Een front-end voor XML Schema

Jan Olaerts

promotor :
Prof. dr. Frank NEVEN

co-promotor :
dr. Geert BEX

Universiteit Maastricht

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

universiteit hasselt

Abstract

Dit document is een thesis voorgedragen tot het behalen van de graad van master in de informatica/ICT/kennistechnologie. Nadat er een beknopte inleiding is gegeven tot XML, wordt beschreven wat een schemataal is en welk nut een schemataal heeft. Vervolgens wordt er een overzicht gegeven van de bestaande schematalen. De besproken schematalen bezitten een zekere uitdrukkingskracht. Op basis van de relatie tussen XML documenten en bomen wordt afgeleid wat de relatie is tussen een XML schema en een boomtaal. Er wordt voor XML Schema, RELAX NG en DTDs exact bepaald met welke klasse van boomtalen ze overeenkomen. Uit een studie blijkt dat er nood is aan een concrete schemataal, gebaseerd op DTDs, die gemakkelijk is in gebruik en die de meest voorkomende schema's aankan. Op basis van de bevindingen in deze studie wordt een dergelijke schemataal, DTD Extra genaamd, geïntroduceerd en in detail uitgewerkt. Er worden tevens tools (een parser, evaluator en transformator) ontwikkeld om de schemataal in de praktijk te kunnen toepassen. Voor elke tool worden de achterliggende logica, de gebruikte hulpmiddelen en de broncode toegelicht. Daarnaast wordt kort beschreven hoe de correcte werking van de tools is gecontroleerd.

Woord vooraf

Deze thesis vormt het sluitstuk van mijn opleiding tot Master in de Informatica met afstudeerrichting Databases aan de Universiteit Hasselt. Er zijn heel wat personen die mij geholpen hebben om dit hoofdstuk van mijn leven tot een goed einde te brengen. Een woord van dank is hier dan ook op zijn plaats.

Allereerst bedank ik mijn promotor, Prof. dr. Frank Neven, voor zijn raad en begeleiding maar ook voor de grote vrijheid die ik heb gekregen om deze thesis te realiseren.

Vervolgens wil ik mijn co-promotor, dr. Geert Jan Bex, bedanken voor de dagelijkse steun, vele tips en uitleg die hij mij heeft gegeven, zowel bij het opstellen van de tekst als tijdens het uitwerken van de implementatie.

Bijzondere dank gaat uit naar mijn ouders en vriendin die mij de kans èn de moed hebben gegeven om deze opleiding aan te vatten. Zonder hen zou ik deze stap nooit hebben ondernomen.

Tenslotte wil ik familie en vrienden bedanken die invloed hebben gehad op het uiteindelijke resultaat. Bedankt voor het nalezen van de tekst, allerhande tips en de broodnodige ontspanning.

Jan Olaerts
20 mei 2007

Inhoudsopgave

Abstract **i**

Woord vooraf **ii**

Inhoudsopgave **iii**

Lijst van figuren **v**

1 Inleiding **1**
1.1 Doelstelling 1
1.2 Overzicht 2

2 XML en XML schematalen **3**
2.1 Wat is XML? 3
2.2 XML schematalen 4
2.3 DTDs 5
2.4 XML Schema 7
2.5 RELAX NG 15
2.6 Schematron 19
2.7 SchemaPath 21
2.8 DTD++ 24

3 Uitdrukkingskracht van XML schematalen **26**
3.1 XML documenten en bomen 26
3.2 XML schema's en boomtalen 27
3.3 Reguliere boomtalen 28
3.4 Lokale boomtalen 29
3.5 Single-type boomtalen 33
3.6 Overzicht 34

4 DTD Extra **36**
4.1 Oorsprong 36
4.2 Definitie en uitdrukkingskracht 38
4.3 Syntax 41

4.4 Semantiek 43
4.5 Voorbeelden 44

5 Methodes en algoritmes 49
5.1 Algemeen 49
5.2 Parsen 49
5.3 Evaluatie 52
5.4 Transformatie 55

6 Implementatie 58
6.1 Algemeen 58
6.2 Platform 58
6.3 Intern datamodel 60
6.4 DTDX Parser 60
6.5 DTDX Evaluator 65
6.6 DTDX Transformator 73

7 Resultaten 79
7.1 Correcte werking 79
7.2 Testdocumenten 79
7.3 Testen 81
7.4 Performantie 82
7.5 Leesbaarheid 82

8 Besluit 91

Bibliografie 93

Lijst van figuren

2.1 XML document 3
2.2 DTD 6
2.3 Externe Document Type Declaration 7
2.4 Interne Document Type Declaration 7
2.5 XML Schema Definition voor bibliotheken 9
2.6 XSD voor bibliotheken in een XML Namespace 11
2.7 XSD voor personen in een XML Namespace 12
2.8 XSD voor bibliotheken met geïmporteerde XML Namespace . 13
2.9 XML Schema ingebouwde datatypes 14
2.10 Element group en all container 16
2.11 Substitution group 17
2.12 RELAX NG 18
2.13 Schematron 20
2.14 SchemaPath 23
2.15 DTD++ 25

3.1 Boomvoorstelling XML document 26
3.2 Vereenvoudigde boomvoorstelling XML document 27
3.3 Een reguliere boomgrammatica 28
3.4 EDTD 29
3.5 Boom over Σ voor de EDTD in Figuur 3.4 30
3.6 Boom over Δ voor de EDTD in Figuur 3.4 30
3.7 Een reguliere boomgrammatica met competing non-terminals 31
3.8 Lokale boomgrammatica 31
3.9 DTD 32
3.10 Single-type EDTD 33
3.11 Element Declarations Consistent 34
3.12 Verhouding in uitdrukkingskracht van boomtalen 35
3.13 Een overzicht van de klassen van boomtalen, het formalisme waarin ze worden uitgedrukt en de overeenkomstige XML schemataal 35

4.1 Overzicht van de in de praktijk aangewende XML Schema eigenschappen . . . 37
4.2 Patroon-gebaseerd schema . . . 38
4.3 Voorouder-gebaseerd schema . . . 40
4.4 DTDX schema voor dvd-winkel . . . 45
4.5 XML document voor het DTDX schema in Figuur 4.4 . . . 46
4.6 DTDX schema met element `c` op even diepte . . . 46
4.7 XML document met element `c` op even diepte . . . 47
4.8 Foutief DTDX schema met element `c` op even diepte . . . 47
4.9 Foutief XML document met element `c` op even diepte . . . 48

5.1 Contextvrije grammatica . . . 50
5.2 Afleiding voor contextvrije grammatica . . . 50
5.3 Afleidingsboom voor contextvrije grammatica . . . 50
5.4 Boomvoorstelling van een XML document dat voldoet aan het schema in Figuur 4.3 . . . 53
5.5 DFA voor ancestor-string $\Sigma^*.store$ . . . 56
5.6 DFA voor ancestor-string $\Sigma^*.regulars$ . . . 56
5.7 DFA voor ancestor-string $\Sigma^*.discounts$ . . . 56
5.8 DFA voor ancestor-string $\Sigma^*.regulars.dvd$ . . . 57
5.9 DFA voor ancestor-string $\Sigma^*.discounts.dvd$ . . . 57
5.10 Single-type EDTD voor het voorbeeld in Figuur 4.3 . . . 57

6.1 Java SAX parsing API . . . 66
6.2 Java DOM parsing API . . . 67
6.3 JAXP SAX packages . . . 68
6.4 JAXP DOM packages . . . 68
6.5 Transformatie van Figuur 4.4 naar XML Schema . . . 74
6.6 Transformatie van Figuur 4.4 naar RELAX NG . . . 75

7.1 DTDX schema met grootouders . . . 80
7.2 Recursief DTDX schema met ouders . . . 80
7.3 Recursief DTDX schema met grootouders . . . 80
7.4 Recursief DTDX schema met even en oneven diepte . . . 81
7.5 Recursief DTDX schema met voorouders . . . 81
7.6 Het aantal toestanden die tijdens de transformatie worden berekend . . . 83
7.7 Het aantal transities die tijdens de transformatie worden berekend . . . 83
7.8 XSD met grootouders . . . 84
7.9 Recursieve XSD met ouders . . . 85
7.10 Recursieve XSD met grootouders . . . 87
7.11 Recursieve XSD met even en oneven diepte . . . 89
7.12 Recursieve XSD met voorouders . . . 89

Hoofdstuk 1

Inleiding

1.1 Doelstelling

Het onderwerp van deze thesis situeert zich in de uitgebreide wereld van XML. Bij de bekendmaking van de thesisvoorstellen werd het onderwerp als volgt omschreven:

> "Uit een recente studie blijkt dat zeer weinig van de geavanceerde features van XML Schema in de praktijk gebruikt worden. De meeste schema's die in de praktijk voorkomen, kunnen eigenlijk gemakkelijk als een extensie van DTDs gedefinieerd worden. Het doel van deze thesis is op basis van het abstracte voorstel in de hogergenoemde paper en concrete voorstellen zoals DTD++, een concrete schemataal gebaseerd op DTDs te definiëren die gemakkelijk is in gebruik en die de meest voorkomende schema's aankan. Voorts dient een parser voor deze taal en een module geschreven te worden die schema's in deze taal omzet naar XML Schema. De omzetting naar XML Schema gebruikt technieken uit de theoretische informatica zoals vertalen van reguliere expressies naar automaten."

Na de toekenning van de thesis werden de details bekend gemaakt. De eerste vereiste was het opstellen van de syntax en het bepalen van de uitdrukkingskracht. Naast een parser was er ook nood aan een evaluator om XML documenten te kunnen controleren. Om het toepassingsgebied van de schemataal te vergroten werd er geopteerd om niet enkel een omzetting te voorzien naar XML Schema maar ook naar RELAX NG. Tijdens de realisatie van de thesis werden hier en daar nog enkele accenten verlegd.

1.2 Overzicht

De eigenlijke tekst begint in Hoofdstuk 2 en gaat van start met een beknopte inleiding tot XML. Vervolgens wordt beschreven wat een XML schemataal is en welk nut een schemataal heeft. De resterende secties in het hoofdstuk geven een overzicht van de bestaande schematalen.

Hoofdstuk 3 gaat na welke uitdrukkingskracht de besproken XML schematalen bezitten. Uit de relatie tussen XML documenten en bomen wordt afgeleid hoe een XML schema zich verhoudt tot een boomtaal. Tenslotte wordt voor RELAX NG, XML Schema en DTDs de equivalente klasse van boomtalen bepaald.

De concrete schemataal wordt toegelicht in Hoofdstuk 4. Nadat de oorsprong van de schemataal in detail is beschreven, worden de nodige definities en de uitdrukkingskracht geformuleerd. Een contextvrije grammatica definieert de syntax van de schemataal waarvan de semantiek wordt gespecificeerd in de volgende sectie. Het hoofdstuk wordt beëindigd met enkele voorbeelden.

Een schemataal kan slechts in de praktijk worden toegepast wanneer er tools beschikbaar zijn. Hoofdstuk 5 beschrijft op formele wijze enkele methodes en algoritmes die bij het ontwikkelen van dergelijke tools kunnen worden gebruikt.

De implementatie komt aan bod in Hoofdstuk 6. Allereerst wordt uitgelegd welk platform er gebruikt is en waarom. Er wordt vervolgd met een beschrijving van het interne datamodel dat wordt gehanteerd om een schema te representeren. Het overige gedeelte van het hoofdstuk bevat een sectie voor elke tool die wordt ontwikkeld, nl. een parser, een evaluator en een transformator. Binnen elke sectie wordt er aandacht besteed aan de gebruikte hulpmiddelen en de broncode. De secties worden afgesloten met een voorbeeld dat het gebruik van de tool illustreert.

Hoofdstuk 7 beschrijft de testresultaten van de tools. Er wordt aandacht besteed aan de gebruikte methodes en de opgestelde testdocumenten. Daarnaast wordt nagegaan wat de efficiëntie van het transformatieproces is en hoe de leesbaarheid van getransformeerde schema's kan worden verbeterd.

Het laatste hoofdstuk, Hoofdstuk 8, recapituleert de belangrijkste elementen uit de tekst en formuleert enkele bemerkingen over de ontwikkelde schemataal.

Hoofdstuk 2

XML en XML schematalen

2.1 Wat is XML?

XML is de afkorting van eXtensible Markup Language en werd in 1998 als een W3C Recommendation gepubliceerd [BPSM+06, Wik06b]. Het is een tekstuele opmaaktaal die gebruikt wordt om allerhande data op een gestructureerde manier te beschrijven. Een dergelijke beschrijving wordt een XML document genoemd en kan zowel door mensen als machines worden gelezen.

```
<?xml version="1.0" encoding="UTF-8"?>
<team>
  <player number="8">
    <first>Dwyane</first>
    <last>Wade</last>
    <position>Guard</position>
  </player>
  <player number="32">
    <first>Shaquille</first>
    <last>O'Neal</last>
    <position>Center</position>
  </player>
</team>
```

Figuur 2.1: XML document

Een XML document is opgebouwd uit tekst, elementen en attributen. De data wordt uitgedrukt in tekst die met behulp van elementen in de gewenste structuur wordt verwerkt. Attributen worden toegekend aan een element en bevatten bijkomende informatie over dat element. De namen van de elementen en attributen zijn niet vastgelegd in de W3C Recommendation, de keuze ervan wordt overgelaten aan de gebruiker. Op die manier is het

mogelijk de semantiek die hoort bij het XML document (gedeeltelijk) op te nemen in het document zelf.

Beschouw het XML document in Figuur 2.1 als voorbeeld. Dit document bevat informatie over een basketbalploeg. De ploeg wordt aangeduid met het element `team` en bestaat uit een aantal spelers die worden beschreven door het element `player`. Naast de elementen `first`, `last` en `position` die respectievelijk de voor- en achternaam en de positie van de speler weergeven, bezit `player` het attribuut `number` dat bijkomende informatie bevat in de vorm van het rugnummer van de speler.

De syntax van een XML document wordt gedefinieerd aan de hand van een aantal basisregels:

- het document bevat:
  - een optionele XML declaratie die de gebruikte versie en het karaktertype bepaald;
  - juist één boomstructuur die uit één of meer elementen bestaat waarvan er precies één als wortel fungeert;
  - geen of meer regels die commentaar, processing instructions of spatiëring bevatten.
- elk element bevat zowel een start als een einde, lege elementen kunnen gebruik maken van een verkorte notatie;
- de elementen worden op een correcte manier in elkaar genest en er is geen overlap tussen elementen;
- de waarden van attributen worden genoteerd tussen aanhalingstekens.

Elk document dat voldoet aan de hierboven vermelde regels wordt beschouwd als een goed gevormd XML document. Naast deze basisregels beschikt de gebruiker over de mogelijkheid bijkomende eisen te stellen aan de structuur en de inhoud van het XML document. Deze bijkomende eisen worden opgenomen in een schema en vormen een beschrijving van het type van het XML document. Er zijn heel wat talen in omloop waarin een dergelijk schema kan worden uitgedrukt, de zogenaamde schematalen.

2.2 XML schematalen

Een XML schemataal stelt de gebruiker in staat om, naast de basisregels betreffende syntax, bijkomende beperkingen op te leggen aan een XML document [MSCV04]. Deze beperkingen kunnen betrekking hebben op zowel de structuur als de inhoud van het document. In de praktijk zijn er heel wat toepassingen aanwezig waarin dit nuttig kan zijn. Denk hierbij aan:

- een omgeving waarin voortdurend gegevens worden uitgewisseld tussen verschillende processen. Elk proces moet er zeker van kunnen zijn dat het te behandelen XML document voldoet aan de afgesproken standaarden voor structuur en inhoud;

- een toepassing die gebruik maakt van gegevens afkomstig uit externe bronnen. Elk van die bronnen stelt zijn data via XML documenten ter beschikking. Het is noodzakelijk dat al deze documenten op identieke wijze kunnen worden behandeld.

Zoals blijkt uit de voorbeelden is het niet voldoende om een schema op te stellen om een bepaald type van XML documenten te specificeren. Er moet ook worden nagegaan of de documenten voldoen aan de regels van het schema. Dit gebeurt met behulp van een validatieproces waarin wordt gecontroleerd of het XML document in alle opzichten voldoet aan de regels van het vermelde XML schema.

In het algemeen worden XML schematalen opgesplitst in twee groepen. Enerzijds zijn er de grammatica-gebaseerde schematalen zoals DTDs en XML Schema. Anderzijds zijn er de regel-gebaseerde schematalen waarvan Schematron het grote boegbeeld is.

De grammatica-gebaseerde schematalen maken gebruik van een top-down werkwijze. Voor elk element en attribuut dat kan voorkomen in een XML document worden er regels gespecificeerd. De regels die betrekking hebben op de inhoud worden beschouwd als type-declaraties van de betreffende objecten. De wijze waarop de validatie van deze schematalen wordt afgehandeld, lijkt sterk op de manier waarmee de invoer van een automaat uit de automatentheorie wordt gevalideerd.

Regel-gebaseerde schematalen hanteren een totaal andere methodologie. De regels waaraan een XML document moet voldoen, worden niet per object ingedeeld maar gegroepeerd in een lijst. Deze lijst kan op twee manieren worden benaderd. Ofwel is de lijst open en wordt alles aanvaard dat niet verboden is, ofwel is de lijst gesloten en wordt alles verworpen dat niet is toegelaten.

2.3 DTDs

Document Type Definitions (DTDs) werden gelijktijdig met XML geïntroduceerd in de reeds vermelde W3C Recommendation [BPSM$^+$06, Wik06a]. Het betreft hier echter geen nieuwe schemataal. DTDs hebben reeds een lange voorgeschiedenis.

In 1986 werd Standard Generalized Markup Language (SGML) aanvaard als ISO-standaard [Wik06f]. SGML is een metataal die, net als XML, gebruikt wordt om de hiërarchische structuur van documenten te specificeren. Om het een en ander te vereenvoudigen werden DTDs toen reeds in het leven geroepen om de structuur van SGML documenten te beschrijven. De syntax die ontwikkeld werd om DTDs in uit te drukken is dan ook gebaseerd op de syntax van SGML. Het was onmogelijk om een XML-gebaseerde syntax te definiëren aangezien XML pas een tiental jaren later werd ontwikkeld.

De versie van DTDs die gehanteerd wordt om XML documenten te beschrijven, is grotendeels gelijk aan de versie voor SGML documenten. Niettemin zijn er toch enkele kleine verschillen tussen beide versies. Zo is het mogelijk om in SGML DTDs te specificeren of de sluitende tag van een element al dan niet verplicht is of om bepaalde elementen te weren als kind van een ander element. Een andere eigenschap die niet werd overgenomen is het gebruik van de `&` operator (analoog aan de container `ALL` in XML Schema). Deze operator laat toe dat, wanneer het inhoudsmodel van een element enkel uit andere elementen bestaat, die elementen in willekeurige volgorde mogen voorkomen.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT team (player)+>
<!ELEMENT player (first, last, position)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT position (#PCDATA)>
<!ATTLIST player number CDATA #REQUIRED>
```

Figuur 2.2: DTD

Figuur 2.2 toont een mogelijk schema voor het XML document in Figuur 2.1. Het type van een XML document wordt in een DTD beschreven aan de hand van element- en attribuutdeclaraties.

Een elementdeclaratie bepaalt welke elementen er mogen voorkomen in een document en wat de onderliggende structuur van die elementen is. Dit kunnen andere elementen zijn, tekst of de combinatie van deze twee. Wanneer er tekst mag voorkomen in een element wordt dit aangeduid met het type `#PCDATA`.

Een attribuutdeclaratie wordt geassocieerd met een specifiek element en beschrijft welke bijkomende informatie er over dat element beschikbaar is. Deze informatie is een tekstwaarde die door drie soorten types kan worden beschreven. Allereerst is er het algemene stringtype dat eender welke waarde omvat en wordt gespecificeerd met `CDATA`. Vervolgens zijn er de token-

types die bijkomende beperkingen opleggen aan de tekstwaarde: `ID`, `IDREF`, `ENTITY` en `NMTOKEN`. Tenslotte zijn er de enumeratietypes die de tekstwaarde beschrijven als een lijst van toegelaten waarden. Elke waarde in de lijst voldoet aan de eisen van het betrokken tokentype. De verschillende enumeratietypes zijn `IDREFS`, `ENTITIES` en `NMTOKENS`.

```
<!DOCTYPE root-element SYSTEM "filename">
```

Figuur 2.3: Externe Document Type Declaration

```
<!DOCTYPE root-element [element-declarations]>
```

Figuur 2.4: Interne Document Type Declaration

Een XML document wordt met behulp van een Document Type Declaration aan een DTD gekoppeld. De DTD bevindt zich meestal in een alleenstaand bestand hoewel dit niet noodzakelijk is. De declaraties kunnen ook rechtstreeks in het XML document worden opgenomen. In Figuur 2.3 en Figuur 2.4 wordt de syntax van zowel de externe als de interne declaratie getoond.

De grote kracht van DTDs is de eenvoudige syntax. Hierdoor is het mogelijk om beknopte schema's op te stellen die ook door menselijke gebruikers kunnen worden geïnterpreteerd. Bovendien zullen nieuwe gebruikers snel in staat zijn om bestaande schema's te begrijpen en zelf nieuwe schema's te ontwerpen.

Naarmate het gebruik van XML evolueerde kwamen de beperkingen van DTDs aan het licht. Zo bleek het beperkte aantal voorgedefinieerde datatypes niet meer toereikend te zijn voor hedendaagse toepassingen. Een ander heikelpunt is de niet-XML-gebaseerde syntax die via overerving van SGML werd verkregen. Daarnaast bieden DTDs geen ondersteuning voor het gebruik van XML Namespaces, een gegeven dat zich in het kader van herbruikbaarheid als een noodzaak heeft geprofileerd. Het is eveneens niet mogelijk om beroep te doen op het `ALL` of `ANY` inhoudsmodel. De genoemde tekortkomingen zijn de belangrijkste maar slechts een greep uit de vele die werden ontdekt.

2.4 XML Schema

Om tegemoet te komen aan de beperkingen van DTDs werden er heel wat nieuwe XML schematalen ontwikkeld. Eén hiervan is XML Schema dat in 2001 als W3C Recommendation werd gepubliceerd [vdV02, Wik06g]. Men

streefde er naar een schemataal te ontwikkelen die vereiste dat de schema's werden opgesteld volgens de XML syntax. Daarnaast moest het mogelijk zijn dat de structuur en de inhoud van elementen preciezer konden worden beschreven en dat het gebruik van XML Namespaces werd ondersteund. Als gevolg van deze eisen werd de specificatie van XML Schema opgesplitst in drie onderdelen, nl. XML Schema Part 0: Primer [FW04], XML Schema Part 1: Structures [TBMM04] en XML Schema Part 2: Datatypes [BM04].

In XML Schema Part 0 wordt een overzicht gegeven van de diverse mogelijkheden die XML Schema biedt. Aan de hand van enkele voorbeelden verwerft de lezer inzicht in de wijze waarop ze kunnen worden gebruikt bij het opstellen van schema's. Part 1 en 2 zijn meer gericht op de ontwikkelaars van toepassingen en geven een formele en gedetailleerde beschrijving van de mogelijkheden.

Een schema dat is opgesteld m.b.v. XML Schema wordt een XML Schema Definition (XSD) genoemd. Figuur 2.5 toont een correct opgestelde XSD om informatie betreffende een bibliotheek te beschrijven.

Het datamodel dat door XML Schema gehanteerd wordt, bestaat uit drie bouwstenen:

- de namen van elementen en attributen;
- de inhoudsmodellen en types van elementen;
- de datatypes.

Na het validatieproces volgens een XSD kan het XML document worden uitgedrukt in termen van dit datamodel. De beschikbare informatie vormt de Post-Schema-Validation Infoset (PSVI). Het bevat het type van het XML document en laat toe om het document als een object te behandelen zoals dit gebeurt in object-georiënteerde programmeertalen.

XML Schema heeft als belangrijkste troef dat het toelaat dat zowat elk mogelijk type van XML documenten kan worden gespecificeerd door ondersteuning te bieden aan een enorme waaier van beperkingen, datatypes en structuren. Deze troef heeft echter ook een negatieve kant. Heel wat gebruikers ervaren XML Schema als een complexe en dubbelzinnige schemataal. Zo kan een subtiel verschil in formulering een grote impact hebben op de specificatie van een documenttype. Dit betekent dat er zeer zorgvuldig moet worden omgegaan met de definitie van de diverse onderdelen van het op te stellen schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" type="bookType" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="bookType">
    <xs:sequence>
      <xs:element name="isbn" type="xs:NMTOKEN" />
      <xs:element name="title" type="xs:string" />
      <xs:element name="authors">
        <xs:complextype>
          <xs:sequence>
            <xs:element ref="person" maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="characters">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="person" maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    <xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required" />
    <xs:attribute name="available"  type="xs:string"
        use="required" />
  </xs:complexType>
  ...
</xs:schema>
```

Figuur 2.5: XML Schema Definition voor bibliotheken

Namen van elementen en attributen

Afhankelijk van de gebruikte relaties en structuren zijn de namen bekend in het hele schema of slechts delen ervan. Uiteraard is het ook mogelijk om bestaande onderdelen van één schema te hergebruiken als onderdeel van een ander schema zodat heuse bibliotheken kunnen worden opgesteld. Hierbij wordt een onderscheid gemaakt tussen *schema inclusion* en *schema inclusion with redefinition*. De eerste methode maakt onderdelen beschikbaar zonder dat er wijzigingen kunnen worden aangebracht. De tweede methode laat dit wel toe. De onderdelen die expliciet opnieuw worden gedefinieerd zijn beschikbaar in de nieuwe vorm terwijl de overige onderdelen impliciet in hun origenele vorm ter beschikking worden gesteld.

