# Università degli Studi di Bologna

**FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI**

**Corso di Laurea in Informatica**


# Un'estensione Object Oriented di un linguaggio di validazione per XML


Tesi di Laurea di:
Giulia Gambini

Relatore:
Chiar.mo Prof. Fabio Vitali



**II Sessione**
**Anno Accademico 2004-2005**

*A nonno*
*sempre con me*

*On ne voit bien qu'avec le cœur.*
*L'essentiel est invisible pour les yeux.*

*Antoine de Saint-Exupéry*

# Indice

**1 Introduzione** **1**

**2 Linguaggi di validazione per XML** **11**
2.1 DTD . . . . . 14
2.2 XML Schema . . . . . 14
2.3 RELAX NG . . . . . 15
2.4 Schematron . . . . . 16
2.5 xlinkit . . . . . 17
2.6 DTD++ . . . . . 17
2.7 Verso il paradigma Object Oriented . . . . . 18

**3 Linguaggi di validazione Object Oriented per XML 21**
3.1 L'approccio Object Oriented . . . . . 21
3.1.1 La classe . . . . . 21
3.1.2 L'oggetto . . . . . 22
3.1.3 Il tipo . . . . . 22
3.1.4 Il sottotipo . . . . . 22
3.1.5 L'ereditarietà . . . . . 23
3.1.6 Il polimorfismo e il binding dinamico . . . 24
3.2 L'approccio Object Oriented in XML Schema . . 26
3.2.1 I tipi . . . . . 26
3.2.1.1 I tipi semplici . . . . . 26
3.2.1.2 I tipi complessi . . . . . 27
3.2.2 La derivazione . . . . . 28

3.2.2.1 Derivazione dei tipi semplici . . . 28
3.2.2.2 Derivazione dei tipi complessi . . 30
3.2.3 Componenti locali e globali . . . . . 32
3.2.4 SubstitutionGroup . . . . . 35
3.3 Stili di progettazione XML Schema . . . . . 37
3.3.1 Russian Doll . . . . . 37
3.3.2 Salami Slice . . . . . 39
3.3.3 Venetian Blind . . . . . 41
3.3.4 Garden of Eden . . . . . 42

**4 Un'estensione Object Oriented di DTD++ 45**
4.1 Introduzione . . . . . 45
4.2 I tipi . . . . . 47
4.2.1 Tipi semplici . . . . . 47
4.2.2 Tipi complessi . . . . . 48
4.3 Derivazione . . . . . 48
4.3.1 Derivazione dei tipi semplici . . . . . 49
4.3.1.1 Restrizione . . . . . 49
4.3.1.2 Lista . . . . . 49
4.3.1.3 Unione . . . . . 50
4.3.2 Derivazione dei tipi complessi . . . . . 52
4.3.2.1 Restrizione . . . . . 52
4.3.2.2 Estensione . . . . . 53
4.4 Componenti locali e globali . . . . . 53
4.4.1 Componenti locali . . . . . 54
4.4.2 Componenti globali . . . . . 55
4.4.3 Elemento radice . . . . . 55
4.5 Estensioni Object Oriented . . . . . 57
4.5.1 Element e Attribute Groups . . . . . 58
4.5.2 SubstitutionGroup . . . . . 59
4.6 Limiti del linguaggio . . . . . 60

**5 PreValidator for DTD++ 63**
5.1 Introduzione . . . . . 63
5.2 Conversione a DTD . . . . . 66
5.2.1 I tipi . . . . . 66
5.2.1.1 I tipi semplici . . . . . 67
5.2.1.2 I tipi complessi . . . . . 68
5.2.2 La derivazione . . . . . 69
5.2.2.1 Derivazione dei tipi semplici . . . 69
5.2.2.2 Derivazione dei tipi complessi . . 70
5.2.3 Componenti locali e globali . . . . . 71
5.2.4 Element e Attribute Groups . . . . . 72
5.2.5 SubstitutionGroup . . . . . 72
5.3 Vantaggi . . . . . 81
5.4 Limiti . . . . . 83

**6 Dettagli implementativi 85**
6.1 Flusso del programma . . . . . 85
6.2 Tecnologie utilizzate . . . . . 86
6.2.1 JFlex . . . . . 87
6.2.2 JavaCup . . . . . 87
6.2.3 MSXML . . . . . 87
6.2.4 Xerces . . . . . 88
6.3 Il programma . . . . . 88
6.3.1 preValidator . . . . . 89
6.3.2 preValidator.documents . . . . . 89
6.3.3 preValidator.types . . . . . 90
6.3.4 preValidator.errors . . . . . 92
6.3.5 preValidator.parsers . . . . . 93
6.3.6 preValidator.parserStruct . . . . . 93
6.3.7 preValidator.utils . . . . . 102
6.4 Utilizzo e installazione . . . . . 103
6.4.1 Utilizzo . . . . . 103
6.4.2 Installazione . . . . . 106

**7 Sviluppi futuri** **109**

**Bibliografia** **113**

**A Riassunto delle dichiarazioni DTD++** **117**

**Ringraziamenti** **123**

# Capitolo 1

# Introduzione

L'oggetto di questa tesi è dimostrare che DTD++ può essere considerato un linguaggio di validazione Object Oriented per XML, andando a colmare il divario tra i linguaggi rule-based e grammar-based.

I linguaggi di validazione per XML permettono di specificare le regole tramite cui accertare la correttezza delle istanze di documenti XML. Grazie a questo processo è quindi possibile verificare l'aderenza di un documento a determinate regole e imporre una struttura comune ai vari documenti prodotti.

Inizialmente tali linguaggi, e in particolare DTD, sono stati utilizzati per ottenere una documentazione semplice e leggibile e per facilitare l'apprendimento del linguaggio e la creazione di tool quali ad esempio browser ed editor.

I linguaggi di validazione basati su XML hanno però portato ad un peggioramento in termini di leggibilità. Infatti la loro grammatica risulta verbosa e porta a scrivere documenti due o tre volte più lunghi e soprattutto più difficili da leggere e capire.

D'altra parte una grammatica basata su XML facilita la realizzazione di tool che trattano, trasformano e utilizzano specifiche sotto forma di schema, mentre far ricorso a DTD porta inevitabilmente ad essere legati ad un parser DTD senza poter

utilizzare tool più sofisticati per le applicazioni aggiuntive dello schema.

I linguaggi di validazione per XML possono essere valutati considerando differenti parametri. In [WC01] vengono confrontati DTD, XML Schema, RELAX NG e Schematron sulla base di cinque caratteristiche che analizzano punti di forza e debolezze dei linguaggi sopra indicati:

- **Content models and datatypes:** quanto sofisticate sono le regole per esprimere vincoli sulle strutture (il numero e l'ordine di elementi e attributi) e sui dati (valori permessi e di default).
- **Modularità:** quanto facilmente può uno schema complesso essere organizzato in moduli indipendenti e quanto è flessibile il riuso di tali moduli.
- **Namespace:** che genere di supporto per i namespace viene fornito e che genere di restrizioni possono essere imposte per qualificare elementi e attributi.
- **Linking:** quali relazioni esplicite possono essere espresse tra elementi e attributi dello stesso documento (ad esempio le relazioni ID/IDREF in DTD).
- **Co-constraints:** se è possibile esprimere vincoli su elementi e attributi basati sulla presenza o sui valori di altri attributi ed elementi.

Da questa analisi, Schematron risulta il linguaggio vincente soprattutto in quanto domina la categoria dei co-constraints. Infatti solo RELAX NG ne fornisce una limitata implementazione mentre nè DTD nè XML Schema li supportano. Tuttavia XML Schema include una vasta libreria di tipi predefiniti e permette sofisticati meccanismi di derivazione e sostituzione rientrando, così, nell'approccio Object Oriented.

Una classificazione effettuata nel dicembre 2001 dal progetto DSDL (Document Schema Definition Language) ISO[DSDL] permette di dividere i linguaggi di validazione in due categorie:

- **grammar-based languages:** descrivono la struttura di un documento formalizzando le regole di produzione che devono essere seguite. Le espressioni che vincolano gli elementi vengono chiamate tipi e corrispondono ai non-terminali della teoria degli automi. Di questo gruppo fanno parte DTD, XML schema, Relax NG (e DTD++).

- **rule-based languages:** forniscono l'insieme delle regole che un documento XML deve seguire dando una specifica aperta (tutto ciò che non è vietato è consentito) oppure chiusa (tutto ciò che non è consentito è vietato). A questo gruppo appartengono Schematron e xlinkit.

I linguaggi appartenenti alla prima categoria vincolano la struttura logica dei documenti XML attraverso espressioni grammaticali e permettono di descrivere in maniera dettagliata l'insieme di valori legali per elementi e attributi. Tuttavia sono limitati nella definizione di vincoli incrociati sulla presenza o assenza di elementi e attributi. DTD e XML Schema, ad esempio, non ne forniscono supporto e RELAX NG solo in maniera limitata.

Dall'altro lato i linguaggi rule-based definiscono meccanismi che consentono di esprimere co-constraints ma non permettono di esprimere allo stesso modo i vincoli strutturali risultando, dunque limitati nella definizione dei valori accettabili.

I linguaggi grammar-based, invece, permettono di esprimere vincoli sulla struttura del documento e ricorrono al concetto di tipo di dato. Tra questi linguaggi il primo è sicuramente **DTD** (Document Type Definition). Introdotto e standardizzato dall'XML 1.0 Recommendation, nasce come versione semplificata

dei DTD SGML. I documenti scritti in DTD risultano compatti e facilmente comprensibili e, grazie a particolari strutture (le **entità parametriche**), si raggiunge un buon grado di modularità e riuso. Tuttavia, essendo stato introdotto prima dell'avvento dei namespace, risulta difficile qualificare gli elementi. Inoltre DTD ha un'espressività limitata che non consente di imporre forti vincoli strutturali sui content model.

Un altro esempio di linguaggio grammar-based è RELAX NG che è uno standard ISO e nasce dall'unione di due linguaggi precedenti TREX e RELAX. Uno schema RELAX NG è caratterizzato da *pattern* che permettono di specificare la struttura e il contenuto di un documento XML. RELAX NG consente poi di realizzare schema modulari grazie a meccanismi di importazione di documenti esterni.

**XML Schema** si pone come linguaggio che va ben oltre l'essere grammar-based. Infatti è un tentativo di descrivere istanze di documenti in un modo il più possibile simile all'approccio Object Oriented. È possibile definire una serie di stringhe (tipi semplici) e associarle sia agli elementi sia agli attributi.

Inoltre XML Schema permette di definire le strutture (tipi complessi) che permettono così di imporre vincoli strutturali sul content model e possono quindi essere assimilati alle classi del paradigma Object Oriented. Inoltre i meccanismi di derivazione consentono di ottenere nuovi tipi che possono essere ulteriormente derivati andando a creare una gerarchia di tipi in maniera analoga a quanto avviene per le classi dell'approccio Object Oriented.

I gruppi di sostituzione, poi, forniscono agli elementi un meccanismo simile al polimorfismo dei sottotipi della programmazione Object Oriented rendendo i content model più flessibili e permettendo di raggiungere una maggiore astrazione dello schema. Infatti è possibile dichiarare gruppi di elementi che potranno essere utilizzati in maniera interscambiabile all'interno del documento

istanza.

Inoltre XML Schema permette di riutilizzare più volte le strutture dichiarate (componente globale) o di nascondere le varie componenti al resto dello schema (componente locale). Grazie all'introduzione di questi concetti è possibile definire strutture riutilizzabili o meno in altri schema andando a realizzare i cosiddetti stili di progettazione XML Schema presentati nel capitolo 3.

Anche se dotato di grande espressività, XML Schema non consente però di dichiarare i co-constraints caratteristica dei linguaggi rule-based.

Tra i linguaggi rule-based ricordiamo ad esempio **Schematron** e **xlinkit**. Schematron permette di esprimere vincoli complessi grazie all'utilizzo di espressioni XPath. Dall'altro lato però le espressioni XPath non consentono di imporre dei vincoli grammaticali. Risulta quindi difficile dichiarare in maniera esplicita il contenuto di elementi o attributi limitando le possibilità di definire strutture complesse.

**xlinkit** è una service application che genera link rule-based e permette di controllare la coerenza di documenti distribuiti e web content. Per esprimere le regole si utilizza CLIX, un linguaggio logico del primo ordine mentre le espressioni XPath consentono di selezionare i nodi desiderati.

Da quanto detto sopra si può notare come non esista un linguaggio di validazione per XML che racchiuda in sé i pregi sia dei linguaggi grammar-based sia dei linguaggi rule-based e che rispetti, inoltre, le cinque caratteristiche presentate in [WC01].

DTD++ si pone come una possibile soluzione al problema fornendo il potere espressivo di XML Schema combinato con la semplicità del paradigma di validazione di DTD e con la possibilità di esprimere i co-constraints.

Sviluppato all'Università di Bologna [VAG03], DTD++ è

un'estensione sintattica dei DTD e permette quindi una maggiore leggibilità dovuta ad una sintassi non XML-based e consente la grande maggioranza di strutture e concetti di XML Schema. Si può così considerare un buon compromesso tra espressività e semplicità. È essenziale, compatto e facile da imparare. DTD++ aumenta le funzionalità di DTD potenziandolo e rendendolo un sottoinsieme significativo di XML Schema.

In particolare viene ripreso il concetto di tipo sia per gli elementi che per gli attributi ed il meccanismo di derivazione che consente di generare una gerarchia di tipi semplici o complessi.

Inoltre con l'introduzione di un costrutto che permette di effettuare un assegnamento di tipo condizionale e di un tipo predefinito #ERROR, è possibile esprimere quei vincoli sulla presenza incrociata di elementi e attributi consentiti da Schematron.

Si elimina, così, il problema di esprimere regole nel linguaggio naturale e di implementarle direttamente all'interno del software. DTD++ 2.0 ha quindi colmato quel divario funzionale tra i linguaggi grammar-based e i linguaggi rule-based permettendo di esprimere forti vincoli e di accostarsi a Schematron.

In questa versione del linguaggio sono stati implementati quei meccanismi che consentono a DTD++ di essere inserito tra i linguaggi di validazione Object Oriented. In particolare si è puntato sulla modularità e il riuso dello schema resi possibili dalle componenti globali e locali. Allo stesso scopo si sono poi introdotti costrutti per poter definire gruppi di elementi e attributi.

Nella nuova versione si sono introdotti i gruppi di sostituzione ed una sintassi che permette di dichiarare sia componenti locali sia componenti globali permettendo di implementare gli stili di progettazione di XML Schema.

Per permettere maggior modularità e riuso, sono stati poi introdotti i gruppi di attributi ed elementi all'interno dei quali è possibile dichiarare delle liste.

Considerando la classificazione effettuata da [WC01] si può notare come DTD++ sia coerente con tutte le cinque caratteristiche. Riprende, infatti, i tipi predefiniti di XML Schema e consente all'utente di definirne di nuovi. Inoltre è possibile esprimere forti vincoli strutturali sui content model.

Con l'introduzione dei gruppi di sostituzione si è raggiunto il polimorfismo del paradigma Object Oriented ottenendo una maggiore astrazione. Inoltre l'inclusione di altri schema e le nozioni di componente locale e componente globale permettono una grande modularità.

È poi possibile associare elementi ed attributi a differenti namespace ampliando l'espressività del vocabolario.

Come DTD, si possono utilizzare ID e IDREF per creare dei link interni al documento. Inoltre grazie al meccanismo di reference è consentito referenziare più volte strutture dichiarate globalmente.

Infine grazie all'introduzione dei co-constraints è possibile esprimere vincoli caratteristici dei linguaggi rule-based.

Come si è visto, nel corso delle versioni la sintassi ha subito cambiamenti che hanno reso il linguaggio più funzionale ed espressivo. Allo stesso modo anche l' architettura e le tecnologie a cui si è fatto ricorso si sono evolute per implementare nuove funzionalità e per permettere una maggiore robustezza e portabilità del software realizzato.

Inizialmente DTD si appoggiava sul parser di Xerces Java 2 opportunamente modificato per poter analizzare le nuove dichiarazioni e permettere di processare la grammatica estesa di DTD++. L'equivalenza tra le strutture dichiarate in DTD++ e in XML

schema ha, così, permesso di ottenere le corrispondenti dichiarazioni XSD e di effettuare la validazione del documento XML.

La scelta di avvalersi di Xerces ha portato il vantaggio di una tecnologia funzionante e ben testata. Tuttavia, nelle versioni successive del parser Xerces Java 2, si ha avuto la perdita delle modifiche precedentemente effettuate sul codice sorgente.

Per sopperire a tali svantaggi, si è realizzato ex novo un preprocessore[Fio03] per validare documenti XML con documenti DTD++ convertiti temporaneamente in XML Schema. Con l'ausilio di validatori esterni è possibile validare l'XML con lo schema generato e riportare all'utente gli eventuali errori. Delegando la validazione a validatori esterni si può far uso delle migliori tecnologie senza dover apportare modifiche al codice e, soprattutto, senza perdere tali modifiche a causa di una nuova versione del parser.

La nuova versione di PreValidator permette di trasformare in maniera indipendente il documento DTD++ in XML Schema o DTD. È poi possibile indicare il percorso in cui salvare la conversione. Nel caso in cui il DTD++ di input contenga dei riferimenti ad altri file, questi vengono a loro volta processati e convertiti seguendo le stesse regole del documento principale.

Queste modifiche hanno, dunque, aumentato le possibilità di ottenere uno standard. Allo stesso modo si potranno ampliare le potenzialità di PreValidator for DTD++ permettendo la conversione ad altri linguaggi schema quali ad esempio RELAX NG o Schematron.

L'adozione di DTD++ e del preprocessore porta, in maniera evidente, ad alcuni vantaggi:

- Ponendosi come un'estensione del DTD, la sintassi di DTD++ permette un apprendimento veloce del linguaggio con una

conseguente diminuzione dei costi per l'aggiornamento del personale.

- La grande espressività di DTD++ permette di scrivere documenti di validazione dettagliati e con vincoli molto forti. Inoltre le estensioni effettuate apportano i benefici del paradigma Object Oriented.

- Grazie alla sintassi concisa di DTD++ si riducono i tempi di realizzazione dei documenti di validazione.

- La conversione a XML Schema o DTD lascia aperta la possibilità di ottenere sempre uno standard permettendo di utilizzare DTD++ per uso interno e di abbandonarne l'utilizzo in qualsiasi momento.

Si può, quindi, utilizzare DTD++ ad un livello produttivo fornendo un livello di validazione semplice ma allo stesso tempo molto potente.

