UNIVERSITÀ DEGLI STUDI DI MODENA E
REGGIO EMILIA
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Da linguaggio standard per WWW a linguaggio standard per OODB: il traduttore XML/ODL$_{I^3}$

Relatore:

Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di:

Andrea Cataldo

Correlatore:

Ing. Alberto Corni

Anno Accademico 1999 - 2000

Parole chiave:

Integrazione di dati
Internet
XML
ODL_{I^3}
Wrapper

RINGRAZIAMENTI

Ringrazio la Professoressa Sonia Bergamaschi per il prezioso supporto fornitomi nella stesura della presente tesi oltre che l'Ing. Alberto Corni e l'Ing Maurizio Vincini per l'altrettanto prezioso aiuto garantitomi nel superamento dei problemi tecnici.

Un ringraziamento particolare alla mia fidanzata Lisa che mi ha dato la forza per raggiungere questo obiettivo ed alla mia famiglia per il supporto e la pazienza garantitami in questi lunghi anni di studi.

Indice

Capitolo 1

Introduzione

L'ambito operativo nel quale si colloca il presente lavoro è quello di un progetto nato dalla collaborazione tra il Dipartimento di Scineze dell'Ingegneria dell'Università di Modena ed il Dipartimento di Ingegneria dell'Università di Milano denominato **MOMIS** (Mediator EnvirOnment for Multiple Information Sources) [1]. Come dice il nome stesso si tratta di un software che permette l'integrazione di dati provenienti da una molteplicità di sorgenti diversi.

La necessità di questo software si è sentita dal momento in cui la diffusione di Internet ha avvicinato alla rete un numero molto alto di utenti ed allo stesso tempo ha aumentato in maniera esponenziale la quantità di dati reperibile. Questo aumento ha portato, però, anche ad un maggior differenziazione dei formati nel quale questi dati sono archiviati creando la necessità di un numero sempre maggiore di strumenti in grado di leggerli e di conseguenza dalla necessità, impensabile dal punto di vista di un fruitore medio e non solo, di conoscere questi strumenti per aver accesso a questi dati.

1.1 Scopo della tesi

Il presente lavoro si prefigge l'obiettivo di integrare una nuova forma di dati all'interno di MOMIS, ed in particolare, di saper leggere il formato XML [2, 3] e di tradurlo nel formato ODL_{I^3} [4, 5] utilizzato da MOMIS quale modello dei dati comuni per poter superare le disomegeneità dei formati dei dati.

Occorre tener conto che questo strumento sarà sicuramente importante in quanto il linguaggio XML si prevede diventi il futuro standard nel quale

verranno create le pagine Web ed è proposto in generale come standard di codifica dei dati.

1.2 Struttura

Questo documento si propone l'obiettivo di descrivere il Wrapper di traduzione XML $\rightarrow$ ODL$_{I^3}$ e l'ambiente all'interno del quale si sivluppa passando per la descrizione delle caratteristiche del linguaggio XML.

Capitolo 2. In questo capitolo si cerca di descrivere brevemente cosa è il progetto MOMIS e la sua architettura, i riferimenti le descrizioni più dettagliate sono in [6, 1, 7, 8].

Capitolo 3. Qui verrà riportata una breve guida al linguaggio ODL$_{I^3}$ ed alle sue caratteristiche principali, rimandando gli eventuali approfondimenti ad altri lavori più pertinenti a questo argomento [4, 5].

Capitolo 4. Il presente capitolo descrive il linguaggio XML, base di partenza del Wrapper, cercando di mettere il lettore in grado di capirne le caratteristiche fondamentali e di assimilarne la grammatica.

Capitolo 5. L'algoritmo utilizzato dal wrapper viene illustrato dettagliatamente in questo capitolo dove viene analizzata nel particolare la sintassi di un DTD XML e si da' una proposta di traduzione in linguaggio ODL$_{I^3}$, ove questo è possibile, oppure viene evidenziato dove non è possibile effettuare queste operazioni di traduzione.

Capitolo 6. Il codice sorgente del wrapper ed il modello utilizzato per la sua scrittura viene illustrato in questo capitolo evidenziando quali sono state le scelte effettuate sia per quanto riguarda la scrittura del codice stesso che per quanto riguarda gli strumenti utilizzati.

Capitolo 7. Infine in questo capitolo si riporta una breve descrizione degli strumenti software utilizzati dal prototipo software e di tutti quei prodotti che sono stato d'ausilio nella stesura della presente tesi, sia per quanto riguarda la scrittura del codice sorgente, sia per quanto riguarda la generazione e la validazione del codice XML.

Capitolo 2

Il progetto MOMIS

Il progetto MOMIS [1] (**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources) nasce nelle università di Modena e Milano da una collaborazione tra il gruppo di lavoro della Professoressa Sonia Bergamaschi (Dipartimento di Scienze dell'Ingegneria di Modena) e della Dottoressa Silvana Castano (Dipartimento di Scienze dell'Informazione dell'Università di Milano) con il preciso intento di sviluppare un sistema di integrazione di basi di dati eterogenee.

In questo capitolo verranno quindi descritte le linee guida che sono alla base di MOMIS in modo da illustrarne le caratteristiche principali del prodotto e di inquadrare l'ambito particolare nel quale la presente tesi si inserisce.

2.1 Introduzione

L'enorme diffusione dei dati attualmente reperibili sulle reti di computer di tutto il mondo, anche attraverso Internet, e la loro disomogeneità per quanto riguarda i formati nei quali sono conservati ha creato un panorama variegato nel quale può diventare complicato ed oneroso districarsi. Infatti è spesso necessario, da parte di un fruitore qualsiasi, procurarsi diversi prodotti software che lo mettano nella posizione di riuscire a decodificare ed ad utilizzare i dati stessi in ogni formato nel quale vengono rintracciati. Questa molteplicità di prodotti pone anche il problema della conoscenza degli stessi per potere, da ognuno di essi, ottenere i risultati voluti.

La strada che quindi occorrerebbe percorrere è quella di avere una certa uniformità nel meccanismo di reperimento e consultazione dei dati ricorrendo

ad uno strumento che si occupi di eliminare le incongruenze e le ridondanze e che quindi sia in grado di dare una visione integrata delle informazioni cercate. E' appunto con questo intento che nasce il progetto MOMIS il quale aspira a diventare questo strumento, denominato più propriamente col nome di *Mediatore*, in grado di svolgere il compito di integrazione.

Per svolgere delle operazioni di integrazione la letteratura attuale ha evidenziato due distinti tipi di approccio ai dati: l'approccio *strutturale* e l'approccio *semantico*. MOMIS segue appunto questo secondo che si caratterizza per le seguenti peculiarità:

- per ogni sorgente sono a disposizione dei metadati, codificati nello schema concettuale;
- nello schema concettuale sono presenti le informazioni semantiche, che possono essere utilmente sfruttate sia nella fase di integrazione delle sorgenti, sia in quella di ottimizzazione delle interrogazioni;
- per poter descrivere in modo uniforme tutte le informazioni che devono essere condivise deve essere presente nel sistema un modello di dati comune;
- per arrivare alla definizione di uno schema globale è realizzata effettivamente una unificazione (parziale o totale) degli schemi concettuali (in modo semi-automatico).

L'utilizzo di tale approccio permette quindi a MOMIS, a differenza di quanto sarebbe potuto avvenire seguendo un approccio strutturale, di avere uno schema globale che permetta una ottimizzazione sematica delle interrogazioni sollevando quindi il progettista dall'onere di ottimizzare le interrogazioni stesse e l'operatore esterno da quello di inserire manualmente le informazioni semantiche.

2.2 Architettura di MOMIS

Andiamo adesso a vedere come è stato strutturato MOMIS cercando di dare un'idea della sua architettura e di come vengono svolte le operazioni fondamentali rimandando ad altri testi [1, 9] l'approfondimento dell'argomento.

MOMIS si appoggia fondamentelmente su due tools già esistenti denominati rispettivamente ARTEMIS (**A**nalysis or **R**equirements **T**ool **E**nvironment

for **M**ultible **I**nformation **S**ystem) [10, 11] e ODB-Tools [12, 13] e li sfrutta allo scopo di avere una architettura I_3 per l'integrazione e l'ottimizzazione delle query.

Vengono inoltre utilizzati un modello di dati comune (ODM_{I^3}) ed un linguaggio di definizione comune (ODL_{I^3} che verrà approfondito più avanti) per la descrizione degli schemi sorgenti e del mediatore. Entrambe questi modelli sono un subset del corrispondente linguaggio ODMG-93 [14], ed in particolare possono essere cosiderati come una estensione che rispetta i propositi del gruppo di lavoro I^3/POB [15] per il linguaggio mediatore standard.

Viene infine utilizzata la logica descrittiva OLCD (**O**bject **L**anguage with **C**omplements allowing **D**escriptives cycles) [16, 17] come linguaggio kernel nell'ambito della quale gli ODB-Tools servono come supporto alla logica stessa.

2.2.1 Struttura del software

Il procedimento di funzionamento di MOMIS avviene principalmente in due fasi separate:

1. viene prima di tutto costruito un Theasurus comune delle relazioni terminologiche [5] basate sulla descrizione dei sorgenti in ODL_{I^3} in modo da creare le basi per identificare le informazioni che dovranno essere integrate nelle loro varie sorgenti

2. in un secondo tempo vengono adottate tecniche di clustering [18, 19] per identificare le classi ODL_{I^3} che descrivono le informazioni simili in modo da poterle integrare in uno schema globale per le sorgenti analizzate [10, 11].

Detto questo possiamo passare all'identificazione dei 4 diversi componenti che costituiscono MOMIS seguendo le basi dell'acrchitettura I_3 e che possono essere denominati come:

- **Wrapper**: possono essere uno o più di uno a seconda seconda della quantità di formati che dovranno essere integrati. Questo componente si occupa della traduzione dallo schema originario della sorgente dei dati in linguaggio ODL_{I^3} oltre che ad occuparsi delle operazioni inverse (provvedere alla traduzione di una query espressa in linguaggio ODL_{I^3} nel linugaggio originario della corrispondente sorgente dati). È

a questo livello che il presente lavoro andrà a collocarsi provvedendo alla traduzione dei dati dal formato XML al formato ODL_{I^3} .

- **Mediatore**: composto da due moduli:
  - il *Global Schema Builder* (GSB) che combina, integra e si occupa di dare una rifinitura ai dati provenienti dal Wrapper;
  - il *Query Manager* (QM) [20] che compie i processi di query e di ottimizzazione ed in particolare genera in modo automatico, tramite l'utilizzo delle tecniche di Descrizione Logica, le query OQL_{I^3} [20] per i vari wrapper partendo dalla singola interrogazione definita dall'utente.
- **Motore ODB-Tools** si occupa della validazione dello schema e della ottimizzazione delle query basandosi sulle Descrizioni Logiche OLCD [4, 21, 17].
- **Tool d'ambiente ARTEMIS** [10, 11] per l'analisi dell'affinità dello schema e per il processo di clustering.

Il tutto può essere agevolemente rappresentato nella Figura 2.1:

2.2.2 Dati semistrutturati e modelli di oggetto

Prima di passare alla descrizione più approfondita di come avvengono le varie fasi del processo di integrazione può essere utile conoscere un po' di teoria riguardante i dati semistrutturati ed i modelli di oggetto su cui si basa MOMIS.

Infatti mentre per quanto riguarda le sorgenti di informazione strutturate (es. database ralazionali e database object-oriented) la descrizione dello schema è sempre disponibile e quindi esso può essere direttamente tradotto nel modello di dati comuni selezionato, per i dati semistrutturati tali informazioni sono direttamente incluse nei dati stessi, si dice quindi che sono strutture "self–describing".

Per un processo di traduzione occorre quindi prima di tutto procedere con la rappresentazione in maniera esplicita delle struttura dei dati. Per fare questo occorre avvalerci di una rappresentazione, in accordo con la letteratura trattante i dati semistrutturati [22, 23, 24], come un grafo.

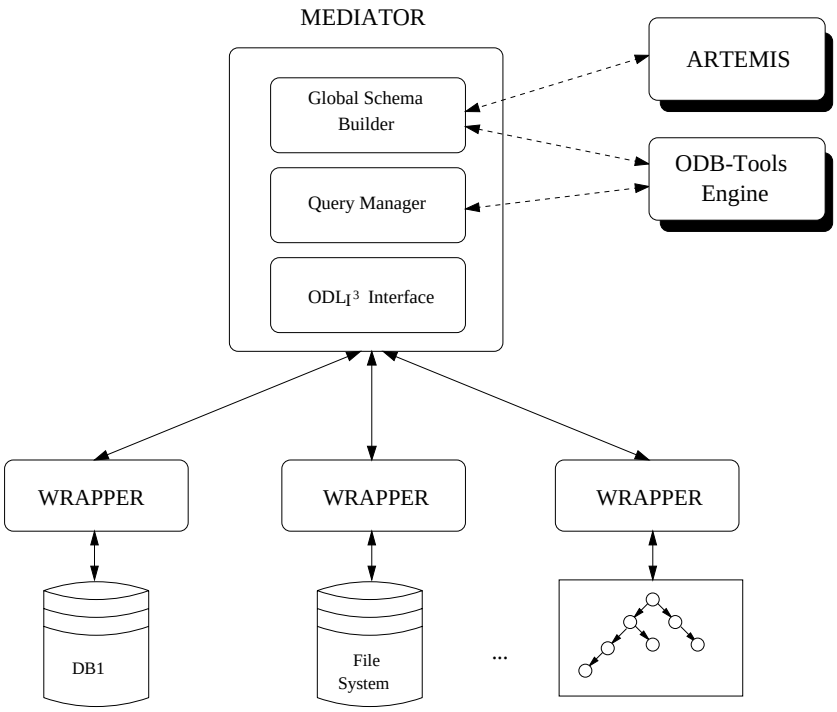

Figura 2.1: Struttura dello schema di MOMIS

Un oggetto semistrutturato quindi può essere visto come una tripla contenente rispettivamente l'identificatore dell'oggetto stesso (*id*), una etichetta che mi dice cosa viene rappresentato nell'oggetto(*label*), ed il valore che assume (*value*) e che può essere di tipo atomico o di tipo complesso. Nel primo caso possiamo imbatterci in valori interi, reali, stringhe o immagini mentre un tipo complesso sono un insieme di oggetti semistrutturati identificati da un set di coppie (*id, label*) e che può essere pensato come il contenitore di tutti gli oggetti che formano il suo valore (oggetti figli). Un oggetto può avere uno o più genitori da cui deriva e per evidenziare che un oggetto *so'* deriva da un oggetto *so* utilizzaziamo una notazione del tipo $so \rightarrow so'$ ed una notazione del tipo *label(so)* per rappresentare l'etichetta di *so*.

Nel modello di dati semistrutturato le etichette sono il più possibile descrittive del concetto rappresentato nell'oggetto che descrivono, quindi in una sorgente di dati gli oggetti che esprimono uno stesso concetto sono identificati da una etichetta unica. Per la rappresentazione dello schema di una sorgente semistrutturata S introduciamo ora il concetto di modelli ad oggetti (*Object patterns*). Tutti gli oggetti *so* di S vengono separati in gruppi distinti, che

assumono la notazione set_l, in modo che tutti gli oggetti appartenenti ad uno stesso gruppo abbiano tutti la stessa etichetta l. Un meddello di oggetti viene poi estratto da ogni gruppo per rappresentare tutti gli oggetti del gruppo stesso. Formalmente un Object Pattern è definito come:

Object pattern: chiamato set_l un gruppo di oggetti di una sorgente semistrutturata S aventi la stessa etichetta l. Un *Object Pattern* di un set_l è una coppia che assume la forma <l,A> dove l è l'etichetta dell'oggetto appartenente a set_l e $A = \bigcup$ *label(so')* in modo che esista almeno un oggetto $so \in set_l$ tale che $so \rightarrow so'$.

Da questa definizione si evince che gli object patterns sono rappresentativi di tutti i differenti oggetti che descrivono lo stesso concetto all'interno della sorgente semistrutturata. In particolare l rappresenta il concetto e set A le proprietà (o gli attributi) caratterizzanti il concetto nella sorgente.