XML Schema biedt ondersteuning voor het gebruik van XML Namespaces waardoor hergebruik van bestaande schema's tot een hoger niveau wordt getild. Een schema en de onderdelen van dat schema worden opgenomen in een bepaalde *namespace* zoals in Figuur 2.6 wordt weergegeven. Door alle schema's die deel uitmaken van een bibliotheek op te nemen in dezelfde *namespace* kunnen er bijvoorbeeld bibliotheken per vakdomein worden opgesteld.

Wanneer er onderdelen worden hergebruikt uit een schema dat tot een bepaalde *namespace* behoort, gebeurt dit rekeninghoudend met de *namespace*. Enerzijds wordt via de *namespace* duidelijk tot welk domein een element behoort, anderzijds kunnen elementen, die gedeeld worden door verschillende domeinen, voor elk domein een domeinspecifieke definitie krijgen en toch dezelfde naam dragen.

Figuur 2.7 beschrijft een minimaal schema om personen voor te stellen. Het schema behoort tot een namespace en maakt geen gebruik van andere namespaces. Dit schema kan zonder problemen worden geïmporteerd in schema's van andere namespaces die gebruik willen maken van de definities. Zo kan het schema in Figuur 2.6 worden aangepast tot het schema in Figuur 2.8.

Inhoudsmodellen en types van elementen

In de inhoudsmodellen van elementen wordt een onderscheid gemaakt tussen een leeg (geen tekst en kindelementen), een eenvoudig (enkel tekst), een complex (enkel kindelementen) en een gemengd (zowel tekst als kindelementen) inhoudsmodel. Het type van een element wordt bepaald door het inhoudsmodel en het al dan niet aanwezig zijn van attributen in het element. Een element met een eenvoudig inhoudsmodel zonder attributen wordt beschouwd als een eenvoudig type terwijl alle andere combinaties van het complexe type zijn.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://dyomeda.com/ns/library"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    xmlns:lib="http://dyomeda.com/ns/library"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" type="lib:bookType" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="bookType">
    <xs:sequence>
      <xs:element name="isbn" type="xs:NMTOKEN" />
      <xs:element name="title" type="xs:string" />
      <xs:element name="authors">
        <xs:complextype>
          <xs:sequence>
            <xs:element ref="lib:person"
                maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="characters">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="lib:person"
                maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    <xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required" />
    <xs:attribute name="available"  type="xs:string"
        use="required" />
  </xs:complexType>
  ...
</xs:schema>
```

Figuur 2.6: XSD voor bibliotheken in een XML Namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://dyomedea.com/ns/people"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    xmlns:ppl="'http://dyomedea.com/ns/people"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="born" type="xs:date" />
        <xs:element name="dead" type="xs:date" minOccurs="0" />
        <xs:element name="qualification" type="xs:string"
            minOccurs="0" />
      </xs:sequence>
      <xs:attribute name="id" type="xs:ID" use="required" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figuur 2.7: XSD voor personen in een XML Namespace

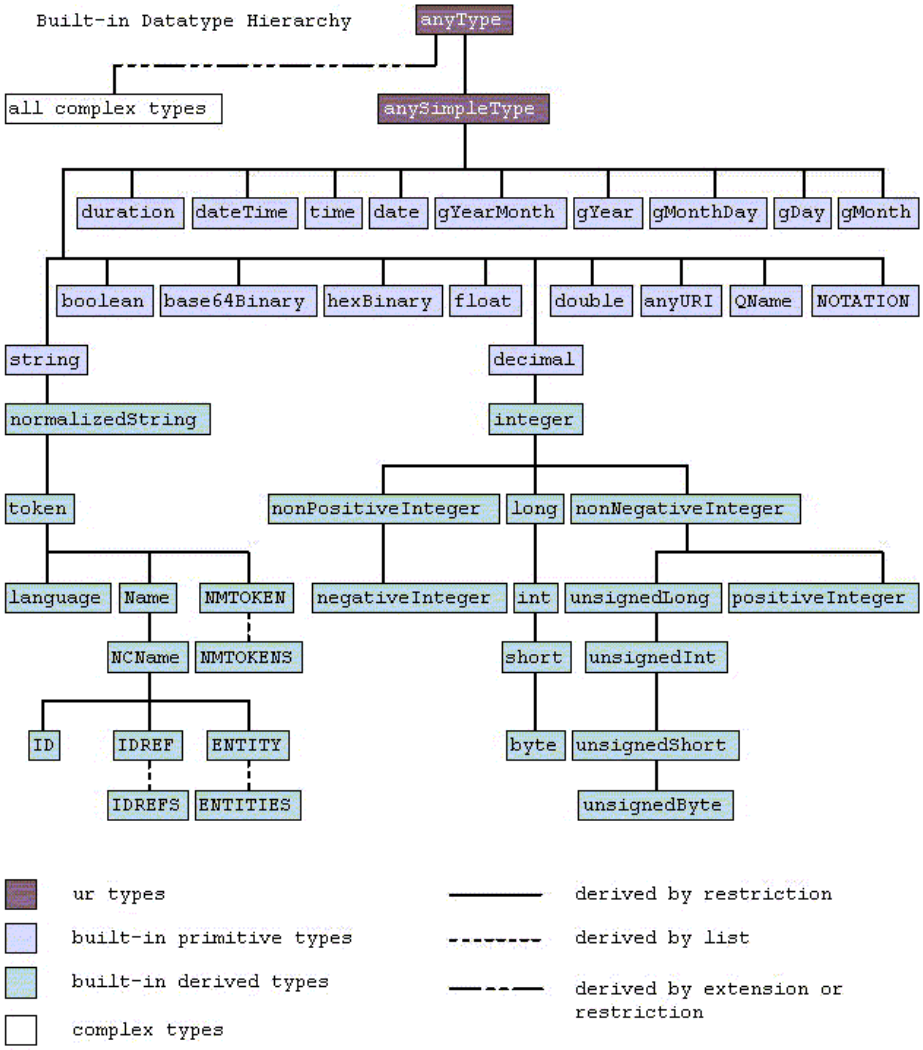
Datatypes

Naast een uitgebreide verzameling van ingebouwde datatypes (Figuur 2.9) kan de gebruiker zelf nieuwe datatypes opstellen. Een eenvoudig datatype kan enkel bekomen worden via afleiding van een ingebouwd of een reeds bestaand datatype. Deze afleiding kan op drie manieren gebeuren. Allereerst is er de beperkende afleiding waarbij het afgeleide datatype ontstaat door bijkomende beperkingen op te leggen aan het af te leiden datatype. Een tweede manier is afleiding via lijst. Dit houdt in dat het nieuwe datatype een lijst van waarden is waarvan elke waarde afzonderlijk voldoet aan de eisen van het af te leiden datatype. De derde manier tenslotte is de afleiding door unie. Hierbij worden de domeinen van meerdere datatypes samengevoegd tot het domein van het afgeleide datatype.

Om een complex datatype te kunnen bekomen moet er eerst een complex datatype bestaan. De creatie van een complex datatype is dan ook de eerste methode die voorhanden is. Daarnaast is er de methode van afleiding die kan worden opgesplitst in twee vormen. De eerste vorm is de beperkende afleiding die analoog is aan de beperkende afleiding van eenvoudige datatypes. De tweede vorm is afleiding door uitbreiding waarbij er nieuwe elementen en/of attributen worden toegevoegd aan het inhoudsmodel van een datatype.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://dyomedea.com/ns/library"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:ppl="http://dyomedea.com/ns/people"
    xmlns:lib="http://dyomedea.com/ns/library">
  <xs:import namespace="http://dyomedea.com/ns/people"
      schemaLocation="simple-2-ns-ppl.xsd" />
  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" type="lib:bookType" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="bookType">
    <xs:sequence>
      <xs:element name="isbn" type="xs:NMTOKEN" />
      <xs:element name="title" type="xs:string" />
      <xs:element name="authors">
        <xs:complextype>
          <xs:sequence>
            <xs:element ref="ppl:person" maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="characters">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="ppl:person" maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    <xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required" />
    <xs:attribute name="available"  type="xs:string"
        use="required" />
  </xs:complexType>
  ...
</xs:schema>
```

Figuur 2.8: XSD voor bibliotheken met geïmporteerde XML Namespace



Figuur 2.9: XML Schema ingebouwde datatypes

Wanneer er complexe datatypes zijn die een gemeenschappelijke groep elementen of attributen hebben in hun inhoudsmodel kan er m.b.v. van een *element* of *attribute group* een herbruikbaar inhoudsmodel worden opgesteld dat kan worden opgenomen in de betrokken datatypes. Het gebruik van deze werkwijze vergt doorgaans geen bijkomende aandacht tenzij er ongeordende inhoudsmodellen, aangeduid met de container *all*, aanwezig zijn. In dat geval is het niet langer mogelijk het aantal voorkomens van een betrokken element of attribuut in te stellen op een waarde groter dan één.

Het schema in Figuur 2.10 definieert twee elementen, `author` en `character`, die beiden een element bevatten om een naam te beschrijven. De twee elementen hanteren een verschillende definitie voor een naam. Voor elke definitie wordt een aangepast element opgesteld, `full-name` en `simple-name`. Deze elementen worden m.b.v. *choice* opgenomen in de *element group* `name`. Nu is het mogelijk om `name` te hergebruiken waar nodig en dit met de definitie naar keuze.

In sommige gevallen kan het interessant zijn om een bepaald type te vervangen door een afgeleid type dat toelaat meer specifieke informatie m.b.t. de context weer te geven. Deze functionaliteit wordt geboden door *substitution groups*. Om te beginnen wordt het hoofd van de *substitution group* gedefinieerd. Dit is het element dat in een XML document kan worden vervangen door een element dat ervan is afgeleid. Het datatype van het hoofd dient zo gekozen te worden dat de inhoudsmodellen en types van de afgeleide elementen geldig zijn t.o.v. dit datatype. Als het hoofd nooit mag voorkomen in een XML document kan dit worden aangegeven door het attribuut `abstract` de waarde `true` te geven. Vervolgens kunnen de afgeleide elementen worden gedefinieerd.

Figuur 2.11 toont een schema dat analoog is aan Figuur 2.10. Eerst wordt het element `name` gedefinieerd dat dienst zal doen als het hoofd van de *substitution group*. Dit element bevat het attribuut `abstract="true"` en is dus niet toegelaten in een XML document. Merk op dat het element eender welk inhoudsmodel toelaat. Vervolgens worden de afgeleide elementen `simple-name` en `full-name` gedefinieerd als lid van de *substitution group*. Deze definities zorgen ervoor dat, telkens `name` gebruikt wordt in een schema, dit in een document mag worden vervangen door `simple-name` of `full-name`.

2.5 RELAX NG

RELAX NG is de afkorting van REgular LAnguage for XML Next Generation [vdV04, Wik06d, CM01, Cla01]. Het is een relatief eenvoudige schemataal die gebaseerd is op RELAX van Murata Makoto en TREX van James Clark.

```
<xs:element name="full-name">
  <xs:complexType>
    <xs:all>
      <xs:element name="first" type="string" minOccurs="0" />
      <xs:element name="middle" type="string" minOccurs="0" />
      <xs:element name="last" type="string" minOccurs="0" />
    </xs:all>
  </xs:complexType>
</xs:element>

<xs:element name="simple-name" type="string" />

<xs:group name="name">
  <xs:choice>
    <xs:element ref="simple-name" />
    <xs:element ref="fill-name" />
  </xs:choice>
</xs:group>

<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="name" />
      <xs:element ref="born" />
      <xs:element ref="dead" minOccurs="0" />
    </xs:sequence>
    <xs:attribute ref="id" />
  </xs:complexType>
</xs:element>

<xs:element name="character">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="name" />
      <xs:element ref="born" />
      <xs:element ref="qualification" />
    </xs:sequence>
    <xs:attribute ref="id" />
  </xs:complexType>
</xs:element>
```

Figuur 2.10: Element group en all container

```
<xs:element name="name" abstract="true" />

<xs:element name="simple-name" type="string"
    substitutionGroup="name" />

<xs:element name="full-name" substitutionGroup="name">
  <xs:complexType>
    <xs:all>
      <xs:element name="first" type="string" minOccurs="0" />
      <xs:element name="middle" type="string" minOccurs="0" />
      <xs:element name="last" type="string" minOccurs="0" />
    </xs:all>
  </xs:complexType>
</xs:element>

<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name" />
      <xs:element ref="born" />
      <xs:element ref="dead" minOccurs="0" />
    </xs:sequence>
    <xs:attribute ref="id" />
  </xs:complexType>
</xs:element>

<xs:element name="character">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name" />
      <xs:element ref="born" />
      <xs:element ref="qualification" />
    </xs:sequence>
    <xs:attribute ref="id" />
  </xs:complexType>
</xs:element>
```

Figuur 2.11: Substitution group

RELAX NG is een XML-gebaseerde technologie. Dit houdt in dat een schema in feite zelf een XML document (schemadocument) is dat gebruikt wordt om andere XML documenten (instantiedocumenten) te valideren. Er bestaat ook een alternatieve, meer compacte notatie die niet gebaseerd is op XML. Beide notaties maken gebruik van de infoset als hoger abstractieniveau. Het XML document wordt op een logische wijze ingedeeld in plaats van het document in zijn tekstuele vorm te verwerken.

In tegenstelling tot DTDs en XML Schema beperkt RELAX NG zich slechts tot het controleren van de structuur van toegelaten elementen, attributen en tekst. Om de controle zo goed mogelijk te verwezenlijken maakt RELAX NG gebruik van patronen om deze structuren te testen. De elementen in het instantiedocument worden vergeleken met de patronen in het schemadocument om na te gaan of ze voldoen aan de gewenste XML structuur.

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="doc">
      <oneOrMore>
        <element name="invoiceLine">
          <group>
            <element name="unit">
              <choice>
               <value>items</value>
               <value>meters</value>
              </choice>
            </element>
            <element name="quantity">
              <text/>
            </element>
          </group>
        </element>
      </oneOrMore>
    </element>
  </start>
</grammar>
```

Figuur 2.12: RELAX NG

Patronen vormen de kern van een RELAX NG schema. Er worden drie basispatronen ter beschikking gesteld, nl. elementen, attributen en tekst. Deze patronen vormen de herbruikbare bouwstenen van RELAX NG. Hiertoe werden volgende functies opgenomen:

- toekennen van een naam aan een patroon;

- hergebruik van een patroon;
- opnieuw definiëren van een patroon;
- groeperen van patronen of keuze tussen patronen door middel van operatoren;
- aangeven van de cardinaliteit van een patroon;
- een patroon als optioneel specificeren.

Het gebruik van patronen in RELAX NG leidt tot twee belangrijke voordelen. Enerzijds is RELAX NG op degelijke wijze wiskundig onderbouwd. Hierdoor onstaat er een zeer goede stabiliteit terwijl ambiguïteit tot een minimum wordt beperkt. Bovendien kunnen bestaande wiskundige formalismen worden hergebruikt. Anderzijds leiden de mogelijkheden van patronen tot een enorme uitdrukkingskracht die toelaat om bijna elke mogelijke XML structuur te beschrijven.

Doordat RELAX NG enkel controle uitvoert op de structuur van een XML document is het aantal ingebouwde datatypes zeer beperkt gebleven. Hiervoor werd een vrij elegante oplossing voorgesteld. Door toe te laten dat er verbindingen kunnen worden gelegd naar zogenaamde datatypebibliotheken kan de validatie van de inhoud van het document worden afgehandeld door die bibliotheken.

2.6 Schematron

Als boegbeeld van de regel-gebaseerde schematalen is Schematron een onderdeel van de Document Schema Definition Languages (DSDL) [Jel06, fS06]. DSDL streeft naar een uitbreidbaar framework van validatie-gerelateerde taken en uitdrukkingen waarin verschillende technologieën samenwerken om één of een verzameling van validatieresultaten te genereren. Hierbij worden ook technologieën vermeld die nog niet ontwikkeld zijn of waarvoor er geen specificatie bestaat.

Een schema opgesteld in Schematron bevat verklaringen in natuurlijke taal die worden uitgedrukt met behulp van de elementen en attributen uit de grammatica van Schematron. Deze grammatica is te bereiken via een namespace URI (Uniform Resource Identifier) en wordt meestal aangeduid met de prefix `sch`. De verklaringen in een schema maken gebruik van de eerder vermelde open specificatie.

In Schematron bestaat een schema uit een verzameling van fasen. Elke fase is een verzameling van patronen en een patroon kan tot meerdere fasen behoren. Een XML document kan dus, afhankelijk van de fase waarin het

document zich bevindt, onderworpen zijn aan andere vereisten. Een patroon is op zijn beurt een verzameling van regels. Elke regel bestaat uit een context en een verzameling beweringen. De context geeft aan wanneer een regel van toepassing is en wordt uitgedrukt met behulp van XPath. Een bewering is een booleaanse functie die wordt geëvalueerd en waarbij er een onderscheid wordt gemaakt tussen positieve (*assert*) en negatieve (*report*) beweringen.