La tesi è così strutturata:

Per prima cosa viene presentata una classificazione dei linguaggi di validazione per XML che permette di dividerli in due gruppi:i grammar-based e i rule-based. A questo segue la presentazione di alcuni linguaggi trattata in maniera più approfondita nel **Capitolo 2-Linguaggi di validazione XML**.
Successivamente vengono illustrati i principali concetti della programmazione Object Oriented. In particolare si pone l'accento sul pricipio di ereditarietà e di polimorfismo. Si passa quindi ad illustrare il concetto di tipo e la derivazione dei tipi semplici e complessi in XML Schema. Dopo aver definito le componenti locali e globali, vengono presentati gli stili di progettazione XML Schema. Infine si presentano i gruppi di sostituzione. Per

maggiori dettagli si rimanda al **Capitolo 3-Linguaggi di validazione Object Oriented per XML**.
Dopo aver ripercorso la storia di DTD++ si presentano le principali caratteristiche del linguaggio. In particolare si introduce la sintassi per dichiarare componenti globali e locali e la struttura ROOT. Si passa poi a illustrare i meccanismi che lo rendono Object Oriented ovvero i gruppi di elementi e di attributi e i gruppi di sostituzione. Per un approfondimento si rimanda al **Capitolo 4-Un'estensione Object Oriented di DTD++**
In seguito viene presentata la tecnologia di cui si è fatto uso nelle varie versioni e i motivi che hanno portato a tali modifiche. Si introducono quindi i cambiamenti apportati dall'attuale versione, in particolare la possibilità di convertire il DTD++ in XML Schema e DTD. In particolare ci si sofferma sulle principali scelte effettuate nelle conversione a DTD. (**Capitolo 5-PreValidator for DTD++**)
Vengono poi illustrati i dettagli implementativi di PreValidator for DTD++ descrivendo il flusso del programma, le tecnologie utilizzate e le classi. Infine sono riportate le istruzioni per l'utilizzo e l'installazione del preprocessore. Per una descrizione più particolareggiata si rimanda al **Capitolo 6-Dettagli implementativi**
Infine sono riportate le osservazioni conclusive e gli sviluppi futuri.

# Capitolo 2

# Linguaggi di validazione per XML

XML è un meta-linguaggio per descrivere linguaggi di markup. Più semplicemente permette di definire i tag e le relazioni strutturali che intercorrono tra i vari tag.

L'obiettivo principale di XML è quello di andare oltre i limiti di HTML mantenendo allo stesso tempo un livello basso di complessità. L'autore di un documento XML può quindi usare tag a proprio piacimento e le informazioni contenutevi saranno rappresentate in maniera completamente indipendente dal come saranno successivamente processate. Con l'applicazione dei fogli di stile, poi, un browser Web può scegliere il modo più appropriato per rappresentare all'utente un documento XML.

Utilizzare XML porta ad una serie di vantaggi. Ad esempio XML può essere scritto interamente in ASCII consentendo una lettura immediata con qualsiasi programma per visualizzare testi. Il definire tag permette poi di facilitare la comprensione del documento che può essere strutturato ad albero a differenza di quanto avviene con altri formati per l'interscambio di dati organizzati linearmente.

Un ulteriore vantaggio di XML è dato dalla possibilità di forzare la convalida del documento imponendo dei vincoli sulla presenza, o meno, degli elementi di markup e sui valori che questi

possono assumere. Per essere corretto un documento XML deve essere ben formato e valido. È detto ben formato quando è conforme ad ogni regola sintattica. Un documento che non è ben formato non viene considerato un XML e sarà rifiutato dal parser che lo processa. Si parla invece di documento valido quando è ben formato e rispetta l'insieme di regole che descrivono i valori consentiti e la posizione delle varie componenti. Validare un documento XML significa quindi verificare l'aderenza a determinate regole. È possibile individuare [Vl01] almeno quattro livelli di validazione:

- **validazione del markup** ovvero controllo sulle strutture del documento;
- **validazione del contenuto** dei singoli nodi(datatyping);
- **validazione dell'integrity** ovvero dei collegamenti dei nodi all'interno di un documento o dei collegamenti tra documenti;
- **ogni altro test** spesso chiamati "business rules".

È in questo ambito che si inseriscono i linguaggi schema. Un **XML Schema** è la descrizione di un tipo di documento XML. Tipicamente esprime vincoli sulle strutture e sul contenuto del documento. Per esprimere in maniera formale tali schema sono nati numerosi linguaggi XML Schema, standard e proprietari, di cui alcuni basati su XML.

Utilizzare i linguaggi di validazione porta numerosi vantaggi. Primi fra tutti la possibilità di esprimere regole attraverso speciali costrutti e di rendere il processo di validazione indipendente dalla piattaforma. Inoltre i linguaggi schema possono essere utilizzati in diversi scenari: ad esempio una community di utenti può accordarsi su uno schema comune e sulla produzione di documenti XML che saranno validi rispetto allo schema in

questione. Prima dell'avvento di SGML o DTD, infatti, i software designer erano costretti a definire speciali formati file o linguaggi per poter condividere dati tra programmi. Un altro settore in cui l'utilizzo di XML può portare benefici, è quello delle applicazioni che, usando una definizione schema, possono guidare un autore nello sviluppo dei documenti e verificare che i dati inseriti siano validi.

A seconda delle proprie caratteristiche i linguaggi schema si dividono in due categorie:

- **grammar-based languages:** descrivono la struttura di un documento formalizzando le regole di produzione che devono essere seguite. Le espressioni che vincolano gli elementi vengono chiamate tipi e corrispondono ai non-terminali della teoria degli automi. Di questo gruppo fanno parte DTD, XML schema, Relax NG.

- **rule-based languages:** forniscono l'insieme delle regole che un documento XML deve seguire dando una specifica aperta (tutto ciò che non è vietato è consentito) o chiusa (tutto ciò che non è consentito è vietato). A questo gruppo appartengono Schematron e xlinkit.

I linguaggi grammar-based offrono particolari vantaggi nelle interrogazioni XML, in quanto la definizione e la conoscenza dello schema aiuta gli utilizzatori a scrivere delle query ottimizzate e facilita loro le operazioni di debugging. I linguaggi rule-based, invece, permettono di formalizzare in maniera semplice i **co-constraints** (constraint incrociati sull'esistenza o inesistenza contemporanea di elementi e attributi) ma non permettono di imporre dei vincoli complessi ai datavalues e appaiono così limitati nella definizione dei vincoli strutturali.

## 2.1 DTD

Tra i vari linguaggi il primo, e forse il più conosciuto, è il DTD (Document Type Definition). Introdotto e standardizzato dall'XML 1.0 Recommendation, il DTD nasce come versione semplificata dei DTD SGML. Il principale vantaggio dei DTD è l'essere supportati da ogni parser che valida XML 1.0.

DTD, inoltre, permette di scrivere documenti compatti e facilmente comprensibili. L'utilizzo delle entità parametriche permette poi di raggiungere un buon grado di modularità e riuso. Tuttavia si scorgono importanti limitazioni: primo fra tutti la mancata gestione dei namespace che rende difficile la qualifica degli elementi.

La limitata espressività non consente, poi, di catturare alcuni aspetti formali di XML. Inoltre, pur permettendo un ragionevole controllo degli elementi strutturati, DTD ha poca flessibilità sui content model misti e non permette vincoli sugli elementi di testo (`#PCDATA` e `CDATA`) a parte le liste di valori negli attributi. In ultimo per allegare documentazione è necessario inserire commenti XML dentro al DTD, commenti che, tuttavia, vengono eliminati dai parser.

## 2.2 XML Schema

XML Schema, o più informalmente XSD, è considerato il successore dei DTD. Pubblicato come W3C Recommandation nel 2001, utilizza un ampio datatyping system grazie al quale la struttura logica del documento XML può essere vincolata in maniera dettagliata. Il primo miglioramento rispetto ai DTD è la sintassi basata su XML che, se da un lato diminuisce la leggibilità e la chiarezza del documento, dall'altro aumenta la flessibilità e la processabilità automatica.

Un altro importante contributo di XML Schema è il Post-

Schema Validation Infoset (PSVI), ovvero l'insieme delle informazioni extra che la validazione aggiunge ai nodi di un documento XML. In questo modo le informazioni implicite nel documento originale sono rese esplicite e si facilita il trattamento del documento come oggetto ricorrendo al paradigma della programmazione Object Oriented.

La validazione XML Schema-based permette di esprimere la struttura e il contenuto di un documento XML in termini di data model (modello che descrive in maniera astratta il modo in cui vengono rappresentati i dati in un sistema informatico) che è implicito durante la validazione. Il data model di XML Schema include:

- **il vocabolario** (nomi di elementi e attributi);
- **il content model** (relazioni/strutture);
- **i datatypes**.

## 2.3 RELAX NG

RELAX NG è un linguaggio schema pubblicato dall'ISO come International Standard nel 2003. Si basa su due linguaggi precedenti, TREX (Tree Regular Expression for XML) [Cla01], sviluppato da James Clark e RELAX (Regular Language description for XML)[Mur00] sviluppato da Murata Makoto. In uno schema RELAX NG la struttura e il contenuto di un documento XML vengono specificati tramite *pattern* che vanno ad identificare una classe di documenti che fanno match con tali pattern. Il maggior vantaggio di RELAX NG rispetto ai DTD e a XML Schema è costituito dagli *attribute-element constraints*, ovvero l'inclusione degli elementi e degli attributi in singole espressioni regolari. In questo modo, pur se in maniera limitata, è possibile definire co-constraint.

RELAX NG offre due sintassi: la prima è XML based e permette di scrivere degli schema che sono, essi stessi, dei documenti XML. La seconda risulta più compatta e consente di definire pattern senza ambiguità. Rispetto a XML Schema, RELAX NG fornisce solo due datatype: string e token. Tuttavia è possibile far riferimento a datatypes definiti esternamente. I più utilizzati sono quelli definiti dal [BM01].

Pur permettendo di esprimere vincoli di occorrenza (tramite i costrutti <`zeroOrMore`> e <`oneOrMore`>), in RELAX NG può essere difficile esprimere un'occorrenza minima e massima in maniera analoga ai `minOccurs` e `maxOccurs` di XML Schema. Inoltre non è possibile specificare valori di default: ciò può portare a delle situazioni in cui non è possibile stabilire se un pattern viene soddisfatto oppure no.

## 2.4 Schematron

Sviluppato da Rick Jelliffe all'Academia Sinica Computing Center (ASCC), Schematron si inserisce nella categoria dei linguaggi schema rule-based. Infatti a differenza dei linguaggi grammar-based, non viene generata una grammatica per un documento XML, ma vengono fatte delle asserzioni su ciò che il documento XML può o non può contenere. Schematron è basato su una semplice azione: prima vengono ricercati determinati **nodi context** (tipicamente elementi) all'interno del documento tramite i meccanismi di XPath, poi, per ognuno di questi nodi, viene verificato che le altre espressioni XPath siano vere.

Grazie a questo approccio è possibile definire strutture che difficilmente possono essere scritte in un linguaggio grammar-based. Inoltre una frase espressa nel linguaggio naturale può essere tradotta facilmente in una regola Schematron. D'altra

parte, però, manca un costrutto esplicito per dichiarare il contenuto di elementi o attributi in quanto tutto è formalizzato tramite espressioni XPath. Di conseguenza vi è una certa difficoltà nell'esprimere alcuni vincoli. Inoltre, essendo l'obiettivo di Schematron la validazione delle strutture XML, non viene fornito in maniera esplicita alcun built-in datatypes.

## 2.5 xlinkit

xlinkit è più di un linguaggio schema, è una service application che genera link rule-based e controlla la coerenza di documenti distribuiti e web content. Un'applicazione xlinkit definisce un insieme di documenti (**documents set**) e un insieme di regole (**rule set**) e verifica la correttezza delle regole in ogni file del document set. Le regole vengono espresse in CLIX (Constraint Language in XML) un linguaggio logico del primo ordine, mentre, per selezionare i nodi da associare ad un quantificatore, si ricorre alle espressioni XPath .

Al contrario di quanto avviene nei linguaggi grammar-based, xlinkit non permette di esprimere vincoli complessi sul contenuto degli elementi e non definisce alcun datatype, ad eccezione di quelli utilizzati all'interno delle espressioni XPath. È invece possibile esprimere co-constraints significativi. Infatti l'utilizzo congiunto delle espressioni XPath e dei connettori booleani porta ad una notevole espressività.

## 2.6 DTD++

DTD++ [VAG03], sviluppato presso la facoltà di Informatica dell'Università di Bologna, è un'estensione sintattica del DTD.

Sfruttandone la semplicità e mantendendone i costrutti tipici, DTD++ ne aumenta l'espressività e colma quel divario fun-

zionale che separa DTD da XML Schema. Permette, quindi, l'utilizzo delle entità parametriche assenti in XML Schema e come quest'ultimo utilizza il concetto di tipo di dato sia per gli elementi che per gli attributi.

Fornisce poi una serie di tipi predefiniti e i principali costrutti di XML Schema, in particolare i meccanismi di derivazione. Inoltre, con la seconda versione è possibile esprimere i co-constraints raggiungendo un potere espressivo pari a quello di Schematron. DTD++ consente, così, allo stesso tempo i benefici dei linguaggi grammar-based e dei linguaggi rule-based.

## 2.7 Verso il paradigma Object Oriented

Dai linguaggi di validazione sopra presentati risulta evidente come i linguaggi rule-based siano più interessati alla **validazione** che alla **definizione** dello schema come invece avviene nei grammar-based.

Tra questi ultimi si è visto come i DTD presentino alcune limitazioni: principalmente non permettono la derivazione dei tipi e limitano il concetto di tipo di dato esclusivamente agli elementi.

RELAX NG, invece, fa uso di pattern che possono essere definiti in maniera globale (e referenziati) e fornisce alcuni metodi di derivazione. Tuttavia non permette l'implementazione di sottotipi e di meccanismi di sostituzione.

XML Schema risulta, quindi, il linguaggio con il maggior potere espressivo. Fornisce infatti il più completo insieme di tipi predefiniti e, grazie a sofisticati meccanismi di derivazione dei tipi, permette di costruire gerarchie di tipi che lo avvicinano alla programmazione Object Oriented.

Si può inserire nello stesso ambito Object Oriented anche DTD++. In particolare nell'ultima versione sono stati implementati meccanismi che consentono di realizzare strutture gerar-

chiche e polimorfiche che rendono DTD++ un linguaggio di validazione Object Oriented, basato su una grammatica compatta e, contemporaneamente, dotato di un'elevata espressività.

# Capitolo 3

# Linguaggi di validazione Object Oriented per XML

## 3.1 L'approccio Object Oriented

La programmazione Object Oriented vede un programma come costituito da unità individuali (o oggetti) e si oppone alla visione tradizionale secondo cui un programma è un insieme di istruzioni assegnate al computer. Ogni **oggetto** può ricevere messaggi, processare dati e mandare messaggi ad altri oggetti. L'obiettivo dell'approccio Object Oriented è quindi quello di ottenere una maggiore flessibilità e modularità.

### 3.1.1 La classe

Il concetto di classe è alla base della programmazione Object Oriented. Di fondamentale importanza è il duplice ruolo che assume: è, al tempo stesso, un **modulo** e un **tipo**.

Mentre un modulo è un'unità software, il tipo è una descrizione statica degli oggetti che verranno creati e manipolati in maniera dinamica durante l'esecuzione del software. Nell'approccio Object Oriented ogni oggetto è istanza di una qualche classe. Un'altra importante caratteristica è data dal fatto che le funzionalità fornite da una classe X, vista come modulo, coinci-

dono con le operazioni permesse alle istanze della stessa classe vista come tipo.

Una classe **astratta** è una classe concepita per essere esclusivamente una superclasse e da cui le sottoclassi possono ereditare. Permette di rappresentare concetti astratti e quindi non è possibile istanziarla. Ad esempio si può considerare l'insieme dei numeri; la classe `Number` rappresenta così il concetto astratto dei numeri. È possibile manipolare ed effettuare operazioni sui i numeri ma non è sensato creare un generico oggetto `Number`.

### 3.1.2 L'oggetto

Un oggetto è un insieme di attributi e di metodi che vengono utilizzate per manipolare tali variabili. Nella programmazione Object Oriented un'istanza di un programma è vista come un insieme di oggetti che interagiscono fra loro. In un linguaggio in cui gli oggetti sono creati a partire da una classe, un oggetto è detto istanza di una classe. Se ogni oggetto ha associato un tipo, due oggetti con la stessa classe avranno lo stesso tipo.

### 3.1.3 Il tipo

Un datatype (o più semplicemente un tipo) è l'insieme dei valori e delle operazioni che vi possono essere effettuate. Con il termine **type system** si intende invece l'insieme delle regole che permettono di dedurre il tipo di ogni espressione all'interno di un programma e di verificare che tali espressioni siano usate senza violare i vincoli imposti dai corrispondenti tipi.

### 3.1.4 Il sottotipo

Un aspetto rilevante dell'approccio Object Oriented è dato dalla nozione di sottotipo. È possibile utilizzare un sottotipo

ogniqualvolta ci si aspetta il tipo da cui questo deriva (supertipo). Insieme al **binding dinamico**, il concetto di sottotipo permette che uno stesso metodo abbia effetti diversi a seconda di chi lo invoca ed esegue.

### 3.1.5 L'ereditarietà

L'ereditarietà è un meccanismo che permette di ottenere nuove classi (**classi derivate**) partendo da classi già definite (**classi base**). In questo modo classi esistenti possono essere contemporaneamente estese, specializzate e riutilizzate. Le classi derivate (dette anche **figlie** o **sottoclassi**) ereditano gli attributi e i metodi della classe base (conosciute anche con il nome di **classi madre** o **superclassi**) ed aggiungono quegli elementi che le caratterizzano. Dal punto di vista della classe, intesa come modulo, si ha estensione in quanto si crea un nuovo modulo aumentando le funzionalità di uno preesistente. Vedendo la classe come un tipo, invece, si ottiene una specializzazione e si realizza un **sottotipo**. In questo modo, quando una classe A deriva da una classe B, le istanze di A costituiscono un sottoinsieme delle istanze di B.

L'eriditarietà viene anche chiamata **generalizzazione** perché tra le istanze della classe figlia e le istanze della classe madre intercorre una relazione **is-a**. Consideriamo ad esempio la classe figlia `arancia` di cui la classe madre `frutto` è una generalizzazione. Tra le due si avrà un relazione del tipo arancia **is-a** frutto. Di conseguenza poiché arancia è un frutto, erediterà tutte le proprietà comuni ai frutti.

Un altro vantaggio dell'eriditarietà è il poter riutilizzare il codice. La classe derivata eredita automaticamente il codice della classe base e aggiunge solo gli attributi e i metodi che ha in più.

### 3.1.6 Il polimorfismo e il binding dinamico

Grazie al meccanismo di ereditarietà si ottiene una notevole flessibilità nella manipolazione degli oggetti e, allo stesso tempo, si mantiene la sicurezza di un sistema di tipi statico. Il **polimorfismo** e il **binding dinamico** sono due tecniche che supportano tale flessibilità.

Il polimorfismo (pluralità di forme) è la capacità di un valore di assumere più tipi. Alla base vi è l'idea di permettere che le stesse definizioni possano essere utilizzate per differenti tipi di dati apportando così una maggiore astrazione. Consideriamo come esempio la gerarchia e le dichiarazioni:

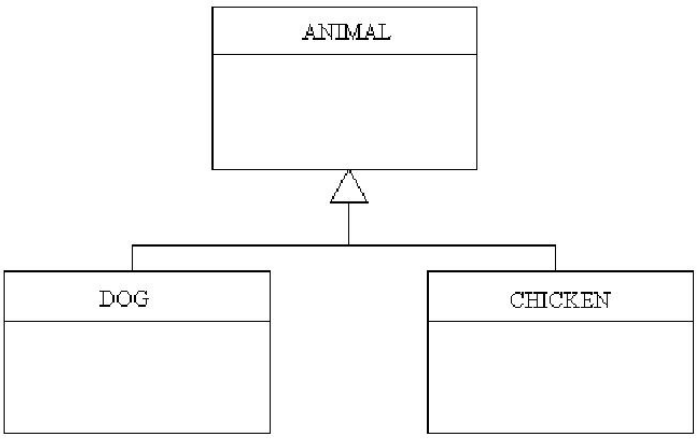



Figura 3.1: 1

```
a: ANIMAL;

d: DOG;

c: CHICKEN;
```

Andiamo ora ad effettuare alcuni assegnamenti:

```
a = d;
a = c;
```

Entrambe le istruzioni sono valide ed assegnano ad una variabile di tipo `ANIMAL` una variabile di tipo `DOG` la prima e una variabile di tipo `CHICKEN` la seconda. Gli assegnamenti in cui il membro sinistro e destro hanno tipo diverso vengono detti polimorfi. Un'entità che come `a` compare in un assegnamento polimorfo è detta entità polimorfa.