Dal momento in cui gli oggetti semistrutturati possono essere eterogenei le etichette in A possono essere definite solo per acluni degli oggetti in set_l, ma non per tutti. Chiameremo questi tipi di etichette "opzionali" e le denoteremo con il simbolo "*".

Una descrizione di modelli di oggetti segue una *open world semantics* tipica dell'approccio della Logica Descrittiva [25, 26, 27]. Gli oggetti che sono conformi ad un modello condividono con esso una minima struttura comune rappresentata da proprietà non opzionali, ma che possono essere successivamente ampliate con proprietà supplementari. In questo modo gli oggetti di una sorgente semistrutturata possono evolversi grazie all'aggiunta di nuove proprietà che saranno quindi riconosciute come istanze valide del corrispondente modello di oggetti nel momento in cui verrà processata una query.

2.2.3 Generazione di un Thesaurus comune

Lo scopo di questa operazione è quello di creare un Thesaurus di relazioni terminologiche [5] che siano in grado di descrivere le conoscenze comuni tra le classi ODL_{I^3} e gli attributi dello schema sorgente.

Le relazioni terminologiche possono essere di 3 tipi diversi e rispettivamente assumono le seguenti caratteristiche:

- **Sinonimia** (SYN) utilizzate per identificare la caratteristica di due

termini distinti, ma che possono essere utilizzati indeffernetemente all'interno di uno stesso contesto senza per questo perderne il significato;

- **Termini più** (Broader Terms - BT) o hipernimia che viene definita tra due termini per i quali uno ha un significato più largo rispetto all'altro. Questa relazione non è simmetrica, tant'è vero che esiste anche l'inverso denominato **Termini restrittivi** (Narrower Terms - NT);
- **Termini collegati** (Related Terms - RT) o associazione positiva utilizzati per legare due termini che sono generalmente usati assieme all'interno di uno stesso contesto.

Andiamo adesso ad analizzare un po' più in dettaglio come queste operazioni vengono fatte, ricordando che il designer ha a disposizione in questa fase gli strumenti offerti dagli ODB-Tools.

Prima di tutto avviene una prima ricerca automatizzata delle relazioni terminologiche esistenti confrontando le informazioni ricavabili direttamente dallo schema della sorgente dei dati e controllando anche le varie relazioni che all'interno dello schema si formano al momento dell'analisi delle foreign key e primary key eventualmente presenti nella struttura stessa.

Successivamente si ha una fase di revisione e di integrazione delle relazioni create in modo automatico per sorgenti relazionali e ad oggetti. Il progettista è quindi in grado di vedere le relazioni create eventualmente correggendole dove non rappresentassero l'effettivo significato della base di dati ed integrandole aggiungedo altre relazioni che non era stato possibile ricavare in maniera automatica.

Una volta finito il lavoro di creazione delle relazioni terminologiche occorre prevedere una fase di validazione delle stesse basate sulla compatibilità dei domini associati con gli attributi. In questo modo è possibile distinguere, tramite un apposito flag, le relazioni valide da quelle non valide. In particolare il controllo che viene effettuato è diverso a seconda della relazione che si stanno analizzando e viene effettuato sulle relazioni tra i nomi degli attributi.

Sfruttando le capacità di deduzione degli ODB-Tools è possibile a questo punto individuare un nuovo set di relazioni utilizzando lo schema virtuale definito nella fase di revisione ed integrazione e dal calcolo di nuove sussunzioni e dalla aggregazione delle relazioni. Queste nuove relazioni semantiche sono tradotte in relazioni terminologiche ed andranno ad arricchire le relazioni individuate nei passi precedenti il che porta al completamento del

Common Thesaurus ricercato.

In tutta questa fase abbiamo visto che si è fatto largo uso degli ODB-Tools che hanno permesso quindi al designer di essere assistito in ogni suo passo.

2.2.4 Ricerca delle affinità

Per *affinità* si intende l'identificazione delle classi ODL_{I^3} che contengono le medesime informazioni oppure che hanno una relazione semantica che lega le varie classi. La base di partenza da cui prende avvio questo processo di ricerca è dato dal Theasurus comune costruito come visto nel paragrafo precedente. L'attività di ricerca delle affinità ci permette quindi di integrare le classi ODL_{I^3} delle diverse sorgenti di dati in uno schema globale tramite l'utilizzo delle tecniche di clustering gerarchico basate appunto sul concetto di affinità. Andiamo ora ad analizzare più in dettaglio come avviene questo processo di ricerca e come vengono trovate ed elaborate le affinità trovate.

Tramite l'utilizzo degli strumenti di ARTEMIS [28] vengono prima di tutto valutati ed assegnati dei coefficienti di affinità alla classi ODL_{I^3} in modo da metterci nella possibilità di valutare il livello di affinità tra le classi stesse. Tali coefficienti assumono la sintassi $\sigma_{\Re}$ per ogni relazione terminologica $\Re$ del Thesaurus Comune e vengono valutati a seconda del tipo di relazione che andranno a "pesare" seguendo l'ordine $\sigma_{SYN} \geq \sigma_{BT/NT} \geq \sigma_{RT}$. Tenendo presente che due termini hanno un coefficiente di affinità se sono connessi tramite un percorso nel Thesaurus possiamo affermare che il livello di affinità dipende sia dalla lunghezza di tale percorso, che dal di tipo di relazione che li lega che dal loro peso. Il livello di affinità assume quindi un valore di 0 nel caso non esistano delle relazioni tra i due termine, 1 nel caso i due termini siano identici ed un valore calcolato come prodotto dei pesi trovati per le relazioni terminologiche che li coinvolgono nel caso la relazione sia più debole.

Per il calcolo dei coefficienti di affinità di cui sopra si opera in 3 diversi modi:

- *Name Affinity Coefficient*: che viene ottenuto dalla misura dell'affinità dei nomi delle due classi rappresentate nel Thesaurus comune;
- *Structural Affinity Coefficient*: ottenuto dalla misura dell'affinità tra gli attributi di due classi;

- *Global Affinity Coefficient*: ottenuto, per le due classi coinvolte, la misura comprensiva del loro livello di affinità calcolata come somma dei pesi dei Name ed Structural Affinity Coefficients.

La bontà dei coefficienti fin qui trovati dipende dalla correttezza delle relazioni terminologiche trovate e dai parametri che intervengono nel calcolo stesso. Oltre a questo bisogna tener conto che interviene anche una sorta di soggettività nella valutazione dovuta anche alla capacità ed esperienza del designer che si è occupato della verifica ed integrazione delle relazioni terminologiche.

Il grado di interattività con il designer introdotto in MOMIS consente di assicurare che tutte le relazioni terminologiche di una sorgente di dati vengano individuate e che ognuna di essa, tramite gli strumenti ODB-Tools, possano essere di conseguenza controllate e validate. Questa particolarità dovrebbe consentire di avere un Thesaurus comune corretto, valido e rappresentante in pieno la realtà che si deve descrivere.

Per quanto riguarda invece il calcolo del livello di affinità abbiamo anche qui un alto livello di interattività che permette sia di avere una correttezza maggiore nella valutazione dei pesi assegnati, sia di fare in modo che MOMIS possa adattarsi maggiormente alle varie realtà di basi di dati che andrà ad esaminare ed elaborare.

Il processo di raggruppamento (clustering) delle classi viene effettuato passando attraverso la creazione di una matrice M di dimensione K, dove K rappresenta il numero totale di classi individuate. In ogni cella di tale matrice verrà inserito un coefficiente GA(c_h,c_k) che rappresenta il grado di affinità tra le classi c_h e c_k. Tale processo di raggruppamento è ancora una volta interattivo e consiste nel mettere ogni classe in un cluster, dopo di che i due cluster che hanno un maggior coefficiente verranno uniti ad ogni iterazione e di conseguenza verranno eliminate contemporaneamente la riga e la colonna corrispondente ai valori uniti ed inserenti una nuova riga e colonna per il nuovo cluster definito e che prende il nome di c_{hk}. Il valore di GA tra c_{hk} ed i rimanenti cluster di M è impostato al massimo valore trovato confrontando i GA di c_h e c_k rispetto ai cluster prima dell'unione. Queste iterazioni continuano fino a quando rimane solamente un cluster e ciò permetterá di creare un albero di classi ODL_{I^3} i cui nodi intermedi corrispondono ai valori di GA trovati in ogni passaggio.

Una volta che l'albero di affinità viene costruito in questo modo occorre,

come ultimo passo, occuparsi della selezione dei cluster che dovranno essere integrati per la definizione dello schema globale. A questa operazione ci si serve ancora una volta dei maccanismi forniti da ARTEMIS oltre che all'ausilio esterno fornito dal livello di interattività garantito anche in qusta fase.

2.2.5 Generazione dello schema globale del Mediatore

Lo schema globale è la visione unificata del Mediatore relativamente alle sorgenti dei dati e le sue classi globali sono generate in modo interattivo utilizzando anche i moduli di supporto del Global Schema Builder (GSB).

Il processo di unificazione degli attributi viene eseguito automaticamente per quanto riguarda i nomi mentre, per la parte riguardante le affinità, vengono seguite le seguenti regole:

1. Nel caso di attributi che hanno il nome dell'affinità dovuta alla relazione di tipo SYN viene scelto uno dei nomi (con l'aiuto anche del Thesaurus Comune) che diventerà anche il nome del corrispondente attributo globale nella classe globale.

2. Per gli attributi che hanno il nome delle affinità dovuta alla relazione di tipo BT e NT viene scelto il termine più ampio dei due che andrà a dare il nome anche al corrispondente attributo globale nella classe globale.

Le informazioni addizionali necessarie per la costruzione dello schema sono ricavate tramite un processo di interazione che riguarda:

- il nome delle classi globali a scelta tra diversi candidati presi dalle relazioni terminologiche create nel Theasurus Comune, oppure un altro nome definito dall'utente;

- il "mapping" tra gli attributi globali e i corrispondenti attributi delle classi dei cluster individuati nel passo precedente. In particolare se un attributo globale è ottenuto da più di una classe appartenente allo stesso cluster è richiesto al designer di specificare il tipo di corrispondenza che tale attributo deve avere. MOMIS mette a disposizione le seguenti corrispondenze:

  - *and* per specificare che un attributo globale corrisponde alla concatenazione degli attributi di una classe c_h appartenente ad uno specifico cluster,

- *union* che specifica la corrispondenza tra un attributo globale ed al massimo uno degli attributi specificati di una classe c_h appartenete ad uno specifico cluster;

- l'assegnazione dei valori di dafault agli attributi globali in corrrispondenza di una classe c_h appartenente ad uno specifico cluster;

- nuovi attributi per le classi globali.

2.2.6 Processi di Query ed ottimizzazione

Una volta definito lo schema globale della base di dati MOMIS è pronto per poter eseguire le query che l'utilizzatore gli sottoporrà.

La prima attività che viene effettuata dal sistema ogni volta che gli viene sottoposta una query è quella di decomporre la query passata in tante sottoquery che vadano successivamente ad interrogare le base di dati delle sorgenti che sono in grado di fornire l'informazione cercata. Questo processo si suddivide principalmente in due fasi distinte:

- Ottimizzazione semantica

- Piano di formulazione della query

Ottimizzazione semantica

In questa fase ci si occupa principalmente di ottimizzare il costo di una query rimpiazzandola con una nuova che abbia le caratteristiche di rispettare le eventuali restrizioni implicite nella query stessa oppure derivate implicitamente dallo schema globale dei dati.

L'ottimizzazione semantica opera soprattutto dal lato dell'introduzione di fattori aggiuntivi nella clausola "where" che porta da un lato un maggior costo dal punto di vista della formulazione, ma come contropartita porta ad un alleggerimento del peso di ricerca dei dati se si riesce a coinvolgere degli indici secondari esistenti nella base di dati tramite i predicati aggiunti.

Piano di formulazione della query

Una volta ottenuto l'ottimizzazione dal punto di vista semantico ci si occupa dell'esplosione di tale query ottimizzata in tante "sottoquery" che dovranno andare ad interrogare le varie basi di dati coinvolte nello schema globale e

che contribuiranno a fornire i risultati desiderati.

Per l'ottenimento di queste sottoquery il Query Manager si occupa di controllore tutti i fattori booleani della clausola where, in particolare viene utilizzato un algoritmo per la generazione delle query locali.

2.3 Il linguaggio ODL$_{I^3}$

Finora si è sempre parlato di ODL$_{I^3}$ e dell'utilizzo delle sue classi, quindi ritengo in questo momento utile illustrare brevemente alcune caratteristiche di questo linguaggio, lasciando al capitolo successivo l'approfondimento a riguardo la sintassi e la struttura del linguaggio stesso.

ODL$_{I^3}$ è un linguaggio di descrizione dei dati ad alto livello, si avvale dello standard di modelli ad oggetti ODMG-93 [14, 29] ed è indipendente dal formato dalla sorgente dei dati stessi. Esso deriva dal linguaggio ODL con delle estensioni riguardanti: l'introduzione di due tipi di rules, il costrutto *union* ed il costrutto *optional* (*), le relazioni terminologiche intra ed inter sorgenti.

Nell'ambito di MOMIS viene appunto sfruttato per le sue caratteristiche che consentono di al mediatore di lavorare allo stesso modo per sorgenti diverse, sia per dati semistrutturati che per dati strutturati.

La traduzione di ogni particolare schema in un formato ODL$_{I^3}$ avviene mediante l'utilizzo di wrapper i quali sono anche responsabili dell'aggiunta di informazioni utili per il mediatore stesso quali il nome ed il tipo della sorgente originaria. Questa operazione di traduzione avviene direttamente sulle basi della sintassi ODL$_{I^3}$ e sulla definizione del modello ad oggetti, in particolare dato un modello <l,A> o una relazione di una orgente relazionale abbiamo che una classe ODL$_{I^3}$ corriponde a l o al nome della relazione e per ogni etichette $l' \in A$ o attributo relazionale un attributo è definito nella corrispondente classe ODL$_{I^3}$.

Capitolo 3

Il linguaggio ODL_{I^3}

Come già detto nel capitolo 2 il punto di arrivo dell'acquisizione degli schemi sorgenti e la base di partenza per le funzionalità di MOMIS stesso è il linguaggio ODL_{I^3} .

I *wrappers* dovranno quindi preoccuparsi di leggere gli schemi sorgenti e di tradurre tali schemi tramite la sintassi ODL_{I^3} i quali verranno successivamente passati al Global Schema Builder per l'elaborazione.

3.1 Il linguaggio ODL

Il linguaggio ODL (**O**bject **D**efinition **L**anguage) è il linguaggio utilizzato per la specifica di schemi ad oggetti proposto dal gruppo di standardizzazione ODMG-93 [14, 29] e riconosciuto come un standard dalla comunità informatica. Tale linguaggio svolge, nell'ambito degli ODBMS, le funzioni di definizione degli schemi che nei sistemi tradizionali sono assegnate al Data Definition Language.

Caratteristiche fondamentali di tale linguaggio, come per altro anche degli altri linguaggi che si basano sul paradigma ad oggetti, sono:

- definizione di tipi classe e tipi valore
- distinzione tra intensione ed estensione di una classe di oggetti
- distinzione di attributi tra semplici e complessi
- definizione di attributici atomici e collezione (set, list, bag)

- definizione di relazioni binarie con relazioni inverse
- dichiarazione delle signature dei metodi.

La sintassi di *ODL* è un ampliamento della sintassi prevista per il linguaggio *IDL* (**I**nterface **D**efinition **L**anguage) che è il linguaggio sviluppato nell'ambito del progetto CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture).

3.2 La estensione I_3: il linguaggio ODL$_{I3}$

ODL$_{I^3}$ deriva direttamente da *ODL* e ne rappresenta una estensione in accordo con le raccomandazioni della proposta di stadardizzazione per i linguaggi di mediazione elaborata presso l'Università del Maryland dal gruppo di lavoro I_3 costituitosi in tale ambito. Infatti il linguaggio *ODL*, pur essendo ben progettato per la conoscenza relativa ad un singolo schema ad oggetti, risulta insufficiente per la descrizione di un insieme si sorgenti di basi di dati eterogenee quale quello richiesto da MOMIS e quindi può essere considerato solamente come una buona base di partenza nel progetto di integrazione.

