5

Zij $\Sigma = \{a, b, c\}$, $A_1 = (\Sigma, \{q_1, q_2\}, \delta_1, \{q_1\})$ en $A_2 = (\Sigma, \{p_1, p_2\}, \delta_2, \{p_1\})$ met

$$\begin{array}{ll}
\delta_1(q_1, a) = {q_2}^* & \delta_2(p_1, a) = {p_2}^* p_1 {p_2}^* \\
\delta_1(q_1, b) = {q_2}^* q_1 {q_2}^* & \delta_2(p_1, b) = {p_2}^* \\
\delta_1(q_1, c) = {q_2}^* q_1 {q_2}^* & \delta_2(p_1, c) = {p_2}^* p_1 {p_2}^* \\
\delta_1(q_2, a) = \emptyset & \delta_2(p_2, a) = {p_2}^* \\
\delta_1(q_2, b) = {q_2}^* & \delta_2(p_2, b) = \emptyset \\
\delta_1(q_2, c) = {q_2}^* & \delta_2(p_2, c) = {p_2}^*
\end{array}$$

Automaat A_1 aanvaardt alle bomen met labels in Σ waarin het symbool a precies één keer voorkomt, automaat A_2 aanvaardt alle bomen met labels in Σ waarin het symbool b precies één keer voorkomt. De toestanden hebben intuïtief de volgende betekenis. Tijdens een run van A_1 (respectievelijk A_2) op een boom t wordt aan een knoop v de toestand q_1 (respectievelijk p_1) toegekend als in de subtree met wortel v precies één knoop met label a (respectievelijk b) voorkomt, in alle andere gevallen krijgt v toestand q_2 (respectievelijk p_2).
We construeren nu boomautomaat A die $\mathrm{L}(A_1) \cap \mathrm{L}(A_2)$ aanvaardt.[4] Dan is $A = (\Sigma, \{(q_1, p_1), (q_1, p_2), (q_2, p_1), (q_2, p_2)\}, \delta, \{(q_1, p_1)\})$ met

$$\begin{array}{l}
\delta((q_1, p_1), a) = (q_2, p_2)^*(q_2, p_1)(q_2, p_2)^* \\
\delta((q_1, p_1), b) = (q_2, p_2)^*(q_1, p_2)(q_2, p_2)^* \\
\delta((q_1, p_1), c) = (q_2, p_2)^*((q_1, p_1) + (q_1, p_2)(q_2, p_2)^*(q_2, p_1) + (q_2, p_1)(q_2, p_2)^*(q_2, p_2))(q_1, p_2)^*
\end{array}$$

$$\begin{array}{l}
\delta((q_1, p_2), a) = (q_2, p_2)^* \\
\delta((q_1, p_2), b) = \emptyset \\
\delta((q_1, p_2), c) = (q_2, p_2)^*(q_1, p_2)(q_2, p_2)^*
\end{array}$$

$$\begin{array}{l}
\delta((q_2, p_1), a) = \emptyset \\
\delta((q_2, p_1), b) = (q_2, p_2)^* \\
\delta((q_2, p_1), c) = (q_2, p_2)^*(q_2, p_1)(q_2, p_2)^*
\end{array}$$

$$\begin{array}{l}
\delta((q_2, p_1), a) = \emptyset \\
\delta((q_2, p_1), b) = \emptyset \\
\delta((q_2, p_1), c) = (q_2, p_2)^*
\end{array}$$

[4] Dezelfde constructie wordt als voorbeeld gebruikt in [GV04]. We hebben hier het voorbeeld getest aan de hand van de implementatie die in hoofdstuk 7 beschreven is.

Aan een knoop v van boom t wordt de eindtoestand (q_1, p_1) toegekend als de subtree met wortel v precies één a en precies één b bevat.

2.2.4 Complement

Zij $A = (\Sigma, Q, \delta, F)$ een deterministische en volledige boomautomaat. We construeren hieruit een boomautomaat $A' = (\Sigma, Q', \delta', F')$ met $Q' = Q$, $\delta' = \delta$ en $F' = Q' - F$. Deze automaat aanvaardt het complement van A, namelijk $\mathcal{T}_\Sigma - \mathrm{L}(A)$.
Indien een boomautomaat A niet-deterministisch is, moet eerst een equivalente deterministische boomautomaat A_{det} geconstrueerd worden. Hierdoor zit het algoritme voor het complementeren van een willekeurige boomautomaat A in EXPTIME. Het volledig maken van A_{det} kan efficiënt geïmplementeerd worden (cf. sectie 2.2.2).

Voorbeeld 2.6

We construeren hier de automaat $A' = (\Sigma, Q', \delta', F')$ die het complement aanvaardt van de boomautomaat A uit voorbeeld 2.2. A' aanvaardt dus alle bomen waarvan de booleaanse formules tot *onwaar* geëvalueerd worden en de bomen die geen correcte booleaanse formule uitdrukken. Omdat A deterministisch en volledig is, krijgen we $Q' = \{q_0, q_1, q_{trash}\}$, $F' = \{q_0, q_{trash}\}$ en de transitiefunctie δ' wordt voorgesteld door de volgende regels (R en S zijn gedefinieerd zoals in voorbeeld 2.1):

$$\begin{array}{ll}
\delta'(q_0, 0) = \epsilon & \delta'(q_1, 0) = \emptyset \\
\delta'(q_0, 1) = \emptyset & \delta'(q_1, 1) = \epsilon \\
\delta'(q_0, \vee) = q_0 {q_0}^* & \delta'(q_1, \vee) = (q_0 + q_1)^* q_1 (q_0 + q_1)^* \\
\delta'(q_0, \wedge) = (q_0 + q_1)^* q_0 (q_0 + q_1)^* & \delta'(q_1, \wedge) = q_1 {q_1}^* \\
\delta'(q_0, \neg) = q_1 & \delta'(q_1, \neg) = q_0
\end{array}$$

$$\begin{array}{l}
\delta'(q_{trash}, 0) = Q^* - \{\epsilon\} \\
\delta'(q_{trash}, 1) = Q^* - \{\epsilon\} \\
\delta'(q_{trash}, \vee) = Q^* - R \\
\delta'(q_{trash}, \wedge) = Q^* - S \\
\delta'(q_{trash}, \neg) = Q^* - \{q_0, q_1\}
\end{array}$$

2.2.5 Leegheid

Het leegheidsprobleem voor boomautomaten kan als volgt geformuleerd worden: gegeven een NTA A, ga na of $\mathrm{L}(A) = \emptyset$. We kunnen de leegheid van een boomautomaat testen door na te gaan of er bereikbare eindtoestanden zijn. Een toestand q van een boomautomaat A is *bereikbaar* indien er een boom t en een run λ van A op t bestaat zodat λ de wortel van t op q afbeeldt. Het

is duidelijk dat $\mathrm{L}(A) \neq \emptyset$ als en slechts als er minstens één eindtoestand van A bereikbaar is.

Indien een boomautomaat van de vorm NTA(NFA) is, zit het leegheidsprobleem in PTIME [Nev02a]. We stellen hier een algoritme in PTIME voor. Zij $A = (\Sigma, Q, \delta, F)$ een NTA(NFA). We construeren iteratief de verzameling R van bereikbare toestanden. Aanvankelijk is R de lege verzameling. Voor elke $\sigma \in \Sigma$ en $q \in Q$ wordt nagegaan of de NFA $\delta(q, \sigma)$ over alfabet Q bereikbare eindtoestanden heeft wanneer enkel symbolen uit R gebruikt worden.[5] Indien ja, wordt q aan R toegevoegd. Dit proces wordt herhaald totdat er geen nieuwe toestanden meer aan R toegevoegd kunnen worden. Indien R geen toestand uit F bevat, kunnen we besluiten dat $\mathrm{L}(A) = \emptyset$, anders geldt $\mathrm{L}(A) \neq \emptyset$. Merk op dat R na de eerste iteratie de toestanden $q \in Q$ bevat waarvoor er een $\sigma \in \Sigma$ bestaat zodat $\epsilon \in \delta(q, \sigma)$.

2.2.6 Inclusie en equivalentie

We bespreken hier nog kort de twee volgende beslissingsproblemen.

Inclusie: Gegeven twee NTA's A_1 en A_2, ga na of $\mathrm{L}(A_1) \subseteq \mathrm{L}(A_2)$.

Equivalentie: Gegeven twee NTA's A_1 en A_2, ga na of $\mathrm{L}(A_1) = \mathrm{L}(A_2)$.

Beide problemen zijn sterk gerelateerd. Om de equivalentie van twee automaten A_1 en A_2 te testen, gaan we na of $\mathrm{L}(A_1) \subseteq \mathrm{L}(A_2)$ en $\mathrm{L}(A_2) \subseteq \mathrm{L}(A_1)$. Indien beide testen een positief resultaat geven, zijn de twee automaten equivalent.
Om te testen of $\mathrm{L}(A_1) \subseteq \mathrm{L}(A_2)$ gaan we na of $\mathrm{L}(A_1) - \mathrm{L}(A_2)$ leeg is. Hiertoe construeren we eerst de automaat A_2' die het complement van A_2 herkent, en gaan vervolgens na of $\mathrm{L}(A_1) \cap \mathrm{L}(A_2')$ leeg is. Omdat het berekenen van het complement van een automaat exponentieel is, heeft dit algoritme in zijn totaliteit een exponentiële tijdscomplexiteit.

2.3 Besluit

Omdat een XML-schema kan geabstraheerd worden als boomautomaat, zijn de hier besproken algoritmes op boomautomaten ook van toepassing op XML-schema's. We moeten er wel rekening mee houden dat de reguliere expressies uit de XML-schema's moeten omgezet worden in een equivalente NFA (polynomiaal in de grootte van de expressie) of DFA (exponentieel in de grootte van de expressie). Terwijl het berekenen van de doorsnede van twee boomautomaten efficiënt gebeurt, is dit niet het geval bij het berekenen van

[5] Zij N de NFA die uit $\delta(q, \sigma)$ verkregen wordt door de transities met symbolen buiten R te verwijderen. We gaan dus na of $\mathrm{L}(N) \neq \emptyset$.

het complement van een boomautomaat omdat deze eerst gedeterminiseerd moet worden (exponentieel algoritme). Hierdoor is ook het algoritme voor het oplossen van het equivalentieprobleem exponentieel. Om na te gaan of een boomautomaat de lege taal aanvaardt, is er wel een efficiënt algoritme bekend.

Hoofdstuk 3

Abstracties van XML-schematalen

De belangrijkste bouwstenen van een XML-document zijn elementen, attributen en datawaarden. We kunnen attributen echter beschouwen als elementen waarbij de ordening niet van belang is. Als we vervolgens de concrete datawaarden buiten beschouwing laten - zij hebben immers geen invloed op de structuur van het XML-document -, kan elk well-formed XML-document voorgesteld worden als een boom over een alfabet van elementnamen. Een XML-schema definieert dus in feite een boomtaal. In dit hoofdstuk bespreken we een aantal abstracte XML-schematalen, die elk een specifieke klasse van boomtalen definiëren. We gaan na op welke wijze validatie en typing van een boom ten opzicht van een schema kan gebeuren. Wanneer XML-documenten op een streaming wijze verwerkt worden, is het interessant ze voor te stellen als een string over een alfabet van begin- en eindtags.

3.1 DTD

We behandelen in deze sectie een DTD in zijn abstracte vorm, namelijk als extended context-vrije grammatica. Bij een extended context-vrije grammatica bevatten de regels aan de rechterkant reguliere expressies. Er wordt hier geen onderscheid gemaakt tussen terminalen en niet-terminalen, alle symbolen worden als terminalen beschouwd. Op die manier definieert een DTD een verzameling bomen.[1]

3.1.1 Definitie

Definitie 3.1 (DTD) *Een DTD is een triplet* (Σ, d, s_d) *met* Σ *een alfabet (de elementnamen),* d *een functie die elk symbool uit* Σ *mapt op een reguliere*

[1] Zie [Sip97] voor een formele definitie van context-vrije grammatica's. Een gewone context-vrije grammatica definieert een verzameling strings.

expressie over Σ *en* $s_d \in \Sigma$ *het startsymbool. We korten* (Σ, d, s_d) *gewoonlijk af tot* d *als* Σ *en* s_d *duidelijk zijn uit de context. Een boom is valid ten opzichte van* d *als de wortel als label* s_d *heeft en voor elke knoop met label* a *de sequentie* $a_1 \ldots a_n$ *van labels van zijn kinderen aanvaard wordt door de taal gedefinieerd door* $d(a)$*.* $L(d)$ *is de verzameling van bomen die voldoen aan* d*.*

De reguliere expressie die geassocieerd wordt met een elementnaam wordt ook wel het *content model* van het element genoemd. Het content model van een element hangt enkel af van zijn naam en niet van zijn context. DTD's definiëren daarom locale boomtalen (*local tree languages*). We spreken af dat voor elk alfabetsymbool a dat niet voorkomt aan de linkerkant van een regel $a \rightarrow \epsilon$ toegevoegd wordt.

We verduidelijken de definitie aan de hand van een voorbeeld.

Voorbeeld 3.1

Neem een DTD met alfabet Σ = {garage, auto, merk, prijs, bouwjaar} en startsymbool garage. De mappingfunctie d wordt door de onderstaande regels uitgedrukt.

$$\begin{array}{lcl} garage & \rightarrow & auto\ auto^* \\ auto & \rightarrow & merk\ prijs\ (bouwjaar + \epsilon) \end{array}$$

Deze DTD definieert een verzameling bomen met *garage* als rootelement. Een garage moet minstens één auto te koop aanbieden. Nieuwe auto's worden voorgesteld aan de hand van een merk en prijs, occasies hebben naast een merk en prijs ook nog een bouwjaar.

3.1.2 Validatie

Om na te gaan of een boom valid is ten opzichte van een DTD moet men controleren of het label van de wortel overeenkomt met het startelement en of de string gevormd door de labels van de kinderen van elk element met label a voldoet aan de reguliere expressie $d(a)$. Dit kan efficiënt geïmplementeerd worden aan de hand van een boomautomaat. Elke DTD (Σ, d, s_d) is om te zetten in een equivalente boomautomaat (Q, δ, F) als volgt[2]:

- $Q = \Sigma$
- $F = \{s_d\}$

[2]Gebaseerd op [GV04] en [Nev02a].

- Voor elke regel $d(a) = e$ met a een elementnaam en e een reguliere expressie wordt de transitie $\delta(a, a) = e$ geconstrueerd. Voor elk paar verschillende symbolen a en a' in Σ wordt de transitie $\delta(a, a') = \emptyset$ aan δ toegevoegd.

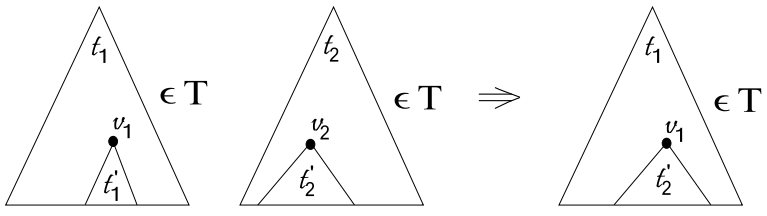
Om na te gaan of een boom t valid is ten opzichte van een DTD d construeren we eerst een equivalente boomautomaat f. Vervolgens runnen we f op input t (lineaire tijdscomplexiteit). Als er een accepterende run bestaat van f op t is t valid.

Bij de implementatie van een boomautomaat kan elke reguliere expressie e uitgedrukt worden als niet-deterministische eindige automaat. Het is niet nodig deze om te zetten in een equivalente deterministische automaat (exponentieel ten opzichte van de grootte van de automaat), omdat het evalueren van een sequentie ten opzichte van een niet-deterministische automaat efficiënt verloopt (lineair ten opzichte van de lengte van de sequentie).

Een alternatief validatiealgoritme, beschreven in [MLMK05], doorloopt een boom in pre-order. Wanneer men een element tegenkomt, gaat men na of zijn kinderen overeenkomen met het content model van dat element. Wanneer men het meest rechtse blad tegenkomt, en alle voorgaande testen zijn geslaagd, wordt de boom valid verklaard. Dit algoritme laat toe een XML-document op streaming wijze te verwerken, wat niet het geval is bij het algoritme gebaseerd op boomautomaten (een XML-boom wordt hier bottom-up geëvalueerd).

3.1.3 Subtree exchange

De expressieve kracht van DTD's kan formeel gekarakteriseerd worden ([PV00] en [MNSB05]). Een reguliere boomtaal T is definieerbaar door een DTD als en slechts als de volgende closure eigenschap geldt: als er twee bomen t_1 en t_2 in T zijn en twee knopen v_1 in t_1 en v_2 in t_2 met hetzelfde label, dan zijn de bomen verkregen door de subbomen van v_1 en v_2 om te wisselen ook in T. Deze eigenschap, *label-guarded subtree exchange* genoemd, wordt geïllustreerd in figuur 3.1.



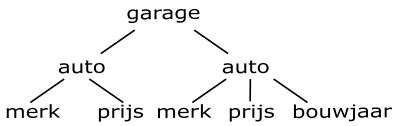
Figuur 3.1: Label-guarded subtree exchange

Deze eigenschap toont duidelijk aan dat DTD's locale boomtalen definiëren.

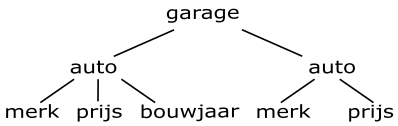
De content van een knoop hangt enkel af van zijn label, vandaar locaal. We kunnen deze eigenschap ook gebruiken om aan te tonen dat bepaalde boomtalen niet kunnen uitgedrukt worden door een DTD.

Voorbeeld 3.2

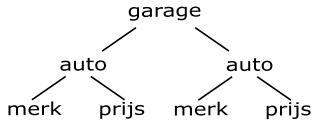
Stel dat we aan de taal gedefinieerd door de DTD uit voorbeeld 3.1 een extra beperking willen opleggen, namelijk dat elke garage tenminste één occasiewagen moet verkopen. Deze taal is niet meer uitdrukbaar door een DTD. De bomen $t_1 =$



en $t_2 =$



zitten in de taal, maar



verkregen uit t_1 door zijn tweede subboom te vervangen door de tweede subboom van t_2 zit niet in de taal.

3.2 EDTD

3.2.1 Definitie

Uit sectie 3.1.3 blijkt dat niet alle reguliere boomtalen kunnen uitgedrukt worden aan de hand van een DTD. Hiervoor is een krachtiger formalisme nodig. Wanneer we de expressieve kracht van DTD's uitbreiden met de notie van *types*, bekomen we een *extended DTD*.[3]

Definitie 3.2 (EDTD) *Een extended DTD (EDTD) is een tupel* $D = (\Sigma, \Delta, d, \mu)$ *met* Σ *een alfabet,* Δ *een alfabet van types,* d *een DTD over* Δ *en* μ *een mapping van* Δ *op* Σ*. Een boom* t *is valid ten opzichte van* D *als er een* $t' \in L(d)$ *bestaat zodat* $t = \mu(t')$ *(*μ *is hier uitgebreid op bomen). L(D) is de verzameling van bomen die voldoen aan D.*

De klasse van talen die door EDTD's gedefinieerd wordt, komt precies overeen met de reguliere boomtalen (*regular tree languages*) [BKMW01]. Dit

[3] Geïntroduceerd door [PV00] als *specialized DTD*.

betekent dat voor elke EDTD een equivalente niet-deterministische boomautomaat bestaat (en omgekeerd). We spreken af dat types altijd van de vorm a^i zijn met $a \in \Sigma$, $i \in \mathbb{N}$ en $\mu(a^i) = a$.

Voorbeeld 3.3

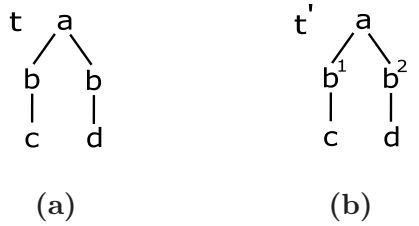
Stel EDTD $D = (\Sigma, \Delta, d, \mu)$ met

- $\Sigma = \{a, b, c, d\}$
- $\Delta = \{a, b^1, b^2, c, d\}$
- d een DTD over Δ met de volgende regels:

$$\begin{array}{lcl} a & \rightarrow & b^1\ b^2 \\ b^1 & \rightarrow & c \\ b^2 & \rightarrow & d \end{array}$$

- $\mu(a) = a$, $\mu(b^1) = b$, $\mu(b^2) = b$, $\mu(c) = c$, $\mu(d) = d$

De boom t in figuur 3.2 (a) is valid ten opzichte van D omdat er een boom t' bestaat die valid is ten opzichte van DTD d en $t = \mu(t')$. De boom t' wordt weergegeven in figuur 3.2 (b). Het root element a heeft twee kinderen b, waarvan het eerste van het type b^1 en het tweede van het type b^2 is.