```
<sch:schema xmlns:sch="http://www.ascc.net/xml/schematron">
  <sch:pattern>
    <sch:rule context="/*">
      <sch:assert test="name()='doc'">
        Root element is not doc.
      </sch:assert>
    </sch:rule>
  </sch:pattern>
  <sch:pattern name="test integer">
    <sch:rule context="quantity[../unit='items']">
      <sch:assert test="floor(.)=number(.)">
        Not an integer value.
      </sch:assert>
    </sch:rule>
  </sch:pattern>
  <sch:pattern name="test decimal">
    <sch:rule context="quantity[../unit='meters']">
      <sch:assert test="number(.)">
        Not a decimal value.
      </sch:assert>
    </sch:rule>
  </sch:pattern>
  <pattern name="items or meters">
    <rule context="unit">
      <assert test="normalize-space(.)='items'
            or normalize-space(.)='meters'">
        The specified value is not permitted.
      </assert>
    </rule>
  </pattern>
</sch:schema>
```

Figuur 2.13: Schematron

Het validatieproces van een XML document volgens een Schematron schema kan drie resultaten opleveren: *valid*, *invalid* of *error*. Het proces bestaat uit twee delen. Eerst wordt de syntax van het schema omgezet in een minimale

syntax. Dit gebeurt in vijf stappen:

- vervang alle *include* elementen door de bron;
- vervang in de abstracte patronen alle parameter referenties in aanwezige attributen die een query bevatten door eigenlijke waarden;
- vervang alle abstracte regels in het schema door de betrokken inhoud;
- zet alle *report* elementen om in *assert* elementen;
- verwijder alle elementen die voor diagnose of documentatie worden gebruikt.

In het tweede deel wordt de minimale syntax gebruikt om na te gaan of het gegeven XML document geldig is met betrekking tot het schema in de betreffende fase. Schematron definieert de geldigheid van een document als volgt:

> "Er bestaat een combinatie van een instantie, een schema en een fase zodat voor elke context, elk patroon, elke regel en elke bewering geldt dat, wanneer de context van een instantie overeenkomt met een regel, er geen andere regel in het patroon is waarmee deze context overeenkomt èn dat de evaluatie van de bewering een positief resultaat oplevert."

Schematron maakt het mogelijk om op eenvoudige wijze co-constraints op te stellen. Dit zijn beperkingen die aangeven of bepaalde elementen en/of attributen al dan niet samen mogen voorkomen in een XML document. Het is bijna onmogelijk om dit type van beperkingen uit te drukken in grammatica-gebaseerde schematalen zoals DTDs, XML Schema of RELAX NG. In tegenstelling tot deze schematalen is het in Schematron echter niet zo eenvoudig om beperkingen op te leggen aan de structuur en waardedomeinen van elementen en attributen.

2.7 SchemaPath

XML Schema biedt geen ondersteuning voor co-constraints. Wanneer een bepaald type van XML documenten hier toch nood aan heeft, worden de co-constraints doorgaans opgenomen in applicatie-specifieke modules. Het spreekt voor zich dat dit geen efficiënte oplossing is. Daarom werd SchemaPath in het leven geroepen [MSCV04].

SchemaPath is een nieuwe schemataal. Het is in feite een uitbreiding van XML Schema, meerbepaald een conservatieve uitbreiding. Dit betekent dat elk correct XML Schema schema ook een correct SchemaPath schema is. In

de praktijk is het dus mogelijk om eerst een schema op te stellen in XML Schema en vervolgens specifieke SchemaPath syntax toe te voegen.

Om ondersteuning te kunnen bieden aan co-constraints werden conditionele declaraties, conditionele elementen en conditionele attributen geïntroduceerd. Een conditionele declaratie is een lijst van alternatieve typedefinities die elk worden geassocieerd met een XPath predicaat en een prioriteit. Een conditioneel element (attribuut) is geldig wanneer het type gelijk is aan het type met de hoogste prioriteit dat voldoet aan het XPath predicaat. Hiertoe werden er slechts één nieuwe constructie en één nieuw eenvoudig datatype toegevoegd aan de syntax van XML Schema:

- element `alt`: een alternatieve typedefinitie binnen een conditionele declaratie die semantisch equivalent is met een niet-conditionele typedefinitie. Met uitzondering van de attributen `cond` en `priority` bevat `alt` dezelfde attributen. Het attribuut `cond` specificeert de conditie waaraan het betreffende type moet voldoen. Deze conditie wordt uitgedrukt met behulp van een XPath predicaat. Met het attribuut `priority` wordt bepaald welke typedefinitie wordt toegepast wanneer een conditioneel element (attribuut) voldoet aan meerdere condities;
- datatype `error`: een eenvoudig type dat zowel aan elementen als attributen kan worden toegewezen. De toekennig van dit type resulteert in een fout tijdens validatie omdat het een leeg waardedomein heeft.

Aangezien SchemaPath een uitbreiding is van XML Schema, spreekt het voor zich dat ook hier een PSVI beschikbaar is. De inhoud van de PSVI wordt niet gewijzigd wanneer het een geldig document betreft. Het nieuwe element `alt` wordt immers enkel gebruikt om te bepalen welk type er geassocieerd moet worden met een conditioneel element en heeft na het validatieproces geen enkel nut meer. In het geval van een ongeldig document kan de inhoud van de PSVI wel wijzigen door de toekenning van het nieuwe datatype `error`.

Elke uitbreiding van XML Schema moet er voor zorgen dat de eigenschappen van XML Schema intact worden gehouden. Dit geldt uiteraard ook voor SchemaPath. Allereerst is er het *validation theorem*. Dat stelt dat de PSVI die wordt opgesteld tijdens het validatieproces een waarheidsgetrouwe representatie van het originele XML document en de type-afleiding is. Het bewijst dat het document correct getypeerd is volgens het schema. Ten tweede moeten *roundtripping* en *reverse-roundtripping* behouden blijven onder dezelfde voorwaarden als in XML Schema. Roundtripping betekent dat het serialiseren van een PSVI in XML en de daaropvolgende deserialisatie dezelfde PSVI oplevert en er geen verlies van data is. Reverse-roundtripping wil zeggen dat, na de validatie van een XML document gevolgd door de serialisatie van de PSVI, opnieuw het oorspronkelijke XML document wordt bekomen.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.cs.unibo.it/SchemaPath/1.0">
  <xsd:element name="doc">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="invoiceLine" type="invoiceLineType"
            maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="invoiceLineType">
    <xsd:sequence>
      <xsd:element name="unit" type="unitType"/>
      <xsd:element name="quantity">
        <xsd:alt cond="../unit='items'" type="xsd:integer"/>
        <xsd:alt cond="../unit='meters'" type="xsd:decimal"/>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="unitType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="items"/>
      <xsd:enumeration value="meters"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Figuur 2.14: SchemaPath

Het validatieproces in SchemaPath wordt afgehandeld door een pre-processor die gebaseerd is op een aantal XSLT stylesheets. Deze stylesheets genereren een afgeleid XML Schema schema en een afgeleid XML document die door een XML Schema validator worden verwerkt. In detail verloopt het proces als volgt. Gegeven een XML document X en een SchemaPath schema S, worden twee XSLT stylesheets T' en T" gebruikt die resulteren in respectievelijk een nieuw schema S' en een niew document X'. T' kan uniform worden toegepast voor eender welk SchemaPath schema in tegenstelling tot T" dat afhankelijk is van X. Dit wordt verholpen door gebruik te maken van een metastylesheet MT dat tijdens het validatieproces T" bepaalt op basis van S. Deze implementatie is vrij inefficiënt maar vormt een goede test voor de uitdrukkingskracht van SchemaPath.

2.8 DTD++

Wanneer er een nieuwe schemataal wordt ontworpen leidt dit meestal tot:

- een lange lijst van mogelijkheden;
- een syntax die gebaseerd is op XML;
- nieuwe en specifieke validatie-paradigma's die heel wat aandacht, leertijd en voorzichtigheid vergen van de gebruiker.

DTD++ werd in 2003 ontworpen als een eigen schemataal die de mogelijkheden bevat die noodzakelijk zijn voor hedendaagse XML-validatie maar met een beperking van nieuwe syntax en validatie-paradigma's [VGA03]. Om dit te bereiken is DTD++ ontworpen als een uitbreiding van DTDs. Dit houdt in dat elke geldige DTD een geldig DTD++ schema is. Omgekeerd geldt het zelfde. Een geldig DTD++ schema dat ontdaan wordt van specifieke mogelijkheden is een geldige DTD. Bovendien beschrijven overeenstemmende DTDs en DTD++ schema's dezelfde klasse van XML documenten.

De kracht van DTDs ligt in het goede evenwicht tussen uitdrukkingskracht en eenvoud. Ze bezitten een compacte syntax die enkel het noodzakelijke bevat en die bovendien gemakkelijk te lezen en te leren is. De syntactische uitbreidingen die nodig waren om DTD++ te ontwerpen werden dan ook tot een minimum beperkt om deze kracht zoveel mogelijk uit te buiten.

Het nadeel van DTDs is de beperkte verzameling van mogelijkheden. Deze werd zorgvuldig uitgebreid met heel wat mogelijkheden waarvan de meeste afkomstig zijn van XML Schema zoals XML namespaces, voorgedefinieerde datatypes, complexe types en diverse inhoudsmodellen. DTD++ is bijgevolg in grote mate semantisch equivalent aan XML Schema.

```
<!ELEMENT doc (invoiceLine)+>
<!ELEMENT invoiceLine (@invoiceLineType;)>
<!ENTITY @ invoiceLineType "(unit, quantity)">
<!ELEMENT unit (#unitType;)>
<!ELEMENT quantity "../unit='items'":5 (#INTEGER)>
<!ELEMENT quantity "../unit='meters'":5 (#DECIMAL)>
<!ENTITY # unitType "(#STRING(items|meters))">
```

Figuur 2.15: DTD++

Validatie volgens een DTD++ schema is gebaseerd op een pre-processor die het betreffende schema vertaalt naar het corresponderende XML Schema schema. Vervolgens worden het vertaalde schema en het XML document als invoer aangeboden aan een XML Schema validator. De foutmeldingen die gegenereerd worden door de validator worden door de pre-processor omgezet zodat ze verwijzen naar het oorspronkelijke DTD++ schema.

In 2004 werd DTD++ 2.0 ontworpen [FMGV04]. Het belangrijkste verschil t.o.v. de vorige versie is de ondersteuning van co-constraints. Om dit te verwezenlijken werd de filosofie van DTD++ 1.0 aangehouden, nl. zo weinig mogelijk wijzigingen aanbrengen in de syntax om bestaande constructies ten volle te kunnen benutten. De nieuwe syntactische elementen die werden toegevoegd kunnen op eenvoudige wijze worden omgezet naar de syntax van SchemaPath. DTD++ 2.0 werd zodanig ontworpen dat het semantisch equivalent is aan een significante subset van SchemaPath.

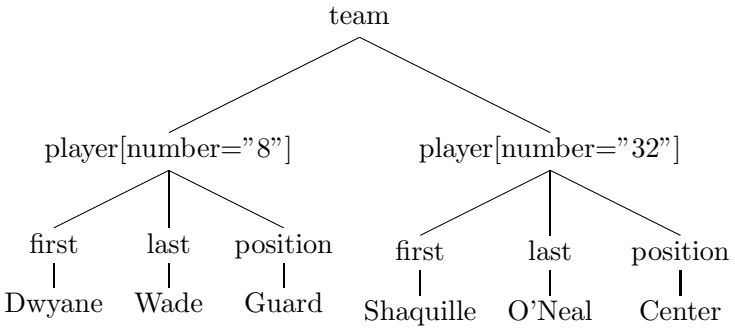
Net zoals bij DTD++ 1.0 wordt de validatie ook hier afgehandeld door een pre-processor. Eerst wordt het DTD++ 2.0 schema omgezet in het corresponderende SchemaPath schema dat op zijn beurt wordt vertaald naar het overeenstemmend XML Schema schema. Dit schema wordt uiteindelijk, samen met het XML document, naar een standaard XML Schema validator gestuurd. Door de semantische equivalentie van de schematalen is het document geldig m.b.t. het DTD++ 2.0 schema als en slechts als het document geldig is volgens het corresponderende XML Schema schema.

Hoofdstuk 3

Uitdrukkingskracht van XML schematalen

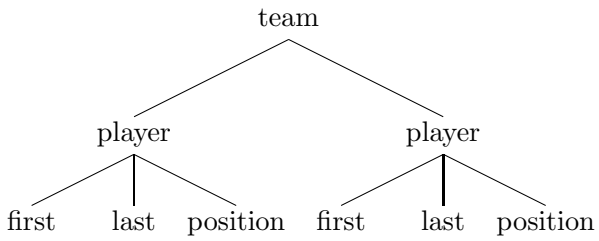
3.1 XML documenten en bomen

In Sectie 2.1 wordt uitgelegd dat een XML document de data op een gestructureerde manier beschrijft. De logische structuur die hier voor wordt aangewend zijn geordende bomen [BMNS05]. Het is bijgevolg mogelijk om een XML document voor te stellen d.m.v. een boomstructuur. De interne knopen van de boom stellen de elementen van het document voor. De eigenlijke gegevens worden weergegeven als de bladeren van de boom. Figuur 3.1 toont de boomstructuur die overeenkomt met het voorbeeld in Figuur 2.1.



Figuur 3.1: Boomvoorstelling XML document

De structuur van XML documenten wordt enkel bepaald door de elementen en de wijze waarop elementen in elkaar worden genest. Aangezien in het vervolg van deze tekst de uitdrukkingskracht van de documentstructuur wordt behandeld, kan de boomvoorstelling van een document sterk worden vereenvoudigd door de datawaarden en attributen van elementen niet voor te stellen. Een dergelijke vereenvoudigde boomvoorstelling voor het voorbeeld in Figuur 2.1 wordt weergegeven in Figuur 3.2.



Figuur 3.2: Vereenvoudigde boomvoorstelling XML document

In formele taal voldoet de boom t die geassocieerd wordt met een XML document aan de volgende recursieve definitie:

- de verzameling van labels in t is gelijk aan de verzameling van de namen van de elementen in het document;
- een document $\langle a \rangle w \langle / a \rangle$, waarbij w enkel tekst bevat, komt overeen met een boom die slechts één knoop heeft met a als label;
- een document $\langle a \rangle x_1 \dots x_k \langle / a \rangle$, waarbij de documenten $x_1, \dots, x_k$ overeenkomstige bomen $t_1, \dots, t_k$ hebben, komt overeen met een boom t met als wortel de knoop gelabeled met a en de bomen $t_1, \dots, t_k$ als kinderen van a, gaande van links naar rechts.

Merk op dat de boomvoorstelling van een XML document een boom is over een eindig alfabet. Dit betekent dat het aantal mogelijke labels die in de boom kunnen voorkomen beperkt zijn. In het algemeen wordt het alfabet bepaald door een XML schema. Een andere eigenschap van deze bomen is dat een knoop een willekeurig aantal kinderen heeft.

3.2 XML schema's en boomtalen

Een XML schema bepaalt welk eindig alfabet gehanteerd wordt voor de labels van de knopen in de boomvoorstelling van een XML document. Dit leidt niet tot een specifieke boom maar tot een verzameling van bomen. Formeel gezien is een XML schema een boomgrammatica die de verzameling bomen m.b.t. een eindig alfabet Σ genereert.

De verzameling bomen die wordt gegenereerd door een boomgrammatica met een eindig alfabet Σ wordt genoteerd met $\mathcal{T}_\Sigma$. Deze verzameling wordt een boomtaal genoemd [BMNS05].

Boomtalen kunnen worden ingedeeld in diverse klassen [MLMK05]. Dit geldt eveneens voor boomtalen die worden gedefinieerd door een XML schema. De klasse waartoe een door een XML schema gedefinieerde boomtaal behoort, is afhankelijk van de schemataal waarin het schema is uitgedrukt.

3.3 Reguliere boomtalen

De meest algemene en tevens ook meest krachtige klasse van boomtalen zijn de reguliere boomtalen [MLMK05]. De grammatica die een reguliere boomtaal genereert heet logischerwijs reguliere boomgrammatica.

Een reguliere boomgrammatica is een 4-tupel (N, T, S, P) met:

- N een eindige verzameling van *non-terminals*;
- T een eindige verzameling van *terminals*;
- S een verzameling van startsymbolen zodat $S \in N$;
- P een eindige verzameling van productieregels. Een productieregel is van de vorm $X \rightarrow a\,r$ met $X \in N$, $a \in T$ en r een reguliere expressie over N. X is de linkerkant, $a\,r$ is de rechterkant en r is het inhoudsmodel van de productieregel.

In Figuur 3.3 wordt een voorbeeld getoond van een correcte reguliere boomgrammatica. Beschouw de productieregel `Doc` $\rightarrow$ `doc (Para1, Para2`*`)`. Hierin is `Doc` de linkerkant, `doc (Para1, Para2`*`)` de rechterkant en `(Para1, Para2`*`)` het inhoudsmodel.

```
N  =  {Doc, Para1, Para2, Pcdata},
T  =  {doc, para, pcdata},
S  =  {Doc},
P  =  {Doc → doc (Para1, Para2*),
       Para1 → para (Pcdata),
       Para2 → para (Pcdata),
       Pcdata → pcdata ϵ}
```

Figuur 3.3: Een reguliere boomgrammatica

Hoofdstuk 2 geeft een beschrijving van de in de praktijk meest gebruikte XML schematalen. Bij één van deze schematalen, meerbepaald RELAX NG, werd reeds vermeld dat ze op degelijke wijze wiskundig is onderbouwd. Meer specifiek, RELAX NG is gebaseerd op de principes van reguliere boomtalen. Bijgevolg beschrijft elk XML schema dat wordt uitgedrukt in RELAX NG een reguliere boomgrammatica die een specifieke reguliere boomtaal definieert.

Er zijn heel wat formalismen in omloop waarin reguliere boomtalen kunnen worden uitgedrukt. In Sectie 3.1 werd reeds vermeld dat in deze tekst de uitdrukkingskracht van XML schematalen wordt beschreven. Om de verbanden tussen de diverse schematalen zo duidelijk mogelijk naar voor te

laten komen, zal er gebruik worden gemaakt van Extended DTD (EDTD) als formalisme [BMNS05].

Een EDTD is een 5-tupel $D = (\Sigma, \Delta, d, s_d, \mu)$ met:

- Σ een eindige verzameling symbolen (elementen);
- Δ een eindige verzameling symbolen (types);
- $d : \Delta \rightarrow regex(\Delta)$ een functie die symbolen uit Δ afbeeldt op reguliere expressies over Δ;
- $s_d \in \Delta$ is het startsymbool;
- $\mu : \Delta \rightarrow \Sigma$ een functie die Δ afbeeldt op Σ.

Een boom t over Σ voldoet aan een EDTD D als er een boom t' over Δ bestaat zodat $t = \mu(t')$. M.a.w. t is de boom die wordt verkregen door in t' elk label δ te vervangen door $\mu(\delta) = \sigma$.

Het voorbeeld in Figuur 3.4 toont een EDTD waarbij Σ = {`store`, `dvd`, `title`, `price`, `discount`} en Δ = {`store`, `reg-dvd`, `dis-dvd`, `title`, `price`, `discount`}. Er geldt μ(`reg-dvd`) = μ(`dis-dvd`) = `dvd`. De overige types worden door μ afgebeeldt op elementen met dezelfde naam als het type.

```
   store  →  (reg-dvd + dis-dvd)* dis-dvd (reg-dvd + dis-dvd)*
reg-dvd   →  title price
dis-dvd   →  title price discount
```

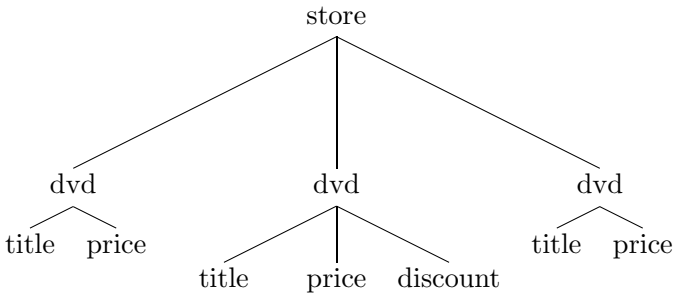
Figuur 3.4: EDTD

De vereenvoudigde boomvoorstelling van een mogelijke boom over Σ voor de EDTD in Figuur 3.4 wordt in Figuur 3.5 getoond. Het is niet moeilijk in te zien dat deze boom voldoet aan de EDTD. Figuur 3.6 toont een boom over Δ die, na toepassing van μ, duidelijk resulteert in de boom uit Figuur 3.5.

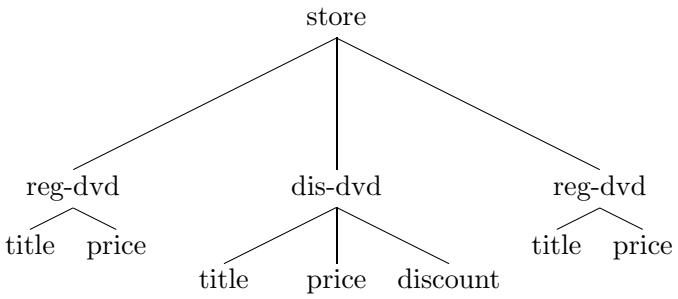
3.4 Lokale boomtalen

Aan de overkoepelende klasse van reguliere boomtalen kunnen beperkingen worden opgelegd om de uitdrukkingskracht te beperken. De minst krachtige klasse van boomtalen is de klasse van lokale boomtalen [MLMK05]. Vooraleer deze te definiëren wordt eerst het begrip *competing non-terminals* ingevoerd.

Twee verschillende *non-terminals* A en B worden als *competing* beschouwd als:



Figuur 3.5: Boom over Σ voor de EDTD in Figuur 3.4



Figuur 3.6: Boom over Δ voor de EDTD in Figuur 3.4

- een productieregel de *non-terminal* A in de linkerkant bevat;
- een andere productieregel de *non-terminal* B in de linkerkant bevat;
- en deze twee productieregels dezelfde *terminal* in de rechterkant bevatten.

Beschouw de reguliere boomgrammatica in Figuur 3.7. Deze bevat *competing non-terminals*. De *non-terminals* `Author1` en `Author2` zijn *competing* omdat de productieregels voor `Author1` en `Author2` beide de *terminal* `author` in de rechterkant bezitten. De rest van de grammatica bevat geen *competing non-terminals*.

```
N  =  {Book, Author1, Son, Article, Author2, Daughter},
T  =  {book, author, son, daughter},
S  =  {Book, Article},
P  =  {Book → book (Author1),
       Author1 → author (Son),
       Son → son ϵ,
       Article → article (Author2),
       Author2 → author (Daughter),
       Daughter → daughter ϵ}
```

Figuur 3.7: Een reguliere boomgrammatica met competing non-terminals

Een boomtaal is een lokale boomtaal wanneer ze gegenereerd wordt door een lokale boomgrammatica. Een lokale boomgrammatica is een reguliere boomgrammatica waar geen *competing non-terminals* in voorkomen. Figuur 3.8 geeft een voorbeeld van een correcte lokale boomgrammatica.

```
N  =  {Book, Author1, Son, Pcdata},
T  =  {book, author, son, pcdata},
S  =  {Book},
P  =  {Book → book (Author1),
       Author1 → author (Son),
       Son → son (Pcdata),
       Pcdata → pcdata ϵ}
```

Figuur 3.8: Lokale boomgrammatica

Het formalisme dat in deze tekst gehanteerd wordt om lokale boomtalen uit te drukken is DTD, de vereenvoudigde versie van Extended DTD dat voor reguliere boomtalen wordt gebruikt [BMNS05]. Het is geen toeval

dat de naam van het formalisme identiek is aan de naam van de XML schemataal. Een XML schema uitgedrukt in DTD beschrijft immers een lokale boomgrammatica.

Een DTD is een 3-tupel (Σ, d, s_d) met Σ een eindig alfabet, d een functie die symbolen uit het alfabet afbeeldt op reguliere expressies over dat alfabet en $s_d \in \Sigma$ het startsymbool. Meestal wordt het 3-tupel afgekort tot d wanneer Σ en s_d duidelijk zijn af te leiden uit de context. De reguliere expressie die geassocieerd wordt met een symbool uit het alfabet wordt ook wel het inhoudsmodel van het overeenkomstige element genoemd.

Figuur 3.9 toont een voorbeeld van een DTD. De beschreven boomtaal is een vereenvoudigde versie van de reguliere boomtaal die in Figuur 3.4 wordt gedefinieerd.

```
store  →  dvd dvd*
  dvd  →  title price (discount + ε)
```

Figuur 3.9: DTD

Merk op dat er in een DTD geen sprake is van een verzameling Δ voor types. Elke elementnaam uit het eindige alfabet Σ komt overeen met precies één reguliere expressie die m.b.v. de functie d enkel geassocieerd wordt met een symbool uit alfabet Σ. Het inhoudsmodel van een element wordt dus niet bepaald door de context van het element maar enkel door de naam van het element, vandaar de term lokaal.

De reguliere expressies in een DTD vertonen een kenmerkende eigenschap: determinisme. Wanneer de invoer van links naar rechts wordt verwerkt, is het te allen tijde mogelijk om te bepalen welk symbool in de reguliere expressie overeenstemt met het volgende symbool van de invoer. Dit wordt ook aangeduid als *one-unambigious*.

Meer formeel, een reguliere expressie r is *one-unambiguous* als en slechts als er geen twee strings wa_iv en wa_jv' bestaan in $L(\overline{r})$ zodat $i \neq j$. Met $\overline{r}$ wordt de reguliere expressie aangeduid die wordt verkregen door in r, voor elke i, het i-de a-element te vervangen door a_i.

Beschouw de reguliere expressie $ab + aa$. Deze expressie is niet *one-unambiguous* omdat $L(a_1b_1 + a_2a_3)$ de strings a_1b_1 en a_2a_3 bevat en $1 \neq 2$. De reguliere expressie $a(b + a)$ daarentegen is wel *one-unambiguous* aangezien $L(a_1(b_1 + a_2))$ enkel de strings a_1b_1 en a_1a_2 bevat. Merk op dat $ab + aa$ en $a(b + a)$ exact dezelfde taal beschrijven.

3.5 Single-type boomtalen

De klasse van single-type boomtalen bezit een uitdrukkingskracht die kleiner is dan de kracht van reguliere boomtalen maar groter dan die van lokale boomtalen [MLMK05]. Een boomtaal is een single-type boomtaal wanneer ze wordt gegenereerd door een single-type boomgrammatica.

Een single-type boomgrammatica is een reguliere boomgrammatica zodat:

- voor elke productieregel geldt dat de non-terminals in het inhoudsmodel niet *competing* zijn met elkaar, en;
- startsymbolen niet *competing* zijn.

De boomgrammatica die wordt getoond in Figuur 3.3 is geen single-type boomgrammatica. De non-terminals `Para1` en `Para2` zijn competing en ze komen beiden voor in het inhoudsmodel van de productieregel voor de non-terminal `Doc`.

Het voorbeeld in Figuur 3.7 daarentegen beschrijft wel een single-type boomgrammatica. Er komen geen productieregels in voor met meer dan één non-terminal in het inhoudsmodel. Bijgevolg kunnen er ook geen competing non-terminals voorkomen in het inhoudsmodel van een productieregel.

Een single-type boomgrammatica kan formeel worden voorgesteld door een single-type EDTD (EDTDST) [BMNS05]. Dat is een EDTD waarvan geen enkele reguliere expressie twee types $\delta \neq \delta'$ bevat zodat $\mu(\delta) = \mu(\delta')$.