Seguendo il lavoro dell'I_3 Workgroup si è potuto quindi ottenere un linguaggio che fosse in grado di supportare sia dei sistemi complessi, quali possono essere quelli ad oggetti, sia i modelli più semplici, quali i file strutturati. Malgrado questi obiettivi si è però sempre cercato di mantenere un certa uniformità rispetto al linguaggio *ODL* e quindi si lavorato sulla sintassi in modo da introdurre il meno possibile delle divergenze.

Il risultato di questo lavoro ha portato appunto ad ODL$_{I^3}$ che ha aggiunto nuovi scopi, rispetto a quanto previsto in *ODL* e tutt'ora presenti, di descrizione che possono essere riassunti nei seguenti punti:

- è stata data al wrapper la possibilità di indicare, per ogni classe, il nome del sorgente di appartenenza ed il relativo tipo;
- nel caso di sorgenti relazionali è possibie definire, per le classi appartenenti a tali sorgenti, le chiavi candidate e le eventuali foreign key;
- sono stati previsti due nuovi costrutti quali *union* e *optional* che servono rispettivamente per avere strutture dati alternative e per indicare

la natura opzionale di un attributo. Queste caratterisitiche sono in accordo con la strategia utilizzata per la descrizione di dati semistrutturati;

- è stato introdotta la possibilità di definire delle grandezze locali e delle grandezze globali;
- viene supportata la dichiarazione di regole di mapping, dette *mapping rule* fra grandezze locali e grandezze globali;
- è possibile la definizione di regole di integrità denominate *if then rule* che possono essere applicate sia sullo schema globale che sui singoli schemi locali;
- è stato introdotto il supporto per la definizione di relazioni terminologiche di sinonimia (SYN), ipernimia (BT), iponimia (NT) ed associazione (RT);
- il linguaggio può essere automaticamente tradotto nella logica descrittiva OLCD usata dagli ODB-Tools e quindi utilizzarne le capacità, nei controlli di consistenza e nell'ottimizzazione semantica, delle interrogazioni.

Nell'appendice B è riportata in BNF la sintassi del linguaggio ODL_{I^3} .

Andiamo adesso ad illustrarne un po' più in dettaglio le caratteristiche principali e soprattutto le estensioni che sono state introdotte rispetto alla sitassi di *ODL*.

3.3 I tipi di dati

In ODL_{I^3} , secondo quanto presente anche nella sintassi del lingaggio *ODL*, è prevista una distinzione fondamentale nei tipi di dati e precisamente:

- **Tipi valore**
- **Tipi classe** denominati più semplicemente **Classi**

I primi sono caratteristici delle variabili e degli attributi semplici ed ogni istanza di questo tipo non ha un proprio identificatore, ma la sua unica proprietà è il suo valore. Diversamente gli attributi complessi, o comunque gli oggetti in generale, sono istanze di classe caratterizzate da un proprio OID,

una propria interfaccia ed un proprio comportamento. Questa differenza è rappresentata anche nella figura 3.1.

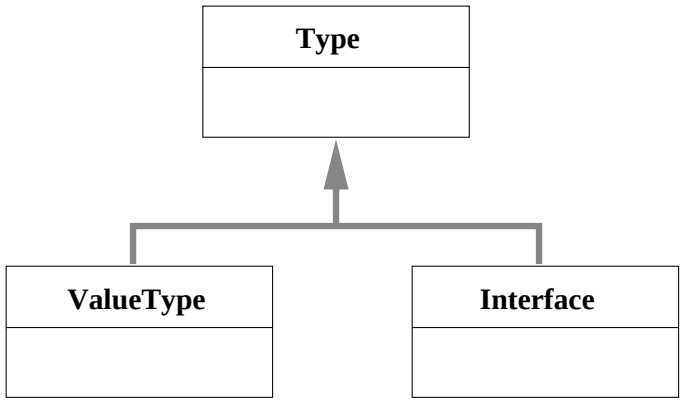

Figura 3.1: I tipi di ODL$_{I^3}$

3.3.1 I tipi valore

Occorre fare una ulteriore distinzione per questi tipi in modo da scendere maggiormente nel dettaglio degli attributi rappresentabili:

- **SimpleType**
- **ConstrType**

come visualizzato nella figura 3.2.

SimpleType

Appartengono a questa categoria tutti i dati di tipo atomico quali possono essere i tipi predefiniti (integer, float, char, boolean, etc...), i vari tipi di collezione **TemplateType** (set, list, bag, array) nonché i tipi **DefinedType** definiti dall'utente.

I **TemplateType** non sono altro che una collezione di istanze tutte omogenee tra di loro di tipo SimpleType. Mentre per i **DefinedType** la sintassi *ODL* prevede una definizione che segue le stesse regole del linguaggio ANSI C: in particolare un tipo definito possiede un nome proprio ed un tipo mappato (è consentito solo il tipo-valore) ed eventualmente un insieme di valori

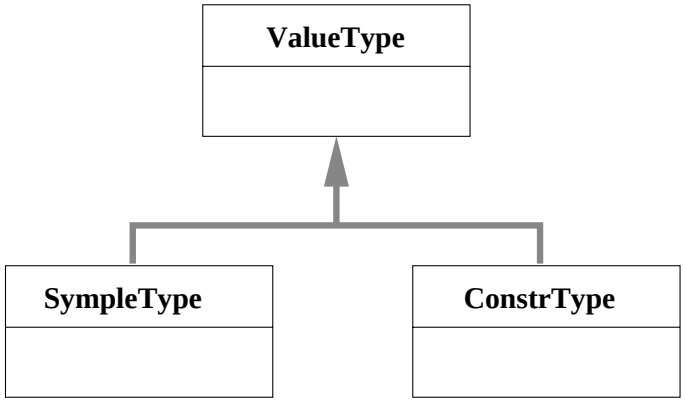

Figura 3.2: I tipi valore

interi, nel caso si tratti di un array e che stanno ad indicare le dimensione dell'array stesso. La particolarità che il tipo riferito sia un generico ValueType piuttosto che un SimpleType consente la dichiarazione di variabili strutturate, ad esempio, tramite una sitnassi semplificata: prima si definisce un tipo nuovo, poi le variabili secondo la sintassi dei tipi atomici.

Una conseguenza del fatto che i TemplateType consentano una collezione di istanze solamente SimpleType è quella di non rendere possibile una dichiarazione diretta di una variabile come *set* di *struct*. Tale risultato è possibile però ottenerlo grazie ad un artificio che consta di due passi: prima di tutto si crea una definizione di un nuovo tipo e poi la si dichiarazione della collezione desiderata. Il tutto è meglio illustrato nel seguente esempio:

```
typedef struct
{
  int a,b;
  boolean x;
} tipostruct;

set<tipostruct> coord1, coord2;
```

La struttura dei SimpleType è graficamente illustrata nella figura 3.3.

ConstrType

Anche gli attributi di questo tipo possono essere dettagliati ulteriormente in **StructType**, **UnionType** ed **EnumType** dove i primi due sono caratteriz-

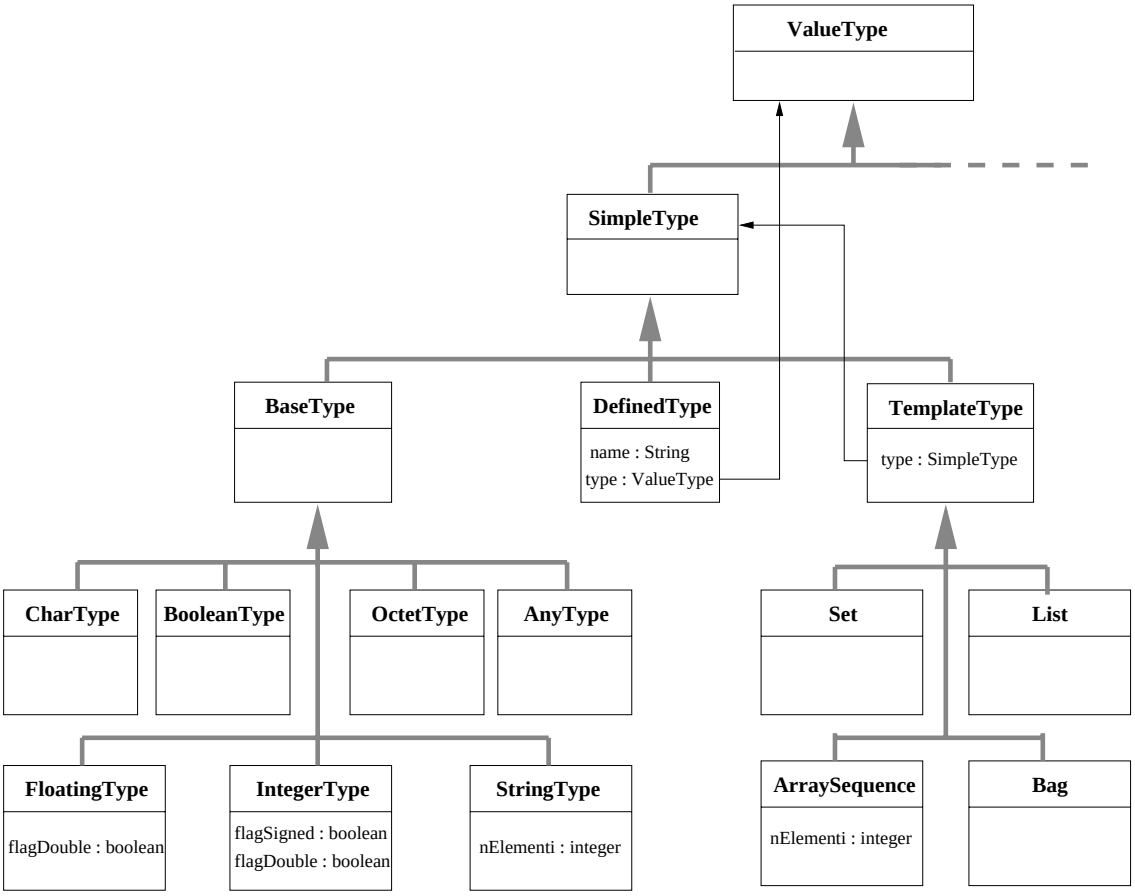

Figura 3.3: I tipi semplici

zati da un tagName opzionale che da' la possibilità all'utente di dichiarare diverse variabili dello stesso tipo senza essere costretto tutte le volte a ridescrivere l'intera struttura. Non bisogna però cadere nell'errore di considerare il tagName come un nome di tipo in quanto non si tratta di un elemento a se stante, ma piuttosto deve essere sempre accompagnato dal tipo-struttura. Il tutto è evidenziato nel seguente esempio:

```
typedef struct nometag
{
  char x;
  boolean y;
  float z;
} tipostruct;
```

che consente la dichiarazione in modo del tutto equivalente delle seguenti variabili:

```
tipostruct var1;
struct nometag var2;
```

Allo stato attuale il tipo UnionType è sensibilmente semplificato. Infatti, a parte il tipo della variabile di switch, tutto ciò che riguarda la definizione non viene interpretata, ma viene solamente archiviata così come è stata definita. Ovviamente la correttezza sintattica della definizione deve sempre essere garantita.

Imvece i tipi **EnumType** hanno un comportamento leggermente diverso da quanto appena illustrato, infatti in essi non c'è altro che un semplice elenco finito di valori e la variable associata dovrà assumere solamente uno dei valori elencati nella dichiarazione.

L'intera struttura dei ConstrType può essere agevolmente essere illustrata nella figura 3.4.

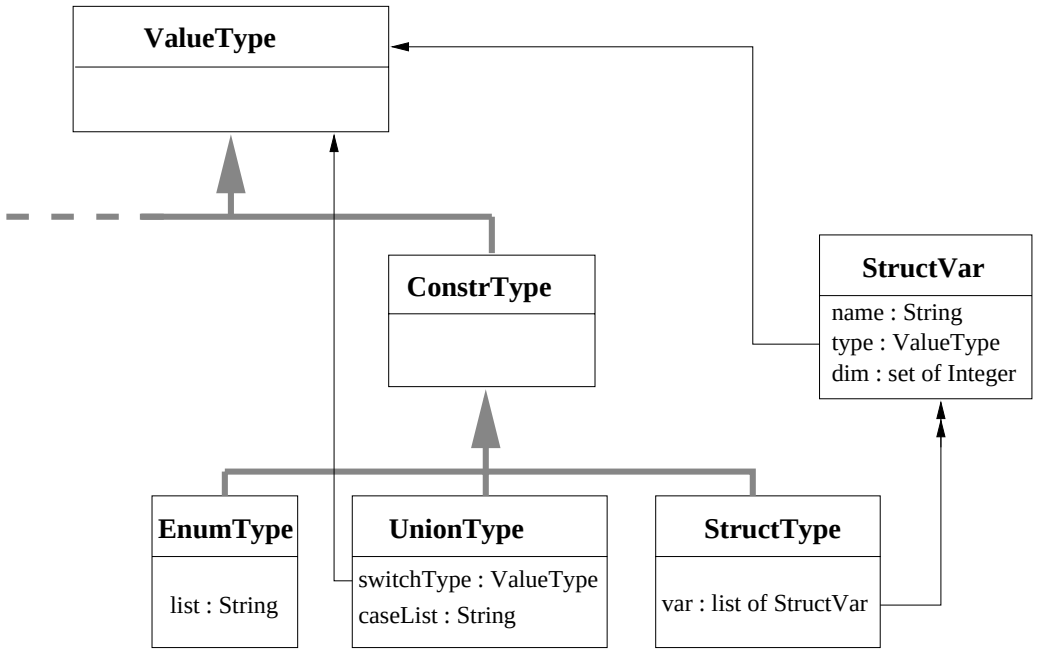

Figura 3.4: ConstrType

3.3.2 I tipi classe

I tipi classi, più brevemente denominti *Classi*, sono identificati dalla parola chiave `interface` e non sono altro che un insieme di diverse dichiarazioni.

Ogni classe è caratterizzata da un proprio nome, dal riferimento al sorgente che l'ha generata ed eventualmente dal riferimento ad una eventuale superclasse da cui deriva secondo un criterio di eriditarietà ammessa per questo linguaggio.

Ereditarietà

Nella figura 3.5 viene illustrato come viene mappato il discorso della ereditarietà all'interno di una classe ODL$_{I^3}$ ed evidenzia come per ogni Interface sia ammessa l'ereditarietà multipla, cioè sia ammessa la possibilità di avere una o più superclassi. Questa proprietà viene rappresentata attraverso la collezione complessa *inheritance* che mappa sulla stessa classe **Interface**.

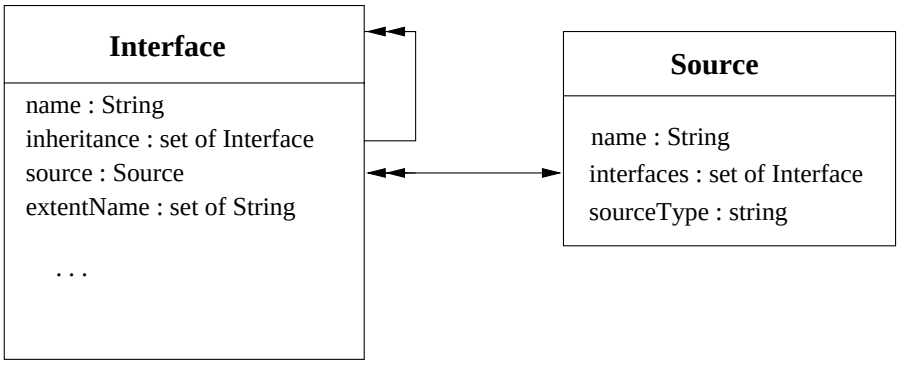

Figura 3.5: Ereditarietà e Source

Source

Nel linguaggio ODL$_{I^3}$ è stato aggiunto un costrutto per la dichiarazione del sorgente che contiene la descrizione della classe. Tale costrutto mi indica, come visualizzato sempre nella figura 3.5, che ogni classe appartiene ad un sorgente caratterizzato da un nome, e dal tipo del sorgente stesso (relazionale, a oggetti, file, semistrutturato).