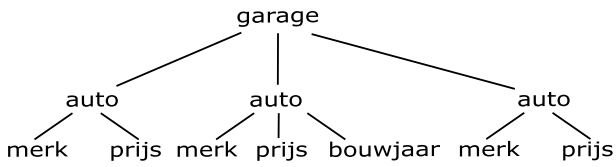
(a) **(b)**

Figuur 3.2: Boom t en getypte boom t'

Voorbeeld 3.4

Laten we nu terugkeren naar het voorbeeld van de garage. Aan de hand van een EDTD kunnen we eisen dat een garage auto's verkoopt waarvan er minstens één occasiewagen is.

$$\begin{array}{lcl} garage & \rightarrow & (auto^1 + auto^2)^*\ auto^2\ (auto^1 + auto^2)^* \\ auto^1 & \rightarrow & merk\ prijs \\ auto^2 & \rightarrow & merk\ prijs\ bouwjaar \end{array}$$



Figuur 3.3: Garage-boom

Hierbij is het alfabet $\Sigma = \{garage, auto, merk, prijs, bouwjaar\}$ en het typealfabet $\Delta = \{garage, auto^1, auto^2, merk, prijs, bouwjaar\}$. De boom uit figuur 3.3 voldoet aan deze EDTD: het eerste en laatste auto-element zijn van het type $auto^1$, het middelste auto-element is van het type $auto^2$.

Het is duidelijk dat de klasse van talen die door EDTD's gedefinieerd wordt de klasse van talen die door DTD's gedefinieerd wordt omvat. Een DTD is immers een EDTD waarbij het elementalfabet gelijk is aan het typealfabet. Daarbij komt nog dat EDTD's strikt krachtiger zijn dan DTD's. Het voorbeeld van de garage toont dit aan: de taal gedefinieerd door bovenstaande EDTD is niet uit te drukken aan de hand van een DTD (cf. sectie 3.1.3 voor een formele benadering).

3.2.2 Validatie en typing

Bij het valideren van een boom t ten opzichte van een EDTD D wordt nagegaan of t aanvaard wordt door D. Typing daarentegen betekent dat men aan elke knoop van t een bepaald type gaat geven, overeenkomstig de regels van D. Bij validatie is de output valid / niet valid, terwijl bij typing de output een getypte boom is. In de praktijk zijn validatie en typing nauw verwant omdat een boom niet expliciet de types weergeeft en ze dus door het validatie-algoritme moeten afgeleid worden.

Typing van DTD's is triviaal: elke elementnaam is uniek en dus kan elk element slechts één type hebben. Bij EDTD's is er geen bovengrens op het aantal types dat met een element geassocieerd kan worden.

Validatie van een boom ten opzichte van een EDTD gebeurt gelijkaardig als bij DTD's. We construeren voor een EDTD $D = (\Sigma, \Delta, d, \mu)$ eerst een equivalente boomautomaat $f = (Q, \delta, F)$ als volgt[4]:

- $Q = \Delta$
- F is een verzameling met één toestand, namelijk het startelement van DTD d

[4]Gebaseerd op [GV04].

- Voor elke regel $d(\tau) = e$ met τ een type en e een reguliere expressie over types wordt de transitie $\delta(\tau, a) = e$ geconstrueerd indien $\mu(\tau) = a$. Omdat δ een totale functie op $Q \times \Sigma$ moet zijn, definiëren we bovendien $\delta(\tau, a) = \emptyset$ wanneer $\mu(\tau) \neq a$.

Vervolgens runnen we f op input t (lineaire tijdscomplexiteit). Als er een accepterende run bestaat van f op t is t valid.

Ook het typing-algoritme kan geïmplementeerd worden aan de hand van een boomautomaat, omdat een accepterende run overeenkomt met een getypte boom. Dit algoritme doorloopt een boom bottom-up in lineaire tijd.

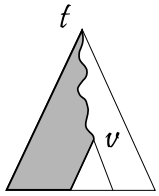
Merk op dat met een bepaalde boom verschillende getypete bomen kunnen overeenkomen. De EDTD met regels $a \rightarrow b^1 + b^2$, $b^1 \rightarrow c$ en $b^2 \rightarrow c$ kan aan een element b uit de boom $a(b(c))$ zowel het type b^1 als b^2 toekennen. Dit wordt ambigue typing genoemd.

3.3 Intermezzo: One-pass preorder typing

Zoals reeds vermeld komt de expressieve kracht van EDTD's overeen met de welbekende en robuuste klasse van reguliere boomtalen. Hiervoor wordt echter een prijs betaald. Het verwerken van XML-documenten gebeurt namelijk bottom-up, waarbij het type van een element pas wordt bepaald na het lezen van zijn content. In de context van streaming XML data of voor SAX-gebaseerde processing is het gewenst om het type van een element te bepalen op het moment dat men de begintag tegenkomt. Een EDTD die het toelaat een XML-document te typen wanneer men de begintag tegenkomt, noemt men *one-pass preorder typeable* (1PPT). Merk op dat niet elke EDTD one-pass preorder typing toelaat. Neem bijvoorbeeld de EDTD bestaande uit de regels $a \rightarrow b^1 + b^2$, $b^1 \rightarrow c$, $b^2 \rightarrow d$ en XML-document $< a >< b >< d/ >< /b >< /a >$. Hier hangt het type van b af van het label van zijn kind. Het is dus onmogelijk om het type van b te bepalen wanneer men zijn begintag $< b >$ tegenkomt, dus zonder eerst naar zijn kind te kijken.

Een alternatieve formulering van 1PPT geeft aan dat het type van een element niet mag afhangen van wat in document orde na dat element volgt. Daarom is het noodzakelijk dat het type van een element uniek wordt bepaald door wat aan dat element voorafgaat (de *preceding* van een element). In figuur 3.4 is de preceding van een knoop v in een boom t gearceerd.

Vooraleer over te gaan tot een formele definitie van 1PPT, formaliseren we eerst het begrip preceding. De *preceding* van een knoop v in een boom t is de deelboom van t bestaande uit alle knopen die in document order vóór v voorkomen en v zelf. Dit noteren we als $\text{preceding}^t(v)$.



Figuur 3.4: Preceding van een knoop v

Definitie 3.3 (1PPT) *Een EDTD $D = (\Sigma, \Delta, d, \mu)$ is one-pass preorder typeable (1PPT) of heeft preceding-based types als er een (partiële) functie $f : \mathcal{T}_\Sigma \to \Delta$ bestaat zodat voor elke boom $t \in L(D)$ er een unieke boom $t' \in L(d)$ is met $\mu(t') = t$ zodat het volgende geldt: voor elke knoop v in t is het label van v in t' gelijk aan $f(\text{preceding}^t(v))$.*

Een gevolg van de definitie van 1PPT is dat aan ieder element een uniek type moet toegekend kunnen worden. Dit is bijvoorbeeld niet het geval bij de EDTD met regels $a \to b^1 + b^2$, $b^1 \to c$, $b^2 \to c$ en boom $a(b(c))$. Aan b kan zowel type b^1 als b^2 toegekend worden, vandaar dat hier sprake is van ambigue typing.

Merk op dat 1PPT een semantische notie is, maar precies daarom heel robuust. Ze definieert de grootste subklasse van EDTD's waarbij typing mogelijk is wanneer een document op streaming wijze wordt verwerkt. In wat volgt gaan we na aan welke restricties een EDTD moet onderworpen worden om 1PPT toe te laten.

3.4 Single-type EDTD's

Men kan een EDTD verbieden regels te hebben waarbij de reguliere expressie twee verschillende types bevatten die overeenkomen met eenzelfde elementnaam. Dit kan geformaliseerd worden als single-type EDTD's.

3.4.1 Definitie

Definitie 3.4 (Single-type EDTD) *Een EDTD (Σ, Δ, d, μ) is single-type indien in geen enkele reguliere expressie twee types $\tau \neq \tau'$ voorkomen met $\mu(\tau) = \mu(\tau')$.*

We spreken bovendien af dat het startelement een uniek type heeft.

Merk op dat deze definitie een louter syntactische beperking is op EDTD's. Gegeven een EDTD kan men gemakkelijk nagaan of deze single-type is door elke reguliere expressie afzonderlijk te bekijken. Komt er in een reguliere expressie eenzelfde element meerdere keren voor, dan moeten deze elementen

hetzelfde type hebben, zoniet is de EDTD niet single-type. De EDTD met als regels $a \rightarrow b^1b^2$, $b^1 \rightarrow c$, $b^2 \rightarrow d$ is niet single-type omdat de reguliere expressie b^1b^2 meerdere b elementen bevat die niet hetzelfde type hebben. Ook de EDTD in voorbeeld 3.4 is niet single-type omdat de regel voor *garage* zowel het $auto^1$ als $auto^2$ type bevat. Een voorbeeld van een single-type EDTD is de volgende.

Voorbeeld 3.5

$$
\begin{array}{lcl}
garage & \rightarrow & nieuw\ occasie \\
nieuw & \rightarrow & (auto^1)* \\
occasie & \rightarrow & auto^2\ (auto^2)* \\
auto^1 & \rightarrow & merk\ prijs \\
auto^2 & \rightarrow & merk\ prijs\ bouwjaar
\end{array}
$$

Deze EDTD bevat de types $auto^1$ en $auto^2$ voor eenzelfde element *auto*, maar ze komen niet tesamen in dezelfde reguliere expressie voor.

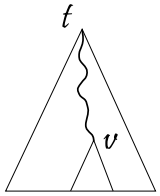
3.4.2 Validatie en typing

Voor single-type EDTD's bestaat er een eenvoudign top-down validatie algoritme dat aan elk element een uniek type toekent ([MNSB05] en [MLMK05]). Dit algoritme gaat als volgt. Ken aan het root element van boom t het uniek type toe. Voor elke interne knoop v met type τ wordt de corresponderende regel $\tau \rightarrow r$ opgezocht en gaat men na of de kinderen van v aanvaard worden door $\mu(r)$. Met $\mu(r)$ wordt de reguliere expressie bedoeld die uit r geconstrueerd wordt door elk type te vervangen door zijn overeenkomende elementnaam. Als deze test niet slaagt, is t niet valid. Als deze test wel slaagt, wordt aan elk kind een uniek type toegekend omwille van de single-type eigenschap. Dit proces eindigt bij de bladeren van t als elke voorafgaande test geslaagd is en t wordt valid verklaard.

Dit algoritme verloopt duidelijk volgens het principe van 1PPT. Van zodra men de openingstag van een element tegenkomt, kan dit element uniek getyped worden. Er kan nooit enige verwarring zijn omtrent het type van een kind, gegeven een getypte parent. Voor elke elementnaam kan in elke regel immers hooguit één type voorkomen.

Men noemt dit soort van typing ook wel ancestor-based typing, omdat het type van een element enkel en alleen afhangt van de (voor)ouders van dat element. Dit wordt schematisch voorgesteld in figuur 3.5. De alternatieve karakterisaties in de volgende sectie maken deze relatie expliciet.

Omdat hier enkel de voorouders het type van een element bepalen, kan men zich afvragen of er geen grotere subklasse EDTD's bestaat die 1PPT toelaten. Uit de definitie van 1PPT volgt immers dat het type van een element



Figuur 3.5: Ancestor-based typing

van de hele preceding van dat element mag afhangen, dus niet noodzakelijk enkel van de voorouders. We geven hierop een antwoord in sectie 3.5.

3.4.3 Alternatieve karakterisaties

We bespreken hier een aantal alternatieve karakterisaties die qua expressiviteit equivalent zijn aan single-type EDTD's.

Ancestor-based schema's

Ancestor-based schema's zijn een instantiatie van de meer algemene pattern-based schema's. Bij pattern-based schema's wordt de context van elementen geëxpliciteerd. In het geval van ancestor-based schema's wordt per element aangegeven aan welk(e) pattern(s) de ancestor-string van dat element moet voldoen. We geven eerst de formele definitie van een pattern-based schema. Neem een taal $\mathcal{P}$ die unaire patterns definieert. Dit betekent dat elk pattern $\phi \in \mathcal{P}$ uit elke boom t een verzameling knopen selecteert, wat we noteren met $\phi(t)$ ($\mathcal{P}$ kan bijvoorbeeld overeenkomen met reguliere expressies of XPath).

Definitie 3.5 *Een $\mathcal{P}$-schema is een triplet $S = (\Sigma, \Delta, R)$ met Σ een alfabet over elementnamen, Δ een alfabet over types en R een verzameling regels van de vorm $\alpha : \phi \rightarrow s$. Hierbij is $\tau \in \Delta$ een type, $\phi \in \mathcal{P}$ een pattern en s een reguliere expressie over Σ. Een boom t is valid ten opzichte van een $\mathcal{P}$-schema als het label van elke knoop tot Σ behoort en er voor elke knoop v in t een regel $\tau : \phi \rightarrow s$ is zodat $v \in \phi(t)$ en de kinderen van v voldoen aan de reguliere expressie s; aan v wordt dan het type τ toegekend.*

Voor de definitie van ancestor-based schema's introduceren we eerst de volgende twee begrippen. Neem een boom t en v een knoop in t. Noteer het label van v als $\text{lab}^t(v)$. De *child-string* van v, genoteerd als $\text{ch-str}^t(v)$, is de string gevormd door de kinderen van v, meer bepaald $\text{lab}^t(v1) \ldots \text{lab}^t(vn)$ indien v n kinderen heeft. De *ancestor-string* van v, genoteerd als $\text{anc-str}^t(v)$, is de string gevormd door de labels op het pad van de wortel tot v, meer bepaald $\text{lab}^t(\epsilon)\ \text{lab}^t(i_1)\ \text{lab}^t(i_1 i_2) \ldots \text{lab}^t(i_1 i_2 \ldots i_k)$ waarbij $v = i_1 i_2 \ldots i_k$. Het superscript t wordt vaak achterwege gelaten.

Definitie 3.6 *Een ancestor-based schema S is een pattern-based schema (Σ, Δ, R) waarbij alle regels van de vorm $\tau : r \rightarrow s$ zijn met r en s reguliere expressies over Σ. Een boom t voldoet aan S als er voor elke knoop v in t een regel $\tau : r \rightarrow s$ is zodat anc-str(v) voldoet aan r en ch-str(v) voldoet aan s.*

Een voorbeeld verduidelijkt deze definitie.

Voorbeeld 3.6

garage:	*garage*	$\rightarrow$	*nieuw occasie*
nieuw:	*garage nieuw*	$\rightarrow$	$(auto)^*$
occasie:	*garage occasie*	$\rightarrow$	*auto* $(auto)^*$
$auto^1$:	Σ^* *nieuw auto*	$\rightarrow$	*merk prijs*
$auto^2$:	Σ^* *occasie auto*	$\rightarrow$	*merk prijs bouwjaar*

Dit schema is equivalent met de single-type EDTD uit voorbeeld 3.5.

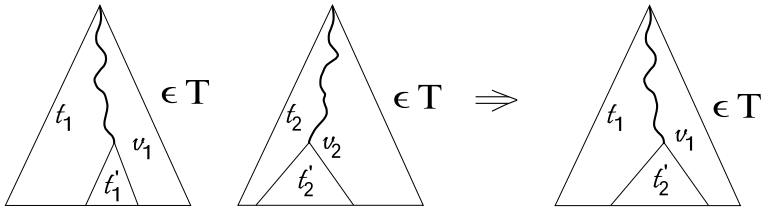
Ancestor-guarded subtree exchange

In sectie 3.1.3 hebben we aangetoond dat niet elke reguliere taal kan uitgedrukt worden door een DTD aan de hand van de eigenschap label-guarded subtree exchange. Single-type EDTD's kunnen op een gelijkaardige manier gekarakteriseerd worden.

Eerst volgen enkele afspraken qua notatiewijze. Wanneer we in een boom t_1 de subtree met wortel v vervangen door een boom t_2 verkrijgen we een nieuwe boom die we noteren als $t_1[v \leftarrow t_2]$. De subtree in t met wortel v noteren we als $\text{subtree}^t(v)$.

Definitie 3.7 *Een boomtaal T is gesloten onder ancestor-guarded subtree exchange als het volgende geldt. Als voor twee bomen t_1, $t_2 \in T$ met als knopen respectievelijk v_1 en v_2 geldt dat $\text{anc-str}^{t_1}(v_1) = \text{anc-str}^{t_2}(v_2)$ dan geldt ook $t_1[v_1 \leftarrow \text{subtree}^{t_2}(v_2)] \in T$.*

Deze definitie wordt geïllustreerd in figuur 3.6.



Figuur 3.6: Ancestor-guarded subtree exchange

Deze eigenschap is een handig middel om aan te tonen dat er reguliere talen bestaan die niet uit te drukken zijn als single-type EDTD. De taal uit voorbeeld 3.4 is niet uitdrukbaar als single-type EDTD. Een tegenvoorbeeld wordt op exact dezelfde manier als in voorbeeld 3.2 geconstrueerd.

Ancestor-based types

Definitie 3.8 *Een EDTD $D = (\Sigma, \Delta, d, \mu)$ heeft ancestor-based types als er een (partiële) functie $f : \Sigma^* \to \Delta$ bestaat zodat voor elke boom $t \in L(D)$ er een unieke boom $t' \in L(d)$ is met $\mu(t') = t$ zodat het volgende geldt: voor elke knoop v in t is het label van v in t' gelijk aan $f(\text{anc-str}^t(v))$.*

Een EDTD is single-type als en slechts deze ancestor-based types heeft. Deze stelling benadrukt dat het type van een knoop bij een single-type EDTD enkel afhangt van zijn ancestor-string. Dit was reeds duidelijk door de beschrijving van het top-down validatie algoritme in sectie 3.4.2, maar het impliceert ook dat elke EDTD waarbij typing op deze manier mogelijk is in feite een single-type EDTD is.

Ancestor-based patterns

Definitie 3.9 *Neem een verzameling bomen T. T kan gekarakteriseerd worden aan de hand van ancestor-based patterns als er een reguliere stringtaal L over $\Sigma \cup \{\#\}$ bestaat zodat voor elke boom t geldt: $t \in T$ als en slechts als $P_{anc}(t) \subseteq L$ met $P_{anc}(t) = \{\text{anc-str}(v)\#\text{ch-str}(v) \mid v \in t\}$.*

Een EDTD D is single-type als en slechts als L(D) gekarakteriseerd kan worden aan de hand van ancestor-based patterns. Deze eigenschap toont aan dat bijvoorbeeld het testen van equivalentie en inclusie kan gereduceerd worden tot het testen van het overeenkomstige probleem bij reguliere stringtalen.

3.4.4 Equivalentiestelling voor ancestor-based schema's

Dat de alternatieve karakterisaties voor single-type EDTD's allemaal equivalent zijn, blijkt uit de volgende stelling. Een boomtaal is homogeen als de wortels van alle bomen uit die taal hetzelfde label hebben.

Stelling 3.1 *Voor homogene reguliere boomtalen T zijn volgende uitspraken equivalent:*

(a) T is definieerbaar door een single-type EDTD.

(b) T is definieerbaar door een EDTD met ancestor-based types.

(c) T is gesloten onder ancestor-guarded subtree exchange.

(d) T kan gekarakteriseerd worden door ancestor-based patterns.

(e) T is definieerbaar door een ancestor-based schema's.

Voor een gedetailleerd bewijs hiervan verwijzen we naar [MNSB05]. In dit bewijs wordt o.a. uitgelegd hoe we uit een gegeven EDTD D een equivalente single-type EDTD D_1 kunnen construeren indien die bestaat. Daarnaast wordt aangetoond dat indien $\mathrm{L}(D) \neq \mathrm{L}(D_1)$ er ook geen andere single-type EDTD $D_2 \neq D_1$ bestaat waarvoor geldt dat $\mathrm{L}(D) = \mathrm{L}(D_2)$. We stellen de constructie van D_1 uit D hieronder voor.

Neem EDTD $D = (\Sigma, \Delta, d, \mu)$ die niet single-type is, maar waarvoor er een equivalente single-type EDTD bestaat. We construeren uit D een equivalente single-type EDTD $D_1 = (\Sigma, \Delta_1, d_1, \mu_1)$. Voor een reguliere expressie r over Δ en $C \subseteq \Delta$ noteren we r^C voor de expressie die uit r verkregen wordt door elke symbool a^i te vervangen door a^C.

- Δ_1 is de verzameling van alle symbolen a^M met $a \in \Sigma$ en $M \subseteq \{1, \ldots, k_a\}$. Voor elk element i uit $\{1, \ldots, k_a\}$ behoort a^i tot Δ.
- Voor elk symbool a^M definiëren we $\mu_1(a^M) = a$.
- Voor elk symbool a^M definiëren we $d_1(a^M) = \bigcup_{i \in M} d(a^i)^{C(a^M)}$, waarbij $C(a^M)$ de verzameling is van alle elementen b^j die voorkomen in $\bigcup_{i \in M} d(a^i)$.