```
    store  →  regulars discounts
  regulars  →  (reg-dvd)*
discounts  →  dis-dvd (dis-dvd)*
  reg-dvd  →  title price
  dis-dvd  →  title price discount
```

Figuur 3.10: Single-type EDTD

In Figuur 3.10 wordt een EDTDST getoond die een single-type boomtaal beschrijft die analoog is aan de reguliere boomtaal in Figuur 3.4. De verschillende types voor het element `dvd` worden ondergebracht in twee nieuwe elementen `regulars` en `discounts`.

Wanneer er wordt nagegaan welke XML schemataal dezelfde uitdrukkingskracht bezit als single-type EDTD, blijkt dat dit XML Schema is. Elk XML Schema schema kan worden uitgedrukt m.b.v. een EDTD maar het omgekeerde is echter niet waar. Niet elke EDTD kan in een XML Schema Document worden uitgedrukt. Dit komt omdat er een beperking is opgelegd aan XML Schema in de vorm van de Element Declarations Consistent (EDC).

EDC stelt dat in het inhoudsmodel van een element geen twee verschillende types aan dezelfde elementnaam mogen worden gekoppeld zoals dat in Figuur 3.11 gebeurt om het element `store` uit het voorbeeld in Figuur 3.4 te definiëren. Bijgevolg kan een XSD slechts een beperkte verzameling van EDTD beschrijven, meerbepaald EDTDST.

```
<xs:element name="store">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="dvd" type="reg-dvd" />
        <xs:element name="dvd" type="dis-dvd" />
      </xs:choice>
      <xs:element name="dvd" type="dis-dvd" />
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="dvd" type="reg-dvd" />
        <xs:element name="dvd" type="dis-dvd" />
      </xs:choice>
    </xs:sequence>
  </xs:complextype>
</xs:element>
```

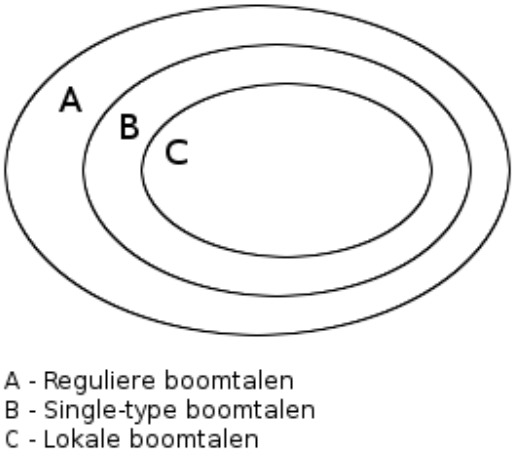
Figuur 3.11: Element Declarations Consistent

Daarnaast is XML Schema nog onderworpen aan een bijkomende beperking, nl. de Unique Particle Attribution (UPA). Deze beperking stelt dat er in een EDTD voor elke reguliere expressie r over Δ een one-unambiguous reguliere expressie $\mu(r)$ over Σ moet bestaan die wordt verkregen door in r elk type $\mathcal{T}$ te vervangen door element $\mu(\mathcal{T})$.

De reguliere expressie $a^1(a^2 + b^1)$ voldoet aan de UPA. Beschouw als tegenvoorbeeld de expressie $r = (a^1 + b^1)^* a^1(a^1 + b^1)$. Elk element a en b moet worden getypeerd als a^1 respectievelijk b^1 maar de expressie $\mu(r) = (a + b)^* a(a + b)$ is niet one-unambiguous, zowel de string $a_1a_2a_3$ als a_2a_3 maken deel uit van $L((a_1 + b_1)^* a_2(a_3 + b_2))$.

3.6 Overzicht

Om de uitdrukkingskracht van XML schematalen te vergelijken, werden drie klassen van boomtalen bestudeerd. De reguliere boomtalen zijn de meest krachtige, de lokale boomtalen het minst krachtig en de single-type boomtalen bevinden zich ertussen. Figuur 3.12 toont een schematische voorstelling van de onderlinge verhouding tussen deze drie klassen.



Figuur 3.12: Verhouding in uitdrukkingskracht van boomtalen

Elke klasse werd geassocieerd met een formalisme waarin de boomtalen die tot die klasse behoren kunnen worden uitgedrukt. Vervolgens werd nagegaan met welke XML schemataal het formalisme, en dus ook de klasse van boomtalen, overeenkomt. Figuur 3.13 geeft een overzicht van de verschillende associaties.

Boomtaal	**Formalisme**	**Schemataal**
Regulier	Extended DTD	RELAX NG
Single-type	Single-type Extended DTD	XML Schema
Lokaal	DTD	DTDs

Figuur 3.13: Een overzicht van de klassen van boomtalen, het formalisme waarin ze worden uitgedrukt en de overeenkomstige XML schemataal

Hoofdstuk 4

DTD Extra (DTDX)

4.1 Oorsprong

Tot op heden worden DTDs het meest gebruikt om schema's op te stellen voor XML documenten. Het grote succes van DTDs is vooral te danken aan de publicatie ervan in de XML W3C Recommendation en aan de rijke voorgeschiedenis van deze XML schemataal. De eenvoudige syntax speelt eveneens een belangrijke rol in het succes. Deze positieve eigenschappen, en nog enkele andere, werden reeds vermeld in Sectie 2.3. Daarnaast worden er in die sectie ook heel wat tekortkomingen aangehaald die aan DTDs zijn verbonden.

Er werden heel wat andere schematalen ontwikkeld die onder andere de tekortkomingen van DTDs aanpakten. Eén van deze schematalen is XML Schema dat eveneens als een W3C Recommendation werd gepubliceerd. XML Schema werd in meer detail beschreven in Sectie 2.4. De uitdrukkingskracht waarover XML Schema beschikt is dan ook veel groter dan die van DTDs zoals werd aangetoond in Hoofdstuk 3. Het verschil in uitdrukkingskracht wordt bekomen doordat XML Schema over een typeringsmechanisme beschikt dat toelaat om meerdere types te associëren met eenzelfde elementnaam op voorwaarde dat de elementnaam deel uitmaakt van verschillende inhoudsmodellen. In een DTD kan er aan een elementnaam slechts één type worden gekoppeld doorheen het hele document. Dit komt doordat het inhoudsmodel van een element enkel bepaald wordt door de naam van dat element.

Een praktijkstudie heeft aangetoond dat de precieze uitdrukkingskracht van XML Schema, en de geschiktheid van die kracht, niet altijd even duidelijk zijn [BMNS05]. De studie maakte gebruik van een uitgebreide verzameling van 819 XSDs die via het Web werden verzameld, waaronder ook enkele standaarden zoals de XML Schema Specification, XHTML, UDDI en RDF. Na deze verzameling grondig te hebben bestudeerd, werd er een duidelijk

overzicht opgesteld van de XML Schema eigenschappen die in de praktijk worden aangewend. Dit overzicht wordt getoond in Figuur 4.1.

Eigenschap	**% XSDs**
Afleiding	
simpleType uitbreiding	18.9
simpleType beperking	45.5
complexType uitbreiding	20.7
complexType beperking	3.6
abstract attributen	9.8
final attributen	0.9
block attributen	0.0
fixed attributen	6.4
substitutionGroup	6.4
herdefiniëring	1.0
Interleaving	
`xsd:all`	5.5
Modulariteit	
namespaces	12.1
import	27.7
Linking	
`key/keyref`	4.1
`unique`	2.9

Figuur 4.1: Overzicht van de in de praktijk aangewende XML Schema eigenschappen

De hamvraag in deze studie luidt als volgt: "In welke mate verschillen de onderzochte XSDs op structureel niveau van DTDs?". Het resultaat is teleurstellend. Van de verzamelde XSDs blijken er slechts 225 syntactisch correct te zijn volgens IBM's SQC. Van de overblijvende XSDs gebruikt de overgrote meerderheid het typeringsmechanisme van XML Schema op dergelijke wijze zodat ze structureel gelijk zijn aan DTDs. Slechts 33 XSDs (5 %) gebruikt het typeringsmechanisme om niet-lokale boomtalen te definiëren. Verrassend genoeg hangt in 30 van deze XSDs het inhoudsmodel van een element enkel af van het parent element.

De mogelijkheden van het typeringsmechanisme van XML Schema worden dus niet ten volle benut. Een mogelijke verklaring hiervoor zijn de complexi-

teit en de dubbelzinnigheid van het mechanisme waardoor het niet altijd even duidelijk is welke uitwerking het toekennen van types zal hebben. Er wordt in de studie geopperd dat de meeste schema's die in de praktijk voorkomen gemakkelijk als een extensie van DTDs kunnen worden gedefinieerd.

De combinatie van het abstracte voorstel uit de praktijkstudie en concrete voorstellen zoals DTD++ heeft geleid tot de ontwikkeling van DTD Extra (DTDX). Het is een concrete, voorouder-gebaseerde schemataal die werd ontwikkeld om de uitdukkingskracht van XML Schema, meerbepaald het typeringsmechanisme, te vatten met behulp van een zo eenvoudig mogelijke syntax. Hiertoe werd de syntax gebaseerd op de constructies van DTDs. Daarnaast wordt er ook tegemoet gekomen aan het beperkt aantal datatypes dat voorhanden is in DTDs door de meest frequent gebruikte datatypes van XML Schema te integreren.

4.2 Definitie en uitdrukkingskracht

DTD Extra is een voorouder-gebaseerde schemataal. Een dergelijke schemataal maakt het mogelijk om op expliciete wijze de voorouder context van een element te specificeren. Een voorouder-gebaseerde schemataal is een specifieke instantie van een meer algemene vorm, nl. de patroon-gebaseerde schematalen [BMNS05].

Patroon-gebaseerde schema's

Vooraleer patroon-gebaseerde schematalen meer formeel te definiëren, wordt er eerst een voorbeeld gegeven. Figuur 4.2 beschrijft een patroon-gebaseerd schema voor de EDTD in Figuur 3.10. Er wordt gebruik gemaakt van XPath als taal om de patronen uit te drukken. Het voorbeeld beschrijft een winkel waar zowel normaal geprijsde dvd's als afgeprijsde dvd's worden aangeboden. Een afgeprijsde dvd bevat een bijkomend element `discount` dat niet aanwezig is in een normaal geprijsde dvd. Het onderscheid tussen de twee soorten dvd's wordt gemaakt m.b.v. het pad vertrekkende van de wortel gaande tot het element `dvd`.

```
                          store  →  regulars discounts
                        regulars  →  dvd*
                       discounts  →  dvd dvd*
regular-dvd:    //regulars/dvd  →  title price
discount-dvd:  //discounts/dvd  →  title price discount
```

Figuur 4.2: Patroon-gebaseerd schema

In het voorbeeld wordt een knoop v met label `dvd` getypeerd als een afgeprijsde dvd als v een knoop met label `discounts` als ouder heeft en de kinderen van v gelabeled zijn met `title`, `price` en `discount`.

De formele definitie van patroon-gebaseerde schematalen maakt gebruik van een taal om patronen uit te drukken. Veronderstel $\mathcal{P}$, een taal die unaire patronen definieert. Elke boom t wordt door elk patroon $p \in \mathcal{P}$ geassocieerd met een verzameling van geselecteerde knopen uit t. Die verzameling knopen wordt genoteerd als $\varphi(t)$.

Een $\mathcal{P}$-schema is een 3-tupel (Σ, Δ, R) met:

- Σ een eindige verzameling symbolen (elementen);
- Δ een eindige verzameling symbolen (types);
- R een eindige verzameling van regels.

Een regel is van de vorm $\tau : \varphi \rightarrow s$ met:

- $\tau \in \Delta$ een type;
- $\varphi \in \mathcal{P}$ een patroon;
- s een reguliere expressie over Σ.

Elke regel van de vorm $a : a \rightarrow s$ met a de naam van een element, wordt een eenvoudige regel genoemd en wordt afgekort tot $a \rightarrow s$. Een DTD kan bijgevolg gezien worden als een $\mathcal{P}$-schema waarvan alle regels eenvoudige regels zijn.

Een boom t is geldig m.b.t. een $\mathcal{P}$-schema $S = (\Sigma, \Delta, R)$ als het label van elke knoop v in t tot Σ behoort en er voor elke v een regel $\tau : \varphi \rightarrow s$ bestaat zodat $v \in \varphi(t)$ en de kinderen van v voldoen aan de reguliere expressie s. In dat geval wordt v getypeerd als τ.

Voorouder-gebaseerde schema's

Verder bouwend op de definities uit de voorgaande sectie kan een voorouder-gebaseerd schema formeel worden gedefinieerd. Het is niet meer dan een instantie van een patroon-gebaseerd schema waarbij reguliere expressies worden gehanteerd als taal om de patronen uit te drukken.

Beschouw een reguliere expressie r. Laat $\varphi(r)$ het patroon zijn dat wordt geassocieerd met r. Dit patroon selecteert die knopen v van een boom t waarvan de opeenvolging van labels op het pad van de wortel tot en met v

voldoen aan r. Binnen r treedt Σ^* op als verzameling van alle Σ-strings en Σ is een wildcard voor een elementnaam.

Een voorouder-gebaseerd schema S is een 3-tupel (Σ, Δ, R) met:

- Σ een eindige verzameling symbolen (elementen);
- Δ een eindige verzameling symbolen (types);
- R een eindige verzameling regels.

Een regel is van de vorm $\tau : r \rightarrow s$ met:

- $\tau \in \Delta$ een type;
- r een reguliere expressie over symbolen van Σ die de ancestor-string van type τ uitdrukt;
- s een reguliere expressie over symbolen van Σ die de child-string van type τ uitdrukt. De expressie s moet one-unambiguous zijn.

De reguliere expressies r en s hebben betrekking op de boomstructuur van een XML document. Beschouw een boom t en een knoop v in t. De ancestor-string is de string die gevormd wordt door concatenatie van de labels die worden gelezen op het pad van de wortel van t tot en met v, genoteerd als $\textit{anc-string}^t(v)$. Indien de kinderen van v de knopen $v_1, \ldots, v_n$ zijn, genummerd van links naar rechts, bestaat de child-string van v, genoteerd als $\textit{ch-string}^t(v)$, uit de string gevormd door de labels van de kinderen.

Figuur 4.3 toont een voorouder-gebaseerd schema voor de EDTD in Figuur 3.10.

	`store`	$\rightarrow$	`regulars discounts`
	`regulars`	$\rightarrow$	`dvd`*
	`discounts`	$\rightarrow$	`dvd dvd`*
`regular-dvd:`	Σ^*`.regulars.dvd`	$\rightarrow$	`title price`
`discount-dvd:`	Σ^*`.discounts.dvd`	$\rightarrow$	`title price discount`

Figuur 4.3: Voorouder-gebaseerd schema

Een knoop v in een boom t voldoet aan de regel $\tau : r \rightarrow s$ indien $\textit{anc-string}^t(v) \in L(r)$ en $\textit{ch-string}(v) \in L(s)$. Voor een reguliere expressie r is $L(r)$ de taal van r. Dit is de verzameling strings die door r worden aanvaard.

Een boom t op zijn beurt voldoet aan een voorouder-gebaseerd schema S wanneer voor elke knoop v in t een regel τ bestaat zodat v voldoet aan die regel.

Uitdrukkingskracht

Het doel van DTD Extra is om de uitdrukkingskracht van XML Schema te vatten. In "Expressiveness of XML Schema" [BMNS05] wordt een formeel bewijs geleverd dat de kracht van de klasse van voorouder-gebaseerde schema's exact gelijk is aan die van de klasse van single-type EDTD's. M.a.w. patroon-gebaseerde schema's met reguliere expressies over ancestor-strings zijn een volwaardige alternatieve syntax voor XML Schema.

4.3 Syntax

Onderstaande contextvrije grammatica beschrijft de syntax van DTD Extra [Sip96]. In het streven naar een eenvoudige syntax werd geopteerd om de grammactica te baseren op de constructies van DTDs.

Merk op dat de naam van een element, type of attribuut minstens één karakter lang moet zijn. De naam begint met een hoofdletter, kleine letter of underscore eventueel gevolgd door geen, meerdere of een combinatie van hoofdletters, kleine letters, underscores, koppeltekens of cijfers.

De default of fixed waarde die kan worden gespecificeerd voor een attribuut wordt steeds omsloten door aanhalingstekens (”). De waarde mag elk mogelijk karakter, uitgezonderd aanhalingstekens, bevatten en dit in eender welke combinatie.

Om de leesbaarheid van een DTDX schema te verhogen is elke vorm van spatiëring toegelaten tussen en in de declaraties. Tijdens het parsen zal deze spatiëring worden genegeerd en heeft dus geen invloed op de semantiek van het schema. Merk op dat spatiëring die voorkomt in de waarde van een attribuut deel uitmaakt van die waarde en dus ook niet wordt genegeerd tijdens het parsen.

```
<schema> ::= <start> <rules> <attributes>

<start> ::= '<!START' <element-name> '>'

<rules> ::= <rule>+

<attributes> ::= <attribute>*

<element-name> ::= [a-z, A-Z, _] ([a-z, A-Z, _, -,
    0-9])*

<rule> ::= <element> | <type>

<attribute> ::= '<!ATTRIBUTE' (<element-name> | <
   type-name>) <attribute-name> <primitive-type>
```

```
    (<default> | <fixed> <required>? | <required>)
    '>'

<element> ::= '<!ELEMENT' <element-name> <hor-reg-
    expr> '>'

<type> ::= '<!TYPE' <type-name> <vert-reg-expr> <
    hor-reg-expr> '>'

<type-name> ::= [a-z, A-Z, _] ([a-z, A-Z, _, -,
    0-9])*

<attribute-name> ::= [a-z, A-Z, _] ([a-z, A-Z, _,
    -, 0-9])*

<default> ::= 'DEFAULT=' <quoted-value>

<fixed> ::= 'FIXED=' <quoted-value> <required>?

<required> ::= 'REQUIRED'

<quoted-value> ::= '"' (~["])* '"'

<hor-reg-expr> ::= <reg-expr>
    | <primitive-type>
    | 'EMPTY'
    | 'ANY'

<vert-reg-expr> ::= <reg-expr>

<reg-expr> ::= <concat-expr> ('|' <reg-expr>)*

<concat-expr> ::= <mult-expr> ('.' <concat-expr>)*

<mult-expr> ::= <factor> <mult>?

<factor> ::= '(' <reg-expr> ')'
    | NCName
    | '$'

<mult> ::= '?' | '*'

<primitive-type> ::= '#STRING'
    | '#BOOLEAN'
    | '#INTEGER'
    | '#FLOAT'
    | '#DOUBLE'
    | '#DURATION'
    | '#DATETIME'
```

```
| '#DATE'
| '#TIME'
```

4.4 Semantiek

Zoals blijkt uit de syntax is de structuur van een schema in DTD Extra zeer eenvoudig. Het schema begint steeds met de verplichte declaratie van het start element. Hiermee wordt expliciet bepaald welk element als wortel van de boomstructuur van het XML document moet optreden. Deze regel mag slechts één keer voorkomen aangezien een boom maar één wortel kan hebben.

De declaratie van het start element wordt gevolgd door een verzameling van regels die bepalen welke elementen er in het XML document mogen voorkomen. Er moet minstens één regel worden gedeclareerd, nl. het element dat dienst zal doen als start element.

Er wordt een onderscheid gemaakt tussen twee soorten regels: type-regels en element-regels. Een type-regel is een regel van de vorm $\tau : r \rightarrow s$ volgens de definitie in Sectie 4.2. De ancestor-string van het element wordt gespecificeerd door de reguliere expressie <`vert-reg-expr`>. Het inhoudsmodel is afhankelijk van <`hor-reg-expr`>. Deze uitdrukking kan een reguliere expressie zijn die de child-string beschrijft maar er zijn ook andere opties. In geval van <`primitive-type`> is de inhoud van het element een waarde die deel uitmaakt van het waardedomein van het gespecificeerde datatype. Daarnaast behoren ook een willekeurige inhoud (`ANY`) en een volledig lege inhoud (`EMPTY`) tot de mogelijkheden.

Een element-regel is gelijkaardig aan een type-regel met dat verschil dat er geen ancestor-string kan worden opgegeven. Deze laatste wordt impliciet door het systeem bepaald en laat toe dat het element eender waar kan voorkomen in het XML document, m.a.w. er wordt geen beperking opgelegd betreffende voorouders.

Het schema wordt afgesloten door een optionele verzameling declaraties die bepalen welke attributen aan welke regel worden toegekend. Een attribuut bevat bijkomende informatie over het inhoudsmodel dat door die regel wordt gespecificeerd. Deze informatie behoort tot het waardedomein van het gespecificeerde datatype. Tenzij expliciet d.m.v. de parameter `REQUIRED` wordt vastgelegd dat het attribuut verplicht moet voorkomen in een XML document, bepaalt de declaratie dat het attribuut optioneel is voor de betreffende regel.

Een attribuut declaratie kan nog één van twee andere parameters bevatten die invloed hebben op de waarde van het attribuut. De parameter `DEFAULT`

maakt het mogelijk om in de declaratie reeds een waarde voor het attribuut te registreren die gebruikt wordt wanneer het attribuut niet expliciet vermeld wordt in een XML document. Bij aanwezigheid van deze parameter is het niet langer mogelijk om het attribuut als vereist attribuut te declareren. De parameter `FIXED` legt een specifieke waarde vast voor een attribuut. Elk voorkomen van het attribuut in een XML document heeft dezelfde waarde en dat is de waarde die in de declaratie is vastgelegd. Het spreekt voor zich dat de waarden die door beide parameters worden opgegeven, moeten behoren tot het waardedomein van het gespecificeerde datatype.

Hoewel de syntax van DTD Extra gebaseerd is op de constructies van DTDs zijn er meer ingebouwde datatypes voorhanden dan in DTDs. Het betreft hier de meest frequent gebruikte datatypes van XML Schema. De waardedomeinen van de DTDX datatypes zijn identiek aan de waardedomeinen van de overeenkomstige XML Schema datatypes.

Het is noodzakelijk op te merken dat er varianten bestaan in de definitie van patroon-gebaseerde schema's en hun bijhorende semantiek. De definities in Sectie 4.2 stellen dat er voor elke knoop v in een boom t minstens één regel $\tau : \varphi \rightarrow s$ aanwezig moet zijn. Een patroon-gebaseerd schema is volgens deze definities, semantisch gezien, existentieel van aard. Daarnaast bestaat er de universele semantiek. Deze vereist dat voor elke regel τ geldt dat voor elke knoop v in $\varphi(\tau)$ de kinderen van v voldoen aan s. Deze twee vormen van semantiek zijn evenwaardig voor elke patroontaal die gesloten is onder de booleaanse operatoren.

Een mogelijk probleem dat zich kan voordoen bij het gebruik van voorouder-gebaseerde schema's is dat er door equivalente ancestor-strings ambiguïteit kan ontstaan bij de toekenning van types aan elementen. Om dit te voorkomen wordt in DTD Extra de beperking opgelegd dat voor elke twee regels τ en τ' geldt dat $\varphi(\tau) \cap \varphi(\tau') = \emptyset$, m.a.w. de doorsnede van de ancestor-strings van elke twee regels moet leeg zijn. Bijgevolg is het onmogelijk dat er equivalente ancestor-strings aanwezig zijn in een schema en kan aan elk element slechts één type worden toegekend.

4.5 Voorbeelden

Om een en ander wat meer te verduidelijken wordt dit hoofdstuk afgesloten met twee beknopte voorbeelden die verschillende aspecten van DTD Extra naar voren brengen.

Dvd-winkel

Figuur 4.4 toont een DTDX schema voor de single-type EDTD in Figuur 3.10. De semantiek van dit schema is vrijwel identiek aan de semantiek van het

patroon-gebaseerde schema in Figuur 4.2 met dat verschil dat er attributen werden toegevoegd.