Poiché non viene a priori esclusa la presenza di classi caratterizzate da uno stesso nome per ottenere una identificazione univoca della classe stessa occorre ricorrere alla coppia *nomeSource-nomeInterface*, cioè al mantenimento anche all'interno dell'oggetto **Source** delle lista di tutte le classi che ne fanno parte. Questo meccanismo risulta efficace anche dal punto di vista di una milgiore navigazione nella struttura dei dati.

Extent

Per facilitare il reperimento indicizzato, e di consegunza rapido, delle informazioni da parte del DBMS il linguaggio ODL_{I^3} da' la possibilita di dichiarare nella classe un'estensione tramite la parola chiave `extent`.

Una estensione rappresenta un insieme di tutte le istanze delle classe all'interno di un particolare database e, nel caso di classe globale, potrebbe tornare utile la dichiarazione di più estensioni associate alla classe stessa attraverso l'uso di un set di nomi in formato stringa.

Key e Foreign Key

Nella sintassi di ODL_{I^3} è stata inserita anche la possibilità di definire delle chiavi candidate, delle chiavi semplici e delle chiavi composte. Infatti per ogni istanza dell'oggetto **KeyList** corrisponde una chiave candidata che coinvolge un solo attributo nel caso di chiave semplice, oppure più attributi nel caso sia necessario descrivere una chiave composta.

Poiché tale linguaggio deve essere in grado di descrivere anche dei database relazionali è stata inserita una sintassi per la descrizione delle foreign key in modo non perdere le informazioni che il modello ralazionale comporta per la realizzazione delle gerarchie di aggregazione. Tale prerogativa consente quindi di mantenere l'informazione in un oggetto **ForeignKey** in cui viene riportata la lista degli attributi componenti, la lista dei medesimi attributi appartenenti alla classe riferita e la classe riferita stessa.

Il tutto è graficamente illustrato nella figura 3.6.

Un esempio in ODL_{I^3} potrebbe essere dato dalle seguente linee di codice:

```
interface Bistro
```

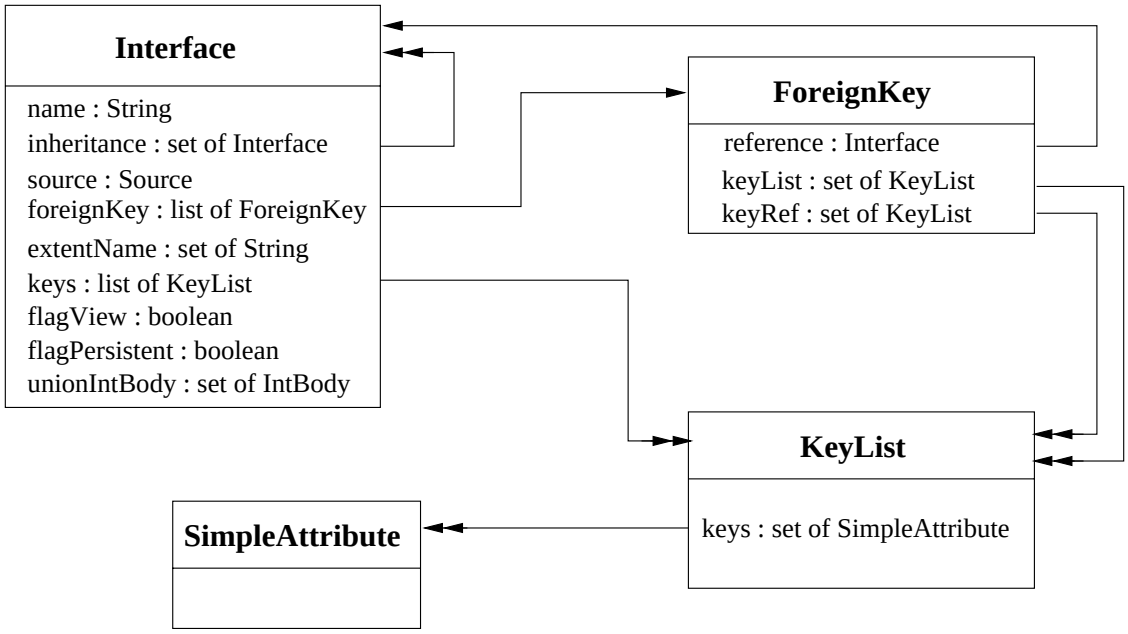

Figura 3.6: Key e Foreign key

```
( source relational Food_Guide
  key r_code
  foreign_key(r_code) references Restaurant,
  foreign_key(pers_id) references Person
)
{  ...  }
```

in cui si evidenziano alcune caratteristiche per le entità relazionali e precisamente:

- vi è una dichiarazione di un sorgente (`Food_Guide`) di tipo relazionale;
- la presenza, obbligatoria nel caso di tabelle relazionali, di una chiave (`r_code`);
- sono state utilizzate due *foreign key* per mantenere l'aggregazione fra la tabella descritta ed altre due tabelle (`Restaurant` e `Person`) componenti la struttura relazionale della base di dati.

Interface Body

Nel *Body* dell'interfaccia vengono descritti tutti i metodi e gli attributi che sono caratteristici della classe (o vista). Diversamente da quanto previsto per il linguaggio *ODL*, in ODL$_{I^3}$ è prevista la possibilità di definire strutture di dati semistrutturati che, per definizione ed a differenza dei dati strutturati, non devono seguire una rigida formattazione.

I dati semistrutturati, infatti, si prestano a rappresentare lo stesso aspetto della realtà attraverso contenuti fortemente differenziati a livello strutturale; attributi che generalmente assumono questa caratteristica potrebbero essere l'*indirizzo*, l'*ora* oppure la *data*. Per quest'ultimo caso, ad esempio, ci si potrebbe trovare di fronte ad un formato dato solamente da una stringa in cui sono racchiuse tutte le informazioni, oppure ad un formato composto da un insieme di campi quali il *giorno* il *mese* e l'*anno*.

Il linugaggio ODL$_{I^3}$ mette a disposizione, per questi casi, lo speciale costrutto `union` che consente di definire più specifiche alternative per una stessa interfaccia. Tale costrutto permette di confrontare degli oggetti che rappresentano due istanze della medesima realtà, ma che sono descritte attraverso scelte strutturali differenti.

Riprendendo l'esempio dell'attributo *data* ci si potrebbe trovare di fronte a delle righe di sorgente come le seguenti:

```
interface Appointment
{
  ...
  attribute struct {
        unsigned int year;
        unsigned short month;
        unsigned short day; } date;
  ...
};
union
{
  ...
  attribute string date;
  ...
};
```

Questo esempio mette anche in evidenza il fatto che ad una stessa classe

possono appartenere più attributi che hanno lo stesso nome, uno diverso per ogni implementazione consentita.

La struttura di una InterfaceBody può essere illustrata tramite la figura 3.7

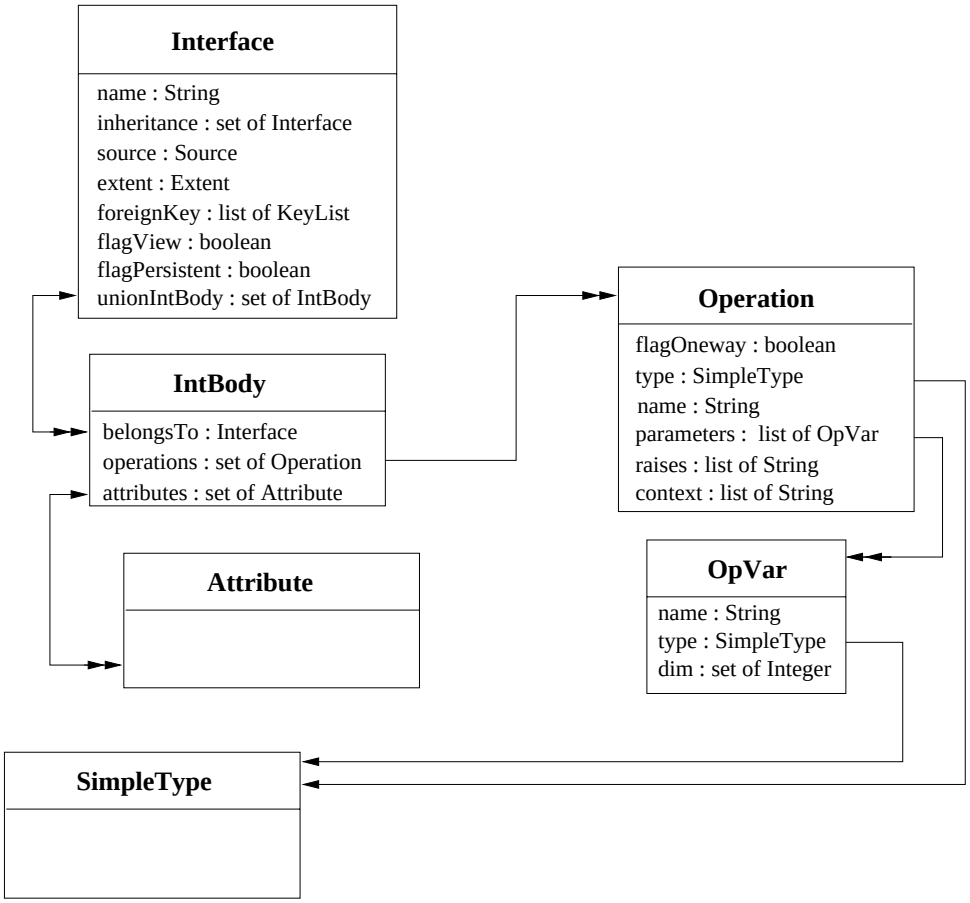

Figura 3.7: Interface body

Operazioni

Nella stessa figura 3.7 sono rappresentate anche le operazioni, che non sono altro che i metodi della classe o meglio la loro signature. Infatti la classe **Operation** contiene le informazioni riguardanti il nome ed il tipo restituito dal metodo ed una lista di parametri passati (di tipo **OpVar**) ognuno avente

un nome, un tipo ed un elenco di eventuali dimensioni di array.

Unica cosa da sottolineare per questa classe riguarda i tipi ammessi per i parametri passati e per il tipo restituito dal motodo stesso che sono previsti dalla sintassi siano istanze della classe **SimpleType**.

Attributi

Sono previsti due categorie di attributi differenziati dal tipo a cui appartengono: gli attributi semplici di tipo-valore gli attributi composti di tipo-classe. Oltre a questo viene data anche la possibilità di dichiarare delle *relationships*, cioè degli attributi complessi che possiedono anche la realzione inversa, e di utilizzare un costrutto per definire delle *mapping rule* in grado di legare grandezze appartenenti allo schema integrato con grandezze degli schemi locali.

Nella figura 3.8 sono evidenziati tre tipi di attributi possibili:

- I **SimpleTypeAttribute** caratterizzati da un tipo generico denominato **Type**, un flag booleano per indicare se si tratta di attributi di sola lettura ed un set di eventuali indici di array.
- Le **Relationship** che servono per mappare una interfaccia e contengono le informazioni sulla inversa attraverso un puntatore alla stessa classe **Relationship**. In questo caso il tipo (*type*) indica se l'attributo è una collezione oppure no, mentre *orderBy* serve per definire il criteri di ordinamento.
- I **GlobalAttribute** hanno le stesse caratteristiche degli attributi locali ma possiedono in più la proprietà di avere un collegamento con un oggetto **MappingRule**.

Mapping rule

Premetto prima di tutto che, qualora la dichiarazione di un attributo sia seguito da un insieme di regole di mapping, l'attributo è automaticamente considerato globale.

Preso un attributo globale è possibile definire diverse regole di mapping a seconda della caratterisitche degli attributi stessi che vengono coinvolti. In particolare si possono presentare i seguenti casi:

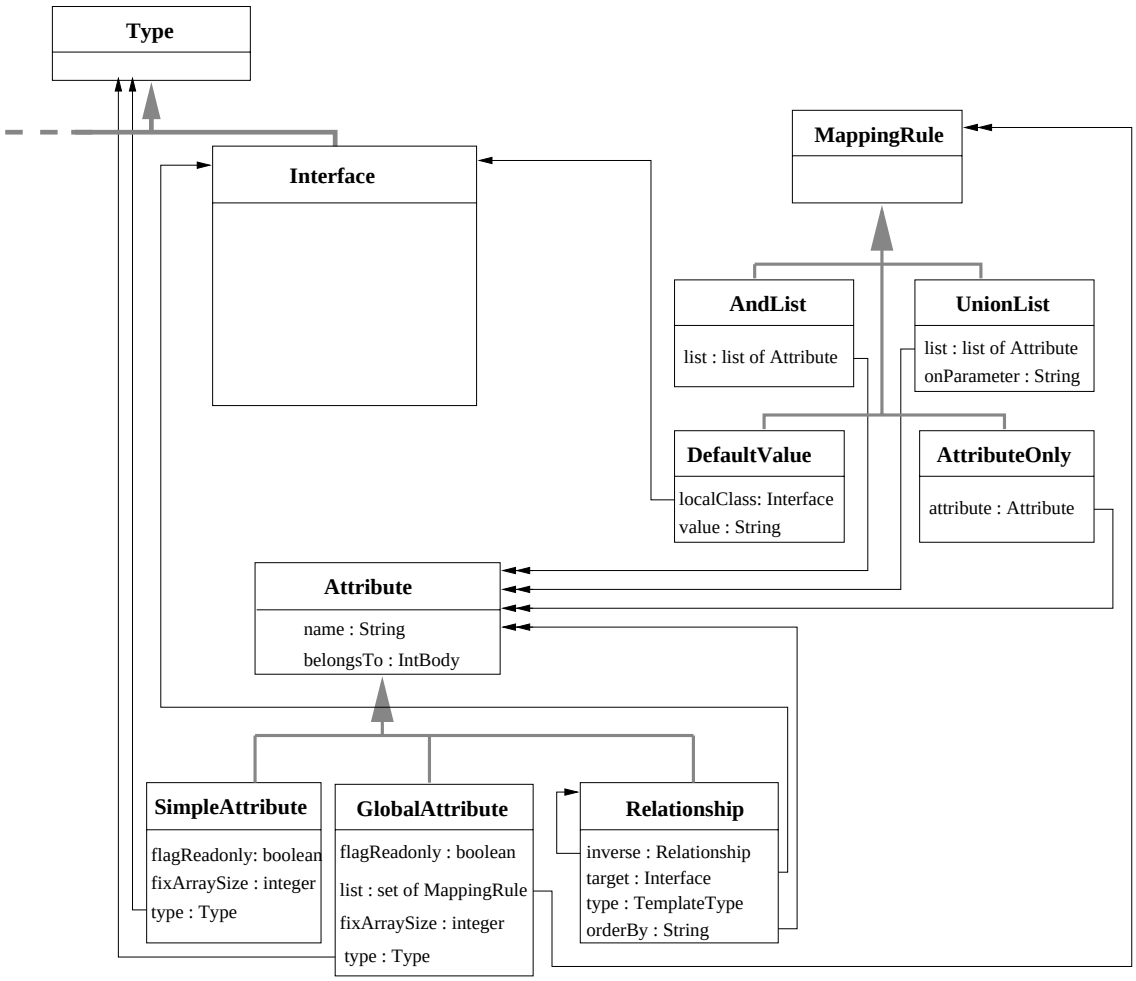

Figura 3.8: Attributi e Mapping rule

- corrispondenza tra l'attributo globale ed un solo attributo locale, che è anche il caso più semplice. Se ricade in questa eventualità, infatti, il sistema possiede le informazioni necessarie per realizzare in modo automatico il meccanismo di mapping e fa coincidere il nome con quello dell'attributo locale;

- corrispondenza tra l'attributo globale ed un insieme di attributi locali tutti appartenenti a classi locali differenti, ma in fusione tra di loro. Come nel caso precedente il mapping si ottiene in maniera automatica, ma il nome potrebbe essere assegnato dal progettista per identificare la grandezza con una denominazione appropriata.