We illustreren dit laatste aan de hand van een voorbeeld. Neem de regels $d(a^1) = a^1 a^2 (b^1 + b^3)$ en $d(a^2) = a^1 + b^3$. We construeren nu $d_1(a^M)$ met $M = \{1, 2\}$. We hebben $C(a^{\{1,2\}}) = \{a^1, a^2, a^3, b^1, b^3\}$ en dus $d_1(a^{\{1,2\}}) = (a^{\{1,2,3\}} a^{\{1,2,3\}} (b^{\{1,3\}} + a^{\{1,2,3\}})) + (a^{\{1,2,3\}} + b^{\{1,3\}})$.
Merk op dat D_1 exponentieel groter is dan D.

3.5 Restrained-competition EDTD's

De beperking van EDTD's tot single-type EDTD's maakt one-pass preorder typing mogelijk. De vraag nu is of er ook EDTD's bestaan die niet single-type zijn maar toch one-pass preorder typeable. Het antwoord hierop is bevestigend. De EDTD met regels $a \to (b^1)^*\ c\ b^2$, $b^1 \to x$, $b^2 \to y$ is niet single-type, maar laat wel 1PPT toe. Hier kan men aan de linkerbroers van b zien als welk type het b element voorkomt: indien c nog niet is tegengekomen is b van het type b^1, anders b^2. Hoewel single-type EDTD's efficiënte en unieke typing toelaten, kunnen zij niet alle 1PPT EDTD's uitdrukken.

3.5.1 Definitie

We versoepelen de single-type beperking op EDTD's en definiëren de ruimere klasse restrained competition EDTD's.

Definitie 3.10 (Restrained competition EDTD) *Een EDTD* (Σ, Δ, d, μ) *is restrained competition indien elke reguliere expressie voldoet aan de restrained competition eigenschap. Een reguliere expressie* r *is restrained competition indien er geen twee strings* $w\tau v$ *en* $w\tau' v'$ *in* $L(r)$ *voorkomen met* $\tau \neq \tau'$ *en* $\mu(\tau) = \mu(\tau')$.

Zoals bij single-type EDTD's spreken we ook hier af dat het startelement met een uniek type kan geassocieerd worden. Intuïtief gezien is een EDTD restrained comptetition als bij het van links naar rechts doorlopen van de kinderen van een knoop het direct duidelijk is welk type met elke knoop geassocieerd moet worden, zonder vooruit te kijken naar zijn rechterbroers. Het type van een knoop wordt dus bepaald door zijn voorouders en hun respectievelijke linkerbroers. Hieruit volgt dat 1PPT mogelijk is.

We keren terug naar het garage voorbeeld. Het volgende voorbeeld geeft een restrained competition EDTD weer die niet single-type is.

Voorbeeld 3.7

$$
\begin{array}{lcl}
garage & \rightarrow & (auto^1)^*\ occasies\ (auto^2)^* \\
occasies & \rightarrow & \epsilon \\
auto^1 & \rightarrow & merk\ prijs \\
auto^2 & \rightarrow & merk\ prijs\ bouwjaar
\end{array}
$$

De expressie $(auto^1)^*\ occasies\ (auto^2)^*$ is restrained competition omdat de types van links naar rechts als volgt worden toegekend: een *auto*-element is van het type $auto^1$ zolang het *occasies*-element niet tegengekomen is, anders is het type $auto^2$.

De reguliere expressie $(auto^1 + auto^2)^*\ auto^2\ (auto^1 + auto^2)^*$ uit voorbeeld 3.4 is niet restrained competition. Neem bijvoorbeeld $w = \epsilon$, $\tau = auto^2$, $\tau' = auto^1$, $v = \epsilon$ en $v' = auto^2$.

Zoals bij single-type EDTD's kan men elke reguliere expressie afzonderlijk bekijken om te zien of een EDTD voldoet aan deze eigenschap. In tegenstelling tot de single-type eigenschap is dit echter een semantische definitie. Toch kan men in polynomiale tijd beslissen of een EDTD restrained competition is (cf. hoofdstuk 4).

3.5.2 Alternatieve karakterisaties

We bespreken hier een aantal alternatieve karakterisaties die qua expressiviteit equivalent zijn aan restrained competition EDTD's en die bijgevolg 1PPT toelaten.

Ancestor-sibling-based schema's

Ancestor-sibling based schema's vormen een syntactische tegenhanger voor restrainded competition EDTD's.

Het komt erop neer een geschikte pattern taal $\mathcal{R}$ te definiëren die uit een boom knopen kan selecteren op basis van hun voorouders en hun respectievelijke linkerbroers. Dit is bijvoorbeeld mogelijk aan de hand van reguliere expressies over symbolen van de vorm $a[r]$ met a een elementnaam een r een reguliere expressie over elementnamen. We korten $a[\Sigma^*]$ af als a. Een symbool $a[r]$ matcht een knoop v als v gelabeld is met a en de string gevormd door de labels van de linkerbroers van v matcht met r. We leggen nu uit hoe een expressie ϕ kan gebruikt worden als unair pattern. Neem een knoop v uit een boom t met $v_1, \ldots, v_n$ het pad van de wortel v_1 tot $v = v_n$. Voor elke i noteren we a_i als label van v_i en w_i de string gevormd door de labels van de linkerbroers van v_i, zonder het label van v_i zelf. Een knoop v wordt geselecteerd door een pattern ϕ als er een string $a_1a_2[r_2]\ldots a_n[r_n] \in \mathrm{L}(\phi)$ bestaat zodat voor elke $i = 2, \ldots, n$ geldt dat $w_i \in \mathrm{L}(r_i)$.

Definitie 3.11 *Een ancestor-sibling-based schema S is een pattern-based schema (Σ, Δ, R) waarbij alle regels van de vorm $\tau : \phi \rightarrow s$ zijn met τ een type uit Δ, ϕ een pattern van $\mathcal{R}$ en s een reguliere expressie over Σ. Een boom t voldoet aan S als er voor elke knoop v in t een regel $\tau : \phi \rightarrow s$ bestaat zodat v geselecteerd wordt door pattern ϕ en ch-str(v) voldoet aan s.*

Voorbeeld 3.8

garage:	*garage*	$\rightarrow$	*auto* occasies auto**
discounts:	*discounts*	$\rightarrow$	ϵ
$auto^1$:	*garage auto*[*auto**]	$\rightarrow$	*merk prijs*
$auto^2$:	*garage auto*[*auto* occasies auto**]	$\rightarrow$	*merk prijs bouwjaar*

Dit schema is equivalent met de restrained competition EDTD uit voorbeeld 3.7. We zouden ook full XPath kunnen gebruiken als pattern taal, maar zijn evaluatie beperken tot de preceding van elke knoop.

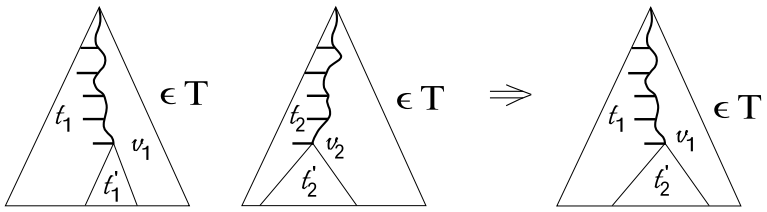
Ancestor-sibling-guarded subtree exchange

Dat niet elke reguliere boomtaal kan uitgedrukt worden als restrained competition EDTD, kan gemakkelijk aangetoond worden aan de hand van de eigenschap ancestor-sibling-guarded subtree exchange.

Hiertoe introduceren we eerst twee nieuwe begrippen. De *left-sibling-string* van v, genoteerd als l-sib-str$^t(v)$, is de string gevormd door de labels van de linkerbroers van v, namelijk lab$^t(u1) \ldots$ lab$^t(uk)$ met $v = uk$. De *ancestor-sibling-string* van v, genoteerd als anc-sib-str$^t(v)$, is de string l-sib-str$^t(v_1)$#l-sib-str$^t(v_2)$#...#l-sib-str$^t(v_n)$#l-sib-str$^t(v)$ gevormd door de concatenatie van de left-sibling strings van alle voorouders v_1, v_2, ..., v_n van v vertrekkend vanaf de wortel v_1.

Definitie 3.12 *Een boomtaal T is gesloten onder ancestor-sibling-guarded subtree exchange als het volgende geldt. Als voor twee bomen t_1, $t_2 \in T$ met als knopen respectievelijk v_1 en v_2 geldt dat anc-sib-str$^{t_1}(v_1)$ = anc-sib-str$^{t_2}(v_2)$ dan geldt ook $t_1[v_1 \leftarrow$ subtree$^{t_2}(v_2)] \in T$.*

Deze definitie wordt geïllustreerd in figuur 3.7.



Figuur 3.7: Ancestor-sibling-guarded subtree exchange

Ancestor-sibling-based types

Definitie 3.13 *Een EDTD $D = (\Sigma, \Delta, d, \mu)$ heeft ancestor-sibling-based types als er een (partiële) functie $f : (\Sigma \cup \{\#\})^* \rightarrow \Delta$ bestaat zodat voor elke boom $t \in L(D)$ er een unieke boom $t' \in L(d)$ is met $\mu(t') = t$ zodat het volgende geldt: voor elke knoop v in t is het label van v in t' gelijk aan f(anc-sib-str$^t(v)$).*

Omdat de ancestor-sibling-string van een element deel uitmaakt van zijn preceding is 1PPT per definitie mogelijk.

Ancestor-sibling-based patterns

Definitie 3.14 *Neem een verzameling bomen T. T kan gekarakteriseerd worden aan de hand van ancestor-sibling-based patterns als er een reguliere stringtaal L bestaat zodat voor elke boom t geldt: $t \in T$ als en slechts als $P_{anc-sib}(t) \subseteq L$ met $P_{anc-sib}(t) = \{$anc-sib-str(v)#ch-str$(v) \mid v \in t\}$.*

Deze eigenschap toont aan dat bijvoorbeeld het testen van equivalentie en inclusie kan gereduceerd worden tot het corresponderende probleem bij reguliere stringtalen.

3.5.3 Equivalentiestelling voor ancestor-sibling-based schema's

Onderstaande stelling geeft aan dat de alternatieve karakterisaties van restrained competition EDTD's terecht als equivalent kunnen beschouwd worden. Meer nog, ze geeft ook aan dat de talen die uitgedrukt kunnen worden aan de hand van een restrained comptetition EDTD precies overeenkomen met de talen die 1PPT zijn.

Stelling 3.2 *Voor homogene reguliere boomtalen T zijn volgende uitspraken equivalent:*

(a) T is definieerbaar door een one-pass preorder typeable EDTD.

(b) T is definieerbaar door een restrained competition EDTD.

(c) T is definieerbaar door een EDTD met ancestor-sibling-based types.

(d) T is gesloten onder ancestor-sibling-guarded subtree exchange.

(e) T kan gekarakteriseerd worden door ancestor-sibling-based patterns.

(f) T is definieerbaar door een ancestor-sibling-based schema's.

Voor een bewijs hiervan verwijzen we naar [MNSB05]. In dit bewijs wordt o.a. uitgelegd hoe we uit een gegeven EDTD D een equivalente restrained competition EDTD D_1 kunnen construeren indien die bestaat. Daarnaast wordt aangetoond dat indien $\mathrm{L}(D) \neq \mathrm{L}(D_1)$ er ook geen andere restrained competition EDTD $D_2 \neq D_1$ bestaat waarvoor geldt dat $\mathrm{L}(D) = \mathrm{L}(D_2)$. We leggen hier aan de hand van een voorbeeld de gedachte achter deze constructie uit.

Neem een EDTD D bestaande met o.a. de regels $a^1 \rightarrow a^1(b^1c + b^2)b^3$, $b^1 \rightarrow a^1a^2$ en $b^2 \rightarrow a^1a^1$. In de regel voor a^1 kunnen de types b^1 en b^2 voorkomen in matching strings met eenzelfde prefix, bijvoorbeeld $a^1b^1cb^3$ en $a^1b^2b^3$, waardoor b^1 en b^2 *competing* types zijn. Door deze types te mergen, krijgen we de regel $a^1 \rightarrow a^1(b^{\{1,2\}}c + b^{\{1,2\}})b^3$, waarvan de reguliere expressie restrained competition is. Nu moeten we een nieuwe regel voor $b^{\{1,2\}}$ construeren die de regels voor b^1 en b^2 combineert: $b^{\{1,2\}} \rightarrow a^1a^2 + a^1a^1$. Ook hier moeten *competing* types gemerged worden, wat resulteert in de regel $b^{\{1,2\}} \rightarrow a^1a^{\{1,2\}} + a^1a^{\{1,2\}}$.
De resulterende EDTD D_1 is exponentieel groter dan D.

Het meest opmerkelijke resultaat is dat (a) in het lijstje van de stelling voorkomt. Hoewel de definitie van 1PPT expliciet toelaat dat het type van een element kan afhangen van de volledige preceding van dat element, blijkt

de ancestor-sibling-string reeds voldoende te zijn. We tonen hier aan dat de klasse van preceding-based types samenvalt met de klasse van ancestor-sibling based types. De implicatie (c) $\Rightarrow$ (a), volgt rechtstreeks uit de definitie van 1PPT. We tonen hier de omgekeerde implicatie aan, namelijk (a) $\Rightarrow$ (c).
Neem een EDTD $D = (\Sigma, \Delta, d, \mu)$ met preceding based types. Om te komen tot een contradictie veronderstellen we dat D types heeft die niet ancestor-sibling gebaseerd zijn. Neem de bomen t_1, $t_2 \in \mathrm{L}(D)$ met knopen $v_1 \in t_1$ en $v_2 \in t_2$ zodat anc-sib-str$^{t_1}(v_1)$ = anc-sib-str$^{t_2}(v_2)$ maar waarbij v_1 een verschillend label heeft in t'_1 dan v_2 in t'_2 met t'_1 en t'_2 de unieke bomen zodat $\mu(t'_1) = t_1$ en $\mu(t'_2) = t_2$. Noem t_1, t_2, v_1 en v_2 het tegenvoorbeeld waarvoor de lengte van anc-sib-str$^{t_1}(v_1)$ minimaal is.
Benoem de linkerbroers van de voorouders van v_1 als $u_1, \ldots, u_n$ in de volgorde waarin ze voorkomen als t_1 breadth-first doorlopen wordt. De corresponderende knopen in t_2 noemen we $w_1, \ldots, w_n$. Omdat we het tegenvoorbeeld zo gekozen hebben dat de lengte van anc-sib-str$^{t_1}(v_1)$ minimaal is, geldt dat voor elke $i \leq n$ het label van u_i in t'_1 hetzelfde is als het label van w_i in t'_2. Neem s de boom die uit t_1 verkregen wordt door de subtree met wortel u_i te vervangen door de subtree met wortel w_i uit t_2 voor elke i. Neem s' de boom met dezelfde verzameling knopen als s. De labels van s' komen overeen met de labels van t'_2 voor de knopen die in s vervangen zijn en de labels van t'_1 voor de overige knopen. Het is duidelijk dat $s' \in \mathrm{L}(d)$. Omdat preceding$^{s}(v_1)$ = preceding$^{t_2}(v_1)$ heeft v_1 in s' en t'_2 hetzelfde label. Maar omdat v_1 ook in t'_1 en s' hetzelfde label heeft, volgt hieruit dat v_1 hetzelfde label moet hebben in t'_1 als v_2 in t'_2, wat een contradictie is.

3.5.4 Validatie en typing

Restrained competition EDTD's zijn even krachtig als 1PPT EDTD's. Er bestaan dus algoritmes die een gegeven XML-document op een streaming manier kunnen valideren en typen. Uit [MNSB05] blijkt dat dit geïmplementeerd kan worden aan de hand van een deterministische pushdown automaat met als maximale stackgrootte de diepte van het XML-document. We gaan hier echter niet dieper op in.

3.6 Besluit

Terwijl aan de hand van een DTD alle locale boomtalen kunnen gedefinieerd worden, komen EDTD's overeen met de klasse van reguliere boomtalen. Omdat niet alle EDTD's one-pass preorder typing toelaten, zijn we op zoek gegaan naar subklassen waarvoor dit wel het geval is. Bij een single-type EDTD hangt het type van een element af van zijn ancestor-string, bij een restrained competition EDTD hangt het type van een element af van zijn

ancestor-sibling string. Hierdoor zijn beide klassen one-pass preorder typeable. Meer nog, restrained competition EDTD's en 1PPT EDTD's zijn qua expressiviteit even krachtig. We kunnen besluiten dat DTD $\subset$ single-type EDTD $\subset$ restrained competition EDTD = 1PPT EDTD $\subset$ EDTD.

Hoofdstuk 4

Recognitie en simplificatie van EDTD's

In de hoofdstuk bespreken we twee belangrijke beslissingsproblemen betreffende EDTD's.[1]

Recognitie: Gegeven een EDTD, ga na of dit een DTD, een single-type EDTD of een restrained competition EDTD is.

Simplificatie: Gegeven een EDTD, ga na of er een equivalente DTD, single-type EDTD of restrained competition EDTD bestaat.

4.1 Recognitie

Omdat de definitie van een DTD en single-type EDTD syntactisch van aard is, kan men rechtstreeks nagaan of een gegeven EDTD hieraan voldoet. Bij een DTD mag er voor elk element slechts één regel gedefinieerd zijn, bij een single-type EDTD kan men voor elke reguliere expressie gemakkelijk nagaan of er geen twee verschillende types voorkomen die geassocieerd zijn met eenzelfde elementnaam. Voor restrained competition EDTD's geldt de volgende eigenschap:

Stelling 4.1 *Er bestaat een algoritme dat nagaat of een EDTD D restrained competition is met een kwadratische tijdscomplexiteit in de grootte van D.*

Het volstaat aan te tonen dat het testen van de restrained competition eigenschap op elke reguliere expressie afzonderlijk in kwadratische tijd mogelijk is. Neem een reguliere expressie r en construeer een equivalente NFA $N_r = (Q, \delta, q_0, F)$ over het alfabet Δ. N_r bestaat uit $\mathcal{O}(n)$ toestanden en wordt berekend in $\mathcal{O}(n \log^2(n))$ tijd, met n de lengte van r [HSW01]. Vervolgens

[1] De resultaten die in de hoofdstuk worden besproken, zijn gebaseerd op [MNSB05], sectie 10.

wordt nagegaan of er twee verschillende toestanden bereikbaar zijn vanuit eenzelfde string en of vanuit die twee toestanden een pad vertrekt naar een eindtoestand. Hiervoor worden er twee verzamelingen geconstrueerd:

- $R = \{q \in Q \mid \exists\ w \in \Delta^*,\ \delta^*(q,w) \in F\}$ is de verzameling toestanden van waaruit een eindtoestand kan bereikt worden, en
- $S = \{(q_1, q_2) \in Q \times Q \mid \exists w \in \Delta^*, \{q_1, q_2\} \subseteq \delta^*(q_0, w)\}$ is de verzameling koppels van toestanden die kunnen bereikt worden door eenzelfde string.

Beide verzamelingen kunnen in lineaire respectievelijk kwadratische tijd samengesteld worden. Nu volgt dat r restrained competition is als en slechts als er geen $(q_1, q_2) \in S$ en a, i en j zijn met $i \neq j$, $\delta(q_1, a^i) \cap R \neq \emptyset$ en $\delta(q_2, a^j) \cap R \neq \emptyset$.

Merk op dat noch N_r, noch R, noch S op voorhand moeten geconstrueerd worden. Vandaar dat dit algoritme tevens te implementeren is in NLOGSPACE waarbij de koppels uit S niet-deterministisch gegenereerd worden.

4.2 Simplificatie

In deze sectie bespreken we de complexiteit van het simplificatieprobleem. Een willekeurige[2] EDTD kan gesimplificeerd worden indien er een equivalente DTD, single-type EDTD of restrained-competition EDTD bestaat. De volgende stelling geeft aan dat het simplificatieprobleem voor zowel een DTD, single-type EDTD als restrained competition EDTD EXPTIME-compleet is. In het bewijs van deze stelling wordt een algoritme voorgesteld dat een equivalent gesimplificeerd schema berekent indien het bestaat.

Stelling 4.2 *Nagaan of een EDTD een equivalente DTD, single-type EDTD of restrained competition EDTD heeft, is EXPTIME-compleet.*

Dit betekent dat elk algoritme dat dit probleem oplost minstens runt in exponentiële tijd (ondergrens) en dat er ook werkelijk een algoritme bestaat dat het probleem oplost in exponentiële tijd (bovengrens). Bovendien is elk ander probleem uit EXPTIME polynomiaal reduceerbaar tot het simplificatieprobleem. Een bewijs voor stelling 4.2 neemt de rest van deze sectie in beslag.

Bewijs.

Ondergrens.

[2]Hier in de betekenis van een EDTD die niet tegelijk ook een DTD, een single-type EDTD of een restrained competition EDTD is.