```
<!START store>
<!ELEMENT store (regulars.discounts)>
<!ELEMENT regulars (dvd*)>
<!ELEMENT discounts (dvd.dvd*)>
<!TYPE regular-dvd store.regulars.dvd title.price>
<!TYPE discount-dvd store.discounts.dvd
title.price.discount>
<!ELEMENT title #STRING>
<!ELEMENT price #FLOAT>
<!ELEMENT discount #INTEGER>
<!ATTRIBUTE store id #INTEGER REQUIRED>
<!ATTRIBUTE store name #STRING DEFAULT="unknown">
```

Figuur 4.4: DTDX schema voor dvd-winkel

Het element `store` heeft twee attributen toegekend gekregen. Het attribuut `id` is een vereist attribuut dat het volgnummer van de winkel registreert als een waarde van het datatype `#INTEGER`. Daarnaast is er het attribuut `name` dat de naam van de winkel bevat. Indien dit attribuut niet wordt vermeld in het XML document, wordt de stringwaarde *unknown* gebruikt. In Figuur 4.5 wordt een XML document getoond dat voldoet aan dit schema.

Even diepte

Het schema in Figuur 4.6 definieert XML documenten waarin slechts drie elementnamen voorkomen.

De wortel van een document is het element `a` dat geen, één of meerdere elementen `b` als kind kan hebben. Het inhoudsmodel van een element `b` is afhankelijk van de diepte van de nesting van het element. Wanneer het element zich op een oneven niveau bevindt, is het element van het type `odd_b` en kan het enkel een element `b` als kind hebben. Bevindt het element `b` zich op een even diepte in de nesting, krijgt het het type `even_b` toegewezen en bestaat het inhoudsmodel uit een optioneel element `b` gevolgd door een element `c`. Dit laatste element bevat een waarde dat behoort tot het domein van het datatype `#STRING`. Figuur 4.7 geeft een XML document weer dat voldoet aan bovenstaand DTDX schema.

Veronderstel dat DTD Extra de bijkomende beperking inzake one-unambiguity die in Sectie 4.4 wordt besproken niet zou opleggen. Het wordt dan mogelijk om een schema op te stellen zoals het schema dat in Figuur 4.8 wordt getoond.

```
<store id="1" name="test">
  <regulars>
    <dvd>
      <title>Amelie</title>
      <price>17</price>
    </dvd>
    <dvd>
      <title>Good bye, Lenin!</title>
      <price>20</price>
    </dvd>
  </regulars>
  <discounts>
    <dvd>
      <title>Gothika</title>
      <price>15</price>
      <discount>4</discount>
    </dvd>
    <dvd>
      <title>Pulp Fiction</title>
      <price>11</price>
      <discount>6</discount>
    </dvd>
  </discounts>
</store>
```

Figuur 4.5: XML document voor het DTDX schema in Figuur 4.4

```
<!START a>
<!ELEMENT a (b*)>
<!TYPE odd_b a.(b.b)*.b (b)>
<!TYPE even_b a.(b.b)*.b.b (b?.c)>
<!ELEMENT c #STRING>
```

Figuur 4.6: DTDX schema met element `c` op even diepte

```
<a>
  <b>
    <b>
      <b>
        <b>
          <c>c1</c>
        </b>
      </b>
      <c>c2</c>
    </b>
  </b>
  <b>
    <b>
      <c>c3</c>
    </b>
  </b>
</a>
```

Figuur 4.7: XML document met element `c` op even diepte

```
<!START a>
<!ELEMENT a (b*)>
<!ELEMENT b ((b?.c) | b)>
<!TYPE c a.(b.b)*.c #STRING>
```

Figuur 4.8: Foutief DTDX schema met element `c` op even diepte

De reguliere expressie voor het inhoudsmodel van element `b` is duidelijk niet one-ambiguous. De strings b_1c_1 en b_2 maken deel uit van $L((b_1?c_1)|b_2)$ en $1 \neq 2$. Verder zorgt de ancestor-string van type `c` er voor dat dit type enkel geldig is op een even diepte van element `b`. Dit schema laat XML documenten toe zoals in Figuur 4.9.

```
<a>
  <b>
    <c>wrong</c>
  </b>
</a>
```

Figuur 4.9: Foutief XML document met element `c` op even diepte

Doordat de child-string van element `b` niet one-unambiguous is, kan er geen onderscheid gemaakt worden tussen een even en oneven diepte. Het is dus mogelijk dat er in een XML document een element `c` voorkomt op een oneven niveau in de nesting van elementen `b`. Wanneer dit gebeurt is er echter geen type dat een inhoudsmodel voor het element definieert.

Hoofdstuk 5

Methodes en algoritmes

5.1 Algemeen

Een XML schemataal kan slechts in de praktijk worden gebruikt als er toepassingen bestaan die de betrokken schema's kunnen verwerken. In Hoofdstuk 6 wordt de implementatie van de voor DTD Extra ontwikkelde tools uitgebreid toegelicht. Dit hoofdstuk bevat een formele beschrijving van enkele methodes en algoritmes die worden aangewend tijdens de implementatie.

5.2 Parsen

De syntax van een DTD Extra schema wordt op formele wijze gespecificeerd door de contextvrije grammatica in Sectie 4.3. Elk schema dat wordt opgesteld moet bijgevolg voldoen aan die grammatica.

Een contextvrije grammatica wordt formeel gedefinieerd als een 4-tupel $G = (V, \Sigma, R, S)$ met:

- V een eindige verzameling niet-terminale symbolen;
- Σ een eindige verzameling terminale symbolen, $V \cap \Sigma = \emptyset$;
- R een eindige verzameling producties. Een productie is van de vorm $X_0 \to X_1 \dots X_n$ met $X_0 \in N$ en $X_i \in N \cup T$ voor $i = 1, \dots, n$;
- $S \in N$ het startsymbool.

Een string s over Σ voldoet aan de grammatica G als s kan worden afgeleid uit het startsymbool S m.b.v. de producties. In elke stap van de afleiding wordt een niet-terminaal symbool X_0 in de huidige string vervangen door de rechterkant $X_1 \dots X_n$ van een productie die X_0 als linkerkant heeft. De huidige string is in het begin gelijk aan de 1-letter string S, op het einde aan de string s en tussenin bestaat de huidige string in het algemeen uit zowel

niet-terminale als terminale symbolen. Deze afleiding kan ook als een boom worden voorgesteld die toepasselijk *afleidingsboom* wordt genoemd [Sip96].

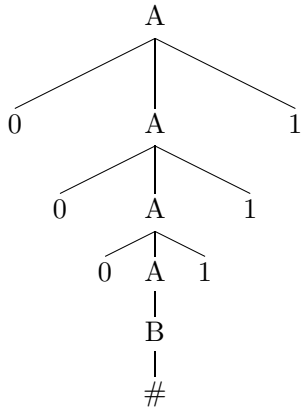
```
A  →  0A1
A  →  B
B  →  #
```

Figuur 5.1: Contextvrije grammatica

Beschouw de eenvoudige grammatica G in Figuur 5.1. Deze grammatica bevat drie producties, twee niet-terminale symbolen `A` en `B` waarbij `A` het startsymbool is en drie terminale symbolen `0`, `1` en `#`. Eén van de strings die kunnen worden gegenereerd door G is `000#111`. De afleiding van deze string is eenvoudig te construeren en wordt getoond in Figuur 5.2. In elke regel wordt het te vervangen niet-terminale symbool onderlijnd. De overeenkomstige afleidingsboom is weergegeven in Figuur 5.3.