- nel caso ci sia una corrispondenza tra l'attributo globale e la fuzione di diversi attributi locali, magari appartenenti alla stessa classe locale, possiamo distinguere due sottocasi:

  - gli attributi locali sono concatenati tramite una condizione *and* il che' implica che l'attributo globale sia ottenuto dal concatenamento degli attributi locali stessi. Tipico esempio di questa eventualità potrebbe essere essere data dall'attributo globale `Owner.name` che mappa in `Person.first_name` e `Person.last_name`.
  - gli attributi locali sono legati da un costrutto del tipo *union* il che' viene risolto tramite la corrispondenza tra l'attributo globale ed uno solo alla volta degli attributi locali in union. Per poter individuare con esattezza quale di questi parametri locali debba essere mappato si è ricorso ad un terzo attributo locale rappresentato nella struttura di figura 3.8 da *onParameter*.

- nel caso non esista una corrispondenza tra l'attributo gloable e gli attributi locali potrebbe essere necessario esprimere un metaconcetto tramite un valore di default. Riprendendo l'esempio di [30] si può notare come si sia presentata l'esigenza di imporre valori di default per l'attributo globale aggiunto `Food_Place.zone` poiché localmente la conoscenza sul reparto è implicita nel nome dei sorgenti. Tali valori potrebbero essere dati da:
  `Food_Place.zone = 'Pacific Coast'` associato a `ED.Fast-Food`
  `Food_Place.zone = 'Atlantic Coast'` per `ID.Restaurant`
  `Food_Place.zone = 'Atlantic Coast'` per `ID.Bistro`
  `Food_Place.zone = 'Atlantic Coast'` per `ID.Brasserie`.

Per riassumere e focalizzare i concetti espressi potrebbero essere analizzate le seguenti righe di codice:

```
interface Food_Place
{
  attribute name
            mapping_rule ED.Fast-Food.name,
                         FD.Restaurant.name,
                         FD.Brasserie.name;
  ...
  attribute category
```

```
            mapping rule ED.Fast-Food.category,
                         FD.Restaurant.category,
                         FD.Bistro.type;
  attribute specialty
            mapping rule ED.Fast-Food.specialty,
                         FD.Restaurant.special dish;
  attribute address
            mapping rule ED.Fast-Food.address,
                         (FD.Restaurant.street and
                          FD.Restaurant.zip code and
                          FD.Brasserie.address);
  attribute price
            mapping rule ED.Fast-Food.midprice,
                         FD.Restaurant.tourist menu price;
  attribute zone
            mapping rule ED.Fast-Food = 'Pacific Coast'',
                         FD.Restaurant = 'Atlantic Coast',
                         FD.Bistro = 'Atlantic Coast',
                         FD.Brasserie = 'Atlantic Coast';
}
```

in cui sono messe in evidenza alcune caratteristiche valide per ogni classe globale:

- allo stato attuale non ci sono superclassi: lo schema integrato ha gerarchie di aggregazione, ma non ereditarietà;
- non viene specificato alcun sorgente, nessuna estensione e nemmeno chiavi o foreign key per la classe `Food_Place`; questo vale per tutte le classi globali, essendo l'informazione sulla sorgente tipica delle grandezze locali;
- ogni attributo globale ha almeno una regola di mapping, cioè un corrispondente locale, sia esso un attributo o un valore di default o entrambe le cose;
- DA RIVEDERE PER L'ESEMPIO MIO. ANDREA poiché gli attributi locali `CD.Patient.physician` e `ID.Patient.doctor_id` sono complessi, il dominio del corrispondente globale `physician` deve a sua volta essere complesso: in particolare conterrà oggetti della classe globale `Medical_Staff`.

La porzione di **Object model** che riguarda i *tipi-classe* può essere riassunto in una sua visione di insieme dalla figura 3.9

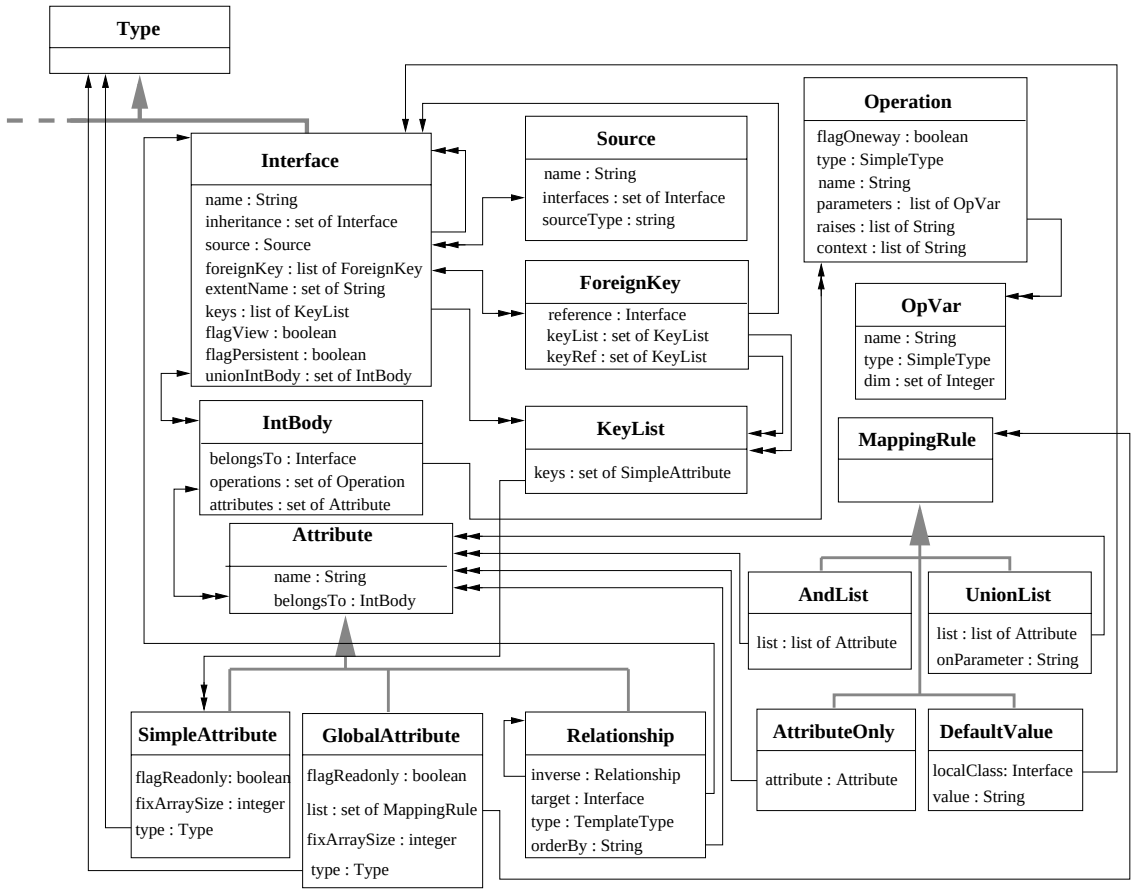

Figura 3.9: Schema completo della rappresentazione dei tipi-classe

3.3.3 Costanti

La sintassi *ODL* prevista per la classe **Constant** è praticamente identica a quella dell'ANSI C e quindi non richiede approfondimenti particolari.

Nella rappresentazione della figura 3.10 sono identificabili tra attributi:

- il nome della costante
- il tipo della costante che deve essere un oggetto **BaseType**
- il valore che assume.

Unica nota da segnalare per questa classe riguarda il formato di archiviazione del valore che è di tipo stringa e quindi dovrà essere convertito nel

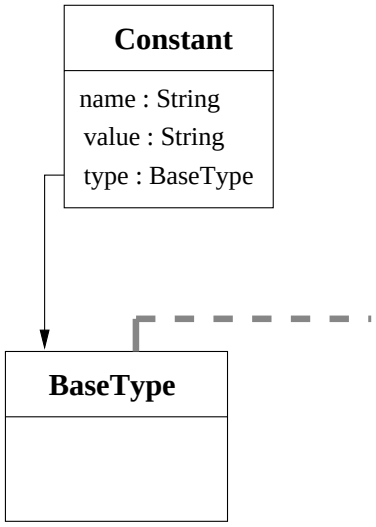

Figura 3.10: Le costanti

formato corretto tramite un operazione di cast da chi si occupa della consultazione della struttura dati.

3.4 Relazioni terminologiche

Altra aggiunta nella sintassi del linguaggio ODL$_{I^3}$ riguarda la possibilità di inserire delle relazioni terminilogiche che possono intercorre tra classi, tra attributi oppure miste tra gli attributi e le classi. Sarà poi il parser che si occuperà di interpretare e capire quali siano gli attributi e/o le classi coinvolte in tali relazioni.

Fondamentalmente i tipi di relazione possibili sono gli stessi visti nel paragrafo 2.2.3 e precisamente:

- Sinonimia (SYN),
- Ipernimia (BT),
- Iponimia (NT),
- Associazione (RT).

Esistono diversi casi per l'archiviazione della strttura dati, tutti rappresentati nella figura 3.11, e precisamente:

- tramite la creazione di un oggetto **InterfaceRel** per quanto riguarda una relazione che coinvolge due classi;
- tramite la creazione di un oggetto **AttributeRel** per quanto riguarda una relazione che coinvolge due attributi;
- tramite la creazione di un oggetto **AttrIntRel** per quanto riguarda il caso misto di una relazione che coinvolge un attributo ed una classe.

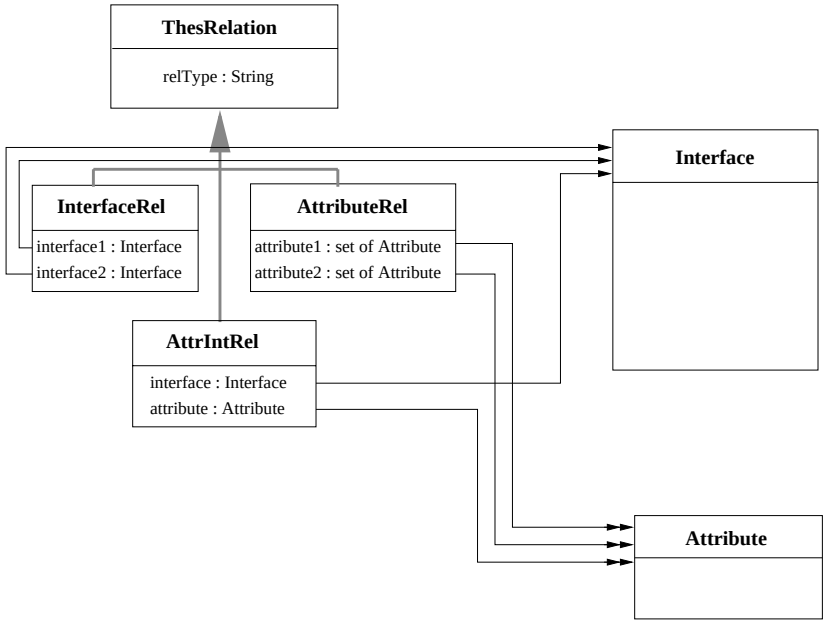

Figura 3.11: Relazioni terminologiche

3.5 Regole di integrità

Ultima particolarità della sintassi di ODL$_{I^3}$ è data dalle regole di integirtà di tipo *if-then* che devono essere verificate per ogni istanza del database. Tale sintassi contiene due forme disitnte e precisamente:

`rule` *nomerule*
`forall` *iteratore* `in` *collezione* : *antecedente* `then` *conseguente*

oppure:

`rule` *nomerule*
`[{ case of` *identifier* : *caselist* `}]`

Dove i termini che rappresentano gli attributi della classe **Rule** (rappresentata nella figura 3.12) assumono il seguente significato:

- *nomerule*: è il nome della regola
- *iteratore*: è in identificatore che rappresenta l'istanza.
- *collezione*: è un insieme di istanze e può essere una classe o parte di essa.
- *antecedente*: è la condizione di *if* che si ottiene ponendo in *and* una serie di predicati booleani.
- *conseguente*: è l'affermazione valida per tutte le istanze che verificano l'*antecedente*. La sintassi di *antecedente* e *conseguente* è la medesima.

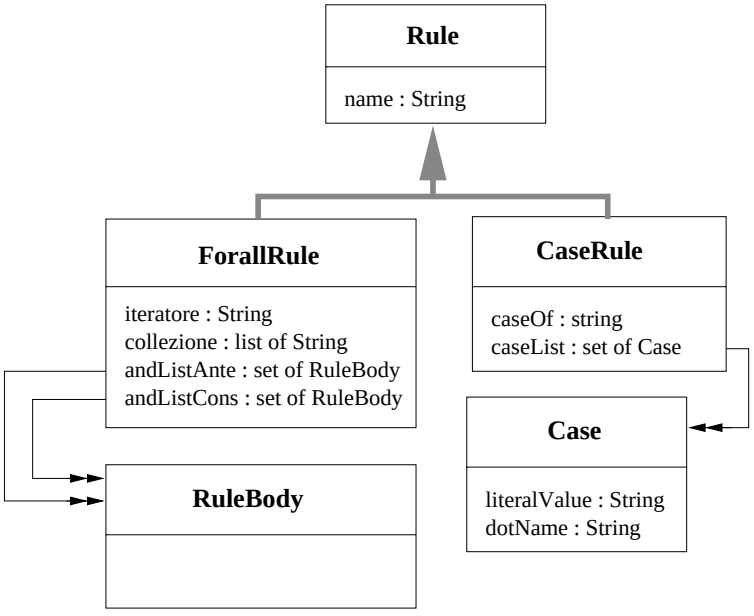

Figura 3.12: Regole di integrità

La differenza fra il costrutto *if* ed il costrutto *case* consiste essenzialmente nel fatto che questo secondo fornisce un criterio di scelta fra più attributi locali che sono mappati in *union* da un unico attributo globale.

Questo tipo di mapping è stato già brevemento trattato nel capitolo 3.3.2 e qui viene sfruttato per l'esigenza di sapere, attraverso una regola definita, in base a quali valori di un dato parametro un attributo globale corrisponde via via ad ognuno degli attributi locali.

Scendendo nel particolare per quanto riguarda le condizioni *antecedente* e *conseguente* possiamo notare che esse sono formate da un predicato booleano oppure da più condizioni poste in *and* tra loro. Ognuno di questi predicati non è altro che un oggetto della classe **RuleBody** che può essere rappresentata nello schema di figura 3.13.

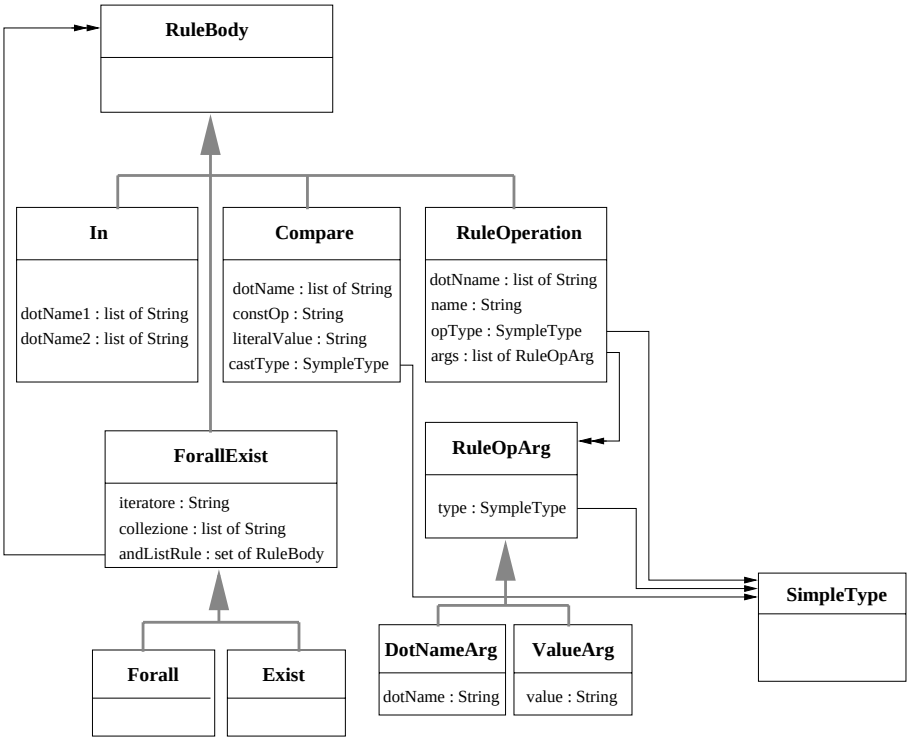

Figura 3.13: Predicati booleani componenti le condizioni di *antecedente* o *conseguente*

Descriviamo un po' in dettaglio le particolarità di ognuna di queste condizioni.