Il meccanismo di ereditarietà permette di vedere un oggetto di tipo `DOG` o di tipo `CHICKEN` come un oggetto di tipo `ANIMAL`. Tra le istanze della classe base e quelle di una sua classe derivata intercorre quindi una relazione di **sottotipo**. Un oggetto di tipo `DOG` ha però alcuni campi in più rispetto ad un oggetto di tipo `ANIMAL`: un assegnamento `a = d` porterà inevitabilmente ad una perdita di informazioni.

Definiamo ora alcune operazioni

```
ANIMAL a = new ANIMAL(...);

DOG d = new DOG(...);

a = d;

x = a.countlegs();
```

Se un metodo viene ridefinito lungo la gerarchia di ereditarietà, il **binding dinamico** prevede che sia eseguita la versione corrispondente al tipo dell'istanza a cui si riferisce la variabile. Di conseguenza poiché al momento della chiamata del metodo la variabile `a` contiene un'istanza di tipo `DOG`, verrà eseguita la versione di `countlegs` definita nella classe `DOG`, pur essendo `a` una variabile di tipo `ANIMAL`. Quindi a run time, richiamando un

metodo su una variabile, verrà eseguita la versione corrispondente al tipo dinamico dell'oggetto cui la variabile si riferisce. Non si avrà necessariamente una coincidenza tra il tipo dinamico e il tipo statico, ma il primo sarà sempre un sotto tipo dell'altro.

## 3.2 L'approccio Object Oriented in XML Schema

XML Schema presenta caratteristiche che vanno oltre l'essere un linguaggio grammar-based. Si pone infatti l'obiettivo di descrivere istanze di documenti con un approccio simile all'Object Oriented. La novità di XML Schema rispetto agli altri linguaggi generativi è rappresentata dal concetto di tipo di dato che descrive il contenuto degli elementi e degli attributi. I tipi riconosciuti da XML Schema sono suddivisi in due categorie: i tipi semplici (utilizzati per elementi e per attributi) e i tipi complessi (utilizzati solo per gli elementi). In XML Schema un elemento o un attributo deve sempre essere associato ad un tipo. Mentre gli elementi e gli attributi possono essere assimilati agli oggetti, i tipi possono essere paragonati alle classi e possono essere derivati in maniera analoga alle classi di un sistema Object Oriented.

### 3.2.1 I tipi

#### 3.2.1.1 I tipi semplici

I tipi semplici non possono contenere elementi o attributi e costituiscono la base dai cui partire per costruire gli altri tipi. XML Schema definisce una grande varietà di tipi semplici che possono essere suddivisi in tre categorie: predefiniti, formalizzati dal W3C [BM01], anonimi e nominati. I tipi anonimi vengono dichiarati all'interno di un elemento o di un attributo.

```
<xs:element name="animal">
  <xs:simpleType>
```

```
    <xs:restriction base="xs:string">
      <xs:enumeration value="dog"/>
      <xs:enumeration value="elephant"/>
      <xs:enumeration value="lion"/>
   </xs:restriction>
  </xs:simpleType>
</xs:element>
```

I tipi nominati, invece, sono dichiarati globalmente come <*simpleType*>.

```
<xs:simpleType name="MyInteger">
  <xs:restriction base="xs:integer">
    <xs:minExclusive value="0"/>
    <xs:maxInclusive value="100"/>
  </xs:restriction>
</xs:simpleType>
```

#### 3.2.1.2 I tipi complessi

I tipi complessi contengono al loro interno sia attributi che elementi. Un tipo definito all'interno di una dichiarazione di elemento è detto anonimo, mentre un tipo globale è detto tipo nominato.

```
<xs:complexType name="authorType">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    xs:element name="nickName" type="xs:token"/>
    <xs:element name="born" type="xs:date"/>
  </xs:sequence>
  <xs:attribute name="password" type="xs:token"/>
</xs:complexType>

<xs:element name="indirizzo">
  <xs:complexType>
```

```
    <xs:element name="via" type="xs:string"/>
    <xs:element name="comune" type="xs:string"/>
    <xs:element name="cap" type="xs:integer"/>
  </xs:complexType>
</xs:element>
```

### 3.2.2 La derivazione

I meccanismi di derivazione consentono di ottenere nuovi tipi che possono essere ulteriormente derivati andando a creare una gerarchia di tipi. Si stabilisce così un'equivalenza con il meccanismo di ereditarietà delle classi del paradigma Object Oriented.

In XML Schema la derivazione dei tipi semplici e complessi avviene in maniera differente. I primi possono essere derivati per restrizione, per lista e per unione, mentre i secondi per restrizione ed estensione del content model.

#### 3.2.2.1 Derivazione dei tipi semplici

**Restrizione**

Nella derivazione per restrizione il nuovo tipo è creato a partire da un tipo preesistente (predefinito o definito dall'utente), detto tipo base. La componente derivata, invece, contiene un insieme di vincoli (**facet**) che restringono il valore del datatype. L'applicabilità dei facet è vincolata al tipo base cui si riferiscono.

```
<xs:simpleType name="myString">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{10}"/>
  </xs:restriction>
</xs:simpleType>
```

Nell'esempio `myString` è definito per restrizione sulle stringhe ed impone agli elementi di questo tipo una lunghezza di dieci

caratteri che, a loro volta, devono assumere valori compresi tra 0 e 9.

#### Lista

Tramite la derivazione per lista si definisce un insieme di valori leciti del tipo atomico derivato. Ad esempio `LotteryNumbers` accetterà una lista di interi non negativi con valore massimo 99.

```
<xs:simpleType name="LotteryNumbers">
  <xs:list>
    <xs:simpleType>
      <xs:restriction base="xs:nonNegativeInteger">
        <xs:maxInclusive value="99"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:list>
</xs:simpleType>
```

#### Unione

Tramite questo processo si uniscono i valori leciti di due tipi semplici ottenendo un nuovo tipo dato dall'unione dei tipi base derivati.

```
<xs:simpleType name="simple">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:gDay">
        <xs:whiteSpace value="collapse"/>
          </xs:restriction>
        </xs:simpleType>
         <xs:simpleType>
      <xs:restriction base="xs:hexBinary"/>
     </xs:simpleType>
```

```
  </xs:union>
</xs:simpleType>
```

#### 3.2.2.2 Derivazione dei tipi complessi

**Restrizione**

L'obiettivo della derivazione per restrizione di un tipo complesso è quello di ridurre l'insieme delle strutture valide per il datatype e di permettere che ogni struttura valida per il tipo derivato sia valida anche per il tipo base. In questo modo se l'applicazione sa come processare il tipo base, sarà sicuramente in grado di processare il tipo derivato.

```
<xs:complexType name="personType">
  <xs:sequence >
    <xs:element name="name" type="xs:string"/>
    <xs:element name="nickName" type="xs:token"
        minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="born" type="xs:date"/>
    <xs:element name="email" type="xs:token"
        minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
  <xs:attribute name="password" type="xs:token"/>
</xs:complexType>
```

Una possibile derivazione per restrizione di `authorType` può essere

```
<xs:complexType name="authorType">
   <xs:complexContent>
      <xs:restriction base="personType">
         <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="nickName" type="xs:token"/>
```

```
            <xs:element name="born" type="xs:date"/>
          </xs:sequence>
          <xs:attribute name="password" type="xs:token"/>
       </xs:restriction>
    </xs:complexContent>
</xs:complexType>
```

Il fatto che la derivazione risulti essere più concisa della definizione, può sembrare in contrasto con l'approccio Object Oriented. In realtà ciò che si vuole ottenere non è la modularità dello schema (una modifica a personType probabilmente porterà ad una modifica dei tipi derivati), ma il fornire una dichiarazione che i tipi derivati possono utilizzare in modo da accrescere la modularità delle applicazioni che processano il documento.

Le istanze di un documento derivato saranno quindi sempre istanze del documento padre. Infatti ogni accesso al documento tramite API standardizzate come DOM[1] e SAX[2] sarà ancora valido nella derivazione per restrizione.

#### Estensione

La derivazione per estensione viene effettuata aggiungendo al content model elementi o attributi. Le nuove componenti devono, però, essere necessariamente poste alla fine delle altre. Questo perché se, un'applicazione è stata disegnata per processare strutture valide per un dato tipo complesso e per ignorare attributi non definiti ed elementi posti dopo quelli definiti nel tipo, tale applicazione sarà in grado di processare anche i tipi derivati per estensione dal tipo base.

Le istanze di un documento derivato saranno sempre istanze del documento padre solo se non vengono fatte assunzioni sul numero di elementi.

[1] http://www.w3.org/DOM/
[2] http://www.saxproject.org/

```
<xs:complexType name="person">
  <xs:sequence>
    <xs:element ref="firstname"/>
    <xs:element ref="secondname"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="personType" >
  <xs:complexContent>
    <xs:extension base="person">
      <xs:sequence>
        <xs:element ref="address"/>
      </xs:sequence>
      <xs:attribute name="age" type="xs:integer"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Anche in questo caso la modularità non è a favore dello schema (qui però una modifica del tipo base sarà automaticamente inserita nel tipo derivato) quanto piuttosto delle applicazioni che utilizzano il documento.

### 3.2.3 Componenti locali e globali

XML Schema permette di definire una serie di componenti con sintassi e caratteristiche differenti. In tabella sono riportate tali strutture.

Le varie componenti possono essere definite globali e riutilizzate tramite il meccanismo di reference oppure possono essere dichiarate (ad eccezione dei gruppi di elementi e di attributi) in maniera anonima all'interno del contesto in cui sono definite.

| componente | XML Schema | DTD++ |
|---|---|---|
| elemento | <xs:element> | <!ELEMENT ....> |
| attributo | <xs:attribute> | <!ATTLIST ...> |
| tipo semplice | <xs:simpleType> | <!ENTITY # ...> |
| tipo complesso | <xs:complexType> | <!ENTITY @ ....> |
| gruppo di elementi | <xs:group> | <!ENTITY @@ ....> |
| gruppo di attributi | <xs:attributeGroup> | <!ENTITY ## ....> |

Anche dal punto di vista della semantica vi sono differenze: mentre gli elementi e gli attributi fanno realmente matching con le produzioni XML e descrivono il documento, gli elementGroup e gli attributeGroup sono dei *contenitori* simili ai pattern generici di RELAX NG (tranne per il fatto che non possono contenere elementi misti, testo o attributi).

Per permettere una maggiore modularità del codice e delle dichiarazioni delle varie componenti, XML Schema permette di dichiarare elementi e tipi sia in maniera globale che in maniera locale. Un elemento o un tipo viene detto **globale** quando è figlio diretto dell'elemento <***schema***>. Solo un elemento definito in questo modo potrà essere utilizzato come radice dell'XML istanza dello schema. Grazie al meccanismo di *reference* le componenti globali possono essere riutilizzate all'interno del documento.

Al contrario un elemento o un tipo è **locale**, e non può essere riutilizzato, quando è definito all'interno di un'altra componente. In questo caso possono esistere in luoghi distinti elementi di tipo anonimo con diversa definizione ma stesso nome. Ad esempio gli elementi `name` e `age` sono dichiarati localmente al tipo `person` e sono, rispettavamente, di tipo `string` e di tipo `integer`.

```
<xs:complexType name="person">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
```

```
    <xs:element name="age" type="xs:integer"/>
  </xs:sequence>
</xs:complexType>
```

In un secondo tipo sarà quindi possibile associare sia a `name` che a `age` un tipo differente.

```
<xs:complexType name="personType">
  <xs:sequence>
    <xs:element name="name" type="xs:TOKEN"/>
    <xs:element name="age" type="xs:nonNegativeInteger"/>
  </xs:sequence>
</xs:complexType>
```

In maniera analoga si può voler dichiarare un tipo anonimo e quindi localmente ad un elemento.

```
<xs:element name="library">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="book" type="xs:string">
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Simmetricamente XML Schema consente di referenziare in maniera esplicita gli elementi globali. Ad esempio il tipo complesso `person` farà riferimento all'elemento globale `age`.

```
<xs:element name="age" type="xs:integer"/>

<xs:complexType name="person">
  <xs:sequence>
    <xs:element ref="age"/>
  </xs:sequence>
</xs:complexType>
```

Allo stesso modo è possibile definire tipi anonimi che fanno riferimento ad elementi globali.

```
<xs:element name="city" type="xs:string"/>

<xs:element name="people" type="xs:integer"/>

<xs:element name="country">
  <xs:complexType >
    <xs:choice>
      <xs:element ref="city"/>
      <xs:element ref="people"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

La nozione di componente globale o locale permette inoltre di implementare diversi stili di progettazione che saranno presentati nella prossima sezione.

### 3.2.4 SubstitutionGroup

I gruppi di sostituzione forniscono agli elementi XML Schema un meccanismo simile al polimorfismo dei sottotipi della programmazione Object Oriented. Un elemento dichiarato **substitution** può essere usato, in maniera interscambiabile, al posto di un elemento globale chiamato **head element**. Ad esempio in un gruppo di sostituzione `Address` con membri `USAAddress` e `UKAddress`, l'elemento `Address` può essere utilizzato in un content model o può essere sostituito da `USAAddress` o da `UKAddress`.

La sola restrizione imposta è che i membri del gruppo di sostituzione e l'head element siano dello stesso tipo o appartengano allo stessa gerarchia di derivazione.

Di seguito viene presentato uno schema e l'istanza che lo valida.

```
<xs:schema
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.example.com"
 xmlns:ex="http://www.example.com"
 elementFormDefault="qualified">

  <xs:element name="book" type="xs:string" />

  <xs:element name="magazine" type="xs:string"
              substitutionGroup="ex:book" />

  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ex:book" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>

<library xmlns="http://www.example.com">
 <magazine>MSDN Magazine</magazine>
 <book>Professional XML Databases</book>
</library>
```

Il content model dell'elemento `library` permette di dichiarare uno o più elementi `book`. Poiché l'elemento `magazine` fa parte del gruppo di sostituzione di `book`, nell'istanza XML, laddove ci si aspetta `magazine`, si può utilizzare l'elemento `book`.

I gruppi di sostituzione rendono così i content model più flessibili e si raggiunge una maggiore astrazione dello schema.

## 3.3 Stili di progettazione XML Schema

XML Schema offre diversi modi di organizzare dichiarazioni di elementi e definizioni di tipo. Si può decidere di utilizzare uno stile piuttosto che un altro quando si vogliono rendere i costrutti riutilizzabili o modificabili da parte di altri schema che includono o importano il documento in cui la struttura è dichiarata.

In XML Schema non esiste il concetto di elemento anonimo. Tuttavia non è il nome a rendere riutilizzabile la dichiarazione di un elemento, ma è necessario che l'elemento sia qualificato con un namespace e che sia dichiarato globalmente. Infatti se un elemento è dichiarato localmente, non può essere riutilizzato anche se è namespace-qualified. Un elemento sia globale sia namespace-qualified, invece, può essere riusato in un altro modulo schema tramite inclusione tra moduli.

I tipi anonimi non sono riutilizzabili in altri moduli schema nè nello stesso modulo in cui sono definiti. Il loro utilizzo è quindi limitato all'unico elemento in cui sono dichiarati. Al contrario i tipi nominati sono riutilizzabili e devono essere definiti come figli diretti dell'elemento <***schema***>.

```
<xs:schema...>
  <xs:complexType name="ReusableType">
    ...
  </xs:complexType>
  ...
</xs:schema>
```

### 3.3.1 Russian Doll

Nello stile Russian Doll le dichiarazioni degli elementi sono annidate una dentro l'altra in maniera progressiva. L'elemento radice, l'unico elemento globale, ha associato un tipo anonimo che contiene tutti gli elementi di secondo livello ciascuno con un

proprio tipo anonimo che contiene elementi di terzo livello e così via. Le componenti sono quindi connesse insieme in una singola unità apportando compattezza allo schema.

```
<xs:schema...>
  <xs:element name="Recipe">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ID" type="cct:IDType"/>
        <xs:element name="Ingredient"
                    maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="IDType"/>
              <xs:element name="Amount" type="xs:string"/>
              <xs:element name="Name" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Step" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="IDType"/>
              <xs:element name="Description"
                          type="xs:string"/>
            <xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    <xs:complexType>
  <xs:element>
<xs:schema>
```

Come si può vedere dall'esempio ogni definizione di elemento è locale ad un elemento contenitore. Ne consegue che, se uno schema è dichiarato `elementFormDefault=unqualified`, il namespace degli elementi locali sarà nascosto all'interno dello schema. Nello stile Russian Doll, inoltre, il content model dell'elemento radice è opaco per gli altri schema e per le altre parti dello stesso schema. Di conseguenza nessuno dei tipi o degli elementi dichiarati al suo interno è riutilizzabile.

### 3.3.2 Salami Slice

Il modello Salami Slice è caratterizzato dal mettere come globali tutti gli elementi e dall'usare i reference per definire i tipi. Gli elementi devono essere qualificati qualunque sia il modello di qualificazione scelto: risultano così visibili per gli altri schema e per le altre parti dello stesso schema e possono quindi essere riutilizzati.

```
<xs:schema...>
  <xs:element name="ID" type="cct:IDType"/>

  <xs:element name="Recipe">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ID"/>
        <xs:element ref="Ingredient" maxOccurs="unbounded"/>
        <xs:element ref="Step" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Ingredient">
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element ref="ID"/>
        <xs:element ref="Amount"/>
        <xs:element ref="Name"/>
      <xs:sequence>
    <xs:complexType>
  </xs:element>

  <xs:element name="Amount" type="xs:string"/>

  <xs:element name="Name" type="xs:string"/>

  <xs:element name="Step" maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ID" type="IDType"/>
        <xs:element ref="Description"/>
      </xs:sequence>
    <xs:complexType>
  </xs:element>

  <xs:element name="Description" type="xs:string"/>
<xs:schema>
```

Come si nota dallo schema sopra riportato, ogni componente è dichiarato nello schema in maniera indipendente e viene poi assemblato con gli altri tramite reference.

Riassumendo si può affermare che gli stili Russian Doll e Salami Slice differiscono tra loro principalmente in quanto:

- **Russian Doll** facilita la gestione flessibile del namespace;
- **Salami Slice** facilita il riuso dei componenti e la modularità.

### 3.3.3 Venetian Blind

Con lo stile Venetian Blind si hanno i vantaggi e i benefici di entrambi gli stili sopra citati.

Il contenuto dello schema viene disassemblato in componenti individuali come nel Salami Slice ma invece di definire elementi, si creano definizioni di tipo. Si ha così un solo elemento globale, l'elemento radice che è generalmente namespace-qualified, e tanti tipi nominali ciascuno dei quali contiene definizioni di elementi locali. Come nel Russian Doll, gli elementi locali possono essere namespace-qualified o namespace-unqualified ma non sono in ogni caso riutilizzabili. Il nome Venetian Blind riflette il fatto che ci siano *slats* individuali e che gli elementi locali possano essere o meno namespace-qualified così come gli *slats* possono essere aperti o chiusi. In questo modo si ottiene sia il riuso che la gestione flessibile dei namespace.