We tonen eerst aan dat het simplificatieprobleem EXPTIME-hard is (ondergrens). Dit doen we door een polynomiale reductie te beschrijven van het universaliteitsprobleem met betrekking tot niet-deterministische boomautomaten ($\mathcal{U}$) naar het simplificatieprobleem ($\mathcal{S}$). Omdat $\mathcal{U}$ EXPTIME-compleet is volgt hieruit dat $\mathcal{S}$ EXPTIME-hard is. We noteren NTA(REG) voor de klasse van niet-deterministische boomautomaten waarbij de transitiefunties worden voorgesteld als reguliere expressies. Het universaliteitsprobleem kan als volgt geformuleerd worden: gegeven een NTA(REG) A over alfabet Σ, ga na of $\mathrm{L}(A) = \mathcal{T}_\Sigma$. $\mathcal{U}$ is zelfs EXPTIME-compleet indien de boomautomaten slechts één eindtoestand hebben en indien de wortel van elke aanvaarde boom hetzelfde label heeft, bijvoorbeeld a.
We stellen nu een polynomiale reductie f voor die input van $\mathcal{U}$, namelijk een boomautomaat A uit NTA(REG), omzet in input van $\mathcal{S}$, namelijk een EDTD D. Neem $A = (Q, \Sigma, \delta, F)$ een NTA(REG) over alfabet $\Sigma = \{a, b\}$ met één eindtoestand $F = \{q_F\}$. Zonder verlies van algemeenheid kunnen we veronderstellen dat A bomen aanvaardt van diepte minstens twee. We kunnen in LOGSPACE een equivalente EDTD $D' = (\Sigma, \Delta, d, \mu)$ construeren als volgt[3]: $\Delta = \{b^q \mid b \in \Sigma, q \in Q\}$, $\mu(b^q) = b$ voor elke $b \in \Sigma$, en d bestaat uit de regels $d(b^q) = r_{b,q}$ waarbij $r_{b,q}$ de reguliere expressie is verkregen uit $\delta(q, b)$ door elk voorkomen van toestand p te vervangen door $(a^p + b^p)$. Als startelement voor DTD d nemen we a^{q_F}. A en D' zijn equivalent omdat elke $t \in \mathrm{L}(d)$ een accepterende run van A op $\mu(t)$ voorstelt.
Vervolgens construeren we uit D' de EDTD's D_1 en D_2 over het alfabet $\Gamma = \{a, b, \alpha, \beta, root\}$. D_1 aanvaardt alle bomen van de vorm $root(\sigma(t_1 \dots t_n))$ waarbij σ gelijk is aan α of β, $t_1, \dots, t_n \in \mathcal{T}_\Sigma$, en de boom die uit t_n verkregen wordt door zijn meest rechtse blad te verwijderen wordt aanvaardt door A. D_2 aanvaardt alle bomen die dezelfde vorm hebben als die voor D_1, maar nu moet het meest rechtse blad van t_n gelijk zijn aan a (respectievelijk b) als σ gelijk is aan α (respectievelijk β). D_1 kan uit D' geconstrueerd worden door D' te simuleren op de subtree met als wortel het meest rechtse kind van σ. D_2 moet enkel het symbool σ vergelijken met het meest rechtse blad.
Ten slotte definiëren we D als de EDTD die $\mathrm{L}(D_1) \cup \mathrm{L}(D_1)$ aanvaardt. Uit de beschrijving hierboven volgt dat D in polynomiale tijd uit A kan geconstrueerd worden.

We moeten nu enkel nog aantonen dat voor alle x uit NTA(REG) geldt: $x \in \mathcal{U} \Leftrightarrow f(x) \in \mathcal{S}$. Dit doen we door de volgende twee implicaties aan te tonen (die tesamen een alternatieve formulering van de stelling uitdrukken):

(i) Als $\mathrm{L}(A) = \mathcal{T}_\Sigma$, dan is $\mathrm{L}(D)$ definieerbaar door een DTD.

[3]We hebben enkel een aantal tellers nodig die overeenkomen met posities op de inputtape van een Turing machine. Een getal n heeft $^2log(n)$ ruimte nodig als binaire voorstelling.

(ii) Als $\mathrm{L}(A) \neq \mathcal{T}_\Sigma$, dan is $\mathrm{L}(D)$ niet definieerbaar door een restrained competition EDTD.

Noem $T = \mathrm{L}(D)$.

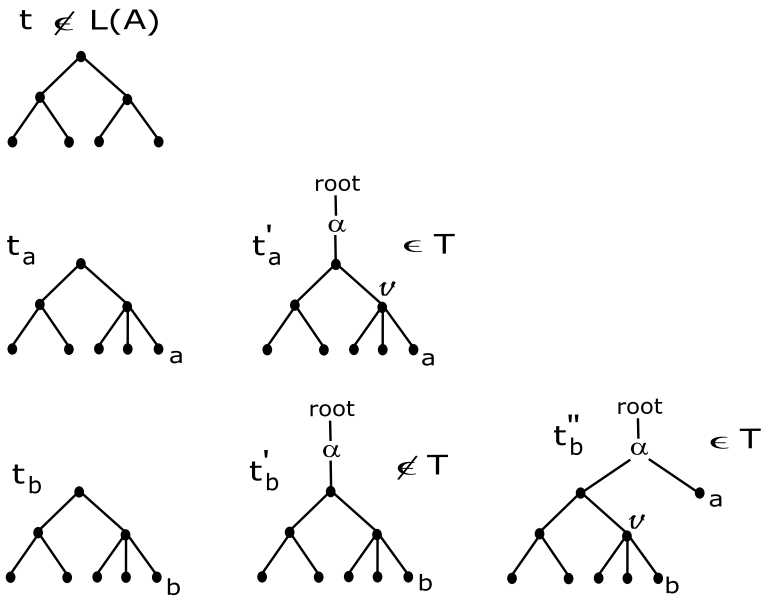
(i) Als $\mathrm{L}(A) = \mathcal{T}_\Sigma$, dan $\mathrm{L}(D_2) \subseteq \mathrm{L}(D_1)$ en $T = \{root(\sigma(t_1 \dots t_2)) \mid \sigma \in \{\alpha, \beta\}, t_1, \dots, t_n \in \mathcal{T}_\Sigma\}$. Het is duidelijk dat T kan gedefinieerd worden door een DTD.

(ii) Als $\mathrm{L}(A) \neq \mathcal{T}_\Sigma$, dan bestaat er een boom $t \notin \mathrm{L}(A)$. We proberen te komen tot een contradictie door te veronderstellen dat T wel definieerbaar is door een restrained competition EDTD. Neem t_a en t_b de bomen die verkregen worden uit t door een a- respectievelijk b-knoop toe te voegen als rechterbroer van het meest rechtse blad. Dan geldt $t'_a := root(\alpha(t_a)) \in T$ en $t'_b := root(\alpha(t_b)) \notin T$. Neem t''_b de boom die uit t'_b kan verkregen worden door een a-blad als meest rechtse kind van α toe te voegen. Uit de definitie van D volgt dat $t''_b \in T$. Neem u het meest rechtse blad van t'_a en v haar ouder. Dan merken we op dat $\text{anc-sib-str}^{t'_a}(v) = \text{anc-sib-str}^{t''_b}(v)$. Omdat T definieerbaar is door een restrained competition EDTD geldt $t'_a[v \leftarrow \text{subtree}^{t''_b}(v)] \in T$. Omdat het meest rechtse blad van $\text{subtree}^{t''_b}(v)$ een b is, moet gelden dat $t \in \mathrm{L}(A)$, wat een contradictie is. De enige veronderstelling die we gemaakt hebben, namelijk dat T definieerbaar is door een restrained competition EDTD, is dus niet waar. We besluiten dat T niet definieerbaar is door een restrained competition EDTD. De afbeeldingen in figuur 4.1 illustreren deze redenering. [4]

Bovengrens.
De exponentiële bovengrens voor single-type en restrained competition EDTD's volgt uit de in sectie 3.4.4 respectievelijk sectie 3.5.3 voorgestelde constructies. Zowel de constructie van de EDTD als het nagaan of deze equivalent is met de originele EDTD kunnen in beide gevallen in exponentiële tijd. Voor DTD's bestaat er een gelijkaardige constructie in polynomiale tijd, maar de equivalentietest neemt ook hier exponentiële tijd in beslag. We gaan hier dieper in op het geval van single-type EDTD's.

Gegeven een EDTD D construeren we de single-type EDTD D_1 zoals beschreven in sectie 3.4.4. We weten hieruit ook dat D een equivalente single-type EDTD heeft als en slechts als D equivalent is aan D_1. De constructie van D_1 gebeurt in exponentiële tijd en D_1 is exponentieel groter dan D. Vervolgens moet nagegaan worden of D en D_1 equivalent zijn. Uit de

[4] Hier volgt extra uitleg over het totstandkomen van de contradictie. $t'_a[v \leftarrow \text{subtree}^{t''_b}(v)]$ kan alleen maar in T zitten als ze aanvaard wordt door D_1 (maar dit is niet het geval want $\alpha \leftrightarrow b$) of als de deelboom met als wortel het enige kind van α aanvaard wordt door D_2. Deze laatste deelboom wordt enkel aanvaard door D_2 als de boom hieruit verkregen door het meest rechtse blad te verwijderen behoort tot $\mathrm{L}(A)$.



Figuur 4.1: Enkele bomen ter illustratie van de contradictie

constructie volgt dat $\mathrm{L}(D) \subseteq \mathrm{L}(D_1)$. We moeten dus enkel controleren of $\mathrm{L}(D_1) \subseteq \mathrm{L}(D)$. We construeren de niet-deterministische boomautomaat A die equivalent is met D en A_1 die equivalent is met D_1 (LOGSPACE) en testen of $\mathrm{L}(A_1) - \mathrm{L}(A)$ leeg is. Hiertoe construeren we boomautomaat A' uit A die het complement van A uitdrukt. Dit neemt exponentiële tijd in beslag omdat A hiervoor eerst gedeterminiseerd moet worden (aan de hand van de welbekende subset constructie). We moeten nu enkel nog nagaan of $\mathrm{L}(A_1) \cap \mathrm{L}(A')$ leeg is. Voor deze emptiness test is een polynomiaal algoritme bekend. Het totale algoritme, bestaande uit de constructie van een single-type EDTD en de equivalentietest, is dus in EXPTIME.

Samengevat.
We hebben aangetoond dat het simplificatieprobleem EXPTIME-hard is (ondergrens) en dat er daadwerkelijk een algoritme bestaat dat het simplificatieprobleem in EXPTIME oplost (bovengrens). Hieruit kunnen we besluiten dat het simplificatieprobleem EXPTIME-compleet is.

Hoofdstuk 5

Het verband tussen de abstracties en bestaande schematalen

In dit hoofdstuk wordt de expressieve kracht van enkele in de praktijk gebruikte schematalen in verband gebracht met de abstracties uit het vorige hoofdstuk. Op deze manier proberen we te achterhalen welke van deze talen one-pass preorder typing toelaten en wat de precieze impact is van de restricties die in sommige talen gedefinieerd zijn.

5.1 Document Type Definition

De Document Type Definition (DTD) zoals voorgesteld door [BPSM$^+$04] komt qua structurele expressiviteit overeen met de abstracte DTD uit sectie 3.1. Dit betekent dat aan de hand van een Document Type Definition locale boomtalen gedefinieerd kunnen worden.

De abstracte DTD met als regels

$$\begin{array}{lcl} garage & \rightarrow & auto\ auto^* \\ auto & \rightarrow & merk\ prijs\ (bouwjaar + \epsilon) \end{array}$$

kan in de praktijk geschreven worden als

$$\begin{array}{lll} \text{<!ELEMENT} & garage & (auto+)\text{>} \\ \text{<!ELEMENT} & auto & (merk\ prijs\ bouwjaar?)\text{>}. \end{array}$$

De XML-specificatie legt echter een bijkomende beperking op aan de reguliere expressies die in DTD's kunnen voorkomen (cf. [BPSM$^+$04] sectie 3.2.1 en appendix E):

> [. . .], it is required that content models in element type declarations be deterministic.

Dergelijke reguliere expressies worden ook one-unambiguous genoemd [BKW98]. Een reguliere expressie is one-unambiguous als men bij het doorlopen van een sequentie van links naar rechts elk symbool van de sequentie onmiddellijk kan mappen op het overeenkomstige symbool uit de reguliere expressie, zonder vooruit te kijken naar de volgende symbolen van de sequentie (zie sectie 2.1.3 voor een formele definitie). Deze beperking is ingevoerd om compabiliteit met bestaande SGML[1]-parsers te behouden. Nochtans is deze beperking niet noodzakelijk voor efficiënte validatie-algoritmes (cf. sectie 3.1.2), waardoor ze aan felle kritiek onderhevig is.[2] Immers, niet alle reguliere expressies zijn te herschrijven als one-unambiguous reguliere expressies.

Typing van XML-documenten is niet echt van belang, omdat elk element slechts met één content model kan geassocieerd worden (dus one-pass preorder typing is mogelijk). Naast het beschreven bottom-up validatie-algoritme kan dit ook top-down gebeuren. Van zodra men een element tegenkomt weet men immers aan welke regel de child-string van dit element moeten voldoen.

De beperkte expressieve kracht van locale boomgrammatica's kan ongunstig zijn bij het ontwerpen van een XML schema aan de hand van een DTD. Neem bijvoorbeeld elementen die een personenauto voorstellen en elementen die een vrachtauto voorstellen. Ontwerpers van DTD's zouden graag een verschillende content definiëren voor eenzelfde auto element. Als men echter voor beide soorten auto's het auto element wil gebruiken, kan dit enkel met eenzelfde content. Als men toch een onderscheid wil maken is men verplicht verschillende auto elementen te introduceren zoals bijvoorbeeld personenAuto, vrachtAuto, 4×4Auto, enz.

5.2 XML Schema

XML Schema [TBMM04] maakt het mogelijk met eenzelfde element verschillende content models te associëren. Ook al wordt deze schemataal in de praktijk vaak gebruikt, toch is de precieze expressieve kracht niet onmiddellijk duidelijk. Hierop wordt in deze sectie dieper ingegaan.

Omdat in het kader van deze thesis vooral de structurele kracht van schematalen van belang is, beperken we de bespreking van XML Schema tot het definiëren van complex content. Complex content wordt gecreëerd door een lijst van kind elementen en eventueel attributen te definiëren. Men gebruikt hiervoor de tag <xs:complexType>. Er wordt een onderscheid gemaakt

[1] SGML is eveneens een markup language en een voorloper van de huidige XML.
[2] Zie bijvoorbeeld [Man01] en p. 98 uit [vdV02].

Compositors	**Particles**
sequence	element
choice	sequence
all	choice
	any
	group

Tabel 5.1: XML Schema - Compositors en particles

tussen *compositors* en *particles* (zie tabel 5.1). Een compositor kan men beschouwen als een soort container waarin particles ondergebracht zijn. XML Schema definieert drie verschillende compositors: *sequence*, om een geordende lijst van particles te definiëren, *choice*, om de keuze van een particle uit een verzameling particles toe te laten, en *all*, om een ongeordende lijst van particles voor te stellen. De sequence en choice compositors kunnen eveneens voorkomen als particles. Alle compositors en particles kunnen de attributen *minOccurs* en *maxOccurs* bevatten, waarmee men kan aangeven hoe vaak ze mogen voorkomen. Terwijl aan een element steeds een naam moet verbonden worden, treedt any op als wildcard. Met group kan men een opeenvolgend stuk schemacode afzonderen en er met een referentie naar verwijzen. Dit alles laat toe complexe structuren te definiëren.

Omdat XML Schema toelaat voor een bepaald element verschillende content models te definiëren, lijkt haar structurele kracht op het eerste gezicht overeen te komen met die van EDTD's. De specificatie legt echter een aantal beperkingen op die het op willekeurige wijze combineren van compositors en particles aan banden legt. We lichten hier twee specifieke constraints toe die het aantal talen dat gedefinieerd kan worden door XML Schema beperken.

5.2.1 Element Declarations Consistent

De Element Declarations Consistent (EDC) constraint wordt in de specificatie als volgt gedefinieerd (cf. [TBMM04] sectie 3.8.6).

> If the particles contains [...] two or more element declaration particles with the same name and target namespace, then all their type definitions must be the same top-level definition, that is, all of the following must be true:
>
> - all their type definitions must have a non-absent name
> - all their type definitions must have the same name
> - all their type definitions must have the same target namespace

Deze constraint drukt uit dat een element geen twee kind-elementen met dezelfde naam mag hebben waarvan de types ofwel geen naam hebben, ofwel verschillende namen hebben, ofwel verschillende namespaces hebben. Om het anders te formuleren: een element mag verschillende kind-elementen hebben met dezelfde naam zolang ze maar exact hetzelfde type hebben. Merk op dat als twee kind-elementen met dezelfde naam een locale type-definitie hebben (ook al betekenen deze definities exact hetzelfde), dat dit een schending is van de eis dat er een niet-afwezige typenaam moet zijn. De desbetreffende types moeten dus globaal gedefinieerd zijn.

Er volgen twee voorbeelden waarin de EDC-constraint geschonden wordt.

```
<xs:complexType name="wrongType">
  <xs:sequence>
    <xs:element name="a" type="xs:int"/>
    <xs:element name="b" type="xs:date"/>
    <xs:element name="a" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

In dit voorbeeld zijn in eenzelfde content model twee elementen gedeclareerd die dezelfde naam hebben, namelijk a, maar elk van een verschillend type, namelijk $xs{:}int$ en $xs{:}string$. Merk op dat de types van deze twee elementen wel een *non-absent name* en dezelfde namespace hebben. Dus als we dit voorbeeld wijzigen en ervoor zorgen dat beide elementen van het type $xs{:}int$ of $xs{:}string$ zijn, voldoet dit content model wel aan de EDC-constraint. Een tweede voorbeeld:

```
<xs:element name="garage">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="auto" type="auto1"/>
        <xs:element name="auto" type="auto2"/>
      </xs:choice>
      <element name="auto" type="auto2"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="auto" type="auto1"/>
        <xs:element name="auto" type="auto2"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Deze definitie komt overeen met de eerste regel van de EDTD uit voorbeeld 3.4. Ook hier wordt de EDC-constraint geschonden omdat als kinderen

van het *garage*-element *auto*-elementen in twee verschillende types kunnen voorkomen, namelijk als *auto1* en *auto2*.

De EDC-constraint komt overeen met het formalisme van single-type EDTD's. Het is eenvoudig in te zien dat de EDC-constraint oplegt dat een element geen kind-elementen mogen voorkomen die dezelfde naam hebben maar een verschillend type. De definitie van single-type EDTD's komt op hetzelfde neer: aan de rechterkant van een regel kan van één element slechts één type voorkomen. De overeenkomst met single-type EDTD's impliceert dat het type van een element enkel afhangt van zijn voorouders en dat elk XML-document dat beschreven is aan de hand van een XML Schema bijgevolg one-pass preorder getyped kan worden.

5.2.2 Unique Particle Attribution

Een andere belangrijke constraint die door de specificatie wordt opgelegd is de Unique Particle Attribution (UPA) constraint. Deze wordt als volgt gedefinieerd (cf. [TBMM04] sectie 3.8.6):

> A content model must be formed such that during validation of an element information item sequence, the particle component contained [...] therein [...] can be uniquely determined without examining the content or attributes of that item, and without any information about the items in the remainder of the sequence.

De UPA-constraint eist dat als men een XML-document in document orde valideert ten opzichte van een XML Schema het element altijd uniek gedetermineerd moet kunnen worden zonder naar zijn attributen, kinderen of de volgende elementen te kijken. We geven een voorbeeld van een XML Schema fragment dat niet voldoet aan de UPA-constraint en leggen uit waaruit we dit kunnen opmaken.

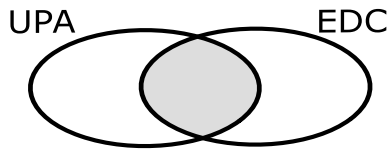
```
<xs:group name="pages">
    <xs:sequence>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="odd-page"/>
            <xs:element ref="even-page"/>
        </xs:sequence>
        <xs:element ref="odd-page" minOccurs="0"/>
    </xs:sequence>
</xs:group>
```

Het probleem hier is dat men, zonder eerst naar het volgende element te kijken, niet kan zeggen of een tag met de naam odd-page nu overeenkomt

met de eerste of tweede declaratie van het odd-page element. Wanneer men een odd-page element tegenkomt kan men dit dus niet direct uniek associëren met een element declaratie uit het schema.

Formeel kunnen we stellen dat een EDTD voldoet aan de UPA-constraint als voor elke reguliere expressie r over het typealfabet Δ geldt dat $\mu(r)$ one-unambiguous is. Met $\mu(r)$ wordt de reguliere expressie genoteerd die we uit r verkrijgen als we elk type τ vervangen door zijn bijhorende elementnaam $\mu(\tau)$.