In

Confronta due oggetti e ritorna un valore vero se *dotName1* (ad esempio un oggetto) è contentuo nella collezione *dotName2*. Viene utilizzato questa denominazione per gli attributi in quanto la sintassi ODL$_{I^3}$ prevede che contengano una lista di stringhe e che siano dei cammini esprimili tramite la *dot-notation* caratteristica di alcuni linguaggi orientati agli oggetti.

Forall e Exist

Per questo predicato è prevista una sintassi identica a quanto già visto per la prima parte della rule in quanto è previsto che si faccia una chiamata ricorsiva a predicati booleani della classe **RuleBody**.

La differenza tra questi due predicati consiste nel numero di match necessari perché la condizione ritornata sia vera, in particolare:

- `Forall` *itaratore* `in` *collezione* : *condizioni_in_and*

  ritorna vero se per tutte le istanze valgono tutte le condizioni in *and*

- `Exist` *itaratore* `in` *collezione* : *condizioni_in_and*

  ritorna vero se esiste almeno una istanza per cui valgono tutte le condizioni in *and*.

Compare

Si tratta del classico operatore di confronto descritto in più di un linguaggio e che ritorna il valore vero o falso a seconda che il confronto tra il valore di una varaibile (ad esempio un attributo) ed il valore di un letterale sia rispettivamente verificato o meno.

Gli operatori di confornto sono i classici segni matematici $=$, $<$, $\leq$, $>$, $\geq$ mentre l'attributo *CastType* serve per fare, quando richiesto, il cast del valore letterale in modo da rendere possibile il confronto con la variabile specificata. In questi casi contiene il tipo della variabile che entrerà nel confronto.

RuleOperation

Questo confronto coinvolge una variabile (*dotName*) e il valore ritornato da una funzione invocata (chiamata operatore) e ritorna vero se l'uguaglianza

tra questi due operandi risulta verificata.

Gli altri attributi della classe **RuleOperation** sono:

- *name*: è il nome dell'operazione;
- *opType*: è il tipo ritornato dall'operazione;
- *args*: è una lista di parametri (oggetti **RuleOpArg**), di ognuno dei quali si conosce:
  - *type*: il tipo del parametro;
  - *dotName* o *value* a seconda che il parametro sia rispettivamente una variabile/attributo/costante o un letterale;

3.6 Un esempio di riferimento

Riporto, come riferimento per i concetti illustrati oltre che come completamento per i richiami fatti nel presente capitolo, l'esempio utilizzato in [30].

```
// Eating_Source (ED):

interface Fast-Food
( source semistructured Eating_Source )
{
  attribute string name;
  attribute Address address;
  attribute integer phone*;
  attribute set<string> specialty;
  attribute string category;
  attribute Restaurant nearby*;
  attribute integer midprice*;
  attribute Owner owner*;
};

interface Address
( source semistructured Eating_Source )
{
  attribute string city;
  attribute string street;
```

```
  attribute string zipcode;
};
  union
{ string; };

interface Owner
( source semistructured Eating_Source )
{
  attribute string name;
  attribute Address address;
  attribute string job;
};

// Food_Guide_Source (FD):

interface Restaurant
( source relational Food_Guide
  key r_code
  foreign_key(pers_id) references Person )
{
  attribute string r_code;
  attribute string name;
  attribute string category;
  attribute string street;
  attribute string zip_code;
  attribute integer pers_id;
  attribute integer tourist_menu_price;
  attribute string special_dish;
};

interface Person
( source relational Food_Guide
  key pers_id)
{
  attribute integer pers_id;
  attribute string first_name;
  attribute string last_name;
  attribute integer qualification;
};

interface Bistro
```

```
( source relational Food_Guide
  key r_code
  foreign_key(r_code) references Restaurant,
  foreign_key(pers_id) references Person)
{
  attribute string r_code;
  attribute set<string> type;
  attribute integer pers_id;
};

interface Brasserie
( source relational Food_Guide
  key b_code )
{
  attribute string b_code;
  attribute string name;
  attribute string address;
};
```

3.7 L'Object Model

L'intero modello ad oggetti, così come è stato costrutito ed utilizzato dal parser ODL_{I^3} , è visualizzabile nella figura 3.14. Dettagli dei tale modello sono stato via via illustrati nei paragrafi precedenti che focalizzavano le diverse sezioni di cui si compone.

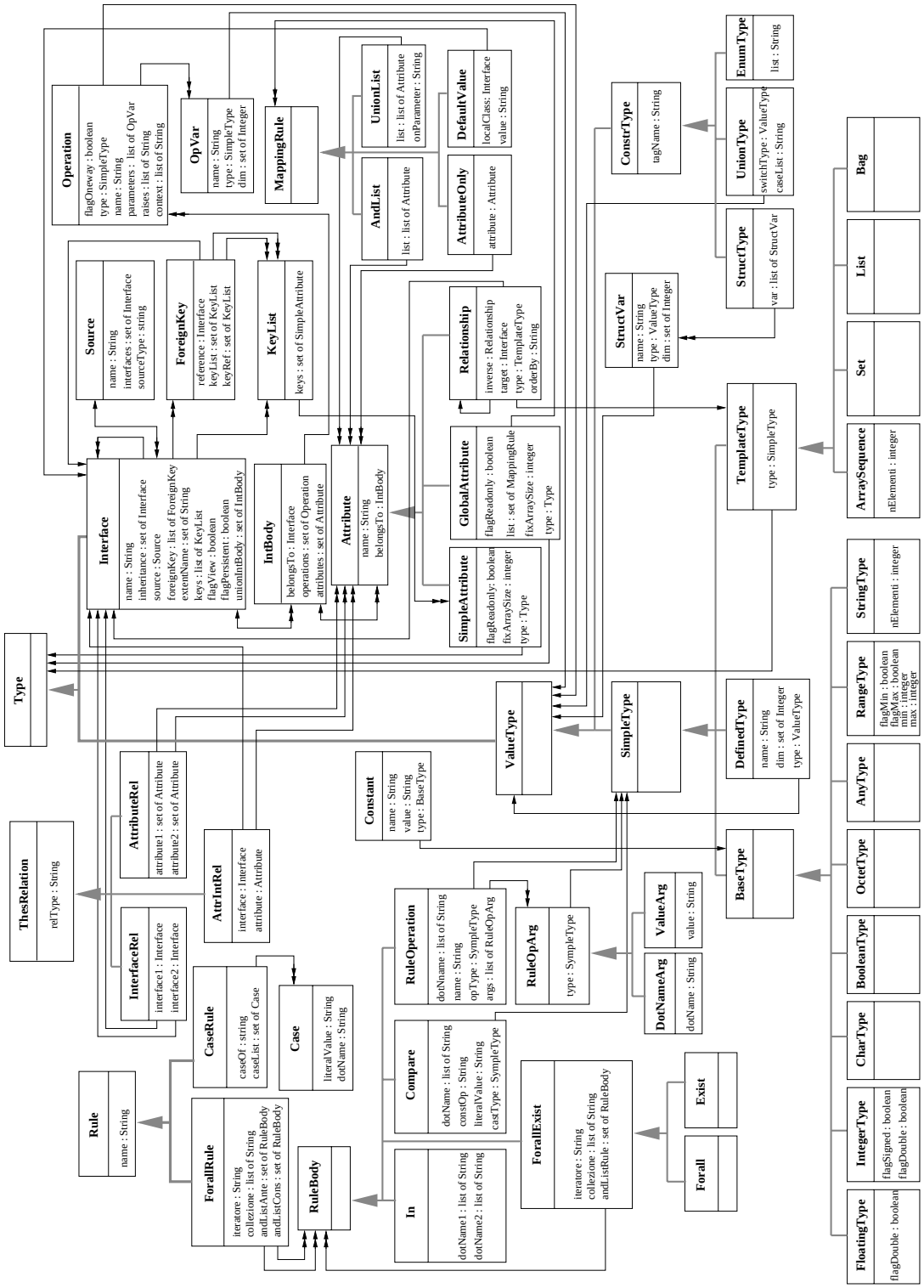

Figura 3.14: L'object model del linguaggio ODL$_{I^3}$

Capitolo 4

Il linguaggio XML

4.1 Introduzione

L'e**X**tensible **M**arkup **L**anguage, abbreviato XML [2], nasce nel 1996 da un gruppo di lavoro costituitosi sotto gli auspici del World Wide Web Consortium (W3C) denominato XML Working Group, anche se originariamente era noto con il nome di SGML Editorial Review Board. Tale nome nasce dal fatto che l'XML deriva direttamente dal linguaggio SGML (pubblicato come ISO8879 [31]), già da tempo conosciuto ed utilizzato in diverse realtà aziendali. L'SGML nasce nel 1986 con il preciso scopo di di poter creare dei documenti strutturati che avessero un formato aperto e stadardizzato oltre all'enorme vantaggio di consentire lo scambio e la manipolazione di dati indipendentemente dalla piattaforma su cui questi fossero stati utilizzati.

L'XML è dunque un linguaggio di Markup dell'ultima generazione, ma prima di tutto occorre definire cosa si intende per lingaggio di Markup e farne una breve panoramica.

4.1.1 I linguaggi di Markup

La produzione di testi utilizzabile su una qualsiasi macchina e comprensibile con il linguaggio umano passa attraverso una serie di filtri che possono essere di due tipi diversi: derivanti da ambienti di Word processing e derivanti da linguaggi di Markup.

Tralasciamo il discorso dei vari Word processing in quanto non sono rilevanti per gli scopi su cui pernea il presente lavoro. Analizzando, invece, i

linguaggi di markup evidenziamo che essi descrivono i meccanismi di strutturazione e di rappresentazione del testo attraverso una sintassi vera e propria che utilizza delle convenzioni spesso standardizzate e quindi utilizzabili su più sistemi e deriva concettualmente dai sistemi usati nelle tipografie per "marcare" le porzioni di testo indicandone le caratteristiche che questo deve assumere.

I linguaggi di markup possono ulteriormente essere distinti in due gruppi, a seconda del meccanismo usato per la formattazione del testo, ovvero per la sua rappresentazione:

- linguaggi di markup di tipo procedurale
- linguaggi di markup di tipo descrittivo

Appartengono al primo gruppo linguaggi quali *Script, TROFF* e *TEX* in cui occorre indicare le istruzioni per la visualizzazione della porzione di testo referenziata, mentre appartengono al secondo gruppo i linguaggi quali *SGML, HTML* e lo stesso *XML* che spostano l'attenzione sui contenuti dei testi lasciando la scelta del tipo di rappresentazione direttamente al software che si occuperà di leggerli.

Il passo che porta dai linguaggi di Markup all'arrivo della diffusione dei testi tramite il Web è relativamente breve e passa dai laboratri del CERN, in Svizzera, all'interno del quale vi lavorava alla fine degli anni ottanta Tim Berners-Lee il quale, partendo dal DTD del linguaggio SGML ed aggiungendovi un meccanismo di link, costruisce un'applicazione ipertestuale che prenderà il nome di HTML e che conoscerà una fortuna incredibile con l'esplosione di Internet.

4.1.2 Dai limiti dell'HTML alla nascita dell'XML

Il successo ottenuto dall'HTML [32] è sotto gli occhi di tutti e noto ai più. Questo successo è stato originato anche alla semplicità d'uso per la creazione di pagine ipertestuali che ha consentito a tantissimi utenti di riversare un'imponente quantità di dati su tutti i server dislocati oramai in ogni angolo della terra.

Ma questo enorme diffondersi di informazioni ha portato alla luce anche il problema della reperibilità dei dati stessi oltre al problema noto con il nome di "dead link", cioè la non corrispondenza di un link con un oggetto che, per

qualche motivo, è stato cancellato oppure semplicemente spostato in un'altro path. Quale fruitore di Internet non si è mai trovato ad usare un motore di ricerca ed ad ottenere come risposta una quantità infinita di link nei quali è necessaria una ulteriore ricerca manuale di ciò di cui si ha realmente necessità di reperire?

Tutto questo è dovuto anche al fatto che l'evoluzione di HTML ha portato oggi a creare una certa confusione tra "tag di struttura" e "tag di rappresentazione". Solo ultimamente il W3C ha rilasciato nelle specifiche la possibilità di utilizzare i fogli di stile (Cascading Style Sheet - CSS [33]) che hanno l'obiettivo di riportare all'esterno del documento i tag di rappresentazione e consentire al documento HTML di ritrovare, almeno in parte, la separazione originaria tra rappresentazione e struttura.

Oltre a queste limitazioni occorre considerare anche l'enorme sviluppo dei servizi che la Rete è in grado di offrire e che si sono scontrati con la rigidità del linguaggio HTML, il quale ricordo era nato semplicemente per un publishing elementare. Questa inadeguatezza ha portato alla nascita di miriadi di applicazioni accessorie per il Web (ricordiamo tra queste Shockwave o Acrobat reader), oltre che a linguaggi che fossero in grado garantire una certa dinamicità dei siti (come può essere il JavaScript o al limite anche gli stessi applet Java). Molte di queste applicazioni, inoltre, non sono indipendenti dalla piattaforma su cui girano obbligando l'utente a procurarsi continuamente la versione corretta ed aggiornata a seconda della piattaforma stessa su cui lavorano.

Non ultimo, bisogna considerare che per la creazione di siti dinamici ed in grado di interagire con una base di dati sia divenuto necessario avvalersi di tecnologie esterne alle specifiche HTML come il caso delle CGI o addirittura di Java.

Rispetto a questo panorama attuale il W3C ha deciso di dare una sterzata verso il ritorno ad una separazione più puntuale tra struttura e rappresentazione. Considerato che i metalinguaggi di markup di tipo descrittivo offrono rispetto ad HTML, sostanzialmente tre vantaggi:

1. l'estensibilità che permette la definizione di set personalizzati di tag

2. la salvaguardia degli elementi strutturali definiti in un file esterno chiamato *Document Type Definition* (DTD)

3. la validazione per la quale ogni documento passa attraverso un controllo che ne attesta la conformità alle regole definite nel DTD

si è pensato di introdurre un nuovo linguaggio per il Web.

L'SGML possedeva già questi vantaggi, ma fu considerato inadeguato in un ambiente diffuso come il Web a causa della sua eccessiva complessità. Quindi si costituisce nel 1996, sotto gli auspici del W3C stesso, l'XML Working Group (originariamente noto come SGML Editorial Review Board) presieduto da Jon Bosak della Sun Microsystems e con la partecipazione attiva dell'XML Special Interest Group (precedentemente noto come SGML Working Group) anch'esso organizzato dal W3C.

L'XML Working Group aveva come obiettivo la definizione di un linguaggio che mantenesse tutte le caratteristiche di un metalinguaggio di tipo descrittivo, ma che possedesse anche una certa facilità di utilizzo. Come risultato del lavoro del Working Group sono state rilasciate nel febbraio 1998 le specifiche della versione 1.0 del linguaggio "eXtensible Markup Language", a tutt'oggi non sostiutite con una nuova versione.

Più che un linguaggio vero e proprio XML può essere considerato come un metalinguaggio che quindi è in grado di creare altri linguaggi o applicazioni, un poco come è successo già con Resource Description Framework (RDF) e Channel Description Format (CDF) attualmente presenti sul Web.

Siccome XML si occupa esclusivamente della definizione dei tag da usare e dalla loro strutturazione essa ha il potenziale sia per permettere di poter utilizzare la sorgente dati con un qualsiasi strumento di visualizzazione a video, quale può essere un browser internet, sia come formato audio per un cellulare passando per una qualsiasi altra forma anche non prevista la momento della stesura dei dati stessi. XML quindi, pur essendo nato propriamente per il mondo Web, ha senso anche fuori da questo contesto, comunque e dovunque qualcuno voglia produrre un documento, a prescindere dal mezzo trasmissivo.