```
<xs:schema...>
  <xs:element name="Recipe" type="RecipeType"/>

  <xs:complexType name="RecipeType">
    <xs:sequence>
      <xs:element name="ID" type="cct:IDType" />
      <xs:element name="Ingredient" type="IngredientType"
                  maxOccurs="unbounded"/>
      <xs:element name="Step" type="StepType"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="IngredientType">
    <xs:sequence>
      <xs:element name="ID" type="cct:IDType" />
      <xs:element name="Amount" type="xs:string"/>
```

```
      <xs:element name="Name" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="StepType">
    <xs:sequence>
      <xs:element name="ID" type="IDType"/>
      <xs:element name="Description" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

<xs:schema>
```

Dall'esempio si nota come nello stile Venetian Blind vi sia il massimo riuso dei tipi. Le dichiarazioni degli elementi sono annidate all'interno dei tipi massimizzando l'incapsulamento dei namespace.

### 3.3.4 Garden of Eden

Lo stile Garden of Eden è un quarto stile in cui i tipi sono nominati e tutti gli elementi sono dichiarati globali. Ogni content model fa poi riferimento alle dichiarazioni degli elementi globali. Tutte le componenti di un documento sono così riutilizzabili in ogni altro documento che lo importa.

```
<xs:schema...>
  <xs:element name="Recipe" type="RecipeType"/>

  <xs:complexType name="RecipeType">
    <xs:sequence>
      <xs:element ref="ID"/>
      <xs:element ref="Ingredient" maxOccurs="unbounded"/>
      <xs:element ref="Step" maxOccurs="unbounded"/>
```

```
    </xs:sequence>
  </xs:complexType>

  <xs:element name="ID" type="cct:IDType" />

  <xs:element name="Ingredient" type="IngredientType" />

  <xs:complexType name="IngredientType">
    <xs:sequence>
      <xs:element ref="ID" />
      <xs:element ref="Amount" />
      <xs:element ref="Name" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="Amount" type="xs:string" />

  <xs:element name="Name" type="xs:string" />

  <xs:element name="Step" type="StepType" />

  <xs:complexType name="StepType">
    <xs:sequence>
      <xs:element ref="ID" />
      <xs:element ref="Description" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="Description" type="xs:string" />

<xs:schema>
```

# Capitolo 4

# Un'estensione Object Oriented di DTD++

## 4.1 Introduzione

La sintassi del DTD è una sintassi semplice, essenziale e compatta. Inoltre le entità parametriche consentono un buon grado di riuso e modularità. Tuttavia risulta un linguaggio poco espressivo che non permette di imporre vincoli precisi. Il concetto di tipo è poi limitato ai soli elementi e non è possibile effettuare alcun tipo di derivazione né dichiarare namespace a cui associare le dichiarazioni di elementi. XML Schema, invece, possiede una gestione dei tipi molto complessa ed è dotato di una grande espressività. Risulta però un linguaggio verboso, a volte astratto e di difficile comprensione.

Il DTD++ nasce quindi per colmare il divario funzionale tra DTD e XML Schema. Preferendo una sintassi più concisa e autodescrittiva, si è mantenuto il paradigma di validazione di DTD e, allo stesso tempo, lo si è arricchito importando le principali componenti di XML Schema.

La prima versione [VAG03] ha visto l'introduzione delle principali funzionalità di XML Schema. In particolare ha fornito supporto per namespace, tipi semplici predefiniti, facet, tipi

semplici e complessi definiti dall'utente e la versione XML Schema del content model ALL.

Come XML Schema e gli altri linguaggi di validazione grammar-based, DTD++ non permetteva di esprimere condizioni sulla reciproca dipendenza di elementi e attributi. La mancanza dei co-constraints in DTD e XML Schema è, tuttavia, sentita in vari ambiti. Infatti tali regole vengono, altrimenti, espresse nel linguaggio naturale e gli sviluppatori devono implementare le regole più importanti direttamente nel software.

Inoltre la validazione dei dati inseriti è un aspetto critico per le infrastrutture e-business in cui si richiede non solo l'utilizzo di una grammatica e di un insieme di datatype, ma anche l'adozione dei co-constraints.

Sia DTD che XML Schema non permettono, neppure, di imporre i vincoli più semplici e comuni. Ad esempio si può volere la presenza non simultanea degli attributi A e B oppure si può voler specificare che se esiste un attributo `gratis`, l'elemento <`prezzo`> non può esistere.

L'importanza dei co-constraints ha così portato ad una nuova versione 2.0 [VFGM03] in cui si è fornita la soluzione per specificare e verificare co-constraints nei documenti XML. Inserendo un'espressione XPath all'interno della dichiarazione di un elemento, si identifica tale elemento come un *elemento condizionale*.

Grazie all'aggiunta di un costrutto che permette di effettuare un assegnamento di tipo condizionale e di un tipo predefinito `#ERROR` è possibile esprimere in DTD++ un elevato numero di co-constraints. Si è così ottenuto un linguaggio semanticamente equivalente ad un significativo sottoinsieme di XML Schema e allo stesso tempo si è fornito supporto per i co-constraints.

DTD++ 2.0 ha quindi colmato quel divario funzionale tra i linguaggi grammar-based e i linguaggi rule-based permettendo di esprimere forti vincoli e di accostarsi a Schematron.

Come si è visto in precedenza, XML Schema fornisce complessi meccanismi di derivazione e di sostituzione di elementi che lo rendono un linguaggio paragonabile al paradigma Object Oriented. In particolare l'ereditarietà e il polimorfismo consentono di ottenere una notevole flessibilità nella manipolazione degli oggetti e rendono i content model più flessibili.

Nell'ultima versione si è, quindi, posto l'accento sugli aspetti Object Oriented per poter raggiunger un maggior riuso e astrazione. Si è, dunque, introdotta una sintassi che consente di specificare le componenti locali o globali e sono stati definiti nuovi costrutti per permettere di dichiarare gruppi di elementi e attributi e gruppi di sostituzione.

Nella sezione 4.2 verrà ripresa la sintassi di DTD++, di cui l'attuale è un'estensione. Nelle sezioni successive si presenteranno in dettaglio le estensioni Object Oriented di questa versione.

## 4.2 I tipi

In DTD++ è possibile definire due categorie di tipo: i tipi semplici e i tipi complessi. Come in XML Schema, un elemento o un attributo deve sempre essere associato ad un tipo.

### 4.2.1 Tipi semplici

DTD++ permette di definire delle entità che hanno lo stesso valore delle dichiarazioni dei tipi semplici in XML Schema. È così possibile utilizzare una serie di tipi predefiniti equivalenti ai tipi predefiniti di cui si parla nella *recommendation XML Schema Part 2*[BM01] o dichiarare dei nuovi tipi semplici.

Ad esempio

```
<!ENTITY # myString (#STRING)>
```

**Tipi semplici**

*Sintassi:*
```
<!ENTITY # nome "(tipobase)">
<!ELEMENT nomeElemento (tipobase)>
<!ATTLIST nomeElemento nomeAttributo (tipobase) default>
<!ATTLIST nomeElemento nomeAttrGroup >
```
*o anche*
```
<!ENTITY # nome "tipobase">
<!ELEMENT nomeElemento tipobase>
<!ATTLIST nomeElemento nomeAttributo tipobase default>
```

| | |
|---|---|
| nome: | nome del nuovo tipo semplice |
| tipobase: | tipo semplice da cui l'elemento deriva |
| nomeElemento: | nome dell'elemento |
| nomeAttributo: | nome dell'attributo |
| default: | #IMPLIED o #REQUIRED o #FIXED " " o " " |
| nomeAttrGroup: | nome dell'attributeGroup sempre racchiuso tra "##" e ";" |

### 4.2.2 Tipi complessi

La dichiarazione di un tipo complesso avviene tramite l'utilizzo di una entità e il simbolo "@". All'interno dei tipi complessi possono apparire sia elementi che attributi.

Ad esempio

```
<!ENTITY @ person "(name,surname)"
          "age #INTEGER #REQUIRED">
```

## 4.3 Derivazione

È possibile derivare i tipi semplici e i tipi complessi creando una gerarchia in maniera analoga alle classi di un sistema Object Oriented.

**Tipo complesso**

| | |
|---|---|
| *Sintassi:* | <!ENTITY @ nome "contentmodel" listaAttributi> |
| *nome:* | nome del tipo complesso |
| *contentmodel:* | contentmodel dell'elemento |
| *listaAttributi:* | "lista di attributi" |

### 4.3.1 Derivazione dei tipi semplici

#### 4.3.1.1 Restrizione

Nella derivazione per restrizione il nuovo tipo è creato a partire da un tipo preesistente (predefinito o definito dall'utente), detto tipo base. La componente derivata, invece, contiene un insieme di vincoli (**facet**) che restringono il valore del datatype.

```
<!ENTITY # myInteger "(#INTEGER[0,101[)">
```

È possibile decidere di non utilizzare le parentesi ottenendo un tipo complesso con l'identico significato.

```
<!ENTITY # myInteger "#INTEGER[0,101[">
```

Da un punto di vista semantico, si noti che non vanno inseriti spazi tra tipobase e i vincoli nè tra i vari vincoli. È possibile, inoltre, applicare più facet contemporaneamente ma la loro applicabilita è vincolata al tipo di dato cui si riferiscono.

#### 4.3.1.2 Lista

La derivazione per lista si ottiene aggiungendo al tipo base il simbolo "+". Tale derivazione permette di definire un insieme di valori leciti del tipo atomico derivato. Ad esempio `LotteryNumbers` accetterà una lista di interi non negativi con valore massimo 99.

**Derivazione per restrizione**

*Sintassi:* <!ENTITY # nome "(tipobase{}[]//()\\)">
<!ELEMENT nomeElemento (tipobase{}[]//()\\)>
<!ATTLIST nomeElemento nomeAttributo
(tipobase{}[]//()\\) default>
*o anche*
<!ENTITY # nome "tipobase {}[]//()\\">
<!ELEMENT nomeElemento tipobase{}[]//()\\>
<!ATTLIST nomeElemento nomeAttributo
tipobase{}[]//()\\ default>

{*lunghezza*}
*[range]*
/*pattern*/
(*enumerazione*)
*whitespace*\: vincoli di derivazione per restrizione

```
<!ENTITY # LotteryNumbers "#NONNEGINTEGER+[,99]">
```

I tipi derivati per lista possono essere ulteriormente derivati per restrizione. In questo caso il simbolo "+" deve essere il primo ad apparire dopo il tipo semplice.

#### 4.3.1.3 Unione

Tramite questo processo si uniscono i valori leciti di due tipi semplici ottenendo un nuovo tipo dato dall'unione dei tipi base derivati. Per esprimere la derivazione per unione i tipi base coinvolti vengono separati dal simbolo "|" e possono essere o meno posti tra parentesi.

```
<!ELEMENT simple ((#GDAY\c\)|(#HEXBINARY))>
```

è del tutto equivalente a

```
<!ELEMENT simple #GDAY\c\|#HEXBINARY>
```

**Derivazione per lista**

*Sintassi:* <!ENTITY # nome "(tipobase+{}[]//()\\)">
<!ELEMENT nomeElemento(tipobase+{}[]//()\\)>
<!ATTLIST nomeElemento nomeAttributo
(tipobase+{}[]//()\\) default>
*o anche*
<!ENTITY # nome "tipobase+ {}[]//()\\">
<!ELEMENT nomeElemento tipobase+{}[]//()\\ >
<!ATTLIST nomeElemento nomeAttributo
tipobase+{}[]//()\\ default>

*+*: derivazione per lista
*{}[]// ()\\:* vincoli di derivazione per restrizione

**Derivazione per unione**

*Sintassi:* <!ENTITY # nome "union">
<!ELEMENT nomeElemento (union)>
<!ATTLIST nomeElemento nomeAttributo union default>
*o anche*
<!ENTITY # nome "union">
<!ELEMENT nomeElemento union>
<!ATTLIST nomeElemento nomeAttributo union default>

*union:* (tipobase+vincoli) |(tipobase+vincoli)|...
*oppure* tipobase+vincoli | tipobase+vincoli |...
*vincoli:* vincoli di derivazione per restrizione

### 4.3.2 Derivazione dei tipi complessi

I tipi complessi possono contenere al loro interno sia elementi che attributi. Questi ultimi, se presenti, devono essere sempre racchiusi tra virgolette. Altrimenti è possibile tralasciarle o inserirle.

Ad esempio

```
<!ENTITY @ personType "(firstname|secondname)">

<!ENTITY @ indirizzo "(via,città)" "">
```

#### 4.3.2.1 Restrizione

Con la derivazione per restrizione è possibile vincolare i tipi complessi definiti dall'utente. Si dichiara quindi un nuovo tipo complesso come restrizione di uno preesistente.

**Derivazione per restrizione**

*Sintassi:* <!ENTITY @ nome tipoBaseRestr "contentmodel" listaAttributi>

*tipoBaseRestr:* tipo complesso base da derivare per restrizione

Ad esempio

```
<!ENTITY @ personType "(firstname,secondname?)"
           "age #INTEGER #REQUIRED
            year CDATA #REQUIRED">

<!ENTITY @ person @personType; "(firstname,secondname)"
           "age #INTEGER #REQUIRED">
```

#### 4.3.2.2 Estensione

La derivazione per estensione permette di aggiungere valori al tipo base. È possibile effettuare la derivazione per estensione

**Derivazione per estensione**

*Sintassi:* `<!ENTITY @ "tipoBaseExt,contentmodel" listaAttr>`
`<!ELEMENT nomeElemento (tipoBaseExt,contentModel)>`

*tipoBaseExt:* `tipo complesso base da derivare per estensione`

aggiungendo solo attributi.

```
<!ENTITY @ personType "(firstname,secondname?)"

<!ENTITY @ person "@personType;"
          "age #INTEGER #REQUIRED">
```

La nuova sintassi prevede anche la derivazione di un tipo complesso anonimo associato ad un elemento.

```
<!ENTITY @ personType "(firstname)">

<!ELEMENT person (@personType;,(firstname,secondname?))>
```

## 4.4 Componenti locali e globali

Per permettere maggiore modularità del codice e delle dichiarazioni delle varie componenti, si sono introdotte due notazioni che permettono di definire elementi e tipi in maniera locale o globale. In questo modo è possibile nascondere la definizione di una componente o renderla riutilizzabile tramite il meccanismo di *reference*.

### 4.4.1 Componenti locali

**Componenti locali**

*Sintassi:* <!ENTITY @ "(nomeElemento)" listaAttributi>
<!ELEMENT nome.nomeElemento tipoBase >

In questo caso possono esistere in luoghi distinti elementi di tipo anonimo con diversa definizione. Ad esempio gli elementi `name` e `age` sono dichiarati localmente al tipo `person` e sono, rispettavamente, di tipo `string` e di tipo `integer`.

```
<!ENTITY @ person "(name,age)">

<!ELEMENT person.name #STRING>

<!ELEMENT person.age #INTEGER>
```

In un secondo tipo sarà quindi possibile associare sia a `name` che a `age` un tipo differente

```
<!ENTITY @ personType "(name,age)">

<!ELEMENT personType.name #TOKEN>

<!ELEMENT personType.age #NONNEGINTEGER>
```

In maniera analoga si può voler dichiarare un tipo anonimo e quindi localmente ad un elemento.

```
<!ELEMENT library "(book)">

<!ELEMENT  book #STRING>
```

dichiara che `library` ha associato un tipo anonimo.

### 4.4.2 Componenti globali

Simmetricamente DTD++ consente di referenziare in maniera esplicita gli elementi globali tramite il simbolo "$".

**Componenti globali**

*Sintassi:*
```
<!ELEMENT nomeElemento tipoBase>
<!ENTITY @ nome "($nomeElemento)" listaAttributi>
<!ELEMENT nome ($nomeElemento)>
```

Ad esempio il tipo complesso `person` farà riferimento all'elemento globale `age`.

```
<!ELEMENT  age  #INTEGER>
<!ENTITY @ person "($age)" >
```

Allo stesso modo è possibile definire tipi anonimi che fanno riferimento ad elementi globali.

```
<!ELEMENT city #STRING>
<!ELEMENT people #INTEGER>
<!ELEMENT country ($city|$people)>
```

`country` avrà associato un tipo complesso anonimo che fa riferimento a agli elementi `city` e `people` entrambi globali.

Grazie è queste estensioni si sono potuti implementare i vari stili di progettazione di XML Schema permettendo di raggiungere, in DTD++, maggior modularità e riuso.

### 4.4.3 Elemento radice

Successivamente si è introdotta la struttura <ROOT> che permette di specificare che un elemento globale debba essere anche una radice.

**Elemento radice**

*Sintassi:* `<!ROOT nomeElemento contentModel >`

Come esempio viene riportato il DTD++ equivalente allo schema secondo lo stile Russian Doll riportato precedentemente.

```
<!ROOT Recipe(ID,Ingredient*,Step*)>

<!ELEMENT Recipe.ID #cct:IDType;>

<!ELEMENT Ingredient (ID,Amount,Name)>

<!ELEMENT Ingredient.ID #IDType;>

<!ELEMENT Amount #STRING>

<!ELEMENT Name #STRING>

<!ELEMENT Step (ID,Description)>

<!ELEMENT Step.ID #IDtype;>

<!ELEMENT Description #STRING>
```

Si noti come l'espressività di DTD++ permette di descrivere strutture complesse senza, però, cadere nella verbosità di XML Schema.

## 4.5 Estensioni Object Oriented

Inizialmente sono stati corretti alcuni bug che hanno permesso di aggiungere leggibilità ed espressività al linguaggio. Per prima cosa si è snellito il linguaggio permettendo di definire i tipi semplici e la loro derivazione per restrizione, lista o unione, senza dover ricorrere all'uso delle parentesi.

In questo modo è possibile associare ad un elemento un tipo semplice senza imporvi alcuna restrizione o vincolo. Ad esempio l'elemento `name` avrà associato il tipo `mystring`

```
<!ENTITY # myString "#STRING">
<!ELEMENT name #myString;>
```

Analogamente per utilizzare un tipo complesso nel content model di un elemento, si può scegliere se racchiuderlo o meno tra parentesi. L'elemento

```
<!ELEMENT address (@USAddress;)>
```

può essere scritto come

```
<!ELEMENT address @USAddress;>
```

mantenendo lo stesso identico significato. Grazie all'eliminazione delle parentesi, non si è risolto solo un problema di leggibilità, ma soprattutto si è evitato che il parser consideri erroneamente una coppia di parentesi come un'ulteriore sottotipatura.

Un'altra importante modifica della sintassi è data dalla derivazione anonima per estensione di tipo complesso.
Ad esempio con

```
<!ELEMENT uno (@test;(a|b)+)>
```

si avrà un'estensione locale del tipo `test` associato all'elemento `uno`.

### 4.5.1 Element e Attribute Groups

Per permettere maggior modularità e riuso, sono stati introdotti in DTD++ i gruppi di elementi e di attributi che possono essere visti come dei contenitori globali all'interno dei quali vengono dichiarati degli elementi o degli attributi. Si hanno, così, delle strutture semplici e flessibili che possono essere riutilizzate più volte.

I due costrutti sono entrambi dichiarati come entità e presentano una sintassi simmetrica.

**Group**

*Sintassi:* <!ENTITY @@ nome "contentModel">

I Group possono essere utilizzati nel content model di un tipo, di un elemento o di un altro Group.
Ad esempio

```
<!ENTITY @@ tuttiblocchi "div|p|img|table">

<!ELEMENT elem (@gerarchiabase;,(@@tuttiblocchi;)+)>
```

dichiara che l'element `elem` ha associato il tipo `gerarchiabase` che viene esteso dal Group `tuttiblocchi`.

Simmetricamente l'attributGroup `HTMLattrs` contiene le definizioni dei tre attribute `class`, `style` e `title`.