```
   A
⇒  0A1
⇒  00A11
⇒  000A111
⇒  000B111
⇒  000#111
```

Figuur 5.2: Afleiding voor contextvrije grammatica



Figuur 5.3: Afleidingsboom voor contextvrije grammatica

Voor eenvoudige grammatica's zoals in Figuur 5.1 is het relatief eenvoudig om manueel te controleren of een string voldoet aan de grammatica. Hoe complexer de grammatica, hoe moeilijker het wordt om manueel een afleiding

te construeren. Een eenvoudig hulpmiddel om die controle uit te voeren is een parser.

Een parser is een toepassing die analyseert of de grammaticale structuur van een invoertekst voldoet aan een op voorhand vastgelegde grammatica. De parser zal de ingevoerde tekst tevens omzetten naar een op voorhand vastgelegde datastructuur [Wik06c].

Proces

Het proces dat wordt doorlopen om de ingevoerde tekst te analyseren bestaat uit drie fasen:

1. lexicale analyse: de ingevoerde tekst is niet meer dan een opeenvolging van karakters. Deze karakters worden sequentieel doorlopen en opgesplitst in groepen die overeenkomen met betekenisvolle symbolen die door de grammatica zijn gedefinieerd. Dergelijke groepen karakters worden tokens genoemd en behandeld alsof ze één karakter voorstellen. Doorgaans komen er ook groepen karakters voor die geen tokens vormen, denk hierbij aan spatiëring en commentaar. Het is de taak van de lexicale analyse om die groepen te verwijderen en enkel een reeks tokens te weerhouden;

2. grammaticale analye: de reeks tokens, die het resultaat is van de lexicale analyse, wordt doorzocht om na te gaan of de opeenvolgende tokens toegelaten uitdrukkingen vormen. De uitdrukkingen zijn gedefinieerd in de producties van de contextvrije grammatica. Ze worden vaak opgebouwd uit recursieve elementen en hun volgorde wordt bepaald door de grammatica;

3. semantische analyse: aan elke toegelaten uitdrukking, die wordt herkend tijdens de grammaticale analyse, zijn een aantal handelingen verbonden in overeenstemming met de semantiek van de grammatica. In deze fase worden de nodige stappen ondernomen om die handelingen tot een goed einde te brengen.

Op basis van het parsing proces kan de taak van een parser op een andere manier worden geformuleerd. Een parser tracht een afleiding te vinden voor de ingevoerde tekst gebruik makend van de producties die aanwezig zijn in de opgegeven grammatica. De afleiding kan op twee manieren tot stand worden gebracht:

- top-down: de parser vertrekt van het startsymbool en tracht dit om te vormen tot de invoer. In elke stap wordt een niet-terminaal symbool vervangen door de bijbehorende productie. Zo worden grote delen opgesplitst in incrementele kleinere delen om uiteindelijk de ingevoerde tekst te construeren;

- bottom-up: de parser start met de input en tracht deze te herschrijven totdat het startsymbool is bereikt. Op elk ogenblik worden de meest eenvoudige elementen gezocht, vervolgens die elementen die de eenvoudige elementen bevatten enzovoort. Dit proces wordt herhaald totdat enkel het startsymbool overblijft. Een andere naam voor deze methode is *shift-reduce* parsing.

5.3 Evaluatie

Nadat een voorouder-gebaseerd schema beschikbaar wordt gemaakt, zullen er XML documenten worden opgesteld volgens de regels in dat schema. Er bestaat echter altijd een mogelijkheid dat een XML document fouten bevat, gaande van eenvoudige spelfouten tot foutieve inhoudsmodellen. Het is dus niet overbodig dergelijke documenten te evalueren op hun correctheid.

De uitdrukkingskracht van de klasse van voorouder-gebaseerde schema's is exact gelijk aan die van de klasse van single-type EDTDs, zoals reeds werd vermeld in Sectie 4.2. Voor deze laatste is er een eenvoudig top-down algoritme voorhanden om een XML document te valideren [BMNS05]. Dit algoritme kan, mits enkele kleine wijzigingen, ook worden toegepast om te controleren of een XML document voldoet aan een voorouder-gebaseerd schema.

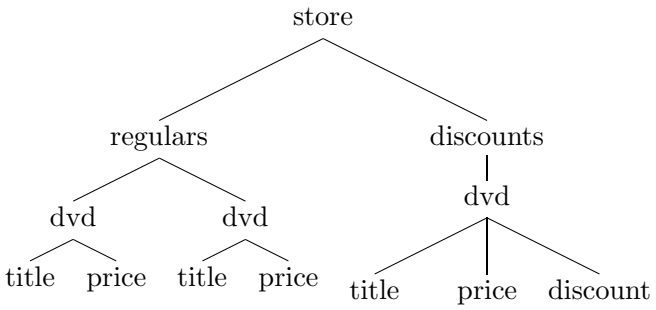
Proces

Het algoritme tracht aan elk element in het XML document een uniek type toe te kennen. Dit is mogelijk omdat het een single-type EDTD betreft en er geen ambiguïteit bestaat bij het bepalen van de types. Bovendien wordt in "Expressiveness of XML Schema" aangetoond dat in single-type EDTDs het type van een knoop u enkel afhankelijk is van de ancestor-string van die knoop. Dergelijke types worden voorouder-gebaseerde types genoemd.

Tijdens het evaluatieproces wordt er gebruik gemaakt van de boomvoorstelling van het document. Er wordt een algemene ancestor-string bijgehouden die bestaat uit alle labels van de knopen op het pad van de wortel tot de bezochte knoop. Voor elke knoop u wordt m.b.v. de ancestor-string de regel $\tau : r \rightarrow s$ bepaald zodat de ancestor-string voldoet aan $\mu(r)$, de reguliere expressie die wordt bekomen door in r elk type te vervangen door het overeenstemmende element. Vervolgens wordt gecontroleerd of de kinderen van u voldoen aan $\mu(s)$. Indien de kinderen niet voldoen, wordt het XML document verworpen. Voldoen de kinderen wel, dan wordt de knoop u getypeerd als τ. Het algoritme aanvaardt het XML document wanneer alle knopen, inclusief de bladeren, worden getypeerd zonder de boom te verwerpen.

Voorbeeld

Beschouw het voorouder-gebaseerd schema in Figuur 4.3. De boomvoorstelling van een XML document dat voldoet aan dit schema wordt getoond in Figuur 5.4.



Figuur 5.4: Boomvoorstelling van een XML document dat voldoet aan het schema in Figuur 4.3

Veronderstel dat het algoritme reeds de knopen `store` en `regulars` heeft getypeerd. Het algoritme gaat verder met het linkerkind van de knoop `regulars`, nl. de knoop `dvd`. De ancestor-string is nu `store.regulars.dvd` en voldoet aan $\mu(r)$ van de regel `regular-dvd:`Σ^*`.regulars.dvd` $\rightarrow$ `title price`. De kinderen van knoop `dvd` voldoen tevens aan $\mu(s)$ zodat de knoop wordt getypeerd als `regular-dvd`.

SAX

Het evaluatieproces verwerkt de inhoud van een XML document. Een eerste methode om dit te realiseren is de Simple API for XML (SAX) [Arm01, Wik06e]. Deze methode gebruikt een sequentieel mechanisme om een XML document te lezen. Tijdens het lezen worden er zogenaamde *events* afgevuurd die een gespecificeerde *callback method* aanroepen. Er zijn verschillende soorten events, o.a. XML Document, XML Text nodes, XML Element nodes, XML Processing Instructions en XML Comments. Elk van deze events wordt tweemaal afgevuurd, een keer bij het begin en een keer bij het einde van het XML onderdeel. XML attributen worden behandeld als een onderdeel van de data die behoort tot een element event.

De sequentiële verwerking van een XML document biedt twee voordelen. Ten eerste wordt er slechts een beperkte hoeveelheid geheugen in beslag genomen. Het geheugenverbruik is enkel afhankelijk van de diepte van de XML boom en de hoeveelheid data die wordt opgeslagen in de XML attributen van een XML element. Het is bijgevolg mogelijk om XML documenten te verwerken die vele malen groter zijn dan het beschikbare geheugen. Ten tweede is SAX *event-driven* van aard. Dit leidt tot een aanzienlijke snelheidswinst

tijdens het verwerken van een XML document. Het is immers niet nodig om geheugen te alloceren wanneer een XML onderdeel moet worden verwerkt omdat de bijhorende callback method wordt aangeroepen.

De voordelen van SAX geven rechtstreeks aanleiding tot het nadeel ervan: een XML document bevindt zich nooit in zijn geheel in het geheugen. Hierdoor wordt het gebruik van sommige vormen van XML validatie (bijna) onmogelijk. Beschouw als voorbeeld een XML document dat IDREF attributen bevat. De waarde van dergelijke attributen moet gelijk zijn aan de waarde van een ID attribuut dat tot een element in het document behoort. Om de IDREF attributen te kunnen verifiëren moet er worden bijgehouden welke ID en IDREF attributen er in het document voorkomen zodat de waarden ervan kunnen worden vergeleken. Een foutief IDREF attribuut wordt bijgevolg pas ontdekt nadat het hele document is gelezen. In dat geval wordt er tijd gespendeerd aan de verwerking van een document dat uiteindelijk wordt verworpen.

Daarnaast zijn er ook een aantal vormen van XML verwerking die vereisen dat een XML document zich in zijn geheel in het geheugen bevindt. Denk hierbij aan XPath en XSLT die op eender welk moment toegang moeten kunnen hebben tot eender welke knoop in de XML boom van het document.

DOM

De tweede methode die kan worden gebruikt om de inhoud van een XML document te verwerken, is het Document Object Model (DOM) [ABM+98, Arm01]. Dit is een API die de logische structuur van een document definieert en bepaald op welke manier een document kan worden benaderd en aangepast. De logische structuur word opgebouwd door het document om te zetten in een verzameling objecten die onderling met elkaar in verbinding staan. Hoewel DOM niet specificeert welke structuur en onderlinge relaties er worden gebruikt om een document voor te stellen, gebeurt dit in het algemeen d.m.v. een boom.

De naam “Document Object Model” werd gekozen omdat het een traditioneel *object model* betreft. Het DOM bepaald:

- de interfaces en objecten die gebruikt worden om een document voor te stellen en te manipuleren;
- de semantiek van die interfaces en objecten;
- de relaties en samenwerking tussen de interfaces en objecten.

Het grote voordeel van DOM komt overeen met het nadeel van SAX: de logische structuur van een XML document bevindt zich volledig in het geheugen. Elke knoop in de DOM boom is makkelijk toegankelijk en kan

eenvoudig worden bereikt door willekeurige navigatie. Alle elementen, hun attributen en hun inhoud kunnen worden aangepast en verwijderd. Bovendien is het mogelijk om nieuwe knopen toe te voegen aan de boom.

Uit het voordeel van DOM blijkt eveneens het nadeel. Enerzijds is er heel wat geheugenruimte nodig om de logische structuur in het geheugen te laden, anderzijds worden alle operaties uitgevoerd in het geheugen wat de snelheid niet ten goede komt.

5.4 Transformatie

In Sectie 4.2 wordt aangetoond dat de uitdrukkingskracht van voorouder-gebaseerde schematalen precies gelijk is aan de uitdrukkingskracht van single-type EDTDs. Elk DTDX schema kan bijgevolg worden getransformeerd in een equivalente single-type EDTD [BMNS05].

Proces

Beschouw het voorouder-gebaseerde schema $S^{VG} = (\Sigma^{VG}, \Delta^{VG}, R^{VG})$ dat voldoet aan alle bepalingen in Sectie 4.2. Duid de verzameling van alle reguliere expressies die een ancestor-string uitdrukken aan met $RE^{VG} = \{r \mid \forall \tau \in \Delta^{VG}, s \in RE(\Sigma^{VG}) : (\tau : r \rightarrow s) \in R^{VG}\}$.

De theoretische informatica heeft aangetoond dat er voor elke reguliere expressie een equivalente deterministische automaat bestaat. Construeer voor elke $r_i \in RE^{VG}$ een equivalente DFA $M_i = (Q_i, \Sigma^{VG}, \delta_i, q_i^0, F_i)$. Bereken het cross product van deze DFAs, nl. $M = M_1 \times \ldots \times M_n$ dat wordt gekarakteriseerd door $(Q, \Sigma^{VG}, \delta, q_0, F)$. De verzameling toestanden, Q, is speciaal te noemen in die zin dat elke toestand maximaal één element bevat dat een eindtoestand is in een M_i. Bovendien kunnen alle toestanden die geen finale toestand van een M_i bevatten, worden verwijderd omdat deze toestanden nooit zullen worden bezocht tijdens een accepting run. Elke toestand van M is een eindtoestand van M, m.a.w. $F = Q$. Merk op dat M deterministisch is aangezien alle M_i deterministisch zijn.

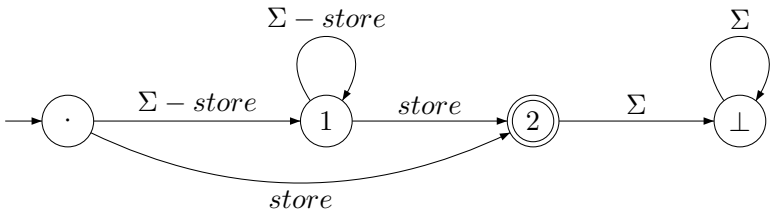
Op basis van de deterministische automaat M wordt de equivalente single-type EDTD $D^{ST} = (\Sigma^{ST}, \Delta^{ST}, d^{ST}, s_d^{ST}, \mu^{ST})$ geconstrueerd met:

- $\Sigma^{ST} = \Sigma^{VG}$;
- $\Delta^{ST} = \Sigma^{VG} \times Q$ en stelt de types voor als het label gevolgd door de toestanden van de DFAs;
- $d^{ST} : (a, q) \rightarrow s$ met s de reguliere expressie die wordt bekomen door in s_i elke τ te vervangen door (τ, q). De index i is de index die overeenkomt met het element van q dat een finale toestand is van M_i;

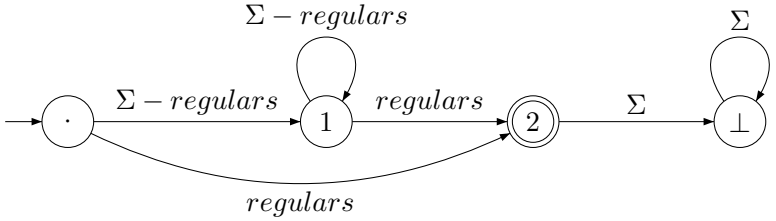
- $s_d^{ST} \in \Delta^{ST}$;
- $\mu^{ST} : (a, q) \rightarrow a$.

Voorbeeld

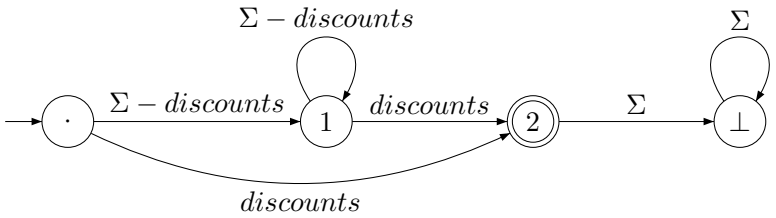
Om de transformatie wat aanschouwelijker te maken wordt hieronder getoond hoe het voorouder-gebaseerde schema in Figuur 4.3 wordt getransformeerd in een single-type EDTD. Allereerst wordt voor elke ancestor-string de equivalente DFA opgesteld. Deze worden getoond in Figuur 5.5, Figuur 5.6, Figuur 5.7, Figuur 5.8 en Figuur 5.9.



Figuur 5.5: DFA voor ancestor-string $\Sigma^*.store$

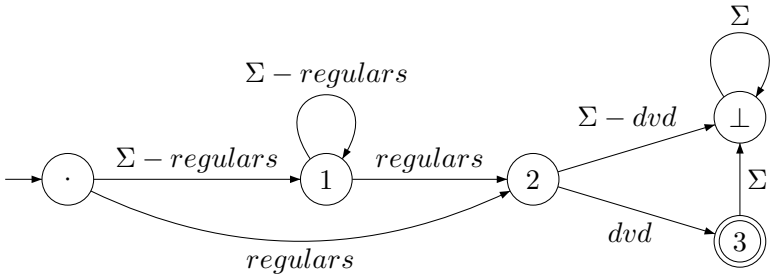


Figuur 5.6: DFA voor ancestor-string $\Sigma^*.regulars$

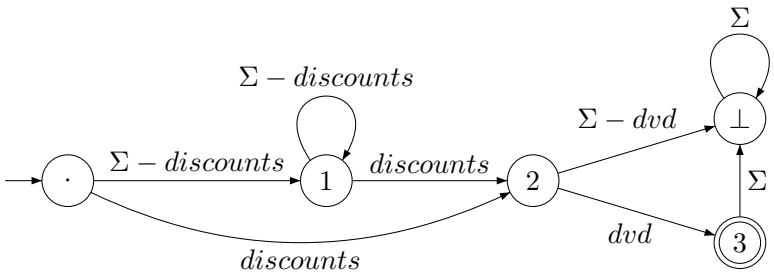


Figuur 5.7: DFA voor ancestor-string $\Sigma^*.discounts$

De types van de single-type EDTD worden aangeduid met $a^{(s_1,s_2,s_3,s_4,s_5)}$ waarbij a de naam van een element is en $s_1, \ldots, s_5$ de toestanden van respectievelijk de automaten $M_1, \ldots, M_5$ zijn.



Figuur 5.8: DFA voor ancestor-string $\Sigma^*.regulars.dvd$



Figuur 5.9: DFA voor ancestor-string $\Sigma^*.discounts.dvd$

De single-type EDTD wordt bekomen door in elke regel van het voorouder-gebaseerde schema de linkerkant te vervangen door het type aangevuld met de verzameling toestanden van M_i. De rechterkant wordt op analoge wijze aangepast. Het resultaat wordt getoond in Figuur 5.10.

$$
\begin{aligned}
store^{(\cdot,\cdot,\cdot,\cdot,\cdot)} &\rightarrow regulars^{(2,1,1,1,1)}\ discounts^{(2,1,1,1,1)} \\
regulars^{(2,1,1,1,1)} &\rightarrow (dvd^{(\perp,2,1,2,1)})^* \\
discounts^{(2,1,1,1,1)} &\rightarrow dvd^{(\perp,1,2,1,2)}\ (dvd^{(\perp,1,2,1,2)})^* \\
dvd^{(\perp,2,1,2,1)} &\rightarrow title\ price \\
dvd^{(\perp,1,2,1,2)} &\rightarrow title\ price\ discount
\end{aligned}
$$

Figuur 5.10: Single-type EDTD voor het voorbeeld in Figuur 4.3

Hoofdstuk 6

Implementatie

6.1 Algemeen

In de vorige hoofdstukken is het theoretische aspect van XML schematalen in het algemeen en DTD Extra in het bijzonder uitvoerig aan bod gekomen. Dit hoofdstuk spitst zich toe op de praktische uitwerking van DTD Extra.

Totnogtoe zijn er drie tools ontwikkeld voor DTD Extra. Het gaat om een parser, een evaluator en een transformator. De eerste twee vormen in feite het absolute minimum om een schemataal te kunnen gebruiken. Ze zijn verantwoordelijk voor het controleren van de correctheid van een schema respectievelijk het nagaan of de inhoud van een XML document conform is aan een schema. De transformator is een tool die toelaat om elk DTDX schema te vertalen naar een schema uitgedrukt in een andere XML schemataal.

6.2 Platform

Eén van de doelen van DTD Extra is te voorzien in een eenvoudige en overzichtelijke syntax. Tijdens de ontwikkeling van de tools wordt getracht dit principe verder te zetten. De keuze van het ontwikkelingsplatform speelt hierin een belangrijke rol. Om het eenvoudig gebruik van de tools te maximaliseren wordt er gebruik gemaakt van de programmeertaal Java. Dit biedt de volgende voordelen:

- platformonafhankelijkheid: dit betekent dat het mogelijk is om de tools op eender welk platform uit te voeren, zonder dat er specifiek aandacht dient te worden besteed aan platformafhankelijke elementen [GM96];
- Java API for XML Processing (JAXP): het onderdeel van de uitgebreide verzameling XML APIs dat enkele interfaces aanbiedt om gebruik te kunnen maken van de SAX, DOM en XSLT APIs in Java [Arm01];

- Java Archive (JAR): het standaard gecomprimeerde archiefformaat om meerdere bestanden te bundelen in één enkel bestand. Een JAR bestand bevat typisch de *class* bestanden die nodig zijn om een toepassing uit te voeren. Het bevat tevens een *manifest* bestand dat beschrijft op welke manier het archief zal worden gebruikt. Daarnaast kunnen er ook externe bronnen worden opgenomen in het archief. Hierdoor wordt het dus mogelijk om elke DTDX tool beschikbaar te maken in één uitvoerbaar bestand dat alle noodzakelijke elementen bevat [Sun06].

Door het gebruik van Java ontstaat er nog een bijkomend voordeel. De systeemvereisten om de DTDX tools te kunnen uitvoeren, worden tot een minimum beperkt. De enige vereiste is dat er een werkende versie van de Java Runtime Environment (JRE) versie 5.0 (of later) voor het Java 2 Platform, Standard Edition (J2SE) aanwezig is op het systeem. Merk op dat een incorrecte werking van de JRE tot foutieve resultaten van de DTDX tools kan leiden. Oudere JRE versies zijn niet aangewezen omdat er in de broncode uitvoerig gebruik wordt gemaakt van *generics*, *enumeratietypes* en de verbeterde *for*-lus [Aus04].

De voor DTD Extra ontwikkelde tools zijn command-line tools. Alle benodigde parameters voor een tool worden dus via de command-line opgegeven. Om deze parameters op een zo eenvoudig mogelijke manier te kunnen verwerken, wordt er gebruikt gemaakt van een zogenaamde command-line parser, meerbepaald Java Simple Argument Parser (JSAP) [Lam06].

Een command-line parser laat toe om de naam en het type van de parameters die kunnen worden opgegeven te specificeren. Tijdens de uitvoer van een programma wordt de command-line geparsed om de aanwezige parameters en hun waarde te identificeren. Vervolgens kunnen de waarden van de parameters worden opgevraagd zodat het programma op de gewenste manier zal functioneren.

Er werd geopteerd voor JSAP omdat deze enkele eigenschappen bezit die ontbreken in andere command-line parsers:

- naast de specificatie van de naam en het type van een parameter is het ook mogelijk om een default waarde vast te leggen;
- tijdens het parsen wordt niet enkel de waarde van een parameter bepaald, er wordt tevens nagegaan of de waarde voldoet aan het type van de parameter;
- bij het opvragen van de waarde van een parameter is er geen typecasting nodig. Het return type komt overeen met het type van de parameter.

6.3 Intern datamodel

Vooraleer er tools kunnen worden ontwikkeld die in staat zijn om een DTDX schema te verwerken, is er nood aan een intern datamodel om het schema voor te stellen. Dit datamodel bevat alle informatie die deel uitmaakt van het schema en stelt deze informatie ter beschikking d.m.v. objecten die gemakkelijk kunnen worden opgevraagd en gemanipuleerd. De tools maken gebruik van het interne datamodel om toegang te krijgen tot de benodigde informatie in het schema zodat ze hun taak probleemloos kunnen vervullen.

In Hoofdstuk 4 wordt DTD Extra voorgesteld. Uit Sectie 4.3 en Sectie 4.4, die de syntax respectievelijk de semantiek van een DTDX schema nauwkeurig beschrijven, kunnen de bouwstenen van het interne datamodel eenvoudig worden afgeleid. Op basis van de onderlinge relaties tussen deze bouwstenen werden de benodigde klassen opgesteld en ingedeeld in packages om tot een robuust ontwerp te komen.

Het interne datamodel voor DTDX is opgesplitst in drie packages:

- package `automaton`: definieert een basisimplementatie voor automaten. Daarnaast zijn er uitbreidingen opgenomen om zowel niet-deterministische (NFA) als deterministische (DFA) eindige automaten te ondersteunen;
- package `schema`: definieert alle klassen die nodig zijn om de logica van voorouder-gebaseerde schemas te vatten;
- package `tree`: een implementatie van binaire bomen die slechts enkele eenvoudige bewerkingen biedt zodat een boom kan worden opgesteld. De gegevens in een boom kunnen worden opgevraagd maar er zijn geen algoritmes aanwezig om een boom te doorlopen.

Meer informatie over de verschillende klassen en hun inhoud is terug te vinden in de JavaDocs die bij de broncode worden geleverd.

6.4 DTDX Parser

De DTDX Parser stelt de gebruiker in staat om te controleren of een DTD Extra schema zowel syntactisch als semantisch correct is. Dit betekent dat door de tool wordt nagegaan of het schema voldoet aan de grammatica beschreven in Sectie 4.3 en de voorwaarde betreffende ancestor-strings in acht is genomen. Bovendien wordt het schema omgezet in het in de vorige sectie besproken interne datamodel voor DTDX.

JavaCC

De DTDX parser werd ontwikkeld m.b.v. een parser generator, een tool die de specificatie van een grammatica leest en die vervolgens omzet in een programma dat in staat is overeenkomsten met de grammatica te herkennen.

Uit het grote aanbod van parser generators voor Java werd Java Compiler Compiler (JavaCC) geselecteerd [jav06]. Volgens de ontwikkelaars de meest populaire parser generator voor Java toepassingen. Het succes is hoofdzakelijk te wijten aan het feit dat JavaCC deel uitmaakt van *java.net*. Dat is een webgebaseerd ontwikkelingsplatform, opgestart en gefinancierd door Sun, om de interactie tussen Java ontwikkelaars te verbeteren en allerlei nieuwe Java technologieën en toepassingen te ontwikkelen. Bovendien wordt java.net gesteund door O'Reilly Media.

Naast de basismogelijkheden van een parser generator biedt JavaCC eveneens ondersteuning voor het opbouwen van bomen (via de inbegrepen tool JJTree), het genereren van documentatie (met de tool JJDoc) en debugging.

Hieronder volgt een lijst van kenmerkende eigenschappen:

- JavaCC genereert top-down parsers. Dit heeft als belangrijkste voordeel dat er meer algemene grammatica's kunnen worden gebruikt. Enkele bijkomende voordelen van een top-down parser zijn het eenvoudige debugging proces en de mogelijkheid om bepaalde waarden zowel opwaarts als neerwaarts door te geven in de afleidingsboom. Standaard worden er LL(1) parsers gegenereerd. Deze parsers onderzoeken enkel het volgende token. Omdat sommige delen van een grammatica niet LL(1) zijn ondersteunt JavaCC syntactische en semantische lookahead om shift-shift ambiguïteiten lokaal op te lossen. Dit houdt in dat een parser enkel op welbepaalde punten LL(k) (k tokens leest) is en elders LL(1). Deze methode wordt toegepast om de performantie van de gegeneerde parser te maximaliseren;

- De lexicale en grammaticale specificaties van een grammatica maken deel uit van het zelfde bestand waardoor het eenvoudiger wordt om de grammatica te bestuderen en te onderhouden. Bovendien laat JavaCC het gebruik van EBNF toe om recursie te beperken en de leesbaarheid te verhogen;

- Het gedrag van JavaCC en van de gegenereerde parsers kan d.m.v. heel wat opties gedetailleerd worden afgesteld. Zo zijn er opties die bepalen hoe een invoer in Unicode moet worden verwerkt. Het gebruik van Unicode karakters is niet beperkt tot de lexicale analyse. Het is eveneens toegelaten dat de lexicale specificaties Unicode karakters bevatten. Zo worden producties beschikbaar die elementen bevatten

die afhankelijk zijn van een bepaalde taal en waarin zodoende bepaalde niet-ASCII karakters voorkomen;

- Een token is hoofdlettergevoelig tenzij expliciet anders bepaald. Dit kan zowel op globaal niveau voor alle tokens of op lokaal niveau voor elk token apart. Bovendien kunnen speciale tokens worden gespecificeerd. Dit zijn tokens die tijdens het parsing proces zullen worden genegeerd maar die wel beschikbaar blijven voor de tools. Een handige toepassing van zulke tokens is de verwerking van commentaar;
- JavaCC bevat enkele opties om een diepgaande analyse te kunnen maken van de stappen tijdens parsing en de verwerking van tokens. De gegenereerde parsers bevatten bijgevolg een uitgebreid systeem voor foutafhandeling. Elke parser is in staat om in geval van een fout tijdens het parsing proces exact aan te geven wat de fout is en waar die zich voordeed.

Broncode

Alle klassen gerelateerd aan de parser zijn opgenomen in het package `parser`. Enkel het document `DTDXParser.jj` bevat eigenhandig geschreven code, de overige klassen zijn gegenereerd door JavaCC. Naast het controleren van de syntax in het DTDX schema worden er nog enkele bijkomende controles uitgevoerd die noodzakelijk zijn om te verzekeren dat het een correct schema betreft.

Het is niet mogelijk een instantie aan te maken van de klasse `DTDXParser`. De parser kan enkel worden aangeroepen met de statische functie `parse()` die als invoer een `InputStream` en de naam van het schema verwacht en als resultaat een `DTDXSchema` instantie teruggeeft. Immers, alle informatie betreffende het schema is beschikbaar via de resulterende `DTDXSchema` instantie. Bovendien is de functie `parse()` de enige aanroepbare functie van `DTDXParser`. Merk op dat de invoer zowel een stuk tekst als een bestand kan zijn.

Vooraleer de eigenlijke verwerking van het opgegeven schema begint, worden er enkele voorbereidingen getroffen:

1. er wordt gecontroleerd of de gespecificeerde invoer kan worden gelezen;
2. een nieuwe instantie van `DTDXSchema` wordt aangemaakt.

Nadat de voorbereiding succesvol werd afgerond, begint de eigenlijke taak van de DTDX Parser. De volgorde van de opeenvolgende stappen is afhankelijk van de opbouw van de grammatica:

1. de naam van het start element wordt geregistreerd in de `DTDXSchema` instantie;
2. de declaraties van de regels worden verwerkt:
   (a) de naam van de regel wordt geregistreerd;
   (b) er wordt een `BinaryTree` instantie opgesteld voor de ancestor-string;
   (c) het inhoudsmodel wordt bepaald. In geval van een reguliere expressie wordt ook hier een `BinaryTree` instantie opgesteld;
   (d) de naam, ancestor-string en child-string worden doorgegeven aan een bijkomende functie die de regel toevoegt aan het `DTDXSchema`. Deze functie controleert dat de naam uniek is en dat er geen conflict is met een bestaande ancestor-string.
3. de declaraties van de attributen worden verwerkt:
   (a) de betrokken regel wordt bepaald;
   (b) de naam van het attribuut wordt bepaald;
   (c) het datatype wordt bepaald;
   (d) bijkomende opties worden geregistreerd;
   (e) een `DTDXAttribute` instantie wordt aangemaakt en samen met de betrokken regel doorgegeven aan een bijkomende functie die het `DTDXAttribute` zal toevoegen aan het `DTDXSchema` nadat er is gecontroleerd dat de betrokken regel bestaat en dat het geen duplicaat attribuut voor die regel betreft.
4. er wordt nagegaan of er een regel aanwezig is in het `DTDXSchema` die van toepassing is op het geregistreerde start element.

Wanneer er zich in één van deze stappen een fout voordoet, wordt er een `ParseException` gegooid die duidelijk vermeld wat er fout liep.

Een codefragment afkomstig uit de broncode van de DTDX Parser licht toe hoe de statische functie wordt gebruikt om een DTDX schema te parsen. Het object `config` bevat het resultaat dat wordt gegenereerd door JSAP na het verwerken van de command-line.

```
try {
    // validate if the specified file containing
       the schema exists
    File schemaFile = config.getFile("schema");

    if (!schemaFile.exists()) {
        System.err.println("Error␣-␣file␣not␣found
           :␣" + schemaFile);
```

```
            System.exit(1);
        }

        // parse the file containing the schema
        System.out.print("Parsing␣DTDX␣schema...");

        DTDXSchema schema = parser.DTDXParser.parse(
            new FileInputStream(schemaFile.
            getAbsoluteFile()), schemaFile.getName());

        System.out.println("succesfull");
        ...
    } catch (FileNotFoundException fnfe) {
        ...
    } catch (parser.ParseException pe) {
        ...
    }
```

Uitvoeren

Met de DTDX Parser kan op eenvoudige wijze worden gecontroleerd of een schema correct is. Het uitvoeren van de tool gebeurt analoog aan het uitvoeren van een andere Java toepassing:

```
Usage: java -jar parser.jar [-v|--verbose]
       [-h|--help] <schema>

  [-v|--verbose]
         Requests verbose output. The internal data model
         for the schema is written to screen.

  [-h|--help]
         Prints this help message.

  <schema>
         The file containing the DTDX schema definition.
```

Onderstaande uitvoer toont het resultaat van de DTDX Parser voor het schema in Figuur 4.4.