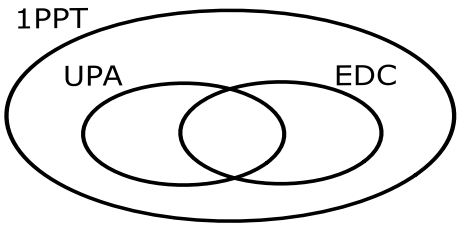
Wat is nu de onderlinge relatie is tussen de UPA- en EDC-constraint? Het antwoord hierop luidt als volgt: een reguliere expressie die voldoet aan de EDC-constraint voldoet niet noodzakelijk aan de UPA-constraint, en omgekeerd, een reguliere expressie die voldoet aan de UPA-constraint voldoet niet noodzakelijk aan de EDC-constraint. We tonen dit aan met een voorbeeld. De expressie $a^1\ a^{2*}\ b^1$ voldoet niet aan de EDC-constraint (het element a komt onder twee verschillende types voor), maar wel aan de UPA-constraint omdat de hieruit afgeleide expressie over elementnamen $a\ a^*\ b$ one-unambiguous is. Voor een tegenvoorbeeld in de andere richting nemen we $r = (a^1 + b^1)^*\ a^1\ (a^1 + b^1)$ die duidelijk voldoet aan de EDC-constraint. De reguliere expressie $\mu(r) = (a + b)^*\ a\ (a + b)$ is echter niet one-unambiguous, zowel $a_1a_2a_3$ als a_2a_3 behoren tot $\mathrm{L}((a_1 + b_1)^*\ a_2\ (a_3 + b_2))$, en ze kan zelfs niet omgevormd worden tot een equivalente one-unambiguous expressie. De verhouding tussen beide constraints in weergegeven in figuur 5.1. Enkel de schema's in de doorsnede kunnen voorkomen als een correct XML Schema.



Figuur 5.1: Relatie tussen de UPA- en EDC-constraint

We gaan nu na op welke manier de UPA-constraint zich verhoudt tot 1PPT. Wanneer we een sequentie matchen ten opzichte van een restrained competition reguliere expressie, hangt het type van een element uit de sequentie enkel af van het deel van de sequentie dat reeds bekeken is. Bij one-unambiguous reguliere expressies over het typealfabet kan men uniek bepalen met welk element uit de reguliere expressie een element uit een sequentie matcht, zonder te kijken naar het vervolg van de sequentie. Bij het matching element uit de reguliere expressie hoort precies één type, waardoor het type van het element uit de sequentie onmiddellijk eenduidig bekend is. We kunnen hieruit besluiten dat de UPA-constraint 1PPT impliceert.

De omgekeerde implicatie geldt echter niet. Er bestaan namelijk reguliere expressies die 1PPT toelaten maar die niet voldoen aan de UPA-constraint, bijvoorbeeld $a^1\ (a^2 + b)^*\ b$. Deze relatie wordt weergegeven in figuur 5.2.



Figuur 5.2: 1PPT en UPA/EDC

5.2.3 Expressiviteit van XML Schema

Uit al het voorgaande kunnen we afleiden dat de expressiviteit van XML Schema overeenkomt met die van single-type EDTD's (cf. EDC) waaraan een extra beperking is opgelegd, namelijk dat de reguliere expressies one-unambiguous moeten zijn (cf. UPA). Het type van een element hangt dus enkel af van zijn voorouders. 1PPT is steeds mogelijk.

1PPT is steeds mogelijk, zelfs al zou er slechts aan één van beide constraints voldaan zijn. Dit roept vragen op over de relevantie van beide constraints. Men zou beide constraints kunnen laten vallen en vervangen door de notie van restrained competition EDTD's. Op die manier zou de volledige klasse van talen waarbij 1PPT mogelijk is uitgedrukt kunnen worden in XML Schema, wat de expressiviteit zou verhogen zonder echter een essentieel verlies aan efficiëntie te veroorzaken. Validatie en typing blijft mogelijk in lineaire tijd (en top-down), toegelaten schema's kunnen herkend worden in kwadratische tijd, en toegelaten schema's kunnen geconstrueerd worden in exponentiële tijd als het tenminste bestaat. De UPA-constraint zorgt echter voor compabiliteit met bestaande SGML-parsers.

5.2.4 Alternatieve syntax

Omdat het type van een element enkel afhangt van zijn voorouders kan het interessant zijn om deze relatie expliciet te maken aan de hand van een pattern-based schemataal (cf. sectie 3.4.3). We stellen hier niet voor om vanaf nul een nieuwe schemataal te ontwikkelen, maar gaan na op welke manier we pattern-based schema's kunnen integreren in een bestaande schemataal. De auteurs van [MNSB05] hebben twee alternatieven ontwikkeld: het eerste is een uitbreiding van de DTD-specificatie, het tweede is een uitbreiding van de XML Schema specificatie. Deze twee alternatieven zijn qua expressiviteit even krachtig als het huidige XML Schema. We illustreren

beide alternatieven aan de hand van een single-type EDTD die een abstractie is van real-life XML schema.[3] Merk op dat het type van j hier afhangt van zijn overgrootouder.

$$\begin{array}{lcl} a & \rightarrow & b + c \\ b & \rightarrow & e\ d^1\ f \\ c & \rightarrow & e\ d^2\ f \\ d^1 & \rightarrow & g\ h^1\ i \\ d^2 & \rightarrow & g\ h^2\ i \end{array} \qquad \begin{array}{lcl} h^1 & \rightarrow & j^1 \\ h^2 & \rightarrow & j^2 \\ j^1 & \rightarrow & k\ l \\ j^2 & \rightarrow & m\ n \end{array}$$

De DTD-specificatie wordt uitgebreid met type declaraties van de vorm <!TYPE typenaam patternExpressie (reguliereExpressie)>.
Als pattern expressie kunnen bijvoorbeeld lineaire XPath expressies gebruikt worden, waarbij alleen de kind- en nakomeling-assen gebruikt worden. Op die manier hangt een type enkel af van zijn voorouders. Elementen die onder verschillende types kunnen voorkomen maar dezelfde naam hebben, worden met een dergelijke TYPE-regel gedefinieerd. Voor de rest wordt de gebruikelijke DTD syntax gebruikt. Dit geeft het volgend resultaat:

<!ELEMENT a $(b + c)$>
<!ELEMENT b $(e\ d\ f)$>
<!ELEMENT c $(e\ d\ f)$>
<!ELEMENT d $(g\ h\ i)$>
<!ELEMENT h (j) >
<!TYPE $j1$ "//b//j" $(k\ l)$>
<!TYPE $j2$ "//c//j" $(m\ n)$>

Merk op dat deze notatie compacter is dan de corresponderende EDTD omdat d en h niet expliciet elk met verschillende types moeten gedefinieerd worden.

Het tweede alternatief bestaat erin de XML Schema specificatie zo uit te breiden dat het onmiddellijk duidelijk is dat het type van een element afhangt van zijn context. Ook hier wordt deze context uitgedrukt als lineaire XPath expressie, waardoor het type van een element afhangt van zijn voorouders. Element j uit het voorbeeld kan als volgt gedefinieerd worden:

```
<xs:element name="j">
  <xs:alt cond="//b//j" type="j1"/>
  <xs:alt cond="//c//j" type="j2"/>
</xs:element>
```

Elke typedefinitie voor element j wordt weergegeven in een alt-tag. Als aan de conditie van het cond-attribuut voldaan is, krijgt het element j het overeenkomstige type uit het type-attribuut.

[3] Dit voorbeeld is overgenomen uit [MNSB05].

Terwijl de uitbreiding van de DTD-specificatie ervoor zorgt dat we krachtigere DTD's bekomen, zorgt de uitbreiding van de XML Schema specificatie niet voor een uitbreiding van expressiviteit. Beide alternatieven komen qua expressiviteit overeen met single-type EDTD's en zijn dus equivalent met het huidige XML Schema.[4]

5.3 RELAX NG

De expressieve kracht van RELAX NG [CM01a] komt precies overeen met die van EDTD's. Met eenzelfde elementnaam kunnen verschillende types geassocieerd worden en hierop worden geen bijkomende beperkingen gedefinieerd. De reguliere expressies mogen zelfs ambigu zijn. We kunnen dus stellen dat aan de hand van een RELAX NG schema de welbekende en robuuste klasse van reguliere boomtalen kan gedefinieerd worden. Een gevolg hiervan is dat 1PPT niet mogelijk is. Aan de hand van niet-deterministische boomautomaten kunnen echter wel efficiënte algoritmes geïmplementeerd worden (cf. sectie 3.2.2).

Voor een uitgebreidere bespreking van RELAX NG en de manier waarop een schema kan geformaliseerd worden als EDTD verwijzen we naar het volgende hoofdstuk. In deze thesis wordt namelijk RELAX NG als uitgangspunt genomen om een aantal eigenschappen te illustreren aan de hand van een in de praktijk gebruikte schemataal.

5.4 Besluit

Terwijl een DTD enkel de locale boomtalen beschrijft, komt RELAX NG overeen met de volledige klasse van reguliere boomtalen. XML Schema ligt hiertussenin en beschrijft de reguliere boomtalen waarbij het type van een element afhangt van zijn voorouders. Om compabiliteit met bestaande SGML-parsers te behouden, is in de DTD en XML Schema specificatie de bijkomende beperking opgelegd dat hun reguliere expressies one-unambiguous moeten zijn. Deze beperking is echter niet nodig om one-pass preorder typing mogelijk te maken.

[4]Om een schemataal te construeren die equivalent is aan restrained-competition EDTD's, kan een alternatieve pattern taal gebruikt worden (cf. sectie 3.5.2).

Hoofdstuk 6

RELAX NG

RELAX NG (REgular LAnguage for XML, New Generation) is een XML-schemataal, ontstaan uit RELAX[1] en TREX[2]. Aanvankelijk werd ze gestandardiseerd door OASIS (Organization for the Advancement of Structured Information Standards), en nadien is RELAX NG als onderdeel van het DSDL-framework[3] opgenomen in een ISO/IEC-standaard (International Organization for Standardization/International Electrotechnical Committee).

RELAX NG is een vrij eenvoudige schemataal met als belangrijkste taak het valideren van de stuctuur van XML-documenten. Voor het beschrijven van de inhoud van een XML-document zijn er slechts twee built-in datatypes voozien, namelijk string en token. Daarnaast kunnen er ook externe datatype bibliotheken ingeplugd worden zodat datatypes van W3C XML Schema en DTD gebruikt kunnen worden.

6.1 Full syntax

Een RELAX NG schema wordt meestal beschreven als een pattern opgebouwd uit subpatterns en een correct schema voldoet aan onderstaande grammatica [CM01a].

```
pattern ::= <element name="QName"> pattern+ </element>
          | <element> nameClass pattern+ </element>
          | <attribute name="QName"> [pattern] </attribute>
          | <attribute> nameClass [pattern] </attribute>
          | <group> pattern+ </group>
```

[1] RELAX staat voor REgular LAnguage description for XML. Zie `http://www.xml.gr.jp/relax/` voor meer informatie.

[2] TREX staat voor Tree Regular Expressions for XML. Zie `http://www.thaiopensource.com/trex/` voor meer informatie.

[3] DSDL is de afkorting van Document Schema Definition Languages. Zie `http://dsdl.org/` voor meer informatie.

```
          | <interleave> pattern+ </interleave>
          | <choice> pattern+ </choice>
          | <optional> pattern+ </optional>
          | <zeroOrMore> pattern+ </zeroOrMore>
          | <oneOrMore> pattern+ </oneOrMore>
          | <list> pattern+ </list>
          | <mixed> pattern+ </mixed>
          | <ref name="NCName"/>
          | <parentRef name="NCName"/>
          | <empty/>
          | <text/>
          | <value [type="NCName"]> string </value>
          | <data type="NCName"> param* [exceptPattern]
            </data>
          | <notAllowed/>
          | <externalRef href="anyURI"/>
          | <grammar> grammarContent* </grammar>
param ::= <param name="NCName"> string </param>
exceptPattern ::= <except> pattern+ </except>
grammarContent ::= start
                 | define
                 | <div> grammarContent* </div>
                 | <include href="anyURI"> includeContent*
                   </include>
includeContent ::= start
                 | define
                 | <div> includeContent* </div>
start  ::= <start [combine="method"]> pattern </start>
define ::= <define name="NCName" [combine="method"]> pattern+
           </define>
method ::= choice
         | interleave
nameClass ::= <name> QName </name>
            | <anyName> [exceptNameClass] </anyName>
            | <nsName> [exceptNameClass] </nsName>
            | <choice> nameClass+ </choice>
exceptNameClass ::= <except> nameClass+ </except>
```

Deze grammatica wordt de *full syntax* genoemd, daarnaast bestaat er ook een *simple syntax* (zie sectie 6.2). In deze grammatica staat *QName* voor qualified name, waarbij er hoogstens één : voorkomt om een optionele prefix te scheiden van de locale naam. In een *NCName* kan geen : voorkomen. *AnyUri* verwijst naar een geldige uri referentie en *string* staat voor een willekeurige string.

Naast de expliciet getoonde attributen kan elk element ook de attributen *ns* en *datatypeLibrary* bevatten. Elke element kan bovendien foreign attributes hebben. Een foreign attribute is een attribuut waarvan de naam een namespace-uri heeft die verschilt van de lege string en de RELAX NG namespace-uri. Elk element behalve *value*, *param* en *name* kunnen ook foreign elements als kinderen hebben. Een foreign element is een element waarvan de naam een namspace-uri heeft die verschilt van de RELAX NG namespace-uri (de lege string is dus wel toegelaten!).

Naast de XML-syntax voor RELAX NG schema's bestaat er ook een equivalente compacte syntax.

6.2 Simplificatie

Om het verwerken van RELAX NG schema's te vergemakkelijken, stelt de specificatie [CM01a] een simplificatie-algoritme voor. Door de toepassing van dit algoritme wordt een willekeurig RELAX NG schema uitgezuiverd en in een soort normaalvorm gebracht. Deze normaalvorm, ook simple syntax genoemd, wordt verkregen door een serie transformatieregels in een welbepaalde vaste volgorde uit te voeren. Een bepaalde transformatieregel wordt toegepast op alle elementen in het schema vooraleer een volgende transformatieregel wordt uitgevoerd. De transformatieregels kunnen ook bijkomende beperkingen definiëren waaraan een correct RELAX NG schema moet voldoen. Hieronder worden een aantal transformatieregels in het kort beschreven; meer informatie is te vinden in de RELAX NG specificatie.

- Verwijder foreign elements en foreign attributes.
- Normalisatie van whitespaces.
- Voeg het *datatypeLibrary*-attribuut toe aan elk *data*- en *value*-element dat er geen heeft. De waarde ervan wordt overgenomen van de dichtstbijzijnde voorouder of is de lege string. Bij elk ander element wordt het *datatypeLibrary*-attribuut verwijderd.
- Vervang elk *externalRef*-element door een nieuw element dat geconstrueerd is aan de hand van de waarde van het *href*-attribuut. Het nieuwe element moet voldoen aan de syntax van *pattern* uit de grammatica hierboven en alle transformatieregels die totnogtoe zijn uitgevoerd worden hierop recursief toegepast.
- Voor elk *include*-element wordt, aan de hand van de waarde van zijn *href*-attribuut, een nieuw *grammar*-element geconstrueerd. Als het *include*-element een *start*-component heeft, moet het *grammar*-element ook een *start*-component hebben en wordt deze uit het *grammar*-element verwijderd. Als het *include*-element een *define*-component

heeft, moet het *grammar*-element ook een *define*-component hebben met dezelfde naam en wordt deze uit het *grammar*-element verwijderd. Vervolgens wordt het *include*-element getransformeerd in een *div*-element: alle attributen van het *include*-element behalve *href* worden attributen van het *div*-element, het nieuwe *grammar*-element en de kinderen van het *include*-element worden de kinderen van het *div*-element, het *grammar*-element wordt hernoemd tot *div*. Alle *externalRef*-elementen worden recursief getransformeerd.

- Vervang het *name*-attribuut van *element* en *attribute* door een *name*-kindelement.
- Voeg het *ns*-attribuut toe aan elk *name*-, *nsName*- en *value*-element dat er geen heeft. De waarde ervan wordt overgenomen van de dichtstbijzijnde voorouder of is de lege string. Bij elk ander element wordt het *ns*-attribuut verwijderd.
- Als een *name*-element een prefix bevat, wordt deze verwijderd en komt er een *ns*-attribuut in de plaats.
- Vervang elke *div*-element door zijn kinderen.
- Transformeer elk *define*-, *oneOrMore*-, *zeroOrMore*-, *optional*-, *list*-, *mixed*- en *except*-element zodat het slechts één kindelement heeft: de kinderen van *except* worden in een *choice*-element ondergebracht, de kinderen van de andere elementen worden in een *group*-element ondergebracht. Transformeer elk *choice*-, *group*- en *interleave*-element zodat het precies twee kinderen heeft: als het slechts één kind heeft, wordt het vervangen door dit kind; als er meer kinderen zijn wordt herhaaldelijk een nesting-algoritme toegepast. Transformeer elk *element*-element zodat het exact twee kinderen heeft, waarvan het eerste een nameclass is en het tweede een pattern (indien er meerdere kinderen zijn worden deze gecombineerd in een *group*-element). Als een *attribute*-element maar één kind heeft, namelijk een nameclass, wordt een *text*-element toegevoegd.
- <*mixed*> *p* </*mixed*> wordt <*interleave*> *p* <*text*/> </*interleave*>, <*optional*> *p* </*optional*> wordt <*choice*> *p* <*empty*/> </*choice*>, <*zeroOrMore*> *p* </*zeroOrMore*> wordt <*choice*> <*oneOreMore*> *p* </*oneOreMore*> <*empty*/> </*choice*>.
- Alle *start*-elementen die bij een bepaald *grammar*-element behoren, worden gecombineerd tot één *start*-element. Voor elk *grammar*-element worden alle *define*-elementen met dezelfde naam gecombineerd tot één *define*-element met die naam. Het manier waarop de verschillende elementen worden gecombineerd tot één enkel element hangt af van de waarde van het *combine*-attribuut (*choice* of *interleave*).

- Het RELAX NG schema wordt zo getransformeerd dat het top-level element *grammar* is en dat er voor de rest geen andere *grammar*-elementen meer voorkomen.
- Transformeer het RELAX NG schema zodat elk *element*-element als kind van een *define*-element voorkomt en het kind van elke *define*-element een *element*-element is.
- Zorg ervoor dat een *notAllowed*-element enkel voorkomt als kind van een *start*- of *element*-element.
- Transformeer het schema zodat een *empty*-element nooit voorkomt als kind van een *group*-, *interleave*- of *oneOrMore*-element of als tweede kind van een *choice*-element.

Het resultaat van al deze transformatieregels is een vlak schema met *grammar* als top-level element. Het *grammar*-element heeft als kinderen een *start*-element gevolgd door nul of meer *define*-elementen. Er zijn geen verwijzingen meer naar externe RELAX NG schema's, alle informatie zit bevat in dit ene RELAX NG document. Een RELAX NG schema waarop het simplificatie-algoritme is toegepast, voldoet aan de volgende grammatica:

```
grammar ::= <grammar> <start> top </start> define* </grammar>
define  ::= <define name="NCName"> <element> nameClass top
            </element> </define>
top     ::= <notAllowed/>
          | pattern
pattern ::= <empty/>
          | nonEmptyPattern
nonEmptyPattern ::= <text/>
          | <data type="NCName" datatypeLibrary="anyURI">
            param* [exceptPattern] </data>
          | <value datatypeLibrary="anyURI" type="NCName"
            ns="string"> string </value>
          | <list> pattern </list>
          | <attribute> nameClass pattern </attribute>
          | <ref name="NCName"/>
          | <oneOrMore> nonEmptyPattern </oneOrMore>
          | <choice> pattern nonEmptyPattern </choice>
          | <group> nonEmptyPattern nonEmptyPattern </group>
          | <interleave> nonEmptyPattern nonEmptyPattern
            </interleave>
param ::= <param name="NCName"> string </param>
exceptPattern ::= <except> pattern </except>
nameClass ::= <anyName> [exceptNameClass] </anyName>
```

```
            | <nsName ns="string"> [exceptNameClass] </nsName>
            | <name ns="string"> NCName </name>
            | <choice> nameClass nameClass </choice>
exceptNameClass ::= <except> nameClass </except>
```

Merk op dat deze grammatica heel wat beknopter is dan de grammatica van de full syntax. Naast de aangegeven elementen en attributen kunnen er geen andere meer voorkomen. We kunnen de verschillende patterns ook een naam geven. De grammar-tag en zijn kinderen vormen een grammarpattern, de element-tag en zijn kinderen vormen een elementpattern, enz. Een nameclasspattern, of afgekort nameclass, kan één van de vier volgende vormen aannemen: anyname-nameclass, nsname-nameclass, name-nameclass en choice-nameclass. De choice en except-tag kunnen elk op twee verschillende manieren voorkomen, als pattern en als nameclass. De except-nameclass is geen nameclass als de vier andere, omdat ze niet zelfstandig maar enkel als kind van een andere nameclass kan voorkomen.

Opmerking 6.1 *Let op het verschil tussen simplificatie van een RELAX NG schema en simplificatie van een EDTD.*

6.3 Restricties

Wanneer een RELAX NG schema aan de full syntax voldoet, betekent dit niet noodzakelijk dat het om een correct RELAX NG schema gaat. De specificatie legt een aantal extra beperkingen op.