```
<!ENTITY ## HTMLattrs
```

**AttributeGroup**

*Sintassi:* <!ENTITY ## nome “listaAttr”>

*listaAttr:* attributeGroup sempre racchiuso tra “##” e “;”
o nome Attributo (tipoBase) default
o nome Attributo tipoBase default

*tipoBase*: predefinito o definito dall’utente
sempre racchiuso tra “#” e “;”

```
"class  #htmlclass;  #IMPLIED
 style  #htmlstyle;  #IMPLIED
 title  #STRING;     #IMPLIED" >
```

Gli attributeGroup possono essere associati ad un tipo complesso, ad un element o ad un altro attributeGroup.

```
<!ENTITY @ hierarchy  "(num?, title?)" "##corereq;">
<!ELEMENT noteref  @markeropt; >
<!ATTLIST noteref  num #STRING #REQUIRED  ##link;>
```

### 4.5.2 SubstitutionGroup

I gruppi di sostituzione permettono di realizzare il polimorfismo, caratteristica tipica dei linguaggi Object Oriented.

Un elemento dichiarato **substitution** può essere usato, in maniera interscambiabile, al posto di un elemento globale detto **head element**. La sola restrizione imposta è che i membri del gruppo di sostituzione e l’head element siano dello stesso tipo o appartengano allo stessa gerarchia di derivazione.

Di seguito è presentato un DTD++ e l’istanza che lo valida.

```
<!TARGETNS ex "http://www.example.com">
```

**SubstitutionGroup**

*Sintassi:* <!ELEMENT nomeSubstitutionGroup nomeElemento>
<!ELEMENT nomeSubstitutionGroup nomeElemento tipo>

*tipo:* stesso tipo o una valida derivazione
del tipo dell'head element

```
<!ELEMENT book #STRING>

<!ELEMENT magazine ex:book>

<!ELEMENT library (ex:book)*>

<library xmlns="http://www.example.com">
 <magazine>MSDN Magazine</magazine>
 <book>Professional XML Databases</book>
</library>
```

Il content model dell'elemento `library` permette di dichiarare uno o più elementi `book`. Poiché l'elemento `magazine` fa parte del gruppo di sostituzione di `book`, nell'istanza XML, laddove ci si aspetta `magazine`, si può utilizzare l'elemento `book`.

## 4.6 Limiti del linguaggio

Come si è visto l' introduzione dei gruppi di sostituzione rappresenta una delle principali caratteristiche Object Oriented. L'unica restrizione imposta ai membri del gruppo di sostituzione, è che l'head element sia globale e che i substitutionGroup abbiano associato lo stesso tipo (o una valida derivazione) dell'-

head element. In questo modo è possibile utilizzare in maniera interscambiabile l'head element o i membri del gruppo di sostituzione.

Una possibile estensione del linguaggio potrebbe prevedere l'introduzione dell'attributo *abstract* che permetterebbe di realizzare un analogo delle classi astratte del paradigma Object Oriented. Infatti un elemento dichiarato abstract non può essere utilizzato nell'istanza del documento ma deve essere necessariamente sostituito.

Inoltre XML Schema prevede l'utilizzo dell'attributo *final* che impone agli elementi e ai datatype di non poter essere sostituiti o derivati all'interno dello schema. Un elemento final non può, quindi, essere scelto come head element di un gruppo di sostituzione, mentre un tipo semplice o complesso dichiarato final non può essere utilizzato come base per una derivazione.

Introducendo in DTD++ la possibilità di dichiarare le componenti come final, si otterrebbe un maggior controllo sui meccanismi di derivazione e sostituzione.

Inoltre DTD++ non fornisce supporto per la gestione dell'unicità. Si potrebbero, quindi, implementare dei costrutti analoghi all'*unique, selector* e *field* e dare all'utente la possibilità di specificare che un elemento o un attributo, debba essere unico all'interno dello stesso scope.

Con l'introduzione di un costrutto *redefine* si renderebbe, poi, il DTD++ ancora più modulare. Infatti, come *include*, consente di includere uno schema che appartiene allo stesso targetNamespace ma in più dà all'utente la possibilità di ridefinire i tipi semplici e complessi e i gruppi di elementi e attributi.

# Capitolo 5

# PreValidator for DTD++

## 5.1 Introduzione

Nella prima versione di DTD++ ci si avvaleva di un parser validante (DTD++ for Xerces) [Amo02] basato sul parser Xerces Java 2. Xerces Java è un progetto open-source che nasce come diretto discendente di XMLJ4 di IBM e si prepone l'obiettivo di creare un parser validante per documenti XML.

Alcune modifiche apportate ai moduli di Xerces dedicati allo scanning del DTD classico, hanno permesso di supportare le nuove dichiarazioni DTD++. In questo modo si è potuto generare una grammatica DTD++ e, da questa, lo schema per validare il documento XML istanza del DTD++.

Principalmente, il parser Xerces Java 2 è stato modificato per poter analizzare le nuove dichiarazioni e salvare i dati. Inoltre, partendo dai dati salvati nelle strutture, è stato creato un parser vero e proprio che ha permesso di generare l'intero DOM corrispondente alla grammatica estesa. L'equivalenza tra le strutture dichiarate in DTD++ e in XML schema ha poi permesso di ottenere le corrispondenti dichiarazioni XML Schema e di effettuare la validazione del documento XML.

La scelta di avvalersi di Xerces ha portato all'indiscutibile vantaggio di aver utilizzato un'architettura testata e ben fun-

zionante supportata dall'ambiente open-source. Tale decisione ha tuttavia lo svantaggio di dover sottostare alle limitazioni imposte da un'architettura preesistente. Inoltre le modifiche effettuate sul codice sorgente del parser Xerces Java 2 non sono disponibili per le versioni successive di DTD++ for Xerces.

Nella versione successiva si è quindi preferito realizzare ex novo un preprocessore per la validazione di documenti XML tramite DTD++ [Fio03]. PreValidator for DTD++ effettua un parsing del DTD++ e poi genera una trasformazione in XML Schema che consente di validare il documento XML preso in input.

Per ottenere una massima flessibilità di utilizzo, i controlli sulla validità dello schema generato non sono stati effettuati dal preprocessore ma delegati al validatore esterno. In particolare sono stati utilizzati MSXML[1] e il GTRI Validation Tool 1.1 che ha permesso di utilizzare Xerces Java (che rientra nel progetto Apache[2]) come validatore.

Scritto interamente in Java per aumentarne la portabilità, PreValidator ricorre a tecnologie esterne solo per le fasi di scanning e di parsing consentendo, così, la massima flessibilità di utilizzo. In questo modo si è ottenuto un controllo completo del flusso di esecuzione senza aver alcuna limitazione sulla implementazione delle funzionalità.

Se da una parte con PreValidator si è perso il vantaggio di utilizzare un'architettura testata e ben funzionante, dall'altra si è guadagnato sull'utilizzo di validatori esterni. Infatti si ha la possibilità di far uso delle migliori tecnologie senza dover apportare modifiche al codice e, soprattutto, senza perdere tali modifiche a causa di una nuova versione del parser.

Come si è visto il PreValidator generava una conversione a XML Schema per poter validare i documenti XML salvandola

[1] http://msdn.microsoft.com/library/en-us/dnanchor/html/anch_xmlprod.asp
[2] http://xml.apache.org

in un file temporaneo che veniva successivamente eliminato. La versione attuale di PreValidator va oltre la validazione e permette all'utente, di salvare la conversione prodotta in un file specificato. È anche possibile indicare il percorso desiderato. Grazie ad alcune modifiche e all'estensione della sintassi, è possibile utilizzare, in maniera indipendente, il preprocessore come convertitore a XML Schema.

In particolare si è fornito supporto per la gestione dei namespace e per i meccanismi di importazione ed inclusione dei file. Per ogni namespace viene generato un file XML Schema (chiamato *namespace*.xsd) che viene successivamente importato all'interno del documento convertito.

Se all'interno del documento DTD++ sono presenti delle entity SYSTEM, il file indicato dal percorso dell'entità, verrà a sua volta convertito in XML Schema e salvato secondo le scelte effettuate.

L'utente, in fase di conversione, può inoltre decidere se applicare uno stile di progettazione XML Schema: Russian Doll, Salami Slice, Venetian Blinds e Garden of Eden. A seconda dello stile scelto, lo schema generato avrà componenti locali o globali. Se non viene selezionato alcuno stile, per default verrà applicato lo stile Salami Slice.

In seguito, per estendere le funzionalità di PreValidator e fornire un'ulteriore apertura verso lo standard, si è data la possibilità di effettuare la conversione a DTD. Nella maggior parte dei casi si avrà una perdita di informazioni dovuta alla maggiore espressività di DTD++. Analogamente a quanto accade nella generazione dello schema, i file richiamati tramite le entity SYSTEM e le entity PUBLIC, vengono processati e convertiti in DTD mantenendo le scelte dell'utente. Nella sezione seguente verranno illustrate le principali scelte effettuate nella generazione dei DTD.

In una prima fase comune vengono quindi effettuate le opera-

zioni di scanning e di parsing dei documenti DTD++. L'unica differenza è data dal comportamento delle entità parametriche. Infatti, mentre in XML Schema è necessario andarle ad espandere già durante la prima fase, in DTD devono essere mantenute e conservate.

Successivamente, in base alla scelta dell'utente, si avrà una conversione a DTD oppure a XML Schema.

Grazie alla grande modularità di PreValidator, si potranno ottenere conversioni ad altri linguaggi di validazione, come ad esempio RELAX NG o Schematron, aumentando la portabilità del preprocessore e fornendo ulteriori possibilità di interscambio.

## 5.2 Conversione a DTD

Come si è visto la conversione a DTD porta, in alcuni casi, alla perdita di informazione. Infatti la sintassi DTD++ è un'estensione della sintassi DTD che ha portato all'introduzione di concetti e costrutti propri di XML Schema. Primo fra tutti il concetto di tipo applicato sia agli elementi che agli attributi e i meccanismi di derivazione.

Nella conversione a DTD si è dunque applicato il concetto secondo cui ogni documento valido in DTD è valido in DTD++ e ogni documento non valido in DTD++ è non valido in DTD.

### 5.2.1 I tipi

Il DTD non consente di dichiarare in alcun modo i tipi. Si è, quindi, fatto uso delle entità parametriche, caratteristica significativa di DTD. In fase di conversione ogni tipo DTD++ è stato così trasformato in un'entità parametrica. Inoltre, a differenza dei tipi, è richiesto che le entità parametriche siano dichiarate prima di poter essere utilizzate.

Si è dovuto quindi verificare che tale ordine fosse rispettato e, nel caso contrario, è stato necessario effettuare uno spostamento. Di conseguenza l'ordine di input e di output potrà non essere lo stesso.

#### 5.2.1.1 I tipi semplici

Poiché i tipi semplici predefiniti di DTD++ non trovano una corrispondenza in DTD, nella conversione a DTD i tipi predefiniti associati ad un elemento o ad un tipo,vengono trasformati in entità parametriche. È stato, quindi, realizzato un file statico "pst.dtd" che, incluso in ogni generazione di DTD, contiene le dichiarazioni di entità equivalenti ai tipi predefiniti di DTD++. Per ogni tipo predefinito ci sarà, così, una corrispondente entità parametrica dichiarata `#PCDATA`.

Ad esempio l'elemento `uri` a cui è associato il tipo `#ANYURI` e il tipo semplice `myInteger`, restrizione di `#INTEGER`,

```
<!ELEMENT uri #ANYURI>
<!ENTITY # myInteger "#INTEGER">
```

diventeranno

```
<!ELEMENT uri (%ANYURI;)>
<!ENTITY % myInteger "%INTEGER;" >
```

e nel file pst.dtd si avranno le dichiarazioni delle entità parametriche `ANYURI` e `INTEGER`.

```
<!ENTITY % ANYURI "#PCDATA">
<!ENTITY % INTEGER "#PCDATA" >
```

Nella conversione degli attribute, invece, si è trasformato ogni tipo stringa nel tipo `#CDATA`.

Ad esempio

```
<!ATTLIST person age #INTEGER>
```

diventerà

```
<!ATTLIST person age CDATA #IMPLIED >
```

#### 5.2.1.2 I tipi complessi

I tipi complessi di DTD++ sono stati trasformati in entità parametriche. Per quanto riguarda gli operatori di ripetizione il range richiesto è stato sostituito con il suo più piccolo sottoinsieme definibile di valori. Infatti, oltre agli operatori classici, DTD++ permette di specificare un valore di ripetizione tramite intervalli con parentesi quadre. Ad esempio l'elemento `a` ripetuto con frequenza `[1,5]` `[0,5]` o `[2,5]` verrà convertito rispettivamente in `a+`, `a*`, `a+`[3]. Ad esempio

```
<!ENTITY @ libro "(autore?,titolo[0,5],pagine[2,9])*">
```

diventerà

```
<!ENTITY % libro "(autore?,titolo*,pagine+)*" >
```

DTD++ permette, inoltre, di esprimere attributi all'interno dei tipi complessi. In questo caso l'elemento che dichiara essere di quel tipo, avrà associato l'attlist dichiarato in tale tipo.

Quindi

```
<!ENTITY @ personType "(name,surname)" "age #STRING">
<!ELEMENT person @personType;>
```

genererà

```
<!ENTITY % personType "(name,surname)" >
<!ELEMENT person %personType; >
<!ATTLIST person age CDATA #IMPLIED >
```

DTD, per di più, non prevede un costrutto analogo a ALL di XML Schema. L'unico modo di esprimere il tipo complesso `uno` presentato di seguito

[3]NOTA: In realtà un intervallo del tipo [2,5] trova la sua conversione in aa+

```
<!ENTITY @ uno "(a & b & c)">
```

è di ricorrere ad una sequenza di choice.

```
<!ENTITY % uno "((a,b,c)|(b,a,c)|(c,a,b)|
                (c,b,a)|(b,c,a)|(a,c,b))" >
```

### 5.2.2 La derivazione

DTD non prevede meccanismi di derivazione dei tipi. In particolare non vi è modo di convertire in DTD i facet e si avrà quindi un'inevitabile perdita di informazioni nella trasformazione da DTD++.

#### 5.2.2.1 Derivazione dei tipi semplici

Come si è visto precedentemente, DTD++ consente tre procedimenti per derivare i tipi semplici: per restrizione, per lista e per unione. In tutti e tre i casi si è associato alle strutture un'entità parametrica tralasciando la derivazione.

Nell'esempio viene mostrato come la derivazione per restrizione e per lista diventi un semplice utilizzo di entità.

```
<!ELEMENT person #personType;\c\>
<!ATTLIST person age #INTEGER+\c\>
```

sarà converito in

```
<!ELEMENT person %personType; >
<!ATTLIST person age CDATA #IMPLIED >
```

Per quanto riguarda la derivazione per unione, invece, si è assunto di richiamare il primo tipo incontrato in quanto entrambi i tipi verrebbero espansi con un dichiarazione `#PCDATA`.

Di conseguenza

```
<!ENTITY # myEntity "#FLOAT|#STRING{10}">
```

diventerà

```
<!ENTITY % myEntity "#FLOAT;" >
```

#### 5.2.2.2 Derivazione dei tipi complessi

Nella derivazione per restrizione viene dichiarato solo il nuovo content model. Se sono presenti degli attributi, verrà seguito lo stesso procedimento indicato sopra.

Ad esempio

```
<!ENTITY @ author @person; "(name,born)"
         "age #INTEGER #IMPLIED id #STRING #IMPLIED">
<!ELEMENT character @author;>
```

sarà convertito in

```
<!ENTITY % author "(name,born)" >
<!ELEMENT character %author; >
<!ATTLIST character age  CDATA #IMPLIED
                    id  CDATA #IMPLIED >
```

Nella derivazione per estensione, invece, bisogna distinguere se questa viene effettuata in maniera anonima oppure no. Nel caso in cui un tipo complesso nominato estenda un altro tipo come nell'esempio sotto riportato

```
<!ENTITY @ basePersons "(author,character)*">
<!ENTITY @ persons "@basePersons;,(editor)[2,100]"
                   "year #GYEAR #IMPLIED">
```

si avrà un DTD della forma

```
<!ENTITY % basePersons "(author,character)*" >
<!ENTITY % persons "%basePersons;,(editor)+" >
```

Altrimenti, se si tratta di una derivazione locale all'interno di un elemento, verrà creata una nuova entità parametrica che sarà chiamata *nomeElemento_entity*.

```
<!ENTITY @ personType "(firstname)">
<!ELEMENT person (@personType;,
         (firstname,secondname[0,1]))>
```

Poichè l'elemento `person` dichiara nel proprio content model un tipo anonimo che estende il tipo complesso `personType`, nel DTD si avrà una nuova entità parametrica `person_entity` e l'elemento `person` avrà associato l'entità `person_entity` come è mostrato qui di seguito.

```
<!ENTITY % personType "(firstname)">
<!ENTITY % person_entity "(%personType;,
          firstname,secondname?)">
<!ELEMENT person %person_entity;>
```

### 5.2.3 Componenti locali e globali

DTD non permette di distinguere tra componente locale e componente globale. In fase di conversione non si è così tenuto conto delle due rappresentazioni applicando le scelte di trasformazione sopra presentate. Le dichiarazioni degli elementi `titolo`, definito localmente a `fronte`, e `autore`, invece referenziato, diventeranno semplicemente due dichiarazione di elementi come previsto dal DTD.

```
<!ELEMENT fronte (titolo,$autore)>
<!ELEMENT fronte.titolo #STRING>
<!ELEMENT autore #STRING>
```

in DTD diventerà

```
<!ELEMENT fronte (titolo,autore)>
<!ELEMENT titolo %STRING;>
<!ELEMENT autore %STRING;>
```

### 5.2.4 Element e Attribute Groups

I gruppi di elementi e di attributi sono stati convertiti in entità parametriche seguendo le stesse regole applicate ai tipi semplici e complessi. Nell'esempio sotto riportato il gruppo di elementi `painting` sarà trasformato nell'entità `painting`.

```
<!ENTITY @@ painting "(monet*,(renoir?|vangogh)[7,9])">

<!ENTITY % painting "monet*,(renoir?|vangogh)+">
```

Analogamente il gruppo di attributi `HTMLattrs`

```
<!ENTITY ## HTMLattrs "class #htmlclass; #IMPLIED
                       style #htmlstyle; #IMPLIED">
```

diverrà un'entità parametrica.

```
<!ENTITY % HTMLattrs "class %htmlclass; #IMPLIED
                      style %htmlstyle; #IMPLIED">
```

### 5.2.5 SubstitutionGroup

Non fornendo DTD un costrutto analogo, i gruppi di sostituzione sono stati assimilati ad un'entità parametrica. Come esempio si supponga di avere un tipo complesso `book` che contiene al suo interno gli elementi `section`, `part` e `paragraph`. L'elemento `part`, a cui è associato il tipo `hierarchy`, sia poi l'head element del gruppo di sostituzione che ha come membri `partie` e `parte`.