```
java -jar parser.jar store.txt

Parsing DTDX schema...succesfull
```

6.5 DTDX Evaluator

De DTDX Evaluator is een tool die kan worden gebruikt om de geldigheid van een XML document t.o.v. een DTDX schema te controleren.

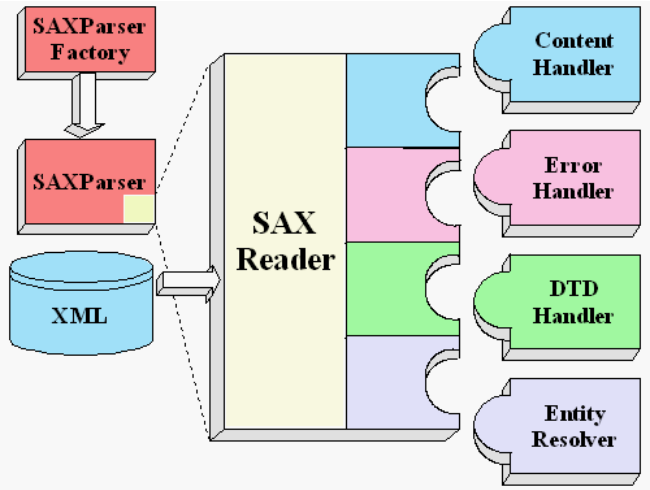
JAXP

In Sectie 6.2 werd reeds aangehaald dat Java enkele interfaces aanbiedt via JAXP om gebruik te kunnen maken van SAX en DOM [Arm01]. De belangrijkste APIs worden gedefinieerd in het package `javax.xml.parsers`. Dit package bevat twee leverancieronafhankelijke *factory classes* `SAXParserFactory` en `DocumentBuilderFactory` die een `SAXParser` respectievelijk `DocumentBuilder` afleveren. De `DocumentBuilder` genereert op zijn beurt een, met DOM overeenkomstig, `Document` object.

Beide factory classes bieden de mogelijkheid om de XML implementatie van een specifieke leverancier te gebruiken zonder dat de code moet worden aangepast. De keuze van implementatie kan worden veranderd door de eigenschappen van de `SAXParserFactory` en `DocumentBuilderFactory` te wijzigen.

De structuur van de SAX APIs wordt getoond in Figuur 6.1. Het proces wordt gestart door het aanmaken van een parser instantie m.b.v. een instantie van de `SAXParserFactory` klasse. De verkregen parser omvat een `SAXReader` object. Zodra de `parse()` method van de parser wordt aangeroepen, zal de `SAXReader` één van de vele callback methods aanroepen die worden gedefinieerd door de interfaces `ContentHandler`, `ErrorHandler`, `DTDHandler` en `EntityResolver`. Hieronder volgt een lijst van de meest belangrijke SAX APIs:

- `SAXParserFactory`: dit object creëert een parser instantie op basis van de ingestelde eigenschappen;
- `SAXParser`: deze interface definieert verscheidene `parse()` methods. In het algemeen wordt er een XML bron en een `DefaultHandler` object doorgegeven aan de parser die de XML verwerkt en de gepaste methods in de `DefaultHandler` aanroept;
- `SAXReader`: vervat in `SAXParser` en verantwoordelijk voor de communicatie met de gedefinieerde SAX event handlers;
- `DefaultHandler`: niet aanwezig in de figuur. De `DefaultHandler` voorziet een implementatie van de `ContentHandler`, `ErrorHandler`, `DTDHandler` en `EntityResolver` interfaces zodat het enkel nodig is de gewenste methods te overschrijven;



Figuur 6.1: Java SAX parsing API

- `ContentHandler`: verantwoordelijk voor het aanroepen van methods zoals `startDocument`, `endDocument`, `startElement` en `endElement` wanneer een XML onderdeel wordt herkend. Deze interface definieert ook methods voor karakters en processing instructions die worden aangeroepen als de parser de tekst in een XML element of een processing instruction te verwerken krijgt;

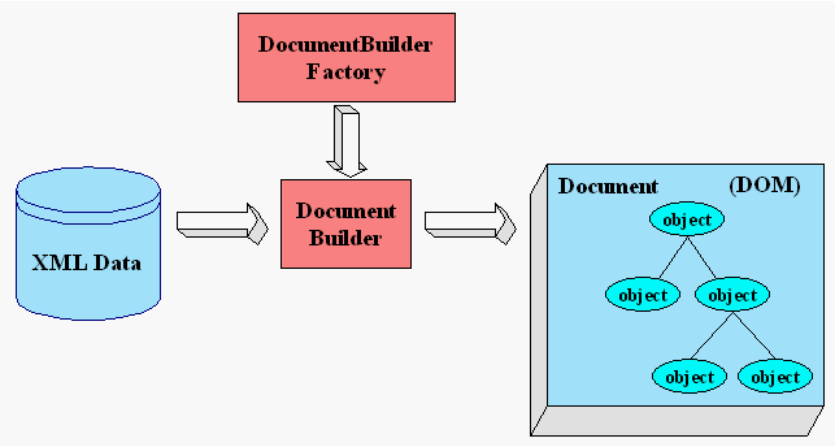
- `ErrorHandler`: methods `error`, `fatalError` en `warning` worden aangeroepen als antwoord op diverse fouten tijdens het parsen. De standaard error handler gooit een `exception` bij fatale fouten en negeert alle andere fouten (inclusief valideringsfouten). Een goed begrip van de SAX parser is dus nodig aangezien de toepassing in sommige gevallen kan herstellen van een fout en in andere gevallen niet. Om te voorzien in een correcte foutafhandeling is het meestal nodig om een eigen error handler toe te voegen;

- `DTDHandler`: definieert methods die gebruikt worden tijdens de verwerking van DTDs om de declaraties van een unparsed entity te herkennen en te behandelen;

- `EntityResolver`: de method `resolveEntity` wordt aangeroepen als de parser data moet verwerken die wordt aangeduid door een URI. Meestal is dat een eenvoudige URL (Uniform Resource Locator) die de locatie van het document specificeert. Een enkele keer is het een URN (Uniform Resource Name), een unieke identiteit. De `EntityResolver` gebruikt dan die identiteit i.p.v. een URL om het document terug te vinden, bijvoorbeeld de lokale kopie van een document.

Een typische toepassing voorziet een implementatie voor de meeste methods in de `ContentHandler` interface. Gezien de werkwijze van de `ErrorHandler` zal een robuuste toepassing ook een implementatie voorzien voor de methods in deze interface.

De implementatie van een DOM parser is typisch een stuk eenvoudiger dan een SAX implementatie. Figuur 6.2 toont de werkwijze.



Figuur 6.2: Java DOM parsing API

Met de klasse `DocumentBuilderFactory` wordt een `DocumentBuilder` instantie aangemaakt die gebruikt wordt om een `Document` object aan te maken. Dit object voldoet aan de eisen die gesteld worden in de specificatie van DOM. De `DocumentBuilder` die wordt verkregen is afhankelijk van de ingestelde eigenschappen van `DocumentBuilderFactory`.

Het `Document` object kan op twee manieren worden aangemaakt door de `DocumentBuilder`:

- method `newDocument()`: genereert een blanco `Document` object dat de `org.w3c.dom.Document` interface implementeert;
- `parse()` methods: genereren een `Document` object op basis van reeds bestaande XML gegevens. Het resultaat is een DOM boom zoals wordt getoond in Figuur 6.2.

Uiteraard bevatten de SAX en DOM APIs heel wat meer klassen en interfaces dan in de vorige paragrafen worden beschreven. Een overzicht van de diverse packages is opgenomen in Figuur 6.3 en Figuur 6.4.

Package	Omschrijving
`org.xml.sax`	Definieert de SAX interfaces. Het package prefix `org.xml` werd bepaald door de groep die de SAX API heeft gedefinieerd.
`org.xml.sax.net`	Definieert SAX extensies die gebruikt worden bij een meer complexe verwerking door SAX zoals het verwerken van DTDs.
`org.xml.sax.helpers`	Bevat hulpklassen die het gebruik van SAX vereenvoudigen, bijvoorbeeld een default handler waarvan enkel die methods moeten worden geïmplementeerd die effectief nodig zijn.
`javax.xml.parsers`	Definieert de `SAXParserFactory` klasse die een `SAXParser` aflevert. Er worden ook exception klassen gedefinieert zodat fouten kunnen worden gerapporteerd.

Figuur 6.3: JAXP SAX packages

Package	Omschrijving
`org.w3c.dom`	Definieert de DOM interfaces voor XML documenten zoals gespecificeerd door het W3C.
`javax.xml.parser`	Definieert de `DocumentBuilderFactory` klasse en `DocumentBuilder` klasse die een object afleveren dat de W3C Document interface implementeert. De te gebruiken *factory* om de *builder* te creëren, wordt bepaald door de `javax.xml.parsers` eigenschappen die kunnen worden ingesteld vanaf de command-line of bij het aanroepen van de method `newInstance`. Het package definieert ook de `ParserConfigurationException` klasse om fouten te rapporteren.

Figuur 6.4: JAXP DOM packages

Broncode

De implementatie van de DTDX Evaluator is opgenomen in het gelijknamige package `evaluator`. Dit package bevat drie klassen en een interface:

- `DOMEvaluator`: een evaluator gebaseerd op de DOM API;
- `EvaluationException`: wordt gegooid wanneer er zich een fout voordoet tijdens de evaluatie, zowel in de `DOMEvaluator` als in de `SAXEvaluator`. Elke instantie bevat een duidelijke omschrijving;
- `Evaluator`: de interface die het algemene gedrag van `DOMEvaluator` en `SAXEvaluator` bepaald;
- `SAXEvaluator`: een evaluator gebaseerd op de SAX API.

In eerste instantie lijkt het onnodig om zowel een evaluator te ontwikkelen gebaseerd op DOM als op SAX. Er is echter een eenvoudige verklaring. Het evaluatieproces dat wordt beschreven in Sectie 5.3 gaat uit van de boomstructuur van een XML document. DOM stelt achter de schermen impliciet een boomstructuur op voor een XML document. Om te verzekeren dat de logica van het evaluatieproces op een correcte manier is geïmplementeerd, werd bijgevolg eerst de klasse `DOMEvaluator` ontwikkeld. Vervolgens werd de bekomen implementatie gebruikt als leidraad om de klasse `SAXEvaluator` te ontwikkelen.

De klasse `DOMEvaluator` kan worden geïnstantieerd door een constructor die als parameter een `DTDXSchema` object verwacht dat het schema representeert waaraan de te evalueren XML documenten moeten voldoen. Er wordt impliciet een variabele `ancestorString` geïnitialiseerd die het pad van de wortel tot het huidige XML element registreert.

Het evaluatieproces wordt gestart zodra de method `parse()` wordt aangeroepen. Deze method krijgt een parameter mee die de lokatie van het te evalueren XML document specificeert. De volgende stappen worden doorlopen:

1. controle of er een geldig `DTDXSchema` object werd gespecificeerd;
2. aanmaken van een DOM `Document` gebaseerd op de data in het opgegeven XML document;
3. controle of de wortel van de DOM boom overeenkomt met de declaratie van het start element in het DTDX schema;
4. aanroepen van de method `validateDOMElement`. Dit is een recursieve method die de DOM boom in in-order doorloopt en enkel de knopen van het type `org.w3c.dom.Node.ELEMENT_NODE` bezoekt. Elke knoop wordt als volgt verwerkt:

(a) voeg de huidige knoop toe aan de variabele `ancestorString`;

(b) identificeer de DTDX regel die voldoet aan `ancestorString`;

(c) bepaal het inhoudsmodel van het XML element en valideer volgens de overeenstemmende DTDX regel;

(d) verifieer de attributen van het XML element;

(e) valideer op recursieve wijze de kinderen van deze knoop;

(f) verwijder de huidige knoop uit de variabele `ancestorString`.

Op analoge wijze kan er een instantie van de klasse `SAXEvaluator` worden aangemaakt. De constructor verwacht dezelfde parameter als de constructor van de `DOMEvaluator`. Ook hier wordt er impliciet een variabele `ancestorString` geïnitialiseerd om het pad van de wortel tot een bepaald XML element te registreren.

Het evaluatieproces wordt eveneens gestart door de method `parse()` aan te roepen maar verloopt volledig anders omdat `SAXEvaluator` gebaseerd is op de `DefaultHandler` API:

1. controle of er een geldig `DTDXSchema` object werd gespecificeerd;
2. initialisatie van een stack om de nesting van XML elementen op te volgen;
3. een instantie van `SAXParser` wordt gegenereerd;
4. de `parse()` method van de `SAXParser` instantie wordt aangeroepen met als parameters de lokatie van het opgegeven XML document en de huidige instantie van `SAXEvaluator` als `DefaultHandler`. De volgende `DefaultHandler` methods werden overschreven:

   - `startElement` (`ContentHandler`): detecteert het begin van een nieuw XML element:

     (a) voeg de naam van het element toe aan de variable `ancestorString`;

     (b) identificeer de DTDX regel die voldoet aan `ancestorString`;

     (c) haal het bovenste item van de stack. Bepaal de DTDX regel die met dit item overeenkomt, pas het item aan en plaats het terug op de stack. Een lege stack betekent dat het start element wordt verwerkt. Controleer of het huidige element voldoet aan de declaratie van het start element in het DTDX schema;

     (d) creëer op basis van het inhoudsmodel van het XML element een nieuw item om op de stack te plaatsen;

     (e) verifieer de attributen van het XML element.

- `endElement` (`ContentHandler`): detecteert het einde van een XML element;
  (a) haal het bovenste item van de stack. Bepaal de DTDX regel die met dit item overeenkomt en valideer het inhoudsmodel;
  (b) verwijder de naam van het element uit de variable `ancestorString`.
- `characters` (`ContentHandler`): detecteert karakters in het XML document. De booleaanse variabele `isRegisterContent` wordt zodanig ingesteld dat enkel karakters uit de inhoud van elementen met een primitief datatype of met een leeg inhoudsmodel worden geregistreerd;
- `setDocumentLocator` (`ContentHandler`): levert een SAX `Locator` af die op elk moment de precieze lokatie in het XML document registreert;
- `fatalError` (`ErrorHandler`): detecteert een onherstelbare fout waardoor de parser nutteloos is geworden. Er wordt een `EvaluationException` gegooid die de precieze omschrijving van de fout bevat;
- `error` (`ErrorHandler`): detecteert een fout waarvan de parser nog kan herstellen. Desalniettemin wordt er een `EvaluationException` gegooid. De fout wordt behandeld als een fatale fout;
- `warning` (`ErrorHandler`): detecteert gebeurtenissen die geen fouten of fatale fouten zijn. Standaard wordt er in dergelijke gevallen geen actie ondernomen omdat de parser doorgaans correct blijft functioneren. Om te verzekeren dat de `SAXEvaluator` geen foute controle uitvoert, wordt een `warning` behandeld alsof het een fatale fout is.

Een codefragment uit de DTDX Evaluator geeft weer hoe de instanties van `DOMEvaluator` en `SAXEvaluator` in de praktijk kunnen worden aangewend.

```
try {
    ...
    DTDXSchema schema = DTDXParser.parse(new
       FileInputStream(schemaFile.getAbsoluteFile
       ()), schemaFile.getName());
    ...

    // determine the evaluator type and evaluate
    // the document against the schema
    System.out.print("Evaluating␣XML␣document␣
       against␣DTDX␣schema...")

    if (DOM.equals(config.getString("type"))) {
```

```
            DOMEvaluator domEvaluator = new
               DOMEvaluator(schema);
            domEvaluator.parse(documentFile);
        } else if (SAX.equals(config.getString("type")
           )) {
            SAXEvaluator saxEvaluator = new
               SAXEvaluator(schema);
            saxEvaluator.parse(documentFile);
        }

        System.out.println("succesfull");
        ...
    } catch (FileNotFoundException fnfe) {
        ...
    } catch (parser.ParseException pe) {
        ...
    } catch (EvaluationException ee) {
        ...
    }
```

Het object `config` bevat het resultaat dat wordt gegenereerd door JSAP na het verwerken van de command-line.

Uitvoeren

Het uitvoeren van de DTDX Evaluator gebeurt analoog aan het uitvoeren van een andere Java toepassing:

```
Usage: java -jar evaluator.jar -t <type> [-v|--verbose]
       [-h|--help] <schema> <document>

  -t <type>
          Selects the evaluator type: DOM or SAX.

  [-v|--verbose]
          Requests verbose output. The internal data model
          for the schema is written to screen.

  [-h|--help]
          Prints this help message.

  <schema>
          The file containing the DTDX schema definition.

  <document>
          The file containing the XML document.
```

Het resultaat van de DTDX Evaluator voor het XML document in Figuur 4.5 en het DTDX schema in Figuur 4.4 wordt getoond in onderstaande uitvoer.

```
java -jar evaluator.jar -t sax store.txt store.xml

Parsing DTDX schema...succesfull
Evaluating XML document against DTDX schema...succesfull
```

6.6 DTDX Transformator

Naast een parser en een evaluator is er nog een derde tool beschikbaar voor DTDX, nl. een transformator. Deze tool is verantwoordelijk voor de vertaling van een DTDX schema naar een schema uitgedrukt in een andere XML schemataal. De DTDX Transformator werd zodanig ontwikkeld dat hij in staat is om elk DTDX schema zowel naar XML Schema als naar RELAX NG te transformeren.

De resulterende transformaties zijn opgesteld volgens de regels van het Venetian Blind design patroon, een patroon dat vaak wordt gebruikt tijdens het ontwerpen van nieuwe XML schema's [KS06]. Dit patroon schrijft voor dat het schema uit individuele componenten wordt opgebouwd. Er wordt slechts één element als globaal element gedeclareerd. De overige elementen worden als lokale types gedeclareerd. Venetian Blind garandeert dat er maar één start element voor het schema beschikbaar is, nl. het globale element. Daarnaast kunnen alle componenten onbeperkt worden hergebruikt. Hierin schuilt ook het nadeel van dit patroon. De gedeclareerde types worden niet altijd voldoende afgeschermd.

In het geval van XML Schema wordt het start element gedeclareerd als een `xsd:element`. De overige XML elementen in het schema worden, afhankelijk van hun inhoudsmodel, als `xsd:simpleType` of `xsd:complexType` gedeclareerd.

De transformatie naar RELAX NG resulteert in een schema dat begint met de declaratie van het start element gebruik makend van de tags `<start>` en `</start>`. Alle andere elementen in het schema worden gedeclareerd met de tags `<define>` en `</define>`.

Ter illustratie tonen Figuur 6.5 en Figuur 6.6 een fragment uit de transformatie van het DTDX schema in Figuur 4.4 naar respectievelijk XML Schema en RELAX NG.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="store" type="store-0-0-0-0-0-0-0-0" />
  <xsd:complexType name="store-0-0-0-0-0-0-0-0">
    <xsd:sequence>
      <xsd:element name="regulars"
           type="regulars-1-1-1-2_1-1-1-1-1" />
      <xsd:element name="discounts"
           type="discounts-1-1-1-2_1-1-1-1-1" />
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:integer"
         use="required" />
    <xsd:attribute name="name" type="xsd:string"
         default="unknown" />
  </xsd:complexType>
  ...
  <xsd:complexType name="dvd-1-1-2_1-1-1-JUNK-1-2">
    <xsd:sequence>
      <xsd:element name="title"
           type="title-1-1-1-1-1-JUNK-1-3" />
      <xsd:element name="price"
           type="price-1-1-1-1-1-JUNK-1-3" />
    </xsd:sequence>
  </xsd:complexType>
  ...
  <xsd:simpleType name="title-1-1-1-1-1-JUNK-1-3">
    <xsd:restriction base="xsd:string" />
  </xsd:simpleType>
  ...
</xsd:schema>
```

Figuur 6.5: Transformatie van Figuur 4.4 naar XML Schema

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
    datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
    xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0">
  <start>
    <element name="store">
      <ref name="store-0-0-0-0-0-0-0-0" />
    </element>
  </start>
  <define name="store-0-0-0-0-0-0-0-0">
    <attribute name="id">
      <data type="integer" />
    </attribute>
    <optional>
      <attribute name="name">
        <choice>
          <value>unknown</value>
          <data type="string" />
        </choice>
      </attribute>
    </optional>
    <group>
      <element name="regulars">
        <ref name="regulars-1-1-1-2_1-1-1-1-1" />
      </element>
      <element name="discounts">
        <ref name="discounts-1-1-1-2_1-1-1-1-1" />
      </element>
    </group>
  </define>
  ...
  <define name="title-1-1-1-1-1-JUNK-1-3">
    <data type="string" />
  </define>
  ...
</grammar>
```

Figuur 6.6: Transformatie van Figuur 4.4 naar RELAX NG

Broncode

Alle klassen gerelateerd aan de DTDX Transformator maken deel uit van het package `transformator`. Er zijn twee versies beschikbaar die enkel van elkaar verschillen in de wijze waarop de voorouderautomaat tot stand wordt gebracht. Het algemene gedrag ligt vast in de interface `Transformator`.

Elke versie is opgenomen in een abstracte klasse, `AbstractTransformator` respectievelijk `AbstractTransformatorOpt`, die voorziet in de algemene afhandeling van de transformatie, m.a.w. het opstellen en doorlopen van de voorouderautomaat. Daarnaast bevat het package een klasse `XXXTransformator` en/of `XXXTransformatorOpt`, afgeleid van de overeenkomstige abstracte klasse, die verantwoordelijk is voor de taalspecifieke implementatie en dit voor elke XML schemataal `XXX` die wordt ondersteund. Indien er zich een fout voordoet tijdens de transformatie, wordt er een `TransformationException` gegooid die een nauwkeurige omschrijving geeft van het probleem.

De transformatie wordt gestart door de method `transform()` aan te roepen. Deze method verwacht een `DTDXSchema` en een `Writer` als parameters. Nadat het nodige is gebeurd om de transformatie te kunnen wegschrijven, worden de volgende stappen doorlopen:

1. schrijf de sequentie die het schema opent;
2. schrijf de declaratie van het start element;
3. bezoek elke toestand van de voorouderautomaat. Schrijf voor elke toestand de overeenkomstige declaratie. Hierbij wordt rekening gehouden met eventuele attributen;
4. schrijf de sequentie die het schema sluit.

Onderstaand codefragment uit de broncode van de DTDX Transformator toont hoe een instantie van `XMLTransformator` en `RNGTransformator` kunnen worden aangemaakt. Het object `config` bevat het resultaat dat wordt gegenereerd door JSAP na het verwerken van de command-line.

```
try {
    ...
    DTDXSchema schema = DTDXParser.parse(new
        FileInputStream(schemaFile.getAbsoluteFile
        ()), schemaFile.getName());
    ...

    // determine the transformator type and
        transform the schema
    ...
```

```
        if (RNG.equals(config.getString("type"))) {
            RNGTransformator rngTransformator = new
               RNGTransformator();
            rngTransformator.transform(schema, new
               FileWriter(transformationFile));
        } else if (XSD.equals(config.getString("type")
           )) {
            XSDTransformator xsdTransformator = new
               XSDTransformator();
            xsdTransformator.transform(schema, new
               FileWriter(transformationFile));
        }
        ...
    } catch (FileNotFoundException fnfe) {
        ...
        System.exit(1);
    } catch (IOException ioe) {
        ...
    } catch (parser.ParseException pe) {
        ...
    } catch (TransformationException te) {
        ...
    }
```

Uitvoeren

Het uitvoeren van de (geoptimaliseerde) DTDX Transformator gebeurt analoog aan het uitvoeren van een andere Java toepassing:

```
Usage: java -jar transformator.jar -t <type> [-v|--verbose]
       [-h|--help] <schema> <transformation>

  -t <type>
          Selects the transformator type: RNG or XSD.

  [-v|--verbose]
          Requests verbose output. The internal data model
          for the schema is written to screen.

  [-h|--help]
          Prints this help message.

  <schema>
          The file containing the DTDX schema definition.

  <transformation>
          The destination file for the transformation.
```

Onderstaande uitvoer toont het resultaat van de DTDX Transformator voor de transformatie van het DTDX schema in Figuur 4.4.

```
java -jar transformator.jar -t xsd store.txt store.xsd

Parsing DTDX schema...succesfull
Transforming DTDX schema...succesfull
```

Hoofdstuk 7

Resultaten

7.1 Correcte werking

Het is van cruciaal belang dat een toepassing correct werkt en de juiste resultaten aflevert. Dit kan worden verzekerd door de toepassing tijdens en na de ontwikkeling aan uitgebreide testen te onderwerpen.

Tijdens de implementatie van de DTDX tools werd er uitvoerig gebruik gemaakt van JUnit [Gam06] om te verzekeren dat elk onderdeel van de broncode naar behoren werkt. Na het afronden van de implementatie werden er voor elke tool bijkomende testen uitgevoerd om na te gaan dat de afgeleverde resultaten juist zijn. Hiertoe werd een verzameling testdocumenten opgesteld die alle eigenschappen van een DTDX schema aan bod laten komen.

7.2 Testdocumenten

Bij het opstellen van de testdocumenten werd er hoofdzakelijk aandacht besteed aan de ancestor-strings die worden gebruikt in de element- en typedeclaraties. Overige eigenschappen, zoals correcte waarden voor datatypes en lege inhoudsmodellen, komen uiteraard ook aan bod maar zijn in zekere mate triviaal.

De verzameling testdocumenten bestaat uit vijf DTDX schema's die elk een specifieke basiseigenschap betreffende ancestor-strings beschrijven. De combinatie van de eigenschappen van twee of meer schema's leidt tot een schema van min of meer onbeperkte complexiteit. De schema's worden getoond in Figuur 7.1, Figuur 7.2, Figuur 7.3, Figuur 7.4 en Figuur 7.5. Het spreekt voor zich dat elk van deze schema's voldoet aan de syntactische en semantische regels.