Een aantal van deze beperkingen worden gecheckt tijdens het simplificatieproces. Zo mag bijvoorbeeld het recursief verwerken van een *externalRef*- of *include*-element niet resulteren in een loop. Dit is enkel het geval indien de waarde van een *href*-attribuut in een gerefereerd schema niet verwijst naar het schema (of de schema's indien de recursiediepte groter is dan 1) van waaruit de referentie wordt aangeroepen. Het is ook niet mogelijk nameclasses te definiëren die geen enkele naam bevatten, wat bijvoorbeeld het geval is indien *<except> <anyName/> </except>* voorkomt. Datatype libraries moeten correct gebruikt worden. Dit betekent dat bij het *data*- en *value*-element de waarde van het *type*-attribuut een geldig type moet zijn in de library die gedefinieerd is door het *datatypeLibrary*-attribuut.[4]

Andere restricties worden gecheckt indien het simplificatieproces volledig is doorlopen. De restricties zijn dus van toepassing op een schema dat aan de simple syntax voldoet.[5]

[4] Een volledig overzicht van de beperkingen die tijdens het simplificatieproces gecontroleerd worden, is te vinden in hoofdstuk 4 van de RELAX NG specificatie. Paragraaf 4.16 is volledig toegewijd aan het beschrijven van een aantal restricties.

[5] Deze restricties worden besproken in hoofdstuk 7 van de RELAX NG specificatie.

Restricties op lijsten. Omdat een lijst, aangegeven door de list-tag, wordt behandeld als een reeks textnodes is het verboden om elementen of attributen in een lijst te laten voorkomen. Het nesten van lijsten is verwarrend en daarom niet toegelaten. Hetzelfde geldt voor de aanwezigheid van het *text*-element in een lijst. Het *interleave*-element mag niet voorkomen als nakomeling van *list* omdat het de implementatie van een RELAX NG schema processor complexer maakt. Samengevat kan een listpattern dus geen van de volgende elementen als nakomeling hebben: *ref* (na simplificatie verwijst ref immers naar een element), *attribute*, *list*, *text* en *interleave*. Een voorbeeld van een niet toegelaten list pattern is het volgende:

```
<list>
  <interleave>
    <data type="integer"/>
    <data type="token"/>
  </interleave>
</list>
```

Restricties op attributen. Volgens de XML-specificatie kunnen attributen geen andere attributen of elementen bevatten. Hierdoor is het niet toegelaten een *attribute*- of *ref*-element als nakomeling van *attribute* te definiëren. Omdat een attribuut met een bepaalde naam slechts één keer kan voorkomen in een element zijn volgende paden niet toegelaten in een RELAX NG schema: *oneOrMore//group//attribute* en *oneOrMore//interleave//attribute*. Als in een group of interleave pattern meerdere attribuut patterns voorkomen, mogen hun nameclasses niet overlappen. Een attribuut dat gedefinieerd wordt met een oneindig aantal mogelijke namen (anyname of nsname nameclass) moet zich bevinden onder een oneOrMore pattern.

Restricties op het exceptpattern. Een except element met als ouder een data element kan enkele data, value of choice elementen bevatten.

Restricties op het startpattern. Het start pattern geeft aan welke elementen kunnen voorkomen als root van een XML-document. Het is dus niet vreemd dat een start element als nakomelingen enkel ref en choice elementen mag hebben.

Content models. RELAX NG definieert drie verschillende content models voor een element:

- empty, wanneer een element 0 of meer attributen heeft en geen verdere inhoud

- simple, wanneer een element 0 of meer attributen heeft en inhoud beschreven door data, value of list patterns
- complex, in alle andere gevallen

Deze definities zijn identiek aan degene die door XML Schema gebruikt worden, behalve dat het mixed content model niet apart wordt genoemd. Ze wijken evenwel lichtjes af van de terminologie in plain XML. Neem als voorbeeld de expressie *<foo> bar </foo>*. RELAX NG ziet deze content als complex content als ze beschreven is door een text pattern en als simple content als ze beschreven is door een ander pattern. Een element dat enkel een text node bevat is dus niet noodzakelijk simple content. Op deze content models zijn restricties gedefinieerd. Om dit te formaliseren wordt de term *groepeerbaar* ingevoerd: het empty content model is groepeerbaar met elke ander content model en het complex content model is groepeerbaar met een ander complex content model. Enkel wanneer twee patterns een content model hebben dat met elkaar groepeerbaar is, mogen deze patterns gecombineerd worden met een group of interleave element. Een pattern kan enkel oneOrMore als ouder hebben als het content model van dit pattern groepeerbaar is met zichzelf. Een gevolg hiervan is dat simple en complex content models in de definitie van een element enkel kunnen voorkomen als alternatieven voor elkaar (gecombineerd door choice), men kan dus kiezen tussen data- of tekstgeoriënteerde inhoud maar tesamen kunnen ze niet voorkomen. Een voorbeeld van een niet toegelaten element pattern is het volgende:

```
<element name="foo">
  <group>
    <data type="integer"/>
    <text/>
  </group>
</element>
```

Restricties op interleave. Elementen die gecombineerd worden door interleave mogen geen overlappende nameclasses hebben. Bovendien mag slechts één text pattern voorkomen tussen de patterns die door interleave gecombineerd worden. Deze restricties hebben geen invloed op de expressieve kracht van RELAX NG, schema's die deze regels schenden kunnen steeds herschreven worden. Door het herschrijven van een schema kan evenwel het modulaire karakter ervan afnemen.

Het doel van deze restricties is het processen van RELAX NG schema's te vergemakkelijken en ervoor te zorgen dat een correct RELAX NG schema zinvol is. Een schema waarbij een attribuut kan voorkomen in een ander

attribuut is niet zinvol, want in een XML-document kan een attribuut enkel tekst bevatten. De laatste reeks restricties, die op interleave, zijn enkel aanwezig omdat de huidige implementatie van RELAX NG validator Jing erop steunt. Volgens James Clark kunnen ze in toekomstige versies van RELAX NG mogelijk geschrapt worden:

> *Better algorithms may be developed that will allow this restriction to be removed in future versions.*[6]

6.4 Beschrijving van RELAX NG aan de hand van voorbeelden

In deze sectie beschrijven we de mogelijkheden van RELAX NG aan de hand van een voorbeeld. We stellen een RELAX NG schema voor dat een acteur definieert en over drie verschillende bestanden verspreid zit, namelijk *actor.rng*, *common.rng* en *film.rng*.

Alvorens over te gaan tot een bespreking van deze schema's, leggen we eerst het gebruik van het *datatypeLibrary*-attribuut uit. Dit attribuut geeft aan welke datatype library gebruikt wordt en kan in elk element voorkomen, ook al heeft het eigenlijk alleen maar betekenis indien er data wordt gedefinieerd. Vandaar dat het in de simple syntax enkel (verplicht!) voorkomt bij het *data*- en *value*-element; het *text*-element legt geen type op aan data en kan dit attribuut in de simple syntax dus niet bevatten. Als een *data*- of *value*-element geen *datatypeLibrary*-attribuut bevat, wordt het overgeërfd van de dichtsbijzijnde voorouder. Als echter geen enkele voorouder een datatype library definieert, wordt de standaard RELAX NG library gebruikt.

Het bestand common.rng bevat een schema dat niet op zichzelf kan gebruikt worden om een XML-document te valideren. Het gaat immers om een grammar-pattern zonder start-tag, terwijl de start-tag nodig is om het root element van een XML-document te definiëren. Het schema ziet er als volgt uit.

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <define name="name-content">
    <element name="first-name"><text/></element>
    <element name="last-name"><text/></element>
  </define>
  <define name="born-element">
    <element name="born">
      <data type="date">
```

[6] Bron: [vdV03] sectie 15.2.6.

```
        <param name="minInclusive">1900-01-01</param>
        <param name="maxInclusive">2099-12-31</param>
        <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
      </data>
    </element>
  </define>
  <define name="available-attribute">
    <attribute name="available">
      <data type="boolean">
        <except>
          <value>1</value>
        </except>
      </data>
    </attribute>
  </define>
</grammar>
```

Het volledige schema maakt gebruik van de XML Schema datatypes; enkel in het *grammar*-element is immers een datatype library gedefinieerd, die door alle nakomelingen wordt overgeërfd. De drie *define*-elementen definiëren onderdelen die in andere schema's kunnen gebruikt worden. De eerste define definieert een sequentie van een first-name en last-name element. Omdat de inhoud van deze elementen geen bepaald datatype moet hebben, bevatten zij het *text*-element. De tweede define definieert het born element met inhoud van type date. Hierop worden nog extra restricties gedefinieerd aan de hand van een aantal *param*-elementen. De waarde van het *name*-attribuut in het *param*-element moet een geldige *facet*[7] zijn die voor het gebruikte datatype in XML Schema is gedefinieerd. De derde define geeft aan dat het available attribuut een boolean waarde moet bevatten. Het exceptpattern zorgt ervoor dat deze waarde enkel true, false of 0 mag zijn. Merk ten slotte op dat een *define*-element om het even welk pattern als content kan hebben; dit moet niet noodzakelijk een elementpattern zijn.

Het bestand actor.rng bevat een grammarpattern dat een XML-document met als root element actor definieert.

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <include href="common.rng"/>
  <start>
    <element name="actor">
      <element name="name">
```

[7]Extra restricties op een voorgedefinieerd datatype worden in de XML Schema terminologie *facets* genoemd.

```
        <ref name="name-content"/>
      </element>
      <optional>
        <ref name="born-element"/>
      </optional>
      <oneOrMore>
        <externalRef href="film.rng"/>
      </oneOrMore>
    </element>
  </start>
  <define name="name-content" combine="choice">
    <text/>
  </define>
</grammar>
```

Een actor element wordt gedefinieerd als een sequentie van een name element, een optioneel born element en één of meerdere film elementen. Aan de hand van het *include*-element wordt de externe grammatica die gedefinieerd is in common.rng gemerged met de huidige grammatica, zodat ze tesamen één grote grammatica vormen. Dit heeft tot gevolg dat er twee *define*-elementen met dezelfde waarde voor het name-attribuut zijn, namelijk name-content. Omdat één van deze *define*-elementen het attribuut *combine* met als waarde *choice* bevat, worden beide *define*-elementen gemerged tot

```
<define name="name-content">
  <choice>
    <group>
      <element name="first-name"><text/></element>
      <element name="last-name"><text/></element>
    </group>
    <text/>
  </choice>
</define>
```

Elk *ref*-element in actor.rng verwijst nu naar een bijhorend *define*-element in de huidige gemergede grammar. Het *externalRef*-element wordt gebruikt om grammatica's te nesten (in plaats van te mergen zoals met het *include*-element). De waarde van het *href*-attribuut geeft aan waar deze grammatica te vinden is, namelijk in het bestand film.rng. Dit betekent dat de inhoud van dit bestand letterlijk gecopieerd wordt op de plaats van het *externalRef*-element, zodat we een grammar in een grammar krijgen.
Het bestand film.rng definieert een film element als een sequentie van een title element, een year element en een director element. Het director element bevat een name element waarvan de content gedefineerd is in de parent grammar. Dit wordt aangegeven met het *parentRef*-element.

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <element name="film">
      <element name="title"><text/></element>
      <element name="year"><text/></element>
      <element name="director">
        <element name="name">
          <parentRef name="name-content"/>
        </element>
      </element>
    </element>
  </start>
</grammar>
```

Het simplificatiealgoritme op RELAX NG schema's zorgt ervoor dat deze verschillende schema's worden samengevoegd tot één groot schema.

We sluiten deze sectie af met een XML-document dat voldoet aan het RELAX NG schema in actor.rng.[8]

```
<actor>
  <name>
    <first-name>Tom</first-name>
    <last-name>Cruise</last-name>
  </name>
  <born>1962-07-03</born>
  <film>
    <title>Eyes Wide Shut</title>
    <year>1999</year>
    <director>
      <name>Stanley Kubrick</name>
    </director>
  </film>
  <film>
    <title>Minority Report</title>
    <year>2002</year>
    <director>
      <name>
        <first-name>Steven</first-name>
        <last-name>Spielberg</last-name>
      </name>
    </director>
```

[8] Dit is getest met behulp van Jing, een validator voor RELAX NG schema's [Cla03a].

```
  </film>
</actor>
```

6.5 Abstractie als EDTD

Een RELAX NG schema kan worden voorgesteld als EDTD. Om dit in te zien vertrekken we van een RELAX NG schema in een licht gewijzigde versie van de simple syntax: elk *element*-element wordt voorgesteld door een nameclass en een referentie (aan de hand van een *ref*-element) naar een *define*-element. Er mogen geen twee *define*-elementen met dezelfde content voorkomen. Ook het *zeroOrMore*-element is toegelaten. De expressiviteit van een RELAX NG schema blijft hierdoor ongewijzigd en het type van een element is op deze manier gemakkelijk en eenduidig vast te stellen.

Voorbeeld 6.1

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element>
      <name ns="">garage</name>
      <ref name="p1"/>
    </element>
  </start>
  <define name="p1">
    <oneOrMore>
      <choice>
        <element>
          <name ns="">auto</name>
          <ref name="p2"/>
        </element>
        <element>
          <name ns="">auto</name>
          <ref name="p3"/>
        </element>
      </choice>
    </oneOrMore>
  </define>
  <define name="p2">
    <group>
      <element>
        <name ns="">merk</name>
        <ref name="p4"/>
      </element>
      <element>
```

```
        <name ns="">prijs</name>
        <ref name="p4"/>
      </element>
    </group>
  </define>
  <define name="p2">
    <group>
      <group>
        <element>
          <name ns="">merk</name>
          <ref name="p4"/>
        </element>
        <element>
          <name ns="">prijs</name>
          <ref name="p4"/>
        </element>
      </group>
      <element>
        <name ns="">bouwjaar</name>
        <ref name="p4"/>
      </element>
    </group>
  </define>
  <define name="p4">
    <text/>
  </define>
</grammar>
```

Met elk element van een RELAX NG schema wordt dus rechtstreeks een bepaald type geassocieerd. De naam van dit type is gelijk aan de naam van het element gevolgd door een superscript met als waarde het *name*-attribuut van zijn *ref*-element. Het content model voor elk type komt overeen met de inhoud van de define-tag waarnaar het superscript verwijst en we kunnen dit uitdrukken als een reguliere expressie over het typealfabet. Tabel 6.1 drukt de vertaling van de belangrijkste patterns uit. Merk op dat de vertaling van interleave slechts een benadering is van wat het interleave pattern werkelijk uitdrukt. Beschouw de twee volgende codefragmenten:

```
<element name="x">
  <interleave>
    <group>
      <element name="a"><text/></element>
      <element name="b"><text/></element>
    </group>
```

<zeroOrMore> p </zeroOrMore>	p*
<oneOrMore> p </oneOrMore>	p p*
<group> p1 p2 </group>	p1 p2
<choice> p1 p2 </choice>	p1 + p2
<interleave> p1 p2 </interleave>	(p1 p2) + (p2 p1)

Tabel 6.1: Vertaling van patterns in reguliere expressies

```
    <group>
      <element name="c"><text/></element>
      <element name="d"><text/></element>
    </group>
  </interleave>
</element>
```

en

```
<element name="x">
  <choice>
    <group>
      <element name="a"><text/></element>
      <element name="b"><text/></element>
      <element name="c"><text/></element>
      <element name="d"><text/></element>
    </group>
    <group>
      <element name="c"><text/></element>
      <element name="d"><text/></element>
      <element name="a"><text/></element>
      <element name="b"><text/></element>
    </group>
  </choice>
</element>.
```

In een EDTD worden deze als volgt voorgesteld:

$$\begin{aligned} x^1 &\rightarrow (a\ b)\ (c\ d) + (c\ d)\ (a\ b) \\ x^2 &\rightarrow (a\ b\ c\ d) + (c\ d\ a\ b) \end{aligned}$$

Beide regels zijn equivalent terwijl de codefragmenten niet equivalent zijn, het eerste fragment aanvaardt een strikt grotere verzameling XML-documenten dan het tweede. De oplossing bestaat erin de reguliere expressie te laten afhangen van de kinderen van interleave. In dit geval, wanneer beide kinderen een group pattern zijn, moet men ervoor zorgen dat de relatieve volgorde binnen elke group behouden blijft en is de correcte oplossing *abcd*

$+ acbd + acdb + bdac + badc + bacd$ (a komt steeds voor b en c komt steeds voor d). Door een exhaustieve opsomming van alle mogelijkheden neemt de grootte van de reguliere expressie sterk toe.
In het kader van deze thesis is de 'naïeve' vertaling van interleave voldoende. Ze heeft enkel invloed op het testen van equivalentie tussen twee welbepaalde EDTD's. Voorbeelden waarop deze test misloopt komen in de praktijk waarschijnlijk niet vaak voor.[9]

Omdat een EDTD geen wildcard elementen kent, beperken we ons tot RELAX NG schema's waarin enkel name-nameclassen gebruikt worden. Dit houdt echter een beperking van de expressiviteit in. Omdat een EDTD geen rekening houdt met het verschil tussen elementen en attributen worden attributen beschouwd als elementen. Om een onderscheid te maken tussen een element en attribuut met dezelfde naam, krijgt elk attribuut de prefix *a*. Elementen met deze prefix, die dus in feite attributen voorstellen, worden niet beschouwd wanneer men nagaat of een EDTD bijvoorbeeld voldoet aan de single-type eigenschap.

Een EDTD definieert precies één startelement, terwijl een RELAX NG schema verschillende startelementen kan hebben. Daarom wordt er bij de vertaling van een RELAX NG schema in een EDTD een nieuw startelement, genaamd *start*, toegevoegd. Het type van *start* noemen we $start^0$ en dit komt overeen met wat in het RELAX NG schema tussen de start-tags staat. Voor de patterns empty, text en notAllowed worden *empty*, *text* en *notAllowed* aan het alfabet en het typealfabet toegevoegd. Ook voor de patterns data, value en list worden elementen aan het alfabet en typealfabet toegevoegd. Omdat deze patterns met een verschillende inhoud kunnen voorkomen, wordt elk verschillend voorkomen aangegeven met een unieke integer. Als bijvoorbeeld *<data type="integer"/>* en *<data type="string"/>* tesamen in een RELAX NG schema voorkomen, worden deze als *data*1 en *data*2 aan het alfabet en typealfabet toegevoegd.
De elementen *start*, *empty*, *text*, *notAllowed*, *data*1 en *data*2 zijn gereserveerde woorden in de resulterende EDTD. Deze elementen moeten niet als zodanig in een te valideren XML-document voorkomen, maar zijn enkel geïntroduceerd om verschillende RELAX NG schema's correct met elkaar te kunnen vergelijken in hun EDTD-representatie (bijvoorbeeld om equivalentie te testen). Hun content model is gelijk aan ϵ.

In een RELAX NG schema is ambiguïteit toegelaten. Als gevolg hiervan kunnen er ambigue reguliere expressies voorkomen in de corresponderende EDTD. In een reguliere expressie kunnen ook twee elementen met dezelfde naam en een verschillend type voorkomen, wat tot gevolg heeft dat de

[9]Het corpus bestaande RELAX NG schema's is voorlopig te klein om dit te kunnen onderzoeken.

corresponderende EDTD niet noodzakelijk een single-type EDTD is. We kunnen besluiten dat de klasse van RELAX NG schema's precies overeenkomt met de klasse van EDTD's. De enige beperking die we aan RELAX NG schema's opleggen is dat ze enkel name-nameclasses bevatten. Omdat elke EDTD uitdrukbaar is als een boomautomaat, kunnen we alle operaties die hierop bekend zijn, bijvoorbeeld de equivalentietest, ook toepassen op RELAX NG schema's.

Hoofdstuk 7

Implementatie

Als onderdeel van deze thesis is een implementatie gemaakt van (een deel van) de hierboven beschreven theorie, meer bepaald van het simplificatiealgoritme beschreven in het bewijs van stelling 4.2. Er wordt nagegaan of een bestaand RELAX NG schema voldoet aan de single-type eigenschap en, indien niet, wordt er een equivalent schema geconstrueerd dat wel single-type is, indien dit tenminste bestaat. Het single-type schema kan vervolgens omgezet worden naar de XML Schema syntax, want het voldoet aan de EDC constraint.

In dit hoofdstuk volgt een bespreking van de gemaakte implementatie.

7.1 Situering

Een schemaconverter is een programma dat een XML-schema in de ene schemataal omzet naar een equivalent schema in een andere taal. Omdat elke schemataal zijn eigen karakteristieken heeft, bestaat er niet steeds een equivalent schema in de andere taal. In dat geval kan er een benaderend schema worden geconstrueerd en moet aan de gebruiker gemeld worden waar het nieuwe schema afwijkt van het oorspronkelijke.