```
<!ENTITY @ book "(section|part|paragraph)">
<!ELEMENT partie part>
<!ELEMENT parte part>
<!ELEMENT part @hierarchy;>
```

Il gruppo di sostituzione diventerà così un'entità parametrica il cui nome sarà *DPPnomeSubGroupSG* e, in questo caso, sarà `DPPpartSG`. Ogni volta in cui si fa riferimento all'head element, il nome dell'head element verrà modificato con quello dell'entità creata.

```
<!ENTITY % DPPpartSG "(part|partie|parte)">
<!ENTITY % book "section|%DPPpartSG;|paragraph">
<!ELEMENT part %hierarchy;>
```

Si noti come all'interno del tipo `book` non venga più fatto riferimento all'elemento `part` ma all'entità `DPPpartSG`.

Nel caso in cui l'head element venga utilizzato all'interno di un content model misto, l'entità generata avrà nome *DPPnomeSubGroupSGMC*. Riprendendo lo stesso esempio verrà quindi generato

```
<!ENTITY % DPPpartSGMC "part|partie|parte">
<!ENTITY % book "(#PCDATA|section|%DPPpartSGMC;
                  |paragraph" >
```

Dall'analisi trattata sopra si nota come DTD++ sia un'estensione sintattica di DTD che permette di esprimere costrutti con un'espressività paragonabile ad XML Schema senza tuttavia cadere nella sua verbosità. Inoltre grazie alle modifiche apportate a Prevalidator è sempre possibile ottenere una conversione standard ad DTD o XML Schema.

Viene ora riportato un esempio di DTD++ e dalle relative trasformazioni in XML Schema e DTD in cui viene descritto un documento.

```
<!ROOT documento @tipoDocumento;>
```

```
<!ATTLIST documento online #BOOLEAN 'no'>
<!ENTITY @ tipoDocumento "(libro|rivista)"
           "tipo #STRING 'scientifico'">
<!ELEMENT libro (@@fronte;,corpo+)>
<!ELEMENT rivista (parte)+>
<!ATTLIST rivista ##rivistaAttr;>
<!ENTITY ## rivistaAttr "uscita
            #STRING(mensile|settimanale)">
<!ENTITY @@ fronte "(titolo,autore[1,100],editore?)">
<!ELEMENT titolo @tipoTitolo;>
<!ENTITY @ tipoTitolo "" "lingua #STRING(it|fr|in)">
<!ELEMENT autore #STRING>
<!ELEMENT editore #STRING>
<!ELEMENT corpo (capitolo+,titolo,para+,parte*)>
<!ELEMENT capitolo (titolo,para+)>
<!ELEMENT parte @parteStile;>
<!ELEMENT partie parte>
<!ELEMENT part parte>
<!ENTITY @ parteStile "@stile;,(capitolo,para)">
<!ELEMENT para @stile;>
<!ENTITY @ stile "(#PCDATA|grassetto|corsivo)*">
<!ELEMENT grassetto #STRING>
<!ELEMENT corsivo #STRING>
```

L'elemento `documento` è dichiarato come elemento radice ed ha associato un tipo complesso e un attributo. Può così essere o un libro o una rivista e, tramite l'attributo `tipo` si identifica se si tratta di un documento scientifico oppure no.

L'elemento `libro` è costituito dal frontespizio e dal corpo che che costituisce il libro vero e proprio. `libro` dichiara quindi al proprio interno un gruppo di elementi `titolo`, `autore`, `editore` e l'elemento `corpo`. Mentre `autore` ed `editore` hanno associato il tipo semplice `#STRING`, titolo ha associato il tipo

complesso `tipoTitolo` che, tramite l'attributo `lingua` identifica se la lingua utilizzata è l'italiano, il francese o l'inglese.

L'elemento `corpo` è una sequenza di capitoli, titoli, paragrafi e parti. L'elemento `parte` viene dichiarato come head element del gruppo di sostituzione di cui fanno parte gli elementi `partie` e `part`. Tutti e tre questi elementi hanno associato il tipo complesso `parteStile` che è ottenuto tramite una derivazione per estensione del tipo `stile`. A sua volta `stile` è un tipo complesso dal content model misto in cui, oltre al testo, può essere presente il grassetto o il corsivo.

Infine l'elemento `rivista` è costituito da uno o più elementi `parte` e ha associato un gruppo di attributi `rivistaAttr` in cui l'attributo `uscita` permette di specificare se si tratta di una rivista settimanale o mensile.

In conversione a XML Schema si avrà:

```
<xsd:schema elementFormDefault="qualified"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="documento">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="tipoDocumento">
                    <xsd:attribute name="online"
                         type="xsd:boolean" default="no"/>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="tipoDocumento" >
        <xsd:choice >
```

```
            <xsd:element ref="libro" />
            <xsd:element ref="rivista" />
        </xsd:choice>
        <xsd:attribute name="tipo" type="xsd:string"
                       default="scientifico" />
    </xsd:complexType>

    <xsd:element name="libro">
        <xsd:complexType >
            <xsd:sequence >
                <xsd:group ref="fronte" />
                <xsd:element ref="corpo" minOccurs="1"
                             maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="rivista">
        <xsd:complexType >
            <xsd:sequence minOccurs="1"
                          maxOccurs="unbounded" >
                <xsd:element ref="parte" />
            </xsd:sequence>
            <xsd:attributeGroup ref="rivistaAttr"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:attributeGroup name="rivistaAttr">
        <xsd:attribute name="uscita" >
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="mensile" />
                    <xsd:enumeration value="settimanale" />
```

```
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
</xsd:attributeGroup>

<xsd:group name="fronte">
    <xsd:sequence>
        <xsd:element ref="titolo"/>
        <xsd:element ref="autore" minOccurs="1"
                     maxOccurs="100" />
        <xsd:element ref="editore" minOccurs="0"
                     maxOccurs="1" />
    </xsd:sequence>
</xsd:group>

<xsd:element name="titolo" type="tipoTitolo" />

<xsd:complexType name="tipoTitolo" >
    <xsd:attribute name="lingua" >
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="it" />
                <xsd:enumeration value="fr" />
                <xsd:enumeration value="in" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
</xsd:complexType>

<xsd:element name="autore" type="xsd:string"/>

<xsd:element name="editore" type="xsd:string"/>
```

```
<xsd:element name="corpo">
    <xsd:complexType >
        <xsd:sequence >
            <xsd:element ref="capitolo" minOccurs="1"
                         maxOccurs="unbounded" />
            <xsd:element ref="titolo" />
            <xsd:element ref="para" minOccurs="1"
                         maxOccurs="unbounded" />
            <xsd:element ref="parte" minOccurs="0"
                         maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="capitolo">
    <xsd:complexType >
        <xsd:sequence >
            <xsd:element ref="titolo" />
            <xsd:element ref="para" minOccurs="1"
                         maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="parte" type="parteStile" />

<xsd:element name="partie" substitutionGroup="parte"/>

<xsd:element name="part" substitutionGroup="parte"/>

<xsd:complexType name="parteStile" mixed="true">
    <xsd:complexContent>
        <xsd:extension base="stile">
```

```
                <xsd:sequence >
                    <xsd:element ref="capitolo" />
                    <xsd:element ref="para" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:element name="para" type="stile" />

    <xsd:complexType name="stile" mixed="true">
        <xsd:choice minOccurs="0" maxOccurs="unbounded" >
            <xsd:element ref="grassetto" />
            <xsd:element ref="corsivo" />
        </xsd:choice>
    </xsd:complexType>

    <xsd:element name="grassetto" type="xsd:string"/>

    <xsd:element name="corsivo" type="xsd:string"/>

</xsd:schema>
```

Come si può notare la conversione ad XML Schema risulta essere più verbosa e di lunghezza due o tre volte maggiore. Lo stile di rappresentazione scelto è il Salami Slice, stile applicato anche per default.

Tutti gli elementi sono così dichiarati globali e vengono dunque riutilizzati più volte. I tipi complessi, anonimi o nominati, vengono definiti tramite riferimenti ai vari elementi globali. Ad esempio il tipo `tipoDocumento` può contenere l'elemento `libro` o l'elemento `rivista` ed ha associato l'attributo `tipo` dichiarato localmente.

Gli attributi associati agli elementi vengono dichiarati all'interno di un tipo complesso anonimo. Ad esempio l'attributo `online`, associato all'elemento `documento`, estende il tipo complesso `tipoDocumento` associato a `documento`.

Infine viene riportato il DTD equivalente.

```
<!ENTITY % PredefinedSimpleTypes SYSTEM "./pst.dtd">
%PredefinedSimpleTypes;

<!ENTITY % tipoDocumento "(libro|rivista)" >
<!ELEMENT documento %tipoDocumento; >
<!ATTLIST documento tipo  CDATA 'scientifico' >
<!ATTLIST documento online  CDATA 'no'  >
<!ELEMENT libro (%fronte;,corpo+)>
<!ENTITY % DPPparteSG "(parte|partie|part)">
<!ELEMENT rivista (%DPPparteSG;)+>
<!ENTITY % rivistaAttr " uscita (mensile|settimanale)
                         #IMPLIED" >
<!ATTLIST rivista %rivistaAttr; >
<!ENTITY % fronte "titolo,autore+,editore?">
<!ENTITY % tipoTitolo "EMPTY" >
<!ELEMENT titolo %tipoTitolo; >
<!ATTLIST titolo lingua (it|fr|in) #IMPLIED >
<!ELEMENT autore (%STRING;) >
<!ELEMENT editore (%STRING;) >
<!ELEMENT corpo (capitolo+,titolo,para+,%DPPparteSG;)>
<!ELEMENT capitolo (titolo,para+)>
<!ENTITY % parteStile "%stile;,(#PCDATA|capitolo|para)" >
<!ELEMENT parte %parteStile; >
<!ENTITY % stile "(#PCDATA|grassetto|corsivo)*" >
<!ELEMENT para %stile; >
<!ELEMENT grassetto (%STRING;) >
<!ELEMENT corsivo (%STRING;) >
```

Per prima cosa, tramite un'entità SYSTEM, vengono incluse le entità parametriche equivalenti ai tipi predefiniti di DTD++. Si noti poi come i tipi, i gruppi di attributi e di elementi siano diventati, a loro volta, delle entità parametriche.

Il gruppo di sostituzione con head element `parte` è stato trasformato nell'entità parametrica `DPPparteSG` e tutti i riferimenti a `parte` sono stati sostituiti con un riferimento all'entità `DPPparteSG`.

È importante notare che `tipoTitolo` è stato trasformato in un'entità dal content model EMPTY. Inoltre l'elemento `titolo` ha ora associato l'attributo `lingua` prima dichiarato localmente all'interno del tipo complesso `tipoTitolo`.

Infine l'esempio di conversione a DTD sopra riportato sottolinea il differente ordine di input e output. Infatti, mentre DTD++ non richiede alcun ordine, in DTD è necessario che ogni entità, prima di essere utilizzata, sia stata dichiarata in precedenza nello stesso file o in un file esterno.

Come si può notare la sintassi di DTD++ risulta concisa e compatta secondo il paradigma del DTD senza tuttavia perdere la grande espressività di XML Schema.

## 5.3 Vantaggi

L'utilizzo di una nuova tecnologia porta sempre ad una valutazione relativa ai vantaggi e ai limiti dovuti alla scelta effettuata. Infatti le esigenze e le aspettative possono variare a seconda del campo e del dominio di applicazione. DTD++ porta con sé una serie di vantaggi in grado di soddisfare esigenze varie e differenti:

- **apprendimento veloce del linguaggio:** poiché DTD++ estende la sintassi di DTD, per utilizzare il nuovo linguaggio è sufficiente imparare soltanto i nuovi costrutti;

- **velocità nello scrivere documenti:** la sintassi di DTD++ è una sintassi essenziale e permette quindi di scrivere rapidamente documenti senza cadere nella verbosità di XML Schema;
- **semplicità e visibilità dei tag:** la semplicità di DTD++ rende un documento più leggibile e comprensibile rispetto al corrispondente XML Schema. In questo modo modificare e correggere eventuali errori risulta essere più veloce e semplice.
- **espressività:** DTD++ è un linguaggio di grande capacità espressiva che permette di scrivere documenti di validazione molto dettagliati e fortemente tipati. Contrariamente a DTD, permette di utilizzare e derivare i tipi predefiniti di XML Schema e di definire una grande varietà di nuovi tipi. Inoltre si possono esprimere quei vincoli tipici dei linguaggi rule-based senza perdere le caratteristiche di un linguaggio grammar-based.
- **utilizzo dei software che sfruttano XML Schema o DTD:** apportando delle modifiche al preprocessore è possibile utilizzare qualsiasi software che prenda in input un file XML Schema o DTD. Ad esempio si può utilizzare PreValidator all'interno dell'editor XML <oXygen/>[4] semplicemente selezionando Strumenti esterni dal menu Strumenti e andando ad aggiungere un nuovo strumento.
- **conversione a XML Schema o DTD:** poter generare un documento XML Schema o DTD permette di ottenere uno standard. In questo modo è ad esempio possibile ricorrere ad un documento DTD++ per un uso interno e adoperare una versione XML Schema o DTD dello stesso documento per un uso esterno.

[4] http://www.oxygenxml.com/

- **validazione effettuata tramite programmi testati:** la validazione è affidata completamente a software testati e largamente usati. Un'estensione della sintassi DTD++ non ricade quindi nel processo di validazione ma è semplicemente risolta con la generazione del corrispondente schema o DTD.

## 5.4 Limiti

Come si è visto in fase di conversione a DTD++, in presenza di operatori di conversione, si è scelto di trasformare l'insieme desiderato nel più piccolo sottoinsieme definibile di valori.

In questo modo un elemento `a` ripetuto da due a cinque volte viene convertito in `a+`, ovvero da una ad infinite volte. In realtà la conversione più appropriata sarebbe `aa+` rispettando così il fatto che il minimo numero di occorrenze possibili è due e non uno.

Inoltre per aumentare la leggibilità dei documenti convertiti restituiti all'utente si potrebbero implementare funzionalità per generare in maniera automatica commenti analogamente a quanto avviene con Javadoc. I commenti dell'utente verrebbero così gestiti in maniera opportuna e inseriti all'interno delle strutture portando ad un maggior leggibilità e chiarezza.

# Capitolo 6

# Dettagli implementativi

In questo capitolo viene descritta l'implementazione del preprocessore per la validazione di documenti XML tramite documenti DTD++. Per permettere una maggiore portabilità, il preprocessore è stato sviluppato interamente utilizzando il linguaggio Java.

La validazione non è effettuata direttamente da preValidator ma viene richiamato un validatore esterno riportando poi gli eventuali errori. Come validatori esterni si sono utilizzati MSXML e Xerces ma con semplici modifiche è possibile utilizzare altri validatori.

Nella nuova versione PreValidator va ben oltre la validazione di un documento XML e, prendendo in input un DTD++, permette di generare un documento XML Schema o DTD. Allo stesso modo sarà possibile ottenere conversioni in altri linguaggi quali ad esempio Relax NG o Schematron.

## 6.1 Flusso del programma

Il file DTD++ viene caricato dal disco e passato alla classe DppDocManager che indicizza il file per righe. Successivamente il lexer riceve il documento e ne estrae le entità parametriche. Nel caso in cui sia richiesta la conversione a XML Schema, le