4.1.3 Obiettivi che si prefigge

Come già ripetuto in precedenza l'obiettivo che si prefigge l'XML è quello di riportare la separazione tra struttura e rappresentazione, ma come si legge nella W3C Raccomandation gli obiettivi progettuali di XML sono:

1. XML deve essere utilizzabile in modo semplice su Internet

2. XML deve supportare un gran numero di applicazioni
3. XML deve essere compatibile con SGML
4. Deve essere facile lo sviluppo di programmi che elaborino documenti XML
5. Il numero di caratteristiche opzionali deve essere mantenuto al minimo possibile, idealmente a zero
6. I documenti XML dovrebbero essere leggibili da un uomo e ragionevolmente chiari
7. La progettazione XML dovrebbe essere rapida
8. La progettazione XML deve essere formale e concisa
9. I documenti XML devono essere facili da creare
10. Non è di nessuna importanza l'economicità nel markup XML

4.1.4 La struttura del linguaggio XML

Il W3C ha voluto mantenere per questo progetto una distinzione tra le specifiche, quindi oltre a quelle riguardanti il linguaggio abbiamo, anche se attualmente sarebbe meglio dire che sono previste, le definizioni per quanto riguarda la rappresesentazione dei dati denominata *eXtensible Style Language (XSL)* [34] e quelle per la gestione dei link denominata *XML Linking Language (XLink)* [35].

Un qualsiasi strumento che deve visualizzare dei dati salvati secondo tali specifiche dovrebbe essere composto di due parti:

- Il Parser
- Il Processor

Il **Parser** si occupa del controllo sintattico e gestisce gli eventuali errori riscontrati, mentre il **Processor** visualizza il documento utilizzando un'altro file nel quale è stata definita la formattazione dei vari tag.

Da questo modo di operare si intravede la separazione tra struttura e rappresentazione. Come si è visto questo è uno degli aspetti chiave della buona costruzione di un ipertesto (tra l'altro garantita anche dalla separazione in

due linguaggi: XML ed XSL).

Il Parser, dal canto suo, effettua un controllo su due livelli, e precisamente valuta prima di tutto la conformità del documento al DTD di riferimento e, in caso di non conformità, esegue un successivo controllo relativo alle regole generali della sintassi XML.

Questo doppio controllo consente di dividere i documenti in due tipi, e precismente:

- docuemnti **Valid**
- documenti **Well-Formed.**

I primi sono quelli che passano il controllo al primo livello di conformità e quindi sono conformi al proprio specifico DTD, diversamente appartengono al secondo tipo se passano il secondo controllo effettuato dal Parser.

4.2 Terminologia

Riportiamo qui un breve glossario delle parole chiave di XML, come riportato dalle specifiche del W3C [3], che dovrebbero descrivere le azioni di un processore XML.

May-Documenti conformi e processori XML possono aderire al comportamento descritto ma la cosa non è obbligatoria.

Must-Documenti conformi e processori XML devono aderire al comportamento descritto, diversamente sono in errore.

Error-Si intende una violazione delle regole inserite nelle specifiche, i risultati ad un errore sono imprevedibili. Un software conforme può rilevare l'errore e riportarlo all'applicazione e può recuperare da esso.

Fatal error-Si tratta di un errore che il processore XML conforme deve rilevare e riportare all'applicazione. Una volta accertato l'errore può continuare l'elaborazione per cercare ulteriori errori da passare all'applicazione. Cosa che non deve invece fare è il proseguio della normale elaborazione, ma può passare i dati non elaborati all'applicazione in modo che possa essere

d'ausilio alla correzione degli errori stessi.

At user option-Un software conforme può o deve (a seconda del contesto della frase) comportarsi come descritto; deve però essere possibile per l'utente abilitare o disabilitare tale funzionalità.

Validity constraint-È una regola che si applica a tutti i documenti XML validi. La violazione di questo vincolo di validità è da considerarsi come un errore e devono, a discrezione dell'utente, essere segnalati dai processori XML validanti.

Well-formedness constraint-È una regola che deve essere applicata a tutti i documenti XML "Well-formed". La violazione di tale vincolo deve essere considerato come un errore fatale.

Match-(*Di stringhe o nomi*): Due stringhe o nomi che sono confrontati devono essere identici. I caratteri che possono essere rappresentati in diversi modi sono identici solamente se hanno la medesima rappresentazione in entrambe le stringhe; a scelta dell'utente tali caratteri possono essere rappresentati in un'unica forma canonica. Nessun "case folding" è eseguito. (*Di stringhe e regole della grammatica*): Una stringa corrisponde ad una produzione grammaticale se appartiene al linguaggio generato da quella produzione. (*Di contenuto e modelli di contenuto*): Un elemento corrisponde con la sua dichiarazione quando si conforma nel modo descritto nel vincolo di elemento valido.

For compatibility-È una caratterisitica di XML introdotto per mantenere la compatibilità con SGML.

For interoperability-Una raccomandazione non vincolante inclusa per aumentare le possibilità che i documenti XML possano essere elaborati dalla base di processori SGML installati ed esistenti prima dell'annesso a ISO 8879 sulle modifiche del WebSGML (WebSGML Adaptations Annex to ISO 8879).

4.3 I documenti XML

In questo capitolo viene dato un cenno alle specifiche che il W3C ha pubblicato riguardo all'XML e quindi le basi necessarie per la conoscenza di questo linguaggio. Per quanto riguarda l'intero documento pubblicato si rimanda all'appendice A in cui è riportata la versione del 10 febbraio 1998 (l'ultima

disponibile al momento della scrittura di questo testo), oppure ai riferimenti Internet del W3C in cui si possono trovare in vari formati e che sono:

- http://www.w3.org/TR/1998/REC- xml-19980210
- http://www.w3.org/TR/1998/ REC-xml-19980210.xml
- http://www.w3.org/TR/1998 /REC-xml-19980210.html
- http://www.w3.org/TR/1998/ REC-xml-19980210.pdf
- http://www.w3.org/TR/1998/REC-xml-19980210.ps

Ai fini di una migliore comprensione del filo conduttore di questo linguaggio riporto la frase iniziale della raccomandazione del W3C:

"L'eXtensible Markup Language (XML) è un subset di SGML [...]. *Il suo scopo è quello di abilitare un generico documento SGML ad essere servito, ricevuto e processato sul Web nello stesso modo in cui è attualmente possibile fare con HTML. XML è stato progettato per una facile implementazione e per l'interoperabilità sia con SGML che con HTML".*

Come già detto il linguaggio XML permette di inserire due tipi di documenti che sono i documenti *Well-Formed* ed i documenti *Valid*. Mentre i primi sono dei documenti in cui vengono semplicemente rispettate le generiche specifiche XML (presenza dei tag di chiusura a meno di empty element comunque vincolanti, differenza tra maiuscolo e minuscolo e controllo dell'indentamento dei tag), i documenti validi devono, invece, possedere obbligatoriamente una sezione denominata DTD in cui viene definita la struttura dei dati e che deve essere chiaramente rispettata.

Il documento è fisicamente composto da entità che a loro volta possono indirizzare altre entità fino a formare una sorta di albero che come radice ha un "document entity", mentre logicamente il documento è composto di dichiarazioni, elementi, commenti, riferimenti a caratteri, e istruzioni di elaborazione, ciascuno dei quali è indicato nel documento da espliciti markup.

4.3.1 I documenti Well-Formed

Perchè un documento sia "Well-formed" occorre che assuma la forma:

```
document ::= prolog element Misc
```

4.3.2 Il prologo

Per prologo, appunto l'elemento definito `prolog`, si definisce la prima parte dei documenti XML che generalmente potrebbero, anche se sarebbe meglio dire che dovrebbero, iniziare con la dichiarazione XML la quale specifica la versione utilizzata ed altre informazioni sul documento stesso che verranno illustrate più avanti. Un tipico esempio di questa definizione può essera data dalla seguente riga:

```
<?xml version="1.0"?>
```

in cui viene indicato il numero di versione "1.0" (per ora l'unica, anche se il W3C non esclude il rilascio di altre versioni) delle specifiche e che di conseguenza dovranno essere rispettate nel documento che segue.

Deve essere considerato un errore la scrittura di un documento che riporti un determinato numero di versione nel prologo mentre i dati non sono conformi con il resto del documento stesso.

I caratteri che si possono utilizzare, sia per i "markup" che per i "character data" sono quelli specificati da ISO/IEC 10646 che comprende anche i caratteri *tab*, *carriage return*, *line feed* ed i caratteri grafici definiti nelle specifiche stesse. I processori XML devono quindi accettare le codifiche UTF-8 ed UTF-16 del 10646, anche se sarebbe auspicabile, data la varietà di codifche attualmente presenti, che siano in grado di utilizzare anche altri tipi di specifiche. Nel caso però venga passato al processore un documento con specifiche non riconosciute, deve essere generato un errore fatale.

Il tipo di codifica utilizzato dal documento deve essere specificato anch'esso nel prologo e può assumere la forma:

```
<?xml encoding='UTF-8'?>
```

che definisce appunto l'uso della UTF-8.

Altra informazione che deve essere inclusa nel prologo riguarda la dichiarazione di documento "stand-alone". Questa dichiarazione informa il processore XML se il documento che sta esaminando è indipendente oppure esistono, o si prevede la possibilità che esistano, delle dichiarazioni di markup esterne. Nel primo caso dovrà assumere il valore `yes`, diversamente assumerà il valore `no` che è anche il valore assunto di default nel caso non venga indicato niente

da parte dell'utente.

Un banale esempio è dato dalla seguente riga di codice:

```
<?xml version="1.0" standalone='yes'?>
```

4.3.3 Markup e Dati

I Markup sono appunti i marcatori dei dati che posso inserire ed assumono la forma di tag di inizio, tag di fine, tag di elemento vuoto, riferimenti a entità, riferimenti a carattere, commenti, delimitatori di sezioni CDATA, dichiarazione di document type, e istruzioni di elaborazione (processing instructions che consentono ai documenti di contenere delle istruzioni per le applicazioni). Il resto del documento XML rappresenta i dati veri o propri.

La definizione di **Commento** può essere intesa come tutto il testo racchiuso tra i delimitatori "<! − −" e "−− >" e che quindi assume una forma simile alla seguente:

```
<!-- dichiarazione per <head> & <body> -->
```

Un appunto che occorre fare in questo punto riguarda il carattere ":" (due punti) che attualmente viene riservato, se contenuto nei nomi di XML, per la definizione e l'utilizzo dei "namespaces" con consistono, come illustrato con maggiore dettaglio in 4.5.5, in una estensione della definizione per gli elementi e gli attributi.

4.4 Il Document Type Declaration - DTD

Il DTD contiene tutte quelle regole che servono per la definizione della struttura di un documento. Come abbiamo già detto la sua presenza non è obbligatoria, ma per chiarezza nella struttura sarebbe consigliabile includerlo sempre. Chiaramente nel momento in cui si intenda generare un documento "Valido" è necessario che questo rispetti le regole definite nel DTD.

XML prevede due possibilità di definizione di un DTD che portano al medesimo risultato: definire la struttura del documento internamente al file dei dati oppure creare in un file esterno la struttura ed utilizzare un costrutto apposito per l'inclusione. In entrambe i casi la sintassi della dichiarazione

rimane identica.

La definizione della struttura del mio documento passa attraverso le dichiarazioni di markup che possono essere di diversi tipi a seconda di cosa si vuole evidenziare. In particolare vi sono le dichiarazioni di elemento, le dichiarazioni di lista di attributi, le dichiarazione di entità e le dichiarazioni di notazione.

4.4.1 Dichiarazioni di elemento

Nel caso di dichiarazione di un elemento, l'unità elementare di un documento XML, la sintassi da rispettare è data da:

```
<!ELEMENT [nome_elemento] ([elenco_sottoelementi])>
```

dove `elenco_sottoelementi` indica cosa dovrebbe essere incluso all'interno di quel determinato elemento. I valori ammessi sono `#PCDATA` nel caso siano contenuti delle stringhe di caratteri, `ANY` nel caso i dati contenuti possono essere di qualsiasi tipo, una lista chiusa degli elementi possibili oppure `EMPTY` nel caso l'elemento non contenga niente. Quest'ultimo caso introduce la definizione di "elemento vuoto" che consiste in un elemento che non contiene dati, che non necessita di un marcatore di chiusura ma che può assumere solamente degli attributi (per fare il parallelo con HTML si può paragonare al tag <IMG>).

Esempi di possibili dichiarazioni di elementi sono date da:

```
<!ELEMENT br EMPTY>
<!ELEMENT p (#PCDATA|emph)* >
<!ELEMENT %name.para; %content.para; >
<!ELEMENT container ANY>
```

4.4.2 Dichiarazione di lista di attributi

Tramite la lista di attributi XML da' la possibilità all'utente di aggiungere delle informazioni addizionali agli elementi che si sono definiti. Già HTML prevedeva una possibilità simile, ad esempio, per quanto riguardava gli allineamenti dei tag (es. `align=''center''`).

In realtà mentre per quanto riguarda l'HTML tali informazioni indicavano principalmente delle reppresentazioni dei tag stessi, con XML ciò non

è più vero in quanto riportano delle vere e proprie informazioni addizionali sulla struttura del documento.

Facendo l'esempio di una libreria potrebbe essere importante distinguere se una entità libro sia nell'edizione lusso oppure nell'edizione economica e quindi potrebbe diventare necessario creare un attributo `EDIZIONE` per permettere di restringere la scelta tra "`libro_lusso`" e "`libro_economico`".

Gli attributi vengono dichiarati nell'"attribute list" (<ATTLIST>) che contiene il nome dell'elemento cui gli attributi si riferiscono, il tipo di dati, la lista dei valori degli attributi stessi e il valore di default. La sintassi di tale comando è data da:

```
<!ATTLIST nome_elemento definizione_attributo>
```

Il `nome_elemento` è ovviamente il nome dell'elemento di cui si vuole descrivere la lsita degli attributi, e può anche non essere stato dichiarato, spetterà al processore XML, eventualmente, provvedere ad una segnalazione all'utente. La `definizione_attributo` può essere di tre tipi, ma tutti devono avere la seguente sintassi:

```
Nome Tipo Dichiarazione_default
```

dove il `Nome` è il nome dell'attributo, il `Tipo` reppresenta il tipo di attributo e può assumere 3 differenti tipologie che sono: stringa (`CDATA`), token (`ID | IDREF | IDREFS | ENTITY | ENTITIES | NMTOKEN | NMTOKENS`) oppure enumerativo. Il significato di tali parole chiave è dato da:

- `ID` si tratta di un nome e deve essere unico all'interno dello stesso documento perché identifica un elemento univocamente; di conseguenza ogni elemento può avere un solo attributo di questo tipo.
- `IDREF` è un riferimento ad un elemento che ha un attributo di tipo ID con valore IDREF.
- `IDREFS` è una lista di IDREF separati da uno spazio bianco.
- `ENTITY` indica il nome di una entità dichiarata nel documento.
- `ENTITIES` è una lista di ENTITY separati da uno spazio bianco.
- `NMTOKEN` è una stringa composta da una sola parola.
- `NMTOKENS` indica una lista di NMTOKEN separati da uno spazio bianco.

Ultimo parametro è la `Dichiarazione_default` che indica il valore che deve assumere per default l'attributo e può essere preceduto dai valori `#REQUIRED` per far si che l'attributo abbia necessariamente un valore ogni volta che è usato l'elemento, `#IMPLIED` nel caso che l'attributo possa non assumere alcun valore, `#FIXED` per permettere all'attributo di non avere alcun valore ma nel caso ce l'abbia deve necessariamente essere quello di default.

Tornando al caso della libreria e del tipo di edizione, nel DTD, dopo la dichiarazione dell'elemento LIBRO si avrà un'"attribute list" con la seguente sintassi:

```
<!ATTLIST LIBRO EDIZIONE (economica|lusso) "economica">
```

Altri esempi di definizione di attributi sono dati da:

```
<!ATTLIST termdef
          id      ID      #REQUIRED
          name    CDATA   #IMPLIED>
<!ATTLIST list
          type    (bullets|ordered|glossary)  "ordered">
<!ATTLIST form
          method  CDATA   #FIXED "POST">
```

4.4.3 Dichiarazioni di entità

Per fare riferimento a file esterni oppure a porzioni di dati diverse l'XML mette a disposizione delle "entities", ovvero delle entità le quali possono contenere sia dei dati convalidabili, cioè formate da markup e tutto quanto è definito dai markup, sia da dati non convalidabili.