```
<!START a>
<!ELEMENT a (b.c)>
<!ELEMENT b (d.e?)> <!-- d_1 -->
<!ELEMENT c (d.e?)> <!-- d_2 -->
<!TYPE d_1 ($*.b.d) (f.g)> <!-- f_1 -->
<!TYPE d_2 ($*.c.d) (f.g)> <!-- f_2 -->
<!TYPE f_1 ($*.b.d.f) (h)>
<!TYPE f_2 ($*.c.d.f) (i)>
<!ELEMENT e EMPTY>
<!ELEMENT g EMPTY>
<!ELEMENT h EMPTY>
<!ELEMENT i EMPTY>
```

Figuur 7.1: DTDX schema met grootouders

```
<!START a>
<!ELEMENT a (b.c)>
<!ELEMENT b (b?.d.e?)> <!-- d_1 -->
<!ELEMENT c (c?.d.e?)> <!-- d_2 -->
<!TYPE d_1 ($*.b.d) (f)>
<!TYPE d_2 ($*.c.d) (g)>
<!ELEMENT e EMPTY>
<!ELEMENT f EMPTY>
<!ELEMENT g EMPTY>
```

Figuur 7.2: Recursief DTDX schema met ouders

```
<!START a>
<!ELEMENT a (b.c)>
<!ELEMENT b (b?.d.e?)> <!-- d_1 -->
<!ELEMENT c (c?.d.e?)> <!-- d_2 -->
<!TYPE d_1 ($*.b.d) (f.g)> <!-- f_1 -->
<!TYPE d_2 ($*.c.d) (f.g)> <!-- f_2 -->
<!TYPE f_1 ($*.b.d.f) (h)>
<!TYPE f_2 ($*.c.d.f) (i)>
<!ELEMENT e EMPTY>
<!ELEMENT g EMPTY>
<!ELEMENT h EMPTY>
<!ELEMENT i EMPTY>
```

Figuur 7.3: Recursief DTDX schema met grootouders

```
<!START a>
<!TYPE a_1 ((a.a)*.a) (a.b)?> <!-- a_2 -->
<!TYPE a_2 ((a.a)*.a.a) (a.c)?> <!-- a_1 -->
<!ELEMENT b EMPTY>
<!ELEMENT c EMPTY>
```

Figuur 7.4: Recursief DTDX schema met even en oneven diepte

```
<!START a>
<!ELEMENT a (b.c)>
<!ELEMENT b (d.e?)> <!-- d_1 -->
<!ELEMENT c (d.e?)> <!-- d_2 -->
<!TYPE d_1 ($.b.$*.d) (d?)> <!-- d_1 -->
<!TYPE d_2 ($.c.$*.d) (d?)> <!-- d_2 -->
<!ELEMENT e EMPTY>
```

Figuur 7.5: Recursief DTDX schema met voorouders

7.3 Testen

De taak van de DTDX Parser is vrij eenvoudig. Correcte schema's worden aanvaard en foutieve schema's worden verworpen. Om na te gaan of de juiste resultaten worden afgeleverd, worden eerst de vijf testschema's ter verwerking aangeboden. Het spreekt voor zich dat elk van deze schema's word aanvaard door de DTDX Parser. Vervolgens worden, van elk van de schema's, enkele duplicaten gemaakt. Elk duplicaat wordt aangepast zodat er een specifieke fout aanwezig is. De duplicaten worden aangeboden aan de DTDX Parser die elk duplicaat verwerpt en dit met de aangebrachte fout als reden. Hieruit kan worden geconcludeerd dat de DTDX Parser naar behoren werkt.

De correcte werking van de DTDX Evaluator kan pas worden geverifieerd zodra er voor elk van de vijf testschema's minstens één representatief XML document beschikbaar is. Er wordt eerst getest met XML documenten waarvan zeker is dat ze voldoen aan de DTDX schema's. Deze documenten worden verkregen met behulp van twee XML generators [Bex07, BMKL02]. Voor alle documenten wordt het juiste resultaat afgeleverd. Vervolgens worden er, zoals bij het testen van de DTDX Parser, duplicaten gemaakt van de XML documenten zodat elk duplicaat een specifieke fout bezit. Bij de verwerking van de duplicaten, wordt elk duplicaat geweigerd waarbij de aangebrachte fout als reden wordt opgegeven. De DTDX Evaluator levert voor alle geteste documenten een juist resultaat af en werkt dus correct.

De DTDX Transformator kan onmiddellijk worden getest met de testdocumenten. Elk DTDX schema wordt door beide versies omgezet naar een XSD en een RNG. Om na te gaan of de transformaties correct zijn verlopen, worden de bekomen XSDs en RNGs gecontroleerd. Hiervoor wordt een beroep gedaan op Topologi Schematron Validator [Top06], Jing [Cla06] en XSV [W3C06]. Deze validators tonen aan dat de getransformeerde schema's geldige XML Schema en RELAX NG schema's zijn. Bijgevolg kan worden vastgesteld dat de DTDX Transformator correct functioneert.

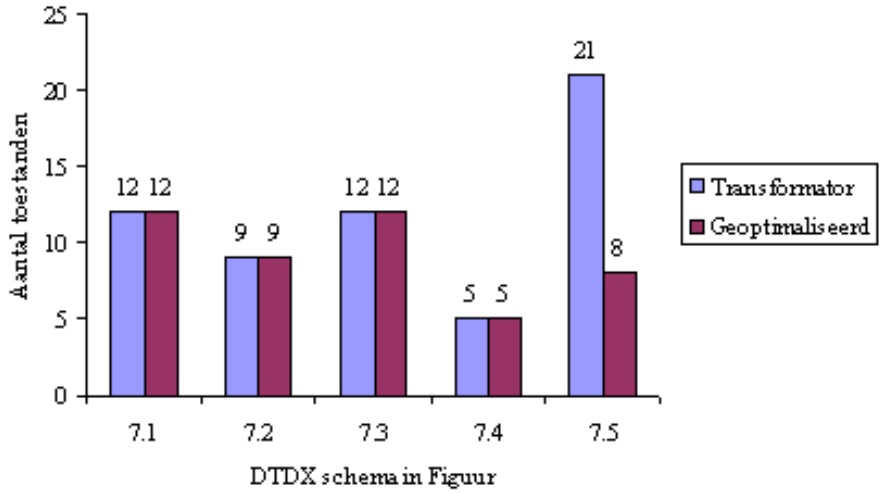
7.4 Performantie

De DTDX transformator en de geoptimaliseerde DTDX Transformator verschillen van elkaar in de wijze waarop de voorouderautomaat tot stand wordt gebracht. Zoals de naam al aangeeft, zal de geoptimaliseerde versie sneller tot een resultaat komen. Dit komt omdat de DTDX Transformator eerst de volledige voorouderautomaat opstelt, zoals wordt beschreven in Sectie 5.4, en vervolgens de automaat op gepaste wijze doorloopt. In bepaalde gevallen zal de voorouderautomaat die op deze manier wordt bekomen heel wat toestanden en transities bevatten die nooit zullen worden bezocht. De geoptimaliseerde DTDX Transformator daarentegen stelt de voorouderautomaat on-the-fly op zodat enkel de toestanden en transities worden berekend die nodig zijn om tot een correcte transformatie te komen.

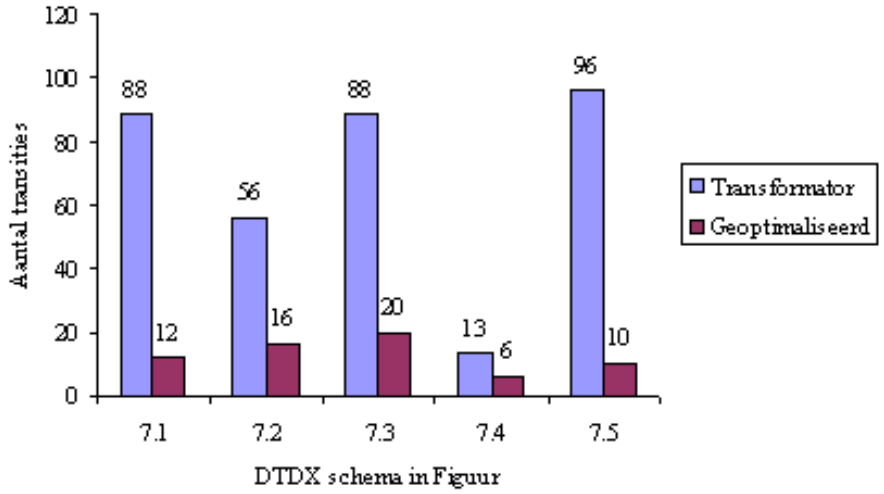
Voor elk van de vijf schema's die werden gebruikt bij het testen van de DTDX tools, werd nagegaan hoeveel toestanden en transities er worden berekend tijdens de transformatie naar XML Schema. De resultaten worden getoond in Figuur 7.6 en Figuur 7.7. Hieruit blijkt dat de DTDX Transformator en de geoptimaliseerde DTDX Transformator in het algemeen hetzelfde aantal toestanden nodig hebben. Het aantal benodigde transities daarentegen ligt bij de geoptimaliseerde DTDX Transformator aanzienlijk lager.

7.5 Leesbaarheid

Beschouw de getransformeerde schema's in Figuur 6.5 en Figuur 6.6. Merk op dat de lengte van de namen van types in een getransformeerd schema afhankelijk is van het aantal ancestor-strings in het DTDX schema. Hoe meer ancestor-strings, hoe langer de namen van types worden. Bij uitgebreide schema's heeft dit een negatieve invloed op de leesbaarheid. Dit kan worden verholpen door, na de transformatie, het verkregen schema op te kuisen. De namen van types kunnen op een alternatieve, kortere manier worden voorgesteld door per element het aantal types te tellen en vervolgens dat resultaat als naam voor het type te gebruiken.



Figuur 7.6: Het aantal toestanden die tijdens de transformatie worden berekend



Figuur 7.7: Het aantal transities die tijdens de transformatie worden berekend

Bij wijze van voorbeeld worden de opgekuiste transformaties van de schema's in Figuur 7.1, Figuur 7.2, Figuur 7.3, Figuur 7.4 en Figuur 7.5 weergegeven in Figuur 7.8, Figuur 7.9, Figuur 7.10, Figuur 7.11 en Figuur 7.12.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="a" type="a-1" />
  <xsd:complexType name="a-1">
    <xsd:sequence>
      <xsd:element name="b" type="b-1" />
      <xsd:element name="c" type="c-1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="b-1">
    <xsd:sequence>
      <xsd:element name="d" type="d-1" />
      <xsd:sequence>
        <xsd:element name="e" type="e-1"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="c-1">
    <xsd:sequence>
      <xsd:element name="d" type="d-2" />
      <xsd:sequence>
        <xsd:element name="e" type="e-2"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="d-1">
    <xsd:sequence>
      <xsd:element name="f" type="f-1" />
      <xsd:element name="g" type="g-1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="e-1" />
  <xsd:complexType name="d-2">
    <xsd:sequence>
      <xsd:element name="f" type="f-2" />
      <xsd:element name="g" type="g-2" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="e-2" />
  <xsd:complexType name="f-1">
    <xsd:sequence>
      <xsd:element name="h" type="h-1" />
```

```
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="g-1" />
  <xsd:complexType name="f-2">
    <xsd:sequence>
      <xsd:element name="i" type="i-1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="g-2" />
  <xsd:complexType name="h-1" />
  <xsd:complexType name="i-1" />
</xsd:schema>
```

Figuur 7.8: XSD met grootouders

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="a" type="a-1" />
  <xsd:complexType name="a-1">
    <xsd:sequence>
      <xsd:element name="b" type="b-1" />
      <xsd:element name="c" type="c-1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="b-1">
    <xsd:sequence>
      <xsd:sequence>
        <xsd:element name="b" type="b-2"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element name="d" type="d-1" />
        <xsd:sequence>
          <xsd:element name="e" type="e-1"
               minOccurs="0" maxOccurs="1" />
        </xsd:sequence>
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="c-1">
    <xsd:sequence>
      <xsd:sequence>
        <xsd:element name="c" type="c-2"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element name="d" type="d-2" />
```

```
      <xsd:sequence>
        <xsd:element name="e" type="e-2"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="b-2">
  <xsd:sequence>
    <xsd:sequence>
      <xsd:element name="b" type="b-2"
        minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="d" type="d-1" />
      <xsd:sequence>
        <xsd:element name="e" type="e-1"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="d-1">
  <xsd:sequence>
    <xsd:element name="f" type="f-1" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="e-1" />
<xsd:complexType name="c-2">
  <xsd:sequence>
    <xsd:sequence>
      <xsd:element name="c" type="c-2"
           minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="d" type="d-2" />
      <xsd:sequence>
        <xsd:element name="e" type="e-2"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="d-2">
  <xsd:sequence>
    <xsd:element name="g" type="g-1" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="e-2" />
```

```
  <xsd:complexType name="f-1" />
  <xsd:complexType name="g-1" />
</xsd:schema>
```

Figuur 7.9: Recursieve XSD met ouders

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="a" type="a-1" />
  <xsd:complexType name="a-1">
    <xsd:sequence>
      <xsd:element name="b" type="b-1" />
      <xsd:element name="c" type="c-1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="b-1">
    <xsd:sequence>
      <xsd:sequence>
        <xsd:element name="b" type="b-2"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element name="d" type="d-1" />
        <xsd:sequence>
          <xsd:element name="e" type="e-1"
               minOccurs="0" maxOccurs="1" />
        </xsd:sequence>
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="c-1">
    <xsd:sequence>
      <xsd:sequence>
        <xsd:element name="c" type="c-2"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element name="d" type="d-2" />
        <xsd:sequence>
          <xsd:element name="e" type="e-2"
               minOccurs="0" maxOccurs="1" />
        </xsd:sequence>
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="b-2">
    <xsd:sequence>
      <xsd:sequence>
```

```
      <xsd:element name="b" type="b-2"
           minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="d" type="d-1" />
      <xsd:sequence>
        <xsd:element name="e" type="e-1"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="d-1">
  <xsd:sequence>
    <xsd:element name="f" type="f-1" />
    <xsd:element name="g" type="g-1" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="e-1" />
<xsd:complexType name="c-2">
  <xsd:sequence>
    <xsd:sequence>
      <xsd:element name="c" type="c-2"
           minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
    <xsd:sequence>
      <xsd:element name="d" type="d-2" />
      <xsd:sequence>
        <xsd:element name="e" type="e-2"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="d-2">
  <xsd:sequence>
    <xsd:element name="f" type="f-2" />
    <xsd:element name="g" type="g-2" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="e-2" />
<xsd:complexType name="f-1">
  <xsd:sequence>
    <xsd:element name="h" type="h-1" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="g-1" />
<xsd:complexType name="f-2">
  <xsd:sequence>
```

```
      <xsd:element name="i" type="i-1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="g-2" />
  <xsd:complexType name="h-1" />
  <xsd:complexType name="i-1" />
</xsd:schema>
```

Figuur 7.10: Recursieve XSD met grootouders

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="a" type="a-1" />
  <xsd:complexType name="a-1">
    <xsd:sequence minOccurs="0" maxOccurs="1">
      <xsd:element name="a" type="a-2" />
      <xsd:element name="b" type="b-1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="a-2">
    <xsd:sequence minOccurs="0" maxOccurs="1">
      <xsd:element name="a" type="a-3" />
      <xsd:element name="c" type="c-1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="b-1" />
  <xsd:complexType name="a-3">
    <xsd:sequence minOccurs="0" maxOccurs="1">
      <xsd:element name="a" type="a-2" />
      <xsd:element name="b" type="b-1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="c-1" />
</xsd:schema>
```

Figuur 7.11: Recursieve XSD met even en oneven diepte

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="a" type="a-1" />
  <xsd:complexType name="a-1">
    <xsd:sequence>
      <xsd:element name="b" type="b-1" />
      <xsd:element name="c" type="c-1" />
    </xsd:sequence>
  </xsd:complexType>
```

```
  <xsd:complexType name="b-1">
    <xsd:sequence>
      <xsd:element name="d" type="d-1" />
      <xsd:sequence>
        <xsd:element name="e" type="e-1"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="c-1">
    <xsd:sequence>
      <xsd:element name="d" type="d-2" />
      <xsd:sequence>
        <xsd:element name="e" type="e-2"
             minOccurs="0" maxOccurs="1" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="d-1">
    <xsd:sequence>
      <xsd:element name="d" type="d-3"
           minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="e-1" />
  <xsd:complexType name="d-2">
    <xsd:sequence>
      <xsd:element name="d" type="d-4"
           minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="e-2" />
  <xsd:complexType name="d-3">
    <xsd:sequence>
      <xsd:element name="d" type="d-3"
           minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="d-4">
    <xsd:sequence>
      <xsd:element name="d" type="d-4"
           minOccurs="0" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Figuur 7.12: Recursieve XSD met voorouders

Hoofdstuk 8

Besluit

Document Type Definitions (DTDs) werden gelijktijdig met XML geïntroduceerd. Omwille van de lange voorgeschiedenis (DTDs werden reeds gebruikt om de structuur van SGML documenten te beschrijven) en de eenvoudige syntax hebben DTDs tot op heden grote successen geboekt. Na verloop van tijd kwamen echter gebreken aan het licht zoals het beperkte aantal voorgedefinieerde datatypes en het gebrek aan ondersteuning voor XML Namespaces. Om tegemoet te komen aan die beperkingen werd XML Schema ontwikkeld, een schemataal die veel krachtiger is dan DTDs. Zo krachtig zelfs dat de precieze uitdrukkingskracht, en de geschiktheid van die uitdrukkingskracht, niet altijd even duidelijk zijn.

Een studie heeft aangetoond dat de meeste XML Schema schema's die in de praktijk voorkomen gemakkelijk kunnen worden gedefinieerd als een extensie van DTDs. De bevindingen in deze studie hebben geleid tot de ontwikkeling van DTD Extra (DTDX). Het is een concrete, voorouder-gebaseerde schemataal die de uitdrukkingskracht van XML Schema, meerbepaald het typeringsmechanisme, tracht te vatten met behulp van een zo eenvoudig mogelijke syntax die is gebaseerd op de constructies van DTDs.

De syntax van DTDX wordt formeel gedefinieerd door de contextvrije grammatica in Sectie 4.3. Een schema begint steeds met de declaratie van het start element. Hierop volgt een verzameling regels die de toegelaten elementen declareren. Elke declaratie specificeert de naam, de ancestor-string en de child-string van het element. Om te voorkomen dat er ambiguïteit ontstaat bij de toekennig van types aan elementen, wordt de beperking opgelegd dat voor elke twee regels geldt dat de doorsnede van de ancestor-strings leeg moet zijn. Het schema wordt afgesloten door een optionele verzameling declaraties die bepalen welke attributen worden toegekend aan welke regel. Een attribuut bevat bijkomende informatie over het element waartoe het behoort.

Om een schemataal te kunnen gebruiken in de praktijk zijn er toepassingen nodig die de opgestelde schema's kunnen verwerken. Een correcte werking kan worden gegarandeerd door de methodes en algoritmes die de toepassingen hanteren op een formele wijze te onderbouwen. Daarnaast is het gebruik van uitgebreide testen tijdens en na het ontwikkelingsproces aangeraden. In het geval van DTDX zijn er momenteel drie tools beschikbaar:

- DTDX Parser: controleert de correctheid van een schema. Zowel de syntax als de beperking inzake ancestor-strings worden in acht genomen;
- DTDX Evaluator: gaat na of een XML document voldoet aan het opgegeven DTDX schema;
- DTDX Transformator: zet een schema om in een equivalent XML Schema schema of RELAX NG schema.

Elke tool werd ontwikkeld met de programmeertaal Java en kan worden uitgevoerd vanaf de command-line.

Hoewel de nodige elementen voorhanden zijn om DTD Extra in de praktijk toe te passen, is er steeds ruimte voor verbetering. Denk hierbij aan ondersteuning voor XML Namespaces zodat het mogelijk wordt om diverse DTDX schema's te combineren zoals dit gebeurt in XML Schema. Een andere mogelijkheid is de integratie van co-constraints naar analogie met DTD++ 2.0.

Bibliografie

[ABM+98] Vidur Apparao, Steve Byrne, Champion Mike, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, and Lauren Wood. Document Object Model (DOM) Level 1 Specification. W3C Recommendation, `http://www.w3.org/TR/REC-DOM-Level-1/`, 1998.

[Arm01] Eric Armstrong. Working with XML: The Java API for Xml Processing (JAXP) Tutorial. Sun Micro Systems, Inc., `http://java.sun.com/webservices/jaxp/dist/1.1/docs/tutorial/index.html`, 2001.

[Aus04] Calvin Austin. J2SE 5.0 in a Nutshell. Sun Micro Systems, Inc., `http://java.sun.com/developer/technicalArticles/releases/j2se15/`, 2004.

[Bex07] Geert Jan Bex. XMLGenerator, 2007.

[BM04] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, `http://www.w3.org/TR/xmlschema-2/`, 2004.

[BMKL02] Denilson Barbosa, Alberto O. Mendelzon, John Keenleyside, and Kelly A. Lyons. ToXgene: An extensible template-based data generator for XML. In *WebDB*, pages 49–54, 2002.

[BMNS05] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In Ellis and Hagino [EH05], pages 712–721.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation, `http://www.w3.org/TR/REC-xml/`, 2006.

[Cla01] James Clark. The Design of RELAX NG. `http://www.thaiopensource.com/relaxng/design.html`, 2001.

[Cla06] Clark, James. Jing - A RELAX NG validator in Java. Thai Open Source Software Center, Ltd., `http://www.thaiopensource.com/relaxng/jing.html`, 2006.

[CM01] James Clark and Murata Makoto. RELAX NG Specification. `http://relaxng.org/spec-20011203.html`, 2001.

[EH05] Allan Ellis and Tatsuya Hagino, editors. *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*. ACM, 2005.

[FMGV04] Davide Fiorello, Paolo Marinelli, Nicola Gessa, and Fabio Vitali. DTD++ 2.0: Adding support for co-constraints. In *Extreme Markup Languages*, 2004.

[fS06] International Organization for Standardization. Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron, 2006.

[FW04] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. W3C Recommendation, `http://www.w3.org/TR/xmlschema-0/`, 2004.

[Gam06] Gamma, Erich and Beck, Kent. JUnit. `http://www.junit.org/index.htm`, 2006.

[GM96] James Gosling and Henry McGilton. Java Language Environment. Sun Micro Systems, Inc., `http://java.sun.com/docs/white/langenv/index.html`, 1996.

[jav06] java.net. JavaCC Home. java.net, `https://javacc.dev.java.net/`, 2006.

[Jel06] Rick Jellife. Academia Sinica Computing Centre's Schematron Home Page. `http://www.ascc.net/xml/schematron`, 2006.

[KS06] Ayub Khan and Marina Sum. Introducing Design Patterns in XML Schemas. Sun Micro Systems, Inc., `http://developers.sun.com/jsenterprise/nb_enterprise_pack/reference/techart/design_patterns.html`, 2006.

[Lam06] Marty Lamb. JSAP - Java Simple Argument Parser (v2.1). Martian Software, Inc., `http://www.martiansoftware.com/jsap/doc/index.html`, 2006.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[MSCV04] Paolo Marinelli, Claudio Sacerdoti Coen, and Fabio Vitali. SchemaPath: Extending XML Schema for Co-Constraints. Technical report, Department Of Computer Science, University of Bologna, 2004.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[Sun06] Sun. Packaging Programs in JAR Files. Sun Micro Systems, Inc., `http://java.sun.com/docs/books/tutorial/deployment/jar/index.html`, 2006.

[TBMM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures Second Edition. W3C Recommendation, `http://www.w3.org/TR/xmlschema-1/`, 2004.

[Top06] Topologi. Topologi Schema Validator. Topologi, `http://www.topologi.com/products/validator`, 2006.

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

[vdV04] Eric van der Vlist. *RELAX NG*. O'Reilly & Associates, Inc., 2004.

[VGA03] Fabio Vitali, Nicola Gessa, and Nicola Amorosi. Datatype- and namespace-aware DTDs: A minimal extension. In *Extreme Markup Languages*, 2003.

[W3C06] W3C. Validator for XML Schema (XSV version 2.10-1). W3C, `http://www.w3.org/2001/03/webdata/xsv`, 2006.

[Wik06a] Wikipedia. Document Type Definition. Wikipedia, The Free Encyclopedia, `http://en.wikipedia.org/wiki/Document_Type_Definition`, 2006.

[Wik06b] Wikipedia. Extensible Markup Language. Wikipedia, The Free Encyclopedia, `http://en.wikipedia.org/wiki/XML`, 2006.

[Wik06c] Wikipedia. Parsing. Wikipedia, The Free Encyclopedia, `http://en.wikipedia.org/wiki/Parsing`, 2006.

[Wik06d] Wikipedia. RELAX NG. Wikipedia, The Free Encyclopedia, `http://en.wikipedia.org/wiki/RELAX_NG`, 2006.

[Wik06e] Wikipedia. Simple API for XML. Wikipedia, The Free Encyclopedia, `http://en.wikipedia.org/wiki/Simple_API_for_XML`, 2006.

[Wik06f] Wikipedia. Standard Generalized Markup Language. Wikipedia, The Free Encyclopedia, `http://en.wikipedia.org/wiki/SGML`, 2006.

[Wik06g] Wikipedia. XML Schema. Wikipedia, The Free Encyclopedia, `http://en.wikipedia.org/wiki/XML`, 2006.

Auteursrechterlijke overeenkomst

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

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Een front-end voor XML Schema
Richting: **Master in de informatica** Jaar: **2007**
in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

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

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

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

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

Ik ga akkoord,

Jan Olaerts

Datum: **22.05.2007**