entità parametriche saranno sostituite. Al contrario, in una conversione a DTD le entità parametriche verranno mantenute. Il lexer, inoltre, inserisce in liste separate i nomi dei tipi semplici, dei tipi complessi, degli attributeGroup e dei Group per poter controllare che componenti referenziate siano state dichiarate nello stesso o in altri documenti DTD++. Se viene incontrata un'entità SYSTEM, il lexer si blocca mantenendo settati i parametri, carica il file associato all'entity SYSTEM e lo processa. Stessa cosa se viene trovata un'entity PUBLIC ed è richiesta una conversione a DTD. Una volta che il documento è stato elaborato dal lexer, viene passato al parser. Al termine del parsing viene così restituito un vettore contenente la lista dei tag. Da questa lista vengono poi estratti i tipi, le attributeList, gli attributeGroup e i Group. AttributeGroup e attributeList vengono successivamente associati agli elementi relativi.

A questo punto la struttura è pronta per emettere la trasformazione del documento in schema o in DTD. Nel passo successivo la lista dei tag viene scorsa e ogni singolo tag emette il proprio schema o DTD. Se si vuole validare il documento ed è stato generato uno schema, viene richiamato il validatore esterno che valida il documento. Gli errori generati dal validatore vengono intercettati dal preprocessore che, tramite il DppDocManager, risolve i numeri di linea e restituisce gli errori sullo standard output con numeri di riga relativi al dpp.

## 6.2 Tecnologie utilizzate

Per la realizzazione del preprocessore sono state utilizzate tecnologie esterne. Per il parsing del documento si sono impiegati Jflex e JavaCup, mentre per la validazione MSXML e Xerces.

### 6.2.1 JFlex

JFlex[1] è un generatore di analizzatore lessicale per java ed è scritto in java. È una reimplementazione di *jlex*, tool molto comune realizzato da Elliot Berk dell'Università di Princeton. JFlex è stato realizzato per lavorare con JavaCup. Inizialmente si era scelto Jlex come lexer per la realizzazione del preprocessore ma le features di JFlex e soprattutto la miglior gestione delle espressioni regolari e degli errori, hanno fatto cadere la scelta su JFlex.

### 6.2.2 JavaCup

JavaCup[2] (Constructor of Useful Parsers) è un sistema per la generazione di parser LARL. Ricopre lo stesso ruolo del largamente usato YACC e ne offre la maggior parte delle features. JavaCup è scritto in java e produce un parser implementato in java. Utilizza JFlex per creare i token.

### 6.2.3 MSXML

MSXML[3] è una API di Microsoft basata su COM per parsare e processare documenti XML. La versione utilizzata include il supporto per il DOM Level 2.0, SAX 2.0, XPath, XSLT e XSD Schema. Per utilizzare MSXML è stata realizzata un'applicazione in C# che effettua l'operazione di validazione. Viene preso in input un file XML e un file XML Schema, si effettua la validazione utilizzando le classi XmlSchemaCollection e XmlValidatingReader e vengono restituiti sullo standard error gli eventuali errori.

[1] http://jflex.de/
[2] http://www.cs.princeton.edu/appel/modern/java/CUP
[3] http://msdn.microsoft.com/library/en-us/dnanchor/html/anch_xmlprod.asp

### 6.2.4 Xerces

Xerces Java è un progetto open-source che nasce come discendente diretto di XMLJ4 di IBM e si prefigge l'obiettivo di creare un parser validante per documenti XML. Fa parte del progetto Apache[4]. Per utilizzarlo come validatore è stato utilizzato il GTRI Validation Tool 1.1. Il tool è stato realizzato dalla Georgia Tech Research Corporation ed è scritto interamente in java. Fondamentalmente è un programma che si limita a rendere Xerces un validatore XML standalone.

## 6.3 Il programma

Il programma è strutturato all'interno del package preValidator che contiene la main class PreValidator.class e sei package:

- preValidator.documents: contiene le classi che gestiscono i file DTD++ e XML;
- preValidator.types: contiene le classi che gestiscono i tipi del documento DTD++;
- preValidator.errors: contiene le classi che gestiscono i messaggi di errore;
- preValidator.parsers: contiene le classi che effettuano lo scanning e il parsing;
- preValidator.parserStruct: contiene le classi che gestiscono la strutturazione e la rappresentazione del documento da generare;
- preValidator.utils: contiene una serie di classi di varia utilità.

[4]http://xml.apache.org

### 6.3.1 preValidator

Package principale contenente la main class PreValidator.

#### PreValidator

PreValidator è la main class del programma. Tramite il metodo pubblico *Start* viene presa in input la lista degli argomenti e si effettuano le chiamate alle operazioni necessarie a trasformare e validare i documenti. Dopo aver processato gli argomenti con il metodo *getArgs*, PreValidator inizializza il DppDocManager e le strutture dati relative alla gestione degli errori. Successivamente viene richiamato il metodo *Process* che effettua l'operazione di parsing e di generazione dello schema o del DTD. Se necessario, poi, prepara una versione modificata del documento XML e grazie al metodo *Validate* il documento viene validato. Al termine si eliminano i file temporanei e, se richiesto, viene stampato sullo stdout la conversione generata. Vi sono, poi, due code in cui vengono salvati i nomi dei file dichiarati con un'entity SYSTEM o PUBLIC. Quando le operazioni sul file principale sono terminate, *Start* richiama il metodo pubblico *getInlcude* in cui si scorre la lista dei file referenziati tramite un'entity SYSTEM. Questi, a loro volta, vengono processati e opportunamente convertiti. Se è richiesta una conversione a DTD, allora lo stesso procedimento viene ripetuto anche per i file definiti con le entity PUBLIC.

### 6.3.2 preValidator.documents

Nel package PreValidator.documents sono contenute le classi che si occupano della gestione dei documenti.

#### DppDocManager

DppDocManager realizza la gestione del file dpp. Prende in input il documento e tramite il metodo *process*, lo separa per linee utilizzando un vettore di LineItem. Terminate queste operazioni, il documento è pronto per essere passato al parser che andrà a creare il DOM da cui si otterrà l'opportuna conversione. *getPostLineByLine* è un metodo che permette di ottenere i riferimenti ai numeri di riga del documento originale utilizzati dal preprocessore per poter elaborare gli errori restituiti dal validatore.

#### DppLineItem

Si occupa di gestire il riferimento tra i numeri di riga attraverso le fasi di trasformazione del documento.

#### XmlDocManager

Gestisce il file XML da validare. Il metodo *setSchemaRef* genera una versione modificata dell'XML inserendo gli attributi xmlns:xi e xsi:noNameSpaceLocation all'interno del primo tag XML.

### 6.3.3 preValidator.types

Vi sono contenute le classi che si occupano della gestione dei tipi del documento DTD++.

#### TypeList

TypeList gestisce la lista dei tipi, degli AttList, degli attributeGroup e dei Group. Ad ogni tag estratto dal dpp corrisponde un tipo che viene inserito nella lista con il metodo *addType*. Se, invece, il tag corrisponde ad un AttList, si utilizza

il metodo *addAttGroup*. I tipi, gli AttList, gli attributeGroup e i Group vengono inseriti in Hashtable. Una volta riempita la struttura dati, i metodi *linkGroup* e *linkEntityGroup* associano, ripettivamente, gli AttList e gli attributeGroup ai relativi tipi.

*linkSubGroup*, *linkSubstitutionGroups* e *createEntitySubstitutionGroup* intervengono quando ci sono dei substitutionGroup per poter effettuare una corretta conversione a DTD. Vi sono, poi, dei metodi che effettuano ricerche all'interno della struttura: *getType* permette di estrarre l'oggetto Type relativo, *getStringType* restituisce il nome da inserire nello schema o nel DTD, *getBaseType* restituisce il nome del tipo da cui è derivato, *getOriginalType* restituisce il nome del tipo base. Il metodo *initPredefined*, richiamato dal costruttore della classe, inizializza i tipi predefiniti.

#### Type e classi derivate

Queste classi definiscono i tipi dei tag. Contengono il nome del tag, la stringa da inserire nella generazione dello schema o del DTD.

#### ParametricList

Gestisce le entità parametriche. Il metodo *put* inserisce le entità parametriche nella lista specificandone la chiave, i valori, il tipo e la posizione all'interno del file di origine. Si è resa ParametricList una classe Singleton per permettere l'utilizzo delle entità parametriche in file diversi da quello in cui sono dichiarate. Con il metodo *adjustList* si effettuano le sostituzioni all'interno della lista stessa.

I metodi *getValue* e *getKey* permettono di estrarre rispettivamente il valore e la chiave dell'entità parametrica e restituiscono null altrimenti. Tramite il metodo *getFileName* è possibile ottenere il nome del file di appartenenza dell'entità desiderata.

#### ScannerFileListCache

Classe Singleton che tiene traccia dei file da processare. Il metodo statico *getInstance* permette di avere una sola istanza della classe. L'Hashtable *hash* contiene la lista dei file. Tramite il metodo pubblico *addFileName* si inserisce il nome di un file all'interno della Hashtable e tramite *exist* si controlla se è già presente un file con lo stesso nome. Con il metodo pubblico *getFileName* vengono ritornate le chiavi della Hashtable.

### 6.3.4 preValidator.errors

Contiene le classi addette alla gestione dei messaggi di errore.

#### ErrorValue

Definisce l'errore e contiene i metodi per la stampa dei messaggi d'errore. Costruttori differenti permettono di costruire l'errore definendone il tipo, la posizione di inizio e fine dell'errore, una serie di valori da visualizzare al momento della stampa e un livello di criticità dell'errore. Con il metodo *toString* si ricava la stringa d'errore con un preciso riferimento alla riga in cui è avvenuto tale errore.

#### ErrorType

La classe ErrorType definisce le varie tipologie d'errore tramite una serie di valori statici.

#### ErrorList

Gestisce la lista degli errori. Il metodo *addElement* permette di aggiungere un ErrorValue. *size* ritorna il numero degli errori e *clear* azzera lo stato della lista. Con il metodo *toString* si crea, invece, una stringa che contiene lo stato d'errore degli ErrorValue contenuti nella lista.

#### ErrorConverter

Classe astratta che elabora i messaggi di errore restituiti dal validatore. Le classi derivate implementano il metodo *Elaborate-Error* che esegue l'operazione di trasformazione.

### 6.3.5 preValidator.parsers

Contiene le classi che effettuano lo scanning e il parsing. Jflex e JavaCup generano automaticamente le classi partendo dai seguenti file:

- parametric.flex genera le classi per la gestione delle entità parametriche. Quando si incontra un'entity SYSTEM o PUBLIC lo scanning del file corrente si interrompe e si processa il file riferito. Ogni entità parametrica, tipo, group o attributeGroup incontrato viene inserito in una propria coda per permettere che una componente possa essere richiamata ed utilizzata in un file differente da quello in cui è dichiarata.
- parser.flex genera le classi che creano i token per il parser;
- parser.par genera le classi che costituiscono il parser.

### 6.3.6 preValidator.parserStruct

In questo package sono contenute le classi che gestiscono la strutturazione e la rappresentazione del documento da generare.

#### Tags

Classe contenitore di tutti i tag. Il campo *items* contiene i tag generati dal parser. Nella conversione a XML Schema l'ordine dei tag è quello in cui sono stati trovati nel file d'origine. Invece nella conversione a DTD, l'ordine può cambiare se un'entità è

usata prima di essere stata dichiarata o se viene creata per permettere un corretto DTD. Il campo *declaration* contiene ciò che va posto all'inizio del documento generato. Il campo *typeList* contiene la lista dei tipi, degli attribute, degli attributeGroup e dei Group e viene inizializzata tramite il metodo *initTypeList*. Se è richiesta una conversione schema secondo lo stile Venetian Blind, Garden of Eden o Russian Doll vengono richiamati rispettivamente i metodi *initTypeListVB*, *initTypeListGE*, *initTypeListRD*.

*getDocType* e *getNameSpace* restituiscono il primo il valore da inserire all'interno del tag DOCTYPE dello schema, il secondo una stringa contenente il namespace dello schema. *CreateNewFile* richiama altri metodi e va a creare i file schema corrispondenti ai namespace dichiarati.

#### ContentModel

Classe astratta che definisce il content model. I seguenti campi definiscono i facet del content model:

- cardinality, le occorrenze minime e massime;
- length, la lunghezza minima, massima o esatta dell'elemento, in caso di numeri decimali definisce il numero di cifre intere e decimali;
- range, il valore minimo e massimo, inclusivo e non inclusivo, che il valore può assumere;
- pattern, un'espressione regolare;
- enumeration, una lista di valori;
- whitespace, la gestione degli spazi bianchi.

Il vettore *Childs* permette di associare al content model altri content model. In questo modo si crea una struttura ad albero. Se il vettore è vuoto, è necessario specificare un valore

nel campo *value*. Si identifica, così, il content model corrente come foglia all'interno dell'albero. Il metodo *add* aggiunge facet o altri child. Per ereditare i facet da un altro content model, senza però sovrascrivere facet già esistenti, si utilizza il metodo *setCard*. *setItem* permette di impostare direttamente il vettore dei childs ed è utilizzato in fase di generazione del DOM del parser.

Il campo *isAnonymous* definisce se il content model è di tipo anonimo. In caso affermativo la conversione a schema conterrà un <`simpleType`> se di tipo semplice e un <`complexType`> se di tipo complesso. Tramite i booleani *isGlobal* e *isLocal* si specifica se il content model deve essere dichiarato globalmente o localmente.

#### SimpleContentModel

Definisce il content model di un tipo semplice. Il campo *simplebaseType* definisce il tipo da cui deriva e deve essere sempre definito. *isList* indica se il content model debba comportarsi o meno come una lista. Il booleano *empty* settato a true identifica il content model senza contenuto e genera un tag di tipo empty.

#### ComplexContentModel

Definisce il content model di tipo complesso. In base al valore del campo *type* vengono gestiti i content model contenuti all'interno del vettore *childs*. CM_TYPE_CHOICE realizza uno schema o un dtd di tipo choice, CM_TYPE_SEQ di tipo sequence, CM_TYPE_ALL di tipo all. In caso di conversione a DTD il tipo CM_TYPE_ALL realizza un DTD di tipo choice.
Ad esempio