La caratteristica delle entità implica una suddivisione in entità interne ed entità esterne a seconda di dove si trovi fisicamente la porzione di dati da includere. La sintassi per la definzione di una entità interna è:

```
<!ENTITY [%] Nome Definizione_in>
```

In questo caso non ci sono elementi di memoria esterni e la dichiarazione definisce il contenuto della dichiarazione stessa.

Allo stesso modo la sintassi:

```
<!ENTITY [%] Nome Definizione_out>
```

viene applicata per le entità esterne, che potrebbero anche fare riferimento a dati presenti su altre macchine. In questo caso è possibile anche definire il tipo del dato che verrà incluso, di conseguenza possiamo estendere la sintassi come:

```
Definzione_out ::= Riferimento | Riferimento Data_decl
```

Il `Riferimento` può essere un identificatore di sistema, quando preceduto dalla parola chiave `SYSTEM`, oppure un identificatore pubblico, qundo preceduto dalla parola chiave `PUBLIC` seguita sia dall'indirizzo pubblico che da quello di sistema. Nel caso di identificatori pubblici il processore XML può cercare di generare un URI alternativo rispetto a quello di sistema in modo da risolvere il riferimento.

Ultima nota riguarda la dichiarazione del tipo del dato, il parametro `Data_decl`, che è una stringa identificate il tipo di dato del riferimento esterno ed è preceduta dalla parola chiave `NDATA`.

Un esempio di dichiarazione interna può essere:

```
<!ENTITY decameron "G.Boccaccio, Il Decamerone, Giunti, Firenze, 1987">
```

Mentre alcuni esempi di dichiarazioni esterne possono essere:

```
<!ENTITY open-hatch
         SYSTEM "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY open-hatch
         PUBLIC "-//Textuality//TEXT Standard open-hatch boierplate//EN"
         "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY hatch-pic
         SYSTEM "../grafix/OpenHatch.gif"
         NDATA gif >
```

4.4.4 Dichiarazione di notazione

Le dichiarazioni di notazioni attribuiscono un nome ad una notazione utilizzata nel DTD per le dichiarazioni di entità oppure nelle specifiche degli attributi, oltre a fornire al processore XML o ad un'applicazione client l'indirizzo di una applicazione esterna che possa fornire un aiuto nella elaborazione dei dati nella notazione data. In sostanza quando ci si riferisce a dei dati esterni è anche necesssario prevedere una dichiarazione di notazione che identifichi il tipo di dati a cui si fa' riferimento.

La sintassi è data da:

```
<!NOTATION Nome (ID_esterno | ID_pubblico)>
```

Nel caso di `ID_esterno` occorre definire il riferimento dell'applicazione di ausilio con la stessa sintassi vista per il `Riferimento` illustrato nella dichiarazione delle entitè esterne, mentre nel caso di `id_pubblico` sarà necessario far precedere l'indirizzo dalla parola chiave `PUBLIC`.

Un esempio è dato dalla seguente dichiarazione del formato GIF per le immagini:

```
<!NOTATION GIF SYSTEM "GIF">
```

4.5 L'inserimento dei dati

Come già più volte detto in questo capitolo i documenti possono essere "Valid" oppure "Well-formed" a seconda della presenza o meno del Document Type Declaration. Chiaramente la presenza di un DTD, esterno o interno che sia, con la relativa struttura comporta che i documenti siano considerati validi dal processore XML solamente se viene rispettata la struttura stessa.

Il documento che prevede la presenza di una struttura definita appunto nel DTD deve contenere nel prologo il tag `DOCTYPE` che può assumero due diverse sintassi a seconda che la struttura sia interna oppure occorra includere un file esterno. Nel primo caso l'intera dichiarazione dovrà essere racchiusa tra '`[`' e '`]`'.

Per illustrarne il funzionamento partiamo da un semplice documento formato da un solo elemento denominato "titolo" la cui linea di codice per la definizione della struttura è sempre data da:

```
<!ELEMENT titolo (#PCDATA)>
```

Nel caso tale documento debba fare riferimento ad un file esterno, denominato esempio1.dtd e contente la struttura appena vista, assumerebbe la forma:

```
<?xml version="1.0"?>
<!DOCTYPE esempio1 SYSTEM "esempio1.dtd">
```

mentre nel caso avessi deciso di mantenere la struttura all'interno avrei avuto la seguente sintassi:

```
<?xml version="1.0"?>
<!DOCTYPE esempio1 [
  <!ELEMENT titolo (#PCDATA)>
]>
```

Per quanto riguarda la creazione di un DTD si rimanda al paragrafo 4.4 in cui sono ampliamente illustrate tutte le dichiarazioni previste.

Vediamo ora la descrizione di come devono essere formati i dati di un documento XML affinchè un processore XML possa processarli correttamente senza generare errori.

4.5.1 Gli elementi

Gli elementi sono la struttura base di ogni documento XML e devono essere racchiusi tra un tag di inizio ed un tag di fine, oppure possono essere degli elementi vuoti e di conseguenza conterrano solamente un tag di elemento vuoto.

I tag di inizio e fine sono simili a quanto era previsto per l'HTML, cioè il nome dell'elemento dovrà essere racchiuso tra parentesi angolari ("<", ">") mentre il tag di chiusura dovrà essere preceduto dal carattere "/". Chiaramente nel tag di inizio e nel tag di fine gli identificatori dell'elemento dovranno essere gli stessi, anche per quanto riguarda i caratteri maiuscoli e minuscoli, in quanto si tratta di nomi case sensitive. Chiaramente nel caso l'elemento abbia una sua dichiarazione nel DTD il nome dell'elemento dovrà essere identico a quanto dichiarato, oltre a mentenere lo stesso formato dei dati, se specificato.

Esempi di elemento sono dati da:

```
<saluti>Hello, World!</saluti>
<nome>Andrea Cataldo</nome>
```

Nel caso si debba definire un elemento vuoto, eventualmente perché nella dichiarazione sia stato definito `EMPTY`, è possibile avere i tag di inizio e fine senza nulla a separarli, ma le specifiche prevedono anche un tag di elemento vuoto racchiuso anch'esso tra parentesi angolari, ma conente prima della chiusura il carattere "/".

Esempi sono dati da:

```
<IMG src="http://www.w3.org/Icons/WWW/w3c_home />
<br></br>
<br/>
```

4.5.2 Gli attributi

Come già detto gli attributi danno la possibilità di aggiungere delle informazioni agli elementi in modo da poter, eventualmente, creare delle ulteriori suddivisioni nell'elemento stesso.

La sintassi prevede di inserire nel tag di inizio il nome dell'attributo stesso seguito dal segno = e dal valore da assegnare all'attributo racchiuso tra doppi apici (").

Riprendendo l'esempio della libreria utilizzato nel paragrafo 4.4.2 possiamo completarlo con la seguente riga:

```
<LIBRO EDIZIONE="economica">La Divina Commedia</LIBRO>
```

Chiaramente nel caso esista una dichiarazione nel DTD e siano stati forniti per l'attributo dei tipi particolari e dei valori di default, tale specifiche dovranno essere rispettate. Per il significato di tipi di attributo e delle parole chiave riguardanti il valore di default si rimanda al paragrafo 4.4.2.

4.5.3 Le entità

Nel caso il mio documento XML debba essere composto, per qualsiasi ragione, da diversi pezzi, dobbiamo introdurre il concetto di entità. Per entità si intende quindi qualsiasi porzione di dato che può essere convalidabile (cioè un insieme di markup o comunque qualsiasi cosa che possa essere definita tramite markup) oppure non convalidabile (cioè a cui non è possibile applicare il concetto di markup).

A differenza di quanto visto per gli elementi e gli attributi le entità necessitano di un'apposita dichiarazione nel DTD sia che contengano dei riferimenti interni al file, sia che contengano i riferimenti ad altri file all'esterno del documento stesso. L'utilizzo di una entità avviene semplicemente attraverso il suo nome.

Come abbiamo accenato nel paragrafo 4.4.3 esistono tre tipi di entità e rispettivamente le **entità interne**, **entità esterne** ed **entità parametriche**.

Entità interne

Le entità interne sono contenute direttamente dentro il documento e dichiarate nel relativo DTD e generalemente sono utilizzate per rappresentare dei caratteri speciali oppure delle scorciatoie per parole o frasi lunghe utilizzate spesso all'interno del documento stesso.

Un esempio può essere dato (dichiarazione nel DTD ed utilizzo nel documento) da:

```
<!ENTITY xml "eXtensible Markup Language">
[...]
&xml;
[...]
```

In XML sono state previste 5 entità predefinite e rispettivamente `<` per il carattere minore (<), `>` per il carattere maggiore (>), `&` per la e commerciale (&), `&quot;` per i doppi apici (") e `&apos;` per l'apostrofo (').

Entità esterne

Come già detto riguarda il referenziamento ad un file esterno. Nel caso si tratti di un testo questo viene rimpiazzato direttamente al posto dell'entità, mentre nel caso si tratti di un file binario il parser non deve preoccuparsi di interpretarlo, ma deve esclusivamente passare il contentuo all'applicazione che si occuperà della lettura.

Da ricordare che nel caso si tratti di file binario potrebbe essere dichiarata una notation (vedi paragrafao 4.4.4) che si occupa di indirizzare l'entità all'applicazione preposta per l'elaborazione.

Entità parametriche

Sono molto simili alle entità interne, ma la differenza consiste nel fatto che quest'ultime vengono semplicemente espanse e poi passate all'applicazione, mentre le entità parametriche vengono espanse e interpretate come parte del DTD.

Una entità parametrica viene dichiarata esclusivamente nel DTD, ma il simbolo che viene utilizzato è il caratere percentuale (/

Un esempio è dato da:

```
<!ENTITY % indir_corto
       '<!ELEMENT indirizzo (citta, provincia)>'>
<!ENTITY % indir_lungo
       '<!ELEMENT indirizzo (via, citta, cap, provincia?)>'>;
```

utilizzato come

```
<!ELEMENT contatto (nome, indirizzo)>
[...]
%indir_corto;
[...]
```

4.5.4 Le sezioni CDATA

Nel caso ci si trovi nella necessità di includere una sequenza di caratteri che verrebbe processata dal processore XML e che dovrebbe invece essere considerata come un dato, occorre racchiuderla in una **sezione CDATA** i cui delimitatori sono "<![CDATA[" e "]]>", ad esempio:

```
<![CDATA[<titolo>Il Wrapper per XML</titolo>]]>
```

4.5.5 I Namespaces

Questa sezione era riportata solamente in fase di sperimentazione al momento della emissione delle specifiche ufficiali [3] ed è stata integrata con un successivo documento datato 14 Gennaio 1999 [36]. Tale documento è stato redatto dai membri del W3C ed approvato dal direttore del W3C stesso e quindi deve essere considerato come materiale ufficiale definitivo e deve essere citato come normativa in riguardo.

Poiché XML si propone lo scopo di migliorare lo stato della riusibilità delle applicazioni Web attraverso un approccio modulare possiamo affermare che esiste un problema di riconoscimento e di eventuale collisione. In pratica il parser XML, ed anche il processore, deve essere in grado di gestire delle eventuali ambiguità che si possono verificare in un approccio modulare, per questo sono stati introdotti i Namescpaces che sono in grado di permettere l'utilizzo di tag che abbiano lo stesso nome, ma che si differenziano nel significato e quindi prevedono dei costrutti che evitino la generazione di ambiguità.

Un esempio potrebbe essere dato, nel caso di un documento che riporti gli articoli di una rivista, dall'elemento "Titolo" che potrebbe essere confuso

tra il titolo dell'articolo ed il titolo onorifico dell'autore dell'articolo stesso. In questo caso è ammessa la creazione di due tag con lo stesso nome evitando l'ambiguità associando delle URI ad elementi che fungono da ambiente. Per cui la seguente definizione:

```
<X xmlns:edi='http://ecommerce.org/schema'></X>
```

applica l'ambiente "edi", rappresentato dal comando "http://ecommerce.org/schema", all'elemento <X>.

La dichiarazione, come vista nell'esempio precedente, si applica sia all'elemento nel quale c'è la dichiarazione sia, a caduta, a tutti quelli che seguono nell'eventuale struttura che si viene a creare fino ad una eventuale nuova dichiarazione.

Altro caso si ritrova nell'esempio:

```
<bk:book xmlns:bk='urn:loc.gov:books'
         xmlns:isbn='urn:ISBN:0-395-36341-6'>
    <bk:title>Cheaper by the Dozen</bk:title>
    <isbn:number>1568491379</isbn:number>
</bk:book>
```

dove all'interno del contesto <bk:book> vi è una alternanza di due Namespaces "bk" e "isbn" che valgono fino ad una nuova dichiarazione.

4.6 Un esempio esplicativo

Per dare una dimostrazione pratica a quanto scritto fino ad ora come esempio di file XML la traduzione dal linguaggio ODL_{I^3} del database, o comunque di una parte di esso, illustarato nel documento [30] a cui sono stati aggiunti alcune particolarità per avere un esempio più completo da un ounto di vista didattico.

Il file DTD, in questo caso chiamato "eating_source.dtd" presenta le seguenti righe di codice:

```
<!ELEMENT Eating_Source (Fast-Food)*>

<!-- Dettaglio per Fast-Food -->
```

```
<!ELEMENT Fast-Food (name,address,phone?,speciality*,
                     category,nearby?,midprice?,owner?)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name id ID #REQUIRED>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT speciality (#PCDATA)>
<!ELEMENT category (#PCDATA)>
<!ELEMENT midprice (#PCDATA)>

<!-- Dettaglio per Address -->

<!ELEMENT address (#PCDATA|address_type)*>
<!ELEMENT address_type (city,street,zipcode)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>

<!-- Dettaglio per nearby (Restaurant) -->

<!ELEMENT nearby (name_rist,category,street,zip_code,
                  person,tourist_menu_price,special_dish)>
<!ELEMENT name_rist (#PCDATA)>
<!ATTLIST name_rist r_code ID #REQUIRED>
<!ELEMENT zip_code (#PCDATA)>
<!ELEMENT person (#PCDATA)>
<!ATTLIST person pers_id IDREF #IMPLIED>
<!ELEMENT tourist_menu_price (#PCDATA)>
<!ELEMENT special_dish (#PCDATA)>

<!-- Dettaglio per owner -->

<!ELEMENT owner (name,address,job)>
<!ELEMENT job (#PCDATA)>
```

Mentre il file contenente i dati, salvato con il nome "eating_source.xml" assume la seguente forma:

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE Eating_Source SYSTEM "Eating_Source.dtd">
```

```
<!-- Dati -->

<Eating_Source>
  <Fast-Food>
    <name id="a">McDocnalds</name>
    <address>via della Spiga, 23 - Milano</address>
    <phone>02-32457299</phone>
    <speciality>BigMac</speciality>
    <speciality>CheesBurger</speciality>
    <category>1</category>
    <owner>
      <name id="p1">Andrea Cataldo</name>
      <address>Guastalla</address>
      <job>NetManager</job>
    </owner>
  </Fast-Food>
  <Fast-Food>
    <name id="b">Burghy</name>
    <address>
      <address_type>
        <city>Milano</city>
        <street>Galleria centrale, 23</street>
        <zipcode>20100</zipcode>
      </address_type>
    </address>
    <phone>02-3999876</phone>
    <speciality>Burghy 1</speciality>
    <speciality>Burghy 2</speciality>
    <category>2</category>
    <nearby>
      <name_rist r_code="Ra">Ristorante Sclavini</name_rist>
      <category>1</category>
      <street>Galleria Centrale, 21</street>
      <zip_code>20100</zip_code>
      <person pers_id="p1"/>
      <tourist_menu_price>50000</tourist_menu_price>
      <special_dish>Cotoletta alla Milanese</special_dish>
    </nearby>
    <midprice>5000</midprice>
  </Fast-Food>
</Eating_Source>
```

4.7 XSL

Poiché lo scopo di questa tesi è, come già ampiamente spiegato