Een voorbeeld van een bestaande schemaconverter is Trang [Cla03c], ontworpen door James Clark, die ook een belangrijke bijdrage heeft geleverd aan de ontwikkeling van de schemataal RELAX NG. Trang kan o.a. schema's in RELAX NG omzetten in XML Schema. Het resulterende schema voldoet echter niet steeds aan de XML Schema specificatie. Zo wordt er bijvoorbeeld geen rekening gehouden met het feit dat een XML Schema moet voldoen aan de EDC-constraint, een beperking die niet geldt in RELAX NG. Indien een RELAX NG schema reguliere expressies heeft die niet one-unambiguous zijn, wordt ook de UPA-constraint geschonden.
Als onderdeel van deze thesis is Trang uitgebreid met een extra functionaliteit: indien een RELAX NG schema niet single-type is, wordt nagegaan

of er een equivalent schema bestaat dat wel single-type is. Indien ja, wordt dit geconstrueerd en als input aan het oorspronkelijk Trang-programma gegeven; indien nee, wordt een foutboodschap uitgeschreven. Op deze manier voldoet het geconstrueerde XML Schema steeds aan de EDC-constraint.

Bij de conversie probeert Trang de structuur van een inputschema zoveel mogelijk te bewaren [Cla03c]:

> [...]; it tries for a translation that preserves all aspects of the input schema that may be significant to a human reader, including the definitions, the way the schema is divided into files, annotations and comments.

Indien een RELAX NG schema over verschillende bestanden is verdeeld waarbij de relatie tussen deze bestanden wordt gelegd aan de hand van de externalRef-tag, krijgen we echter de boodschap dat dit feature nog niet ondersteund wordt door Trang (er wordt dus geen output in XML Schema gegenereerd). Dit geldt ook voor geneste grammatica's in het algemeen[1] en het gebruik van de parentRef-tag. Wanneer een extern schema wordt ingevoerd aan de hand van de include-tag, zijn herdefinities niet mogelijk. In dit laatste geval wordt wel een XML Schema output gegenereerd, maar eventuele herdefinities worden volledig genegeerd. Indien we in al deze gevallen toch een corresponderend XML Schema willen verkrijgen, kunnen we op het oorspronkelijke RELAX NG schema het simplificatie-algoritme uit sectie 6.2 toepassen. Het resulterende RELAX NG schema wordt steeds volledig door Trang ondersteund. De oorspronkelijke verdeling in verschillende bestanden, evenals annotaties en commentaar zijn dan echter verdwenen in het XML Schema.

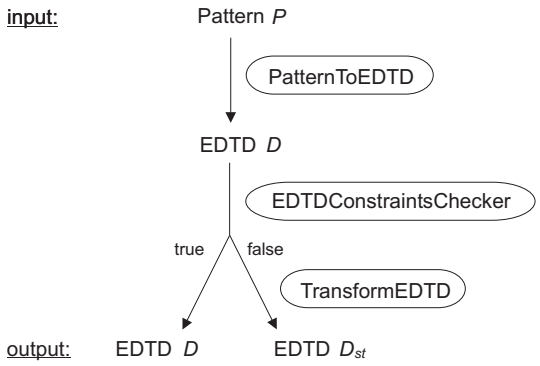
Om na te gaan of een RELAX NG schema voldoet aan de single-type eigenschap, wordt het volledige schema in eenzelfde *Pattern*-datastructuur ondergebracht. Om dit te realiseren wordt gebruik gemaakt van de RELAX NG schemaparser van Jing [Cla03a], een RELAX NG schema validator, eveneens ontworpen door James Clark. Deze parser maakt gebruik van het simplificatiealgoritme, waardoor bijvoorbeeld geneste grammatica's niet meer voorkomen en verwijzingen naar externe schema's verdwenen zijn. Dit is nodig om op een eenvoudige manier te kunnen nagaan of een gegeven schema single-type is.

7.2 Toepassing van het simplificatiealgoritme

De input van Trang is een bestand met een RELAX NG schema. Het schema wordt met behulp van Jings RELAX NG schemaparser ingelezen en

[1] De geneste grammatica hoeft niet noodzakelijk in een apart bestand te staan.

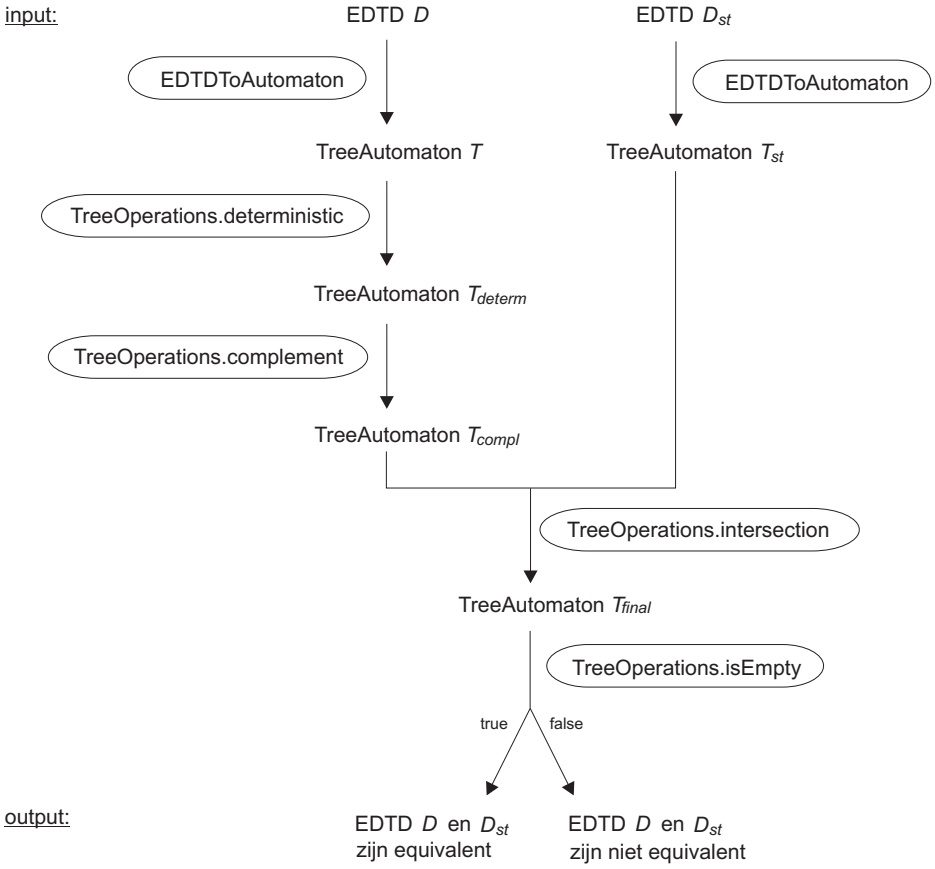
voorgesteld als een instantie van de *Pattern*-klasse. Dit pattern P wordt vervolgens omgezet naar een EDTD D en er wordt getest of D single-type is. Als D single-type is, wordt het als RELAX NG schema naar een bestand geschreven. Dit bestand wordt dan als input gegeven aan de oorspronkelijke Trang-implementatie die het omzet naar XML Schema. Het corresponderende XML Schema zal steeds voldoen aan de EDC-constraint. Als D niet single-type is, wordt uit D een single-type EDTD D_{st} geconstrueerd volgens het algoritme beschreven in sectie 3.4.4. Dit proces wordt schematisch voorgesteld in figuur 7.1. De ovalen stellen klassen voor die verantwoordelijk zijn voor een bepaalde bewerking.



Figuur 7.1: Schematische voorstelling van het algoritme (deel 1)

Indien D niet single-type is, moet getest worden of D en D_{st} equivalent zijn. Omdat volgens de constructie geldt dat $\mathrm{L}(D) \subseteq \mathrm{L}(D_{st})$, moet enkel getest worden of $\mathrm{L}(D_{st}) \subseteq \mathrm{L}(D)$. Deze test wordt geïmplementeerd aan de hand van boomautomaten. De EDTD's D en D_{st} worden beide omgezet in hun equivalente boomautomaat T en T_{st}. De boomautomaten worden voorgesteld door instanties van de klasse *TreeAutomaton*. Uit T construeren we T_{compl} die het complement van T definieert, en vervolgens gaan we na of $\mathrm{L}(T_{compl}) \cap \mathrm{L}(T_{st})$ leeg is. Als deze test *true* oplevert, dan zijn D en D_{st} equivalent, en kan D_{st} als input gegeven worden aan de eigenlijke schemaconvertor. Als het resultaat van deze test *false* is, definieert D_{st} een strikt grotere taal dan D. Ook in dit geval geven we D_{st} als input aan de schemaconvertor, maar wordt aan de gebruiker een waarschuwing gegeven. Dit proces wordt weergegeven in figuur 7.2. De ovalen stellen hier klassen en functies in klassen voor.

Het single-type maken van een RELAX NG schema gebeurt eigenlijk op twee niveaus. Bij het inlezen van een schema door de Jing-schemaparser worden de eenvoudige gevallen ontdekt. Een schema met de regels $a \to b^1 b^2$, $b^1 \to c$ en $b^2 \to c$ wordt intern onmiddellijk voorgesteld als $a \to bb$ en $b \to c$. Enkel



Figuur 7.2: Schematische voorstelling van het algoritme (deel 2)

indien dit resultaat nog niet single-type is, wordt het simplificatiealgoritme zoals hierboven beschreven toegepast. Dat er voor het schema met de regels $a \rightarrow b^1b^2$, $b^1 \rightarrow x + y$ en $b^2 \rightarrow y + x$ een equivalent single-type schema bestaat, wordt niet door de Jing-parser ontdekt.

Het simplificatiealgoritme is EXPTIME-compleet. Zowel het construeren van een single-type EDTD als het testen of beide EDTD's equivalent zijn, is exponentieel. Door het implementeren van een aantal optimalisaties is het algoritme in vele gevallen wel praktisch uitvoerbaar. Deze optimalisaties worden verderop in dit hoofdstuk besproken.

7.3 Voorstelling van de belangrijkste datastructuren

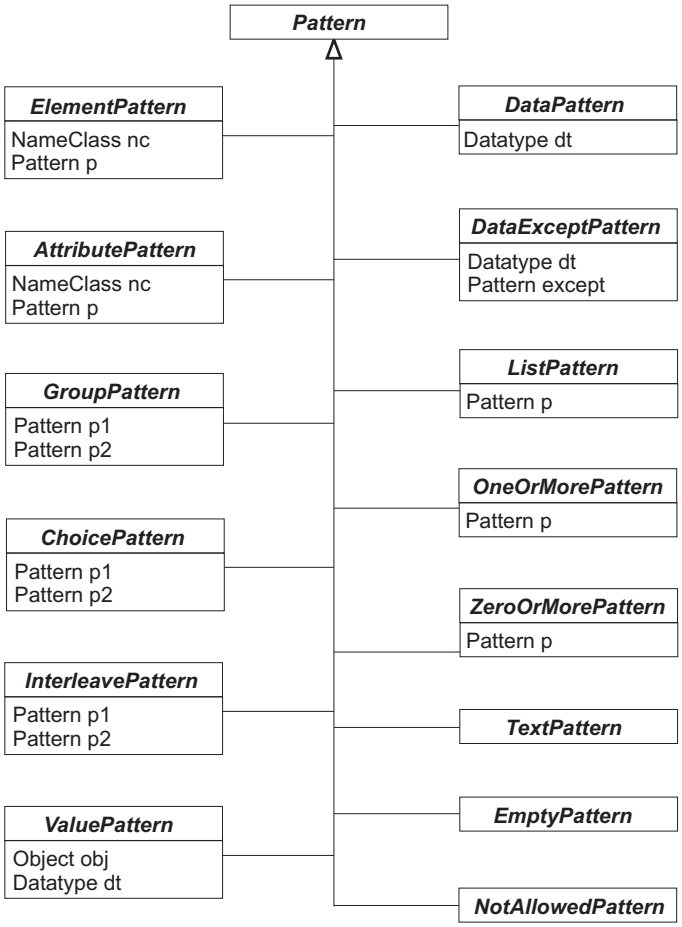
7.3.1 Patterns

Een RELAX NG schema dat uit een file ingelezen is, wordt voorgesteld door de klasse *Pattern*. Dit is een abstracte klasse waarvan de subklassen de concrete patterns van een RELAX NG schema voorstellen zoals bijvoorbeeld het element- of choicepattern. De klassenhiërarchie wordt getoond in figuur 7.3. Enkel de onderdelen die relevant zijn in deze context worden weergegeven.[2]

Het spreekt voor zich welke specifieke RELAX NG patterns door de verschillende subklassen worden uitgedrukt. Zowel het *ElementPattern* als het *AttributePattern* bevatten een membervariabele van het type *NameClass*. In deze implementatie worden enkel nameclasses beschouwd die precies één naam uitdrukken. Ze worden voorgesteld door de klasse *SimpleNameClass*, een subklasse van *NameClass*, en bevatten een *Name*-object als representatie van een element- of attribuutnaam. Het *Datatype*-object in de klassen *DataPattern*, *DataExceptPattern* en *ValuePattern* legt beperkingen op aan de datawaarden die in een XML-document mogen voorkomen. Een concrete *Datatype*-instantie geeft bijvoorbeeld aan dat de data van het type string moet zijn met een maximumlengte van vijf karakters. Wat ook opvalt is de geneste structuur van deze klassen, waardoor een RELAX NG schema als een boom wordt voorgesteld.

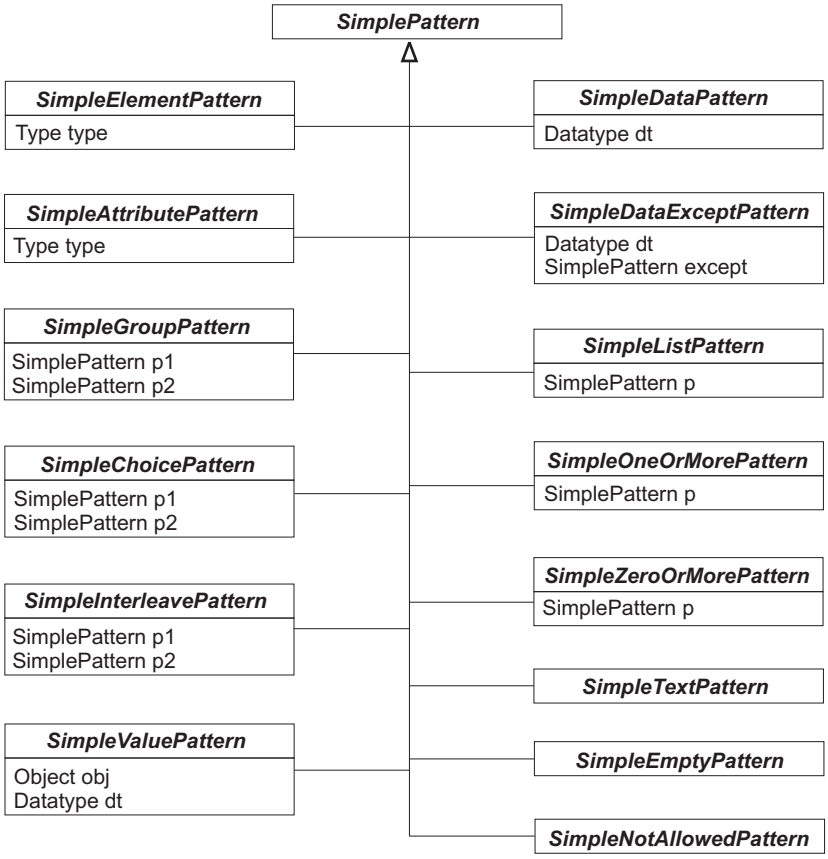
Om te controleren of een RELAX NG schema voldoet aan de single-type eigenschap, is het nodig per element duidelijk aan te geven als welk type het voorkomt. Dit wordt gerealiseerd door de *Pattern*-voorstelling gedeeltelijk te ontnesten. De ontneste versie van een RELAX NG schema wordt voorgesteld

[2]Deze klassen behoren tot de Jing-implementatie, uitgezonderd de klasse *ZeroOrMorePattern*.



Figuur 7.3: Hiërarchie van de klasse *Pattern*

door de klasse *SimplePattern*. De klassenhiërarchie is gelijkaardig aan die van *Pattern* en wordt getoond in figuur 7.4.



Figuur 7.4: Hiërarchie van de klasse *SimplePattern*

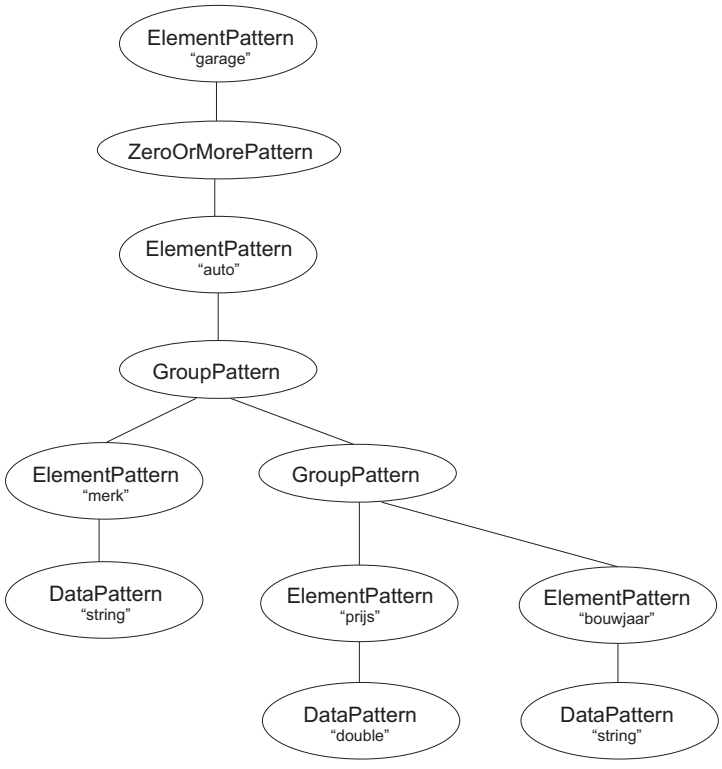
Enkel de klassen *SimpleElementPattern* en *SimpleAttributePattern* zijn fundamenteel verschillend van hun tegenhanger in de *Pattern*-voorstelling. Ze bevatten elk een *Type*-object, dat aangeeft als welk type het element of attribuut voorkomt. De naam van het element of attribuut zit hierin vervat. Het gevolg is dat een RELAX NG schema niet meer als één grote boom wordt voorgesteld, maar is opgesplitst in een verzameling kleinere bomen. De *SimplePattern*-klasse wordt gebruikt in de voorstelling van een EDTD.

Voorbeeld 7.1

We tonen een RELAX NG fragment en zijn corresponderende *Pattern*-voorstelling in figuur 7.5. Het gebruik van de *SimplePattern*-klasse komt aan bod in de volgende paragraaf.

```
<element name="garage">
```

```
  <zeroOrMore>
    <element name="auto">
      <group>
        <element name="merk">
          <data type="string"/>
        </element>
        <element name="prijs">
          <data type="double"/>
        </element>
        <element name="bouwjaar">
          <data type="string"/>
        </element>
      </group>
    </element>
  </zeroOrMore>
</element>
```



Figuur 7.5: Het garage-element als *Pattern*

Ook als de RELAX NG code op een vlakke manier geschreven wordt aan de hand van de define-tag, blijft de *Pattern*-voorstelling dezelfde als in figuur 7.5.

7.3.2 EDTD's

Een EDTD $D = (\Sigma, \Delta, d, \mu)$ wordt voorgesteld door een instantie van de klasse *EDTD*. De membervariabelen *alphabet* en *typeAlphabet* stellen respectievelijk het alfabet en het type-alfabet van D voor. *Alphabet* is een verzameling *Name*-instanties, die een element- of attribuutnaam representeren bestaande uit een namespace-uri en een locale naam, beide van het type string. *TypeAlphabet* is een verzameling *Type*-instanties, die het type van een element representeren. De mappingfunctie is van de vorm $\mu(a^i) = a$ waardoor deze niet expliciet moet voorgesteld worden. De grammaticaregels worden voorgesteld door de hashmap *rules*: de key hiervan is een integer die het type van een element voorstelt en wordt gekoppeld aan een *SimplePattern*-instantie.