```
<!ENTITY @ uno  "a & b & c">
```

diventerà

```
<!ENTITY % uno "(a,b,c)|(a,c,b)|(b,a,c)|
                (b,c,a)|(c,a,b)|(c,b,a)" >
```

Se di tipo CM_TYPE_VALUE viene restituito un tag schema o DTD di tipo element che riferisce ad un tipo base definito nel campo *value*. CM_TYPE_ANY realizza uno schema o un DTD di tipo any. Il tipo di any viene registrato nel campo *anyType* e la lista dei namespace nel vettore *anyNameSpaceList*. Il boolean *isMixed*, infine, indica se si tratta di un content model misto.

#### ComplexReferenceContentModel

Questa classe definisce un content model che ha come unico elemento un tipo complesso. Viene utilizzato nelle dichiarazioni di elementi che contengono un'entità complessa

```
<!ELEMENT nomeElemento (@tipocomplesso;)>
```

#### ComplexContentModelProperty

Permette una derivazione per estensione anonima. Viene, quindi, usato nelle dichiarazione di elementi del tipo

```
<!ELEMENT a (@tipo;,b)>
```

Il campo *derivedType* identifica il nome del tipo da cui deriva, mentre il booleano *derivedType_isMixed* è posto a true se il tipo da cui deriva ha content model misto.

#### Group

Definisce il content model di un Group. In base al valore del campo *type* vengono gestiti i content model contenuti all'interno del vettore childs. CM_TYPE_CHOICE realizza uno schema o un dtd di tipo choice, CM_TYPE_SEQ di tipo sequence.

#### Tag

Classe astratta da cui derivano tutte le classi relative ai tag del DTD++. I campi *left* e *right* contengono la posizione iniziale e finale del tag all'interno del documento DTD++. Il metodo astratto *getSchema* permette, per ciascuna delle classi derivate, di restituire una rappresentazione in XML Schema della propria struttura. *getSchemaRD*, *getSchemaVB*, *getSchemaGE*, *getSchemaSS* rendono possibile la conversione secondo gli stili Russian Doll, Venetian Blind, Garden of Eden e Salami Slice. Analogamente il metodo astratto *ToDtd* permette ad ogni classe derivata, una rappresentazione in DTD. I booleani *isLocal* e *isGlobal* identificano, rispettivamente, il tag come locale o globale. Il campo *style* memorizza lo stile XML Schema da utilizzare. *current_file*, *current_prefix* e *namespaceList*, invece, vengono utilizzati quando si crea un XML Schema e vi sono differenti namespace.

#### AttList

Classe che definisce un'attlist. Genera degli attribute all'interno di un tipo complesso. Il campo *attributes* è un vettore di classi Attribute, il campo *name* è una stringa con il nome dell'attributeGroup.

#### Attribute

Definisce un attribute. Il campo *name* contiene il nome dell'attributo, *contentModel* il Content model e *defaultValue* il valore di default dell'attributo.

#### AttributeDefault

Classe che definisce il valore di default di un attribute. Il campo *type* contiene il tipo di attributo e può assumere i valo-

ri REQUIRED, IMPLIED, FIXED e DEFAULT. Il valore di default dell'attributo viene invece settato nel campo *value*.

#### AttributeVector

Classe contente un vettore di attribute. I metodi pubblici *add* e *remove* permettono di aggiungere o rimuovere un attribute dalla lista. Con *getAttributeByName* si recupera la posizione dell'attribute all'interno della lista altrimenti viene restituito -1.

#### EntityCharRef

Classe che definisce l'entità interna del DTD.

#### EntitySimple

Classe che definisce un'entity simple. Viene creata indicando il nome e il content model. Se deriva da un'altra entity il metodo *getParent* restituisce una stringa contenente il nome dell'entity da cui deriva. All'interno del campo *simpleContentModel* viene definito il content model. Il metodo *isUnion* restituisce true se il content model è di tipo union.

#### EntityComplex

Classe che definisce un'entity complex. Il nome dell'entità è contenuto nel campo *name*, mentre il vettore *attlist* contiene una lista di oggetti Attribute. La gestione del content model è delegata interamente al campo *property* di tipo EntityComplex-Property. Il booleano *isCreated* è settato a true se l'entità è stata creata per poter gestire uno stile di rappresentazione XML Schema.

#### EntityComplexProperty

Classe che definisce le proprietà di un tipo complesso. Nel caso in cui il tipo derivi per restrizione o estensione vengono settati rispettivamente a true i campi *isRestricted* e *isExtended* e viene assegnato il nome del tipo da cui si ereditano le proprietà nel campo *derivedType*. Il campo *contentModel* contiene il riferimento al content model del tipo. I costruttori permettono di creare un tipo non derivato o derivato per estensione. Per effettuare una derivazione per restrizione è necessario chiamare il metodo *setRestriction* dopo la creazione dell'oggetto. Il metodo *getIsMixed* ritorna true nel caso in cui il content model sia di tipo mixed mentre il booleano *derivedType_isMixed* informa se il tipo da cui deriva ha content model mixed oppure no.

#### EntityAttlist

Classe che definisce un attributeGroup. Il campo *name* contiene il nome dell'attributeGroup mentre *attlist* è il vettore degli attributi.

#### EntityGroup

Classe che definice un Group. La stringa *name* contiene il nome del group mentre il campo *contentModel* contiene il riferimento al content model del Group.

#### EntityGroupProperty

Definisce le proprietà di un Group. Il campo *contentModel* contiene il riferimento al content model del Group.

#### EntityParametric

Classe che gestisce le entità parametriche. Utilizzata in conversione a DTD. Il campo *name* contiene il nome dell'entità. Il

booleano *isCreated* è settato a true quando l'entità viene creata per gestire i substitutionGroup in DTD. Il campo *att* contiene la lista degli attributi.

#### EntitySystem

Classe che gestisce le entità SYSTEM o PUBLIC. Viene utilizzata per la conversione a DTD. La stringa *name* contiene il nome dell'entity e il campo *isPublic*, se settato a true, indica che si tratta di un'entity PUBLIC. Altrimenti sarà di tipo SYSTEM.

#### FacetCardinality

Classe che definisce il numero di occorrenze del content model. Genera la stringa `maxOccurs=x minOccurs=y` all'interno dei tag schema.

#### FacetEnumeration

Classe che definisce un facet di tipo Enumeration. Il campo *items* contiene la lista dei valori di enumerazione.

#### FacetLength

Classe che definisce un facet per la lunghezza delle stringhe e le caratteristiche dei numeri decimali. Viene stampato, in maniera esclusiva, il facet length o i facet minLength e maxLenght o i facet totalDigits e fractionDigits.

#### FacetPattern

Classe che definisce un facet per la definizione dei pattern.

#### FacetRange

Classe che definisce un facet per il range dei numeri. I valori possono essere di tipo inclusivo o esclusivo.

#### FacetWhiteSpace

Classe che definisce un facet di tipo WhiteSpace.

#### Pair

Classe che rappresenta una coppia di valori separati da un punto o da una virgola. Viene utilizzata dalle classi FacetRange, FacetCardinality e FacetLength per assegnarne i rispettivi valori.

#### Element

Classe che definisce un element. Il campo *name* contiene il nome dell'elemento mentre il campo *contentModel* memorizza il content model. Se il booleano *attGroup* è impostato a true allora l'element avrà associato degli attribute o farà riferimento ad attributeGroup. Il campo *occurs* di tipo FacetCardinality definisce le occorrenze. Il booleano *root* settato a true permette di identificare l'elemento come root. *name1* è una stringa che viene utilizzata per memorizzare il nome del tipo in cui l'elemento viene dichiarato in maniera locale o il nome dell'element substitution. In quest'ultimo caso il campo *isSubstitution* sarà settato a true a significare che si tratta di un element substitution.

#### SchemaPathState

Classe che gestisce SchemaPath.

#### Comment

Classe che definisce il Tag commento. Il valore viene inserito comprensivo dei tag di inizio e fine commento dei DTD che vengono poi filtrati direttamente dal costruttore.

#### TargetNS

Classe che definisce il Tag TARGETNS.

### 6.3.7 preValidator.utils

Package contenente classi di vario tipo utilizzate dal programma per la stampa e l'accesso al disco.

#### Debug

Classe per la stampa dei messaggi di debug del parser e del lexer.

#### Disk

Classe che gestisce gli accessi al disco. Il metodo *LoadFromFile* carica un file di testo dal disco e restituisce in una stringa il suo contenuto. I due metodi, entrambi chiamati *SaveToFile*, salvano il contenuto di una stringa sul disco e differiscono per il numero di parametri presi in input. Nel secondo è infatti possibile specificare il nome della directory in cui salvare il file.

#### IndentSchema

*IndentSchema* gestisce l'indentazione nella generazione dello schema. Il campo *numTab* tiene memoria del numero di indentazioni in un dato momento. Tale valore viene aumentato o diminuito grazie ai metodi *addTab* e *delTab*. Con il metodo

*getTab* si restituisce la stringa da anteporre alla corrente riga dello schema.

#### PrintSchema

Classe di utility per la stampa dello schema. Il metodo statico *countLine* conta le righe di una stringa.

#### TemporaryNames

Si occupa della generazione di elementi temporanei.

#### Verbose

Classe utilizzata per la stampa della modalità verbose.

## 6.4 Utilizzo e installazione

### 6.4.1 Utilizzo

L'utilizzo del preprocessore avviene tramite linea di comando. I file sono passati in input dal file system e l'output può essere emesso sia su file che sullo standard output. Gli eventuali errori verranno visualizzati sullo standard error.
La sintassi per utilizzare PreValidator è la seguente:

java -jar PreValidator.jar [-v] [-nv] [-schema fileName -dtd fileName ] [-out] [-salamislice | -ss | -gardenofeden | -goe | -venetianblinds | -vb | -russiandolls | -rd] [-no-validator-error] [-stdout-error] [-xerces | -msxml | -validator fileName] [-no-force-schema] [-nodelete-temp] [file.xml] <file.dpp>

**-dtd fileName Save DTD to file**
Viene salvata la trasformazione del documento DTD++ in DTD nel file specificato.

**-schema fileName Save Schema to file**
Viene salvata la trasformazione del documento DTD++ in XML Schema nel file specificato.

**-salamislice, -ss Salami Slice schema style**
Lo schema sarà generato secondo lo stile Salami Slice. Se non viene specificato alcun stile, per default si utilizza Salami Slice.

**-gardenofeden, -goe Garden Of Eden schema style**
Lo schema sarà generato secondo lo stile Garden Of Eden.

**-venetianblinds, -vb Venetian Blinds schema style**
Lo schema sarà generato secondo lo stile Venetian Blinds.

**-russiandolls, -rd Russian Dolls schema style**
Lo schema sarà generato secondo lo stile Russian Dolls.

**-validator fileName Validator file name**
Determina quale validatore utilizzare. Per default si utilizza il validator.exe che deve trovarsi nella stessa directory del preprocessore.

**-xerces Validate with Xerces**
Abilita Xerces come validatore. Equivale a lanciare PreValidator con i parametri -validator "java -jar validateXML-1.1-bin.jar-in" -no-validator-error.

**-msxml Validate with MSXML**
Abilita MSXML come validatore. Equivale ad eseguire il processo con i parametri "-validator validator.exe". Se non viene specificato un validatore, per default si utilizzerà questa configurazione.

**-v Verbose mode**
Segnala sullo standard output lo stato in cui si trova il preprocessore durante l'esecuzione. Si possono specificare altri due livelli di verbose-mode tramite le opzioni -v5 e -v6. Con la prima si stampano sullo standard output le informazioni di debug del parser, con la seconda sia quelle relative al parser sia quelle relative allo scanner.

**-nv Do not validate**
Disabilita l'operazione di validazione e il processore si limita a trasformare il documento DTD++ in XML Schema.

**-outputDirectory directoryName Save schema or DTD to directory**
Salva la trasformazione del documento DTD++ nella directoty specificata.

**-out Print Schema or DTD to stdout**
Stampa la trasformazione del documento DTD++ sullo standard output. Con questa opzione viene disabilitata l'operazione di validazione.

**-stdout-error Error message on stdout**
Imposta il preprocessore in modo da ricevere i messaggi di errore del validatore sullo standard output invece che sullo standard error.

**-no-validator-error Do not process validator error**
Disabilita il processing degli errori restituiti dal validatore. L'errore viene stampato sullo standard error così come ricevuto dal validatore. Utile se il messaggio d'errore restituito dal validatore non è compatibile con il preprocessore.

**-no-force-schema Do not specify schema**
Non passa il nome dello schema come parametro al validatore ma lo inserisce direttamente nel file XML. Viene così creato un file XML temporaneo.

**-nodelete-temp Do not delete temp file**
Se abilitato non cancella i file temporanei creati durante l'esecuzione.

**file.xml**
Il file XML che si vuol validare. Se omesso, viene disabilitata l'operazione di validazione.

**file.dpp**
Il file DTD++ che si vuole processare.

### 6.4.2 Installazione

Il preprocessore funziona senza particolari requisiti hardware e software. È però necessario avere installato una JVM 1.4.1[5] o successiva.
All'interno del pacchetto vi sono quattro file:

- **PreValidator.jar** il preprocessore;
- **validateXML-1.1-bin.jar GTRI XML Validation Tool**[6] il validatore Xerces;
- **validator.exe** il validatore MSXML. Per funzionare ne-

[5] http://java.sun.com/
[6] http://justicexml.gtri.gatech.edu/extern/v2/validateXML-1.1

cessita del runtime di DotNet[7] versione 1.1 e del MSXML 4.0[8].

- **pst.dtd** file DTD contenente i tipi predefiniti di XML Schema.

[7] msdn.microsoft.com/netframework/downloads
[8] http://msdn.microsoft.com/library/en-us/dnanchor/html/anch_xmlprod.asp

# Capitolo 7

# Sviluppi futuri

DTD++ è un linguaggio caratterizzato da una semplicità d'apprendimento e da una leggibilità che si contrappone alla verbosità degli altri linguaggi di validazione per XML. Inoltre si pone come punto di incontro tra i linguaggi grammar-based e i linguaggi rule-based permettendo così di espremire allo stesso tempo sia vincoli strutturali sia co-constraints.

Grazie alle nuove estensioni introdotte DTD++ è diventato un linguaggio Object Oriented permettendo maggior flessibilità d'utilizzo e maggior riuso. L'introduzione del concetto di componente locale e componente globale ha poi permesso di riutilizzare più volte le strutture e di organizzare e gestire lo schema in vari modi.

Per poter avvicinarsi ancora di più al paradigma Object Oriented, la sintassi potrebbe essere estesa con l'introduzione dell'attributo *abstract* presente in XML schema. In questo modo un elemento dichiarato abstract non potrà essere utilizzato analogamente alle classi astratte dell'approccio Object Oriented.

Allo stesso modo l'inserimento dell'attributo *final*, ripreso dalla sintassi di XML Schema, permetterebbe un maggior controllo sui meccanismi di derivazione e di sostituzione impedendo agli elementi e ai tipi di poter essere sostituiti o derivati all'interno dello schema.

L'introduzione di un costrutto analogo al *redefine* di XML Schema renderebbe il DTD++ ancora più modulare. Infatti si avrebbe la possibilità di includere uno schema che appartiene allo stesso targetNamespace ma in più l'utente potrebbe ridefinire i tipi semplici e complessi e i gruppi di elementi e attributi.

Anche PreValidator for DTD++, come il linguaggio, ha subito un'evoluzione: da semplice validatore di documenti XML si è trasformato in un convertitore a XML Schema permettendo anche di poter scegliere uno stile di progettazione. In ogni momento si ha così la possibilità di abbandonare DTD++ in favore di uno standard.

Inoltre si è permesso di poter convertire un documento DTD++ in un DTD aumentando ulteriormente la portabilità di DTD++ e permettendo all'utente di scegliere il linguaggio in cui il documento sarà restituito.

Grazie alla grande modularità di PreValidator sarà possibile ottenere conversioni ad altri linguaggi di validazione per XML come ad esempio RELAX NG o Schematron. Si avranno così ulteriori possibilità di ottenere uno standard e di utilizzare DTD++ come formato interscambio.

Inoltre per aumentare la leggibilità dei documenti convertiti restituiti all'utente si potrebbero implementare funzionalità per generare in maniera automatica commenti analogamente a quanto avviene con Javadoc. I commenti dell'utente verrebbero così gestiti in maniera opportuna e inseriti all'interno delle strutture portando ad un maggior leggibilità e chiarezza.

Come si è visto, in fase di conversione a DTD si è assunto il principio secondo cui ogni documento valido in DTD++ sarà un documento valido in DTD e un documento non valido in DTD, non sarà valido nemmeno in DTD++. Inoltre per gli operatori di ripetizione si è sostituito l'insieme richiesto con il suo più piccolo sottoinsieme definibile di valori.

In questo modo però un intervallo del tipo `a[2,5]` verrà

convertito in `a+`, ovvero da una a infinite volte perdendo inevitabilmente delle informazioni. In realtà la conversione più appropriata sarebbe `aa+` rispettando così il fatto che il minimo numero di occorrenze possibili è due e non uno.

Una diversa gestione in fase di conversione permetterebbe una maggiore aderenza a quanto richiesto dall'utente e, di conseguenza, una conversione più efficiente.

# Bibliografia

[Amo02] Nicola Amorosi, *Un'estensione sintattica dei DTD per la validazione dei documenti XML* Tesi di Laurea discussa presso l'Università di Bologna, marzo 2001 http://tesi.fabio.web.cs.unibo.it/Tesi/DtdPlusPlus

[BM01] P. V. Biron and A. Malhotra, *XML Schema Part 2: Datatypes*. http://www.w3.org/TR/xmlschema-2, May 2001. W3C Recommendation.

[Cap01] Sara Capecchi. *Il pattern Decorator per l'estensione dei linguaggi orientati agli oggetti* Tesi di Laurea discussa presso l'Università di Firenze, 2001 http://music.dsi.unifi.it/thesis/tesi-decorator.ps.gz

[Cla01] James Clark. *TREX - Tree Regular Expressions for XML. Language Specification.* http://www.thaiopensource.com/trex/spec.html, February 2001

[Cos03] Roger L. Costello *XML Schemas: Best Practices*. 17 Feb 2003. http://www.xfront.com/BestPracticesHomepage.html.

[DSDL] *ISO/IEC 19757 - DSDL Document Schema Definition Languages December 2004* http://www.dsdl.org

[Fio03] Davide Fiorello, *Un processore per la validazione di documenti XML: DTD++* Tesi di Laurea discussa

presso l'Università di Bologna, marzo 2003 http://tesi.fabio.web.cs.unibo.it/Tesi/DtdPlusPlus

[Kni00] Günter Kniesel, *Dynamic Object-Based Inheritance with Subtyping*
Tesi di dottorato presso l'univeristà di Bonn, 2000 http://javalab.cs.uni-bonn.de/data2/papers/darwin/darwinDiss.pdf

[MAl02] Eve Maler *Schema Design Rules for UBL...and Maybe For You*, in XML Conference & Exposition 2002. December 8-13, 2002 Baltimore Convention Center http://www.idealliance.org/papers/xml02/dx_xml02/papers/05-01-02/05-01-02.html

[MSCV] Paolo Marinelli, Claudio Sacerdoti Coen, Fabio Vitali *SchemaPath: Extending XML Schema for Co-Constraints, in Technical Report UBLCS-2004-13* June 2004. ftp://ftp.cs.unibo.it/pub/techreports/2004/2004-13.pdf

[Mur00] Makoto Murata. *RELAX (REgular LAnguage description for XML)*. http://www.xml.gr.jp/relax, 2000.

[VAG03] Fabio Vitali, Nicola Amorosi, Nicola Gessa. *Datatype- and namespace-aware DTDs: a minimal extension*, in Proceedings of Extreme Markup 2003. August 2003. Montreal, Canada. http://tesi.fabio.web.cs.unibo.it/Tesi/DtdPlusPlus

[VFGM03] Fabio Vitali, Davide Fiorello, Nicola Gessa, Paolo Marinelli. *DTD++ 2.0: Adding support for co-constraints*, in Extreme Markup Languages 2003, August 2003. http://tesi.fabio.web.cs.unibo.it/Tesi/DtdPlusPlus

[Vl01] Eric van der Vlist. *Using XML Schema* October 17, 2001 http://www.xml.org/pub/a/2001/11/29/schemas/part1.html

[Vl01b] Eric van der Vlist. *Comparing XML Schema Languages* Dicember 12, 2001 http://www.xml.org/pub/a/2001/12/12/schemacompare.html

[Vl02] Eric van der Vlist. *The W3C's Object-Oriented descriptions for XML* (First Edition) O'Reilly

[Vl02b] Eric van der Vlist. *Tutorial: XML Schema languages*, 2002 http://xml-coverpages.org/vanderVlist-SchemaTutorial2002.html

[WC01] Norman Walsh e John Cowan, *Schema Language Comparison.* December 2001.http:// nwalsh.com/xml2001/schematownhall/slides/.

[W3C] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, François Yergeau *Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation* 04 February 2004 http://www.w3.org/TR/REC-xml/

# Appendice A

# Riassunto delle dichiarazioni DTD++

Qui di seguito si riportano le dichiarazioni del DTD++

**Tipi semplici**

*Sintassi:* `<!ENTITY # nome "(tipobase)">`
`<!ELEMENT nomeElemento (tipobase)>`
`<!ATTLIST nomeElemento nomeAttributo (tipobase) default>`
`<!ATTLIST nomeElemento nomeAttrGroup >`

*o anche*

`<!ENTITY # nome "tipobase">`
`<!ELEMENT nomeElemento tipobase>`
`<!ATTLIST nomeElemento nomeAttributo tipobase default>`

nome: nome del nuovo tipo semplice
tipobase: tipo semplice da cui l'elemento deriva
nomeElemento: nome dell'elemento
nomeAttributo: nome dell'attributo
default: #IMPLIED o #REQUIRED o #FIXED " " o " "
nomeAttrGroup: nome dell'attributeGroup sempre racchiuso tra "##" e ";"

**Derivazione per restrizione**

*Sintassi:*
```
<!ENTITY # nome "(tipobase{}[]//()\\)">
<!ELEMENT nomeElemento (tipobase{}[]//()\\)>
<!ATTLIST nomeElemento nomeAttributo
(tipobase{}[]//()\\) default>
```
*o anche*
```
<!ENTITY # nome "tipobase {}[]//()\\">
<!ELEMENT nomeElemento tipobase{}[]//()\\>
<!ATTLIST nomeElemento nomeAttributo
tipobase{}[]//()\\ default>
```

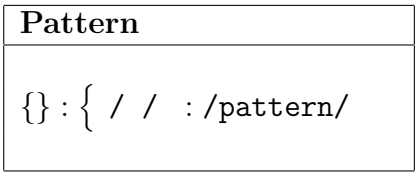
*{lunghezza}[range]*
*/pattern/*
*(enumerazione)*
*\whitespace*: vincoli di derivazione per restrizione

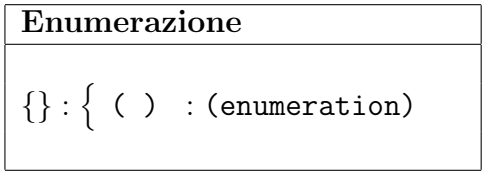
**Lunghezza**

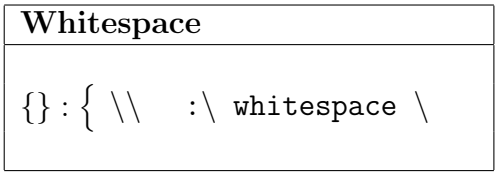
$$\{\} : \begin{cases} \{\} & : \{\texttt{length}\} \\ \{,\} & : \{\texttt{minLength,maxLength}\} \\ \{.\} & : \{\texttt{totalDigits.fractionDigits}\} \end{cases}$$

**Range**

$$[,] : \begin{cases} [,] & : [\texttt{minInclusive,maxInclusive}] \\ [,[ & : [\texttt{minInclusive,maxExclusive}[ \\ ],] & : ]\texttt{minExclusive,maxInclusive}] \\ ],[ & : ]\texttt{minExclusive,maxExclusive}[ \end{cases}$$




**Derivazione per lista**

*Sintassi:* `<!ENTITY # nome "(tipobase+{}[]//()\\)">`
`<!ELEMENT nomeElemento(tipobase+{}[]//()\\)>`
`<!ATTLIST nomeElemento nomeAttributo`
`(tipobase+{}[]//()\\) default>`

*o anche*

`<!ENTITY # nome "tipobase+ {}[]//()\\">`
`<!ELEMENT nomeElemento tipobase+{}[]//()\\ >`
`<!ATTLIST nomeElemento nomeAttributo`
`tipobase+{}[]//()\\ default>`

*+*: derivazione per lista
*{}[]// ()\\:* vincoli di derivazione per restrizione

**Derivazione per unione**

*Sintassi:* <!ENTITY # nome "union">
<!ELEMENT nomeElemento (union)>
<!ATTLIST nomeElemento nomeAttributo union default>
*o anche*
<!ENTITY # nome "union">
<!ELEMENT nomeElemento union>
<!ATTLIST nomeElemento nomeAttributo union default>

*union:* (tipobase+vincoli) |(tipobase+vincoli)|...
*o union:* tipobase+vincoli | tipobase+vincoli |...
*vincoli:* vincoli di derivazione per restrizione

**Tipo complesso**

*Sintassi:* <!ENTITY @ nome "contentmodel" listaAttributi>
*nome:* nome del tipo complesso
*contentmodel:* contentmodel dell'elemento
*listaAttributi:* "lista di attributi"

**Derivazione per estensione**

*Sintassi:* <!ENTITY @ "tipoBaseExt,contentmodel" listaAttributi>
<!ELEMENT nomeElemento "(tipoBaseExt,contentModel)">

*tipoBaseExt:* tipo complesso base da derivare per estensione

**Derivazione per restrizione**

*Sintassi:* <!ENTITY @ nome tipoBaseRestr "contentmodel" listaAttributi>

*tipoBaseRestr:* tipo complesso base da derivare per restrizione

**Elemento radice**

*Sintassi:* <!ROOT nomeElemento contentModel >

**Group**

*Sintassi:* <!ENTITY @@ nomeGroup "contentModel">

**AttributeGroup**

*Sintassi:* <!ENTITY ## nome "listaAttributi">

*listaAttributi:* attributeGroup sempre racchiuso tra "##" e ";"
o nome Attributo (tipoBase) default
o nome Attributo tipoBase default

*tipoBase*: predefinito o definito dall'utente e sempre racchiuso tra "#" e ";"

**SubstitutionGroup**

*Sintassi:* <!ELEMENT nomeSubstitutionGroup nomeElemento>
<!ELEMENT nomeSubstitutionGroup nomeElemento tipo>

*tipo:* stesso tipo o una valida derivazione
del tipo dell'head element

**Content Model misto**

*Sintassi:* <!ENTITY @ nome "#PCDATA *contentmodel*">
<!ELEMENT nome #PCDATA *contentmodel*>

*contentmodel:* (content model dell'elemento)

**TARGETNS**

*Sintassi:* <!TARGETNS *prefisso* "URI">

*listaAttributi:* attributeGroup sempre racchiuso tra "##" e ";"
o nome Attributo (tipoBase) default
o nome Attributo tipoBase default

*prefisso:* prefisso associato al namespace
URI: namespace

**ANY**

*Sintassi:* <!ENTITY @ nome "ANY*[,]*{*,*}" listaAttributi>

# RINGRAZIAMENTI

Il primo ringraziamento, come promesso, va a Davide Fiorello per la sua gentilezza e disponibilità. Ringrazio poi il Prof. Fabio Vitali per la sua simpatia e per il suo sostegno costante. Un grazie mille a Stefano Zacchiroli per l'aiuto datomi per la stesura della tesi.

Ma soprattutto ringrazio mia madre, mio padre, Francesco, le mie nonne, i miei zii e tutta la mia famiglia per aver reso possibile tutto questo e per essermi stati vicini in ogni momento.

Ringrazio poi Claudina, con cui ho condiviso questa bella avventura e tante risate e per la sopportazione reciproca.
Sabrina, Gian, Piero, Noè, Salvo, il Pira, il Tosi, Fra, Vas e le tante persone conosciute in questi anni che mi sono state vicine aiutandomi.

L'ultimo ringraziamento va a Max per esserci sempre e comunque e per trovare sempre le giuste parole ma soprattutto per farmi ridere.