Name
String namespaceUri
String localName

Type
Name name
String type

EDTD
Set<Name> alphabet
Set<Type> typeAlphabet
Map<Integer, SimplePattern> rules

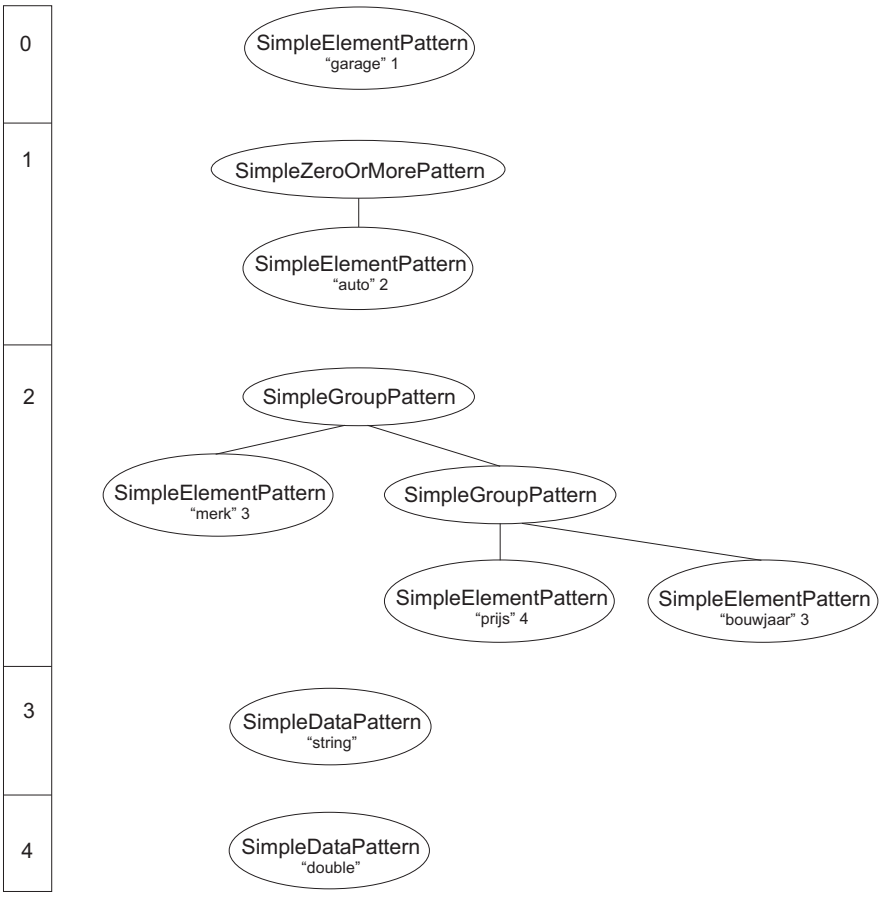
Voorbeeld 7.2

In figuur 7.5 toonden we de *Pattern*-voorstelling van het garage-element. Figuur 7.6 toont de regels van de corresponderende EDTD.

De reguliere expressies aan de rechterkant van de regels worden opgeslaan als *SimplePattern*-objecten. Merk op dat de regel voor integer 3 slechts één keer voorkomt in plaats van twee keer (voor $merk^3$ en $bouwjaar^3$). De interne voorstelling met integers als key van de *rules*-map vermindert dus redundantie. Eenzelfde content model wordt slechts één keer opgeslaan. Dit levert tijd- en plaatswinst op bij het berekenen van een mogelijk equivalent single-type schema. Algemeen kunnen we stellen: als twee elementen eenzelfde integer als typeaanduiding hebben, dan hebben ze hetzelfde content model; als twee elementen een verschillende integer als typeaanduiding hebben, dan hebben hebben ze niet hetzelfde content model.

7.3.3 Boomautomaten

Een boomautomaat $T = (\Sigma, Q, \delta, F)$ wordt voorgesteld door een instantie van de klasse *TreeAutomaton*. De vier membervariabelen zijn alle van het type *Set*. *States* en *acceptStates* representeren respectievelijk de verzameling toestanden Q en eindtoestanden F. Het alfabet Σ wordt gerepresenteerd door de variabele *alphabet*, de transitiefunctie δ wordt voorgesteld door de variabele *transitions*.



Figuur 7.6: Het garage-element als *EDTD*

TreeAutomaton
Set<String> alphabet
Set<String> states
Set<String> acceptStates
Set<Transition> transitions

Transition
FiniteStateAutomaton childExpression
String symbol
String toState

Terwijl de eerste drie membervariabelen van *TreeAutomaton* verzamelingen van strings zijn, is de laatste een verzameling van *Transition*-objecten. De klasse *Transition* representeert een transitie van een boomautomaat. Een transitie drukt het volgende uit: in een boom wordt aan een node v met als naam *symbol* de toestand *toState* toegekend als de toestandsstring van de kinderen van v aanvaard wordt door de stringautomaat *childExpression*.

7.3.4 Stringautomaten

Voor de representatie van stringautomaten wordt gebruik gemaakt van het externe package JFLAP [Rod05]. De klasse *FiniteStateAutomaton*, overgeërfd van de meer algemene *Automaton*-klasse, stelt een NFA $(Q, \Sigma, \delta, q_0, F)$ voor. De membervariabelen *states*, *initialState* en *finalStates* stellen respectievelijk de verzameling toestanden Q, de begintoestand q_0 en de verzameling eindtoestanden F voor. De transitiefunctie δ wordt voorgesteld door de membervariabele *transitions*, die een verzameling *FSATransition*-objecten bevat. Voor elke transitie $\delta(q, a) = q'$ bevat deze verzameling een corresponderend *FSATransition*-object. Het alfabet Σ is niet expliciet voorgesteld, maar kan afgeleid worden uit de symbolen in de transitiefunctie.

7.4 Optimalisaties

Het simplificatiealgoritme is een exponentieel algoritme, wat vragen oproept in verband met de praktische uitvoerbaarheid van het algoritme. Er zijn een aantal optimalisaties geïmplementeerd die ervoor zorgen dat het algoritme op eenvoudige gevallen toch uitvoerbaar is.

7.4.1 Constructie van de single-type EDTD

De beschrijving van het algoritme om uit een gegeven EDTD D een mogelijk equivalente single-type EDTD D_{st} te construeren, geeft aan dat D_{st} exponentieel meer types dan D kan bevatten. Deze types moeten echter niet allemaal geconstrueerd worden, omdat ze ofwel niet bereikbaar zijn ofwel

omdat hun reguliere expressie de lege verzameling uitdrukt. We vertrekken van de bestaande regels in D. Indien er reguliere expressies voorkomen die niet single-type zijn, worden deze single-type gemaakt door voor elk element dat met verschillende types voorkomt een nieuw type te construeren en de conflicterende types in de reguliere expressie hierdoor te vervangen. Nadat deze types zijn gedefinieerd, wordt hun reguliere expressies geconstrueerd. Deze nieuwe reguliere expressies kunnen nu op hun beurt de single-type eigenschap schenden. Een nieuwe iteratie wordt ingezet. Omdat er maximaal 2^n types mogelijk zijn, stopt dit proces steeds. We illustreren dit aan de hand van een voorbeeld.

Voorbeeld 7.3

$$\begin{array}{lcllcl} a & \rightarrow & (b^1cb^2)^* & p^1 & \rightarrow & r \\ b^1 & \rightarrow & x^1 & p^2 & \rightarrow & s \\ b^2 & \rightarrow & x^2 & c & \rightarrow & \epsilon \\ x^1 & \rightarrow & p^1p^2 & r & \rightarrow & \epsilon \\ x^2 & \rightarrow & p^1 & s & \rightarrow & \epsilon \end{array}$$

De EDTD bestaande uit bovenstaande regels is duidelijk niet single-type. Zowel de regel voor a als die voor x^1 schenden deze eigenschap. Deze twee regels worden vervangen door $a \rightarrow (b^{\{1,2\}}cb^{\{1,2\}})^*$ en $x^1 \rightarrow p^{\{1,2\}}p^{\{1,2\}}$. Er worden nu twee nieuwe types gecreëerd met een bijhorende reguliere expressie, namelijk $b^{\{1,2\}} \rightarrow (x^1 + x^2)$ en $p^{\{1,2\}} \rightarrow r + s$. De regel voor $(b^{\{1,2\}}$ is niet single-type en een nieuwe iteratie wordt ingezet. We construeren het nieuwe type $x^{\{1,2\}}$ met bijhorende reguliere expressie, namelijk $x^{\{1,2\}} \rightarrow p^1p^2 + p^1$. Omdat deze regel niet single-type is, wordt ze vervangen door $x^{\{1,2\}} \rightarrow p^{\{1,2\}}p^{\{1,2\}} + p^{\{1,2\}}$. Deze vervanging introduceert geen nieuw type en dus stopt het iteratief proces. Het resultaat is volgende single-type EDTD:

$$\begin{array}{lcllcl} a & \rightarrow & (b^{\{1,2\}}cb^{\{1,2\}})^* & c & \rightarrow & \epsilon \\ b^1 & \rightarrow & x^1 & r & \rightarrow & \epsilon \\ b^2 & \rightarrow & x^2 & s & \rightarrow & \epsilon \\ x^1 & \rightarrow & p^{\{1,2\}}p^{\{1,2\}} & b^{\{1,2\}} & \rightarrow & (x^{\{1,2\}} + x^{\{1,2\}}) \\ x^2 & \rightarrow & p^1 & p^{\{1,2\}} & \rightarrow & r + s \\ p^1 & \rightarrow & r & x^{\{1,2\}} & \rightarrow & p^{\{1,2\}}p^{\{1,2\}} + p^{\{1,2\}} \\ p^2 & \rightarrow & s & & & \end{array}$$

Deze EDTD bevat een groot aantal onbereikbare types, dus blijven enkel a, $b^{\{1,2\}}$, c, $x^{\{1,2\}}$, $p^{\{1,2\}}$, r en s over. Het aantal regels en types wordt dus nog eens met de helft verminderd.

7.4.2 Determiniseren van een boomautomaat

Ook het deterministisch maken van een boomautomaat zit in EXPTIME. Indien een niet-deterministische automaat A n toestanden bevat, dan heeft

zijn deterministische tegenhanger A'er 2^n omdat elke mogelijke deelverzameling van de toestanden in A een toestand van A' wordt. Het is echter niet nodig al deze toestanden te creëren. Boomautomaten die een RELAX NG schema representeren zijn van een speciale vorm. Als men door een alfabetsymbool a te lezen in een toestand q_1 kan geraken, is het niet mogelijk in toestand q_1 te geraken door een ander alfabetsymbool dan a te lezen. Dit kan als volgt geformaliseerd worden: voor elk koppel transities $\delta(q_1, a)$ en $\delta(q_1, b)$ geldt dat minstens één van beide de lege verzameling definieert indien $a \neq b$. Dit heeft een invloed op het construeren van de verzameling toestanden Q' in A'. Per alfabetsymbool wordt de machtsverzameling van toestanden berekend die met dat alfabetsymbool voorkomen. Omdat het aantal toestanden dat met een bepaald alfabetsymbool voorkomt meestal vrij klein is, is deze machtsverzameling niet al te groot. Als er k alfabetsymbolen voorkomen, worden er dus k machtsverzamelingen gecreëerd. De unie van al deze machtsverzamelingen vormt de uiteindelijk verzameling toestanden van A'. We illustreren dit aan de hand van een voorbeeld.

Voorbeeld 7.4

Neem de niet-deterministische boomautomaat $A = (\Sigma, Q, \delta, F)$ met

- $\Sigma = \{a, b, c, p, x, y, z\}$
- $Q = \{a^1, b^1, b^2, c^1, c^2, c^3, p^1, p^2, p^3, x^1, y^1, z^1\}$
- $F = \{a^1\}$

en waarbij enkel de volgende transities een niet-lege taal definiëren:

$\delta(a^1, a)$	$\delta(p^1, p)$
$\delta(b^1, b)$	$\delta(p^2, p)$
$\delta(b^2, b)$	$\delta(p^3, p)$
$\delta(c^1, c)$	$\delta(x^1, x)$
$\delta(c^2, c)$	$\delta(y^1, y)$
$\delta(c^3, c)$	$\delta(z^1, z)$

Dan geldt voor de equivalente boomautomaat $A' = (\Sigma, Q', \delta', F')$ dat

- $\Sigma = \{a, b, c, p, x, y, z\}$
- $Q = \{\{a^1\}, \{b^1\}, \{b^2\}, \{b^1, b^2\}, \{c^1\}, \{c^2\}, \{c^3\}, \{c^1, c^2\}, \{c^1, c^3\}, \{c^2, c^3\}, \{c^1, c^2, c^3\}, \{p^1\}, \{p^2\}, \{p^3\}, \{p^1, p^2\}, \{p^1, p^3\}, \{p^2, p^3\}, \{p^1, p^2, p^3\}, \{x^1\}, \{y^1\}, \{z^1\}\}$
- $F = \{\{a^1\}\}$

Voor elke toestand wordt er slechts één niet-lege tranitie gedefinieerd, met als alfabetsymbool r indien de toestand van de vorm $\{r^1, \ldots, r^n\}$ is.

Boomautomaten die een RELAX NG schema representeren hebben in de praktijk eerder een grote verzameling alfabetsymbolen en in verhouding een beperkte verzameling toestanden. Hierdoor blijft de verzameling toestanden van zijn deterministische tegenhanger vrij klein. Omdat het berekenen van $\delta'(R, a)$ voor elke toestand R in Q' een zware operatie is en afhangt van het aantal toestanden in Q', zorgt de beperking van het aantal toestanden in Q' voor een aanzienlijke tijdswinst.

De tranitiefunctie wordt als volgt geconstrueerd. Voor elke toestand $R = \{r^1, \ldots, r^n\}$ in Q' berekenen we

$$\delta'(R, r) = \bigcap_{q \in R} f(\delta(q, r)) - \bigcup_{q \in (Q-R)} f(\delta(q, r))$$

met als substitutie $f : Q \to Q'$ zodat $f(q) = \{S \mid S \subseteq Q \text{ en } q \in S\}$. Deze substitutie zorgt ervoor dat een stringautomaat een groot aantal bogen krijgt indien $f(q)$ een grote verzameling definieert. Door toepassing van de optimalisatie is elke verzameling $f(q)$ vrij klein en dit heeft een gunstige invloed op de tijd die nodig is om elke transitie te construeren. Doordat bovendien een minimaliseringsalgoritme is geïmplementeerd op stringautomaten, hebben deze steeds een minimaal aantal toestanden.

7.5 Evaluatie

Het implementeren van de optimalisaties zorgt voor een enorme tijdswinst bij het uitvoeren van het simplificatiealgoritme. Toen de optimalisatie voor het deterministisch maken van een boomautomaat nog niet was geïmplementeerd, was dit algoritme zelfs niet uitvoerbaar voor eenvoudige RELAX NG schema's waarin slechts acht verschillende types voorkwamen.

Hoofdstuk 8

Besluit

DTD's hebben een eerder beperkte expressiviteit. Het type van een element hangt enkel af van de naam dit element, waardoor ze de klasse van locale boomtalen definiëren. Bij EDTD's kan eenzelfde elementnaam met verschillende types geassocieerd worden. Het type van een element hangt hier dus af van de context van het element. EDTD's vallen samen met de robuuste en welbekende klasse van reguliere boomtalen. Voor deze grote expressieve uitdrukkingskracht wordt echter een prijs betaald. Het verwerken van XML-documenten gebeurt namelijk bottom-up, waarbij het type van een element pas wordt bepaald na het lezen van zijn content. In de context van streaming XML is het gewenst om het type van een element te bepalen op het moment dat men de begintag tegenkomt. Dit betekent dat het type van een element enkel afhangt van zijn preceding. Een EDTD die het toelaat een XML-document te typen wanneer men de begintag tegenkomt, noemt men one-pass preorder typeable. Merk op dat dit een semantische eigenschap is; ze definieert niet hoe een EDTD er precies uit moet zien, maar beschrijft hoe een EDTD zich moet gedragen. Niet elke EDTD is one-pass preorder typeable. We spreken in deze context van efficiënte typing indien one-pass preorder typing mogelijk is.
Bij single-type EDTD's wordt er een beperking opgelegd aan de reguliere expressies die aan de rechterkant van een regel voorkomen. Hierdoor hangt het type van een element enkel af van zijn voorouders en is one-pass preorder typing steeds mogelijk. De beperking tot single-type reguliere expressies is echter te strikt voor zijn doel, namelijk efficiënte typing toelaten. Restrained competition EDTD's vormen de grootste subklasse van EDTD's die one-pass preorder typing toelaten. Ook al zijn restrained competition reguliere expressies semantisch gedefinieerd, toch bestaat er een kwadratisch algoritme dat nagaat of een reguliere expressie hieraan voldoet. Er bestaat ook een syntactische tegenhanger die dezelfde klasse van talen definieert, namelijk ancestor-sibling based schema's. Nagaan of een willekeurige EDTD een equivalente single-type EDTD of restrained competition EDTD heeft en

deze construeren, is EXPTIME-compleet.
Omdat RELAX NG geen essentiële beperkingen oplegt aan het content model van een element, komt deze schemataal qua expressiviteit overeen met EDTD's. One-pass preorder typing is niet altijd mogelijk. De EDC-constraint in XML Schema zorgt ervoor dat deze schemataal overeenkomt met single-type EDTD's. Onafhankelijk van de EDC-constraint zorgt ook de UPA-constraint ervoor dat XML Schema one-pass preorder typeable is. Indien deze constraints enkel gedefinieerd zijn om efficiënte typing toe te laten, kunnen beide constraints beter vervangen worden door de notie van restrained competition reguliere expressies.
RELAX NG drukt een strikt grotere klasse van boomtalen uit dan XML Schema. De bestaande schemaconvertor Trang houdt hier echter geen rekening mee bij de vertaling van een RELAX NG schema naar XML Schema. Door de implementatie van het simplificatiealgoritme worden wel steeds correcte XML Schema's geconstrueerd worden, indien het tenminste bestaat. Hoewel dit algoritme een exponentiële tijdscomplexiteit heeft, zorgen een aantal optimalisaties ervoor dat het in de praktijk toch in vele gevallen uitvoerbaar is.

Bibliografie

[BKMW01] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1, april 3, 2001. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[BKW98] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Information and Computation*, 140(2):229–253, 1998.

[BM04] Paul V. Biron and Ashok Malhotra. Xml schema part 2: Datatypes. Technical report, World Wide Web Consortium, oktober 2004. `http://www.w3.org/TR/xmlschema-2/`.

[BMNS05] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of xsds: from practice to theory, there and back again. In *Proceedings of the 14th international conference on World Wide Web (WWW)*, pages 712–721, New York, 2005. ACM Press.

[BPSM+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). Technical report, World Wide Web Consortium, februari 2004. `http://www.w3.org/TR/REC-xml/`.

[CDG+97] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997.

[Cla03a] James Clark. Jing. a relax ng validator in java, 2003. `http://www.thaiopensource.com/relaxng/jing.html`.

[Cla03b] James Clark. Relax ng, 2003. `http://www.relaxng.org`.

[Cla03c] James Clark. Trang. multi-format schema converter based on relax ng, 2003. `http://www.thaiopensource.com/relaxng/trang.html`.

[CM01a] James Clark and Makoto Murata. Relax ng specification. Technical report, OASIS, december 2001. `http://www.oasis-open.org/committees/relax-ng/spec.html`.

[CM01b] James Clark and Makoto Murata. Relax ng tutorial. Technical report, OASIS, december 2001. `http://www.oasis-open.org/committees/relax-ng/tutorial.html`.

[FW04] David C. Fallside and Priscilla Walmsley. Xml schema part 0: Primer. Technical report, World Wide Web Consortium, oktober 2004. `http://www.w3.org/TR/xmlschema-0/`.

[GV04] Marc Gyssens and Stijn Vansummeren. Logica en databases. cursus 2de lic Informatica, Universiteit Hasselt, 2004.

[HSW01] Juraj Hromkovic, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small epsilon-free nondeterministic finite automata. *Journal of Computer and System Sciences*, 62(4):565–588, 2001.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Man01] Murali Mani. Keeping chess alive - do we need 1-unambiguous content models? In *Extreme Markup Languages*, augustus 2001.

[Man03] Murali Mani. *Data Modeling using XML Schemas*. PhD thesis, University of California, Los Angeles, juli 2003.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. In *ACM Transactions of Internet Technology (TOIT)*, november 2005.

[MNSB05] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness of xml schema: the element declaration consistent constraint explained. 2005.

[Nev02a] Frank Neven. Automata, logic, and xml. In *Conference for Computer Science Logic (CSL)*, pages 2–26, Berlijn, 2002. Springer.

[Nev02b] Frank Neven. Automata theory for xml researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[PV00] Yannis Papakonstantinou and Victor Vianu. Dtd inference for views of xml data. In *Proceedings of the 19th Symposium on Principles of Database Systems (PODS)*, pages 35–46, New York, 2000. ACM Press.

[PV03] Yannis Papakonstantinou and Victor Vianu. Incremental validation of xml documents. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, pages 47–63, Berlijn, 2003. Springer.

[Rod05] Susan H. Rodger. Jflap, 2005. `http://www.jflap.org/`.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1997.

[SMT05] C. M. Sperberg-McQueen and Henry Thompson. Xml schema, 2005. `http://www.w3.org/XML/Schema`.

[TBMM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. Xml schema part 1: Structures. Technical report, World Wide Web Consortium, oktober 2004. `http://www.w3.org/TR/xmlschema-1/`.

[vdV02] Eric van der Vlist. *XML Schema*. O'Reilly, 2002.

[vdV03] Eric van der Vlist. *RELAX NG*. O'Reilly, 2003. `http://books.xmlschemata.org/relaxng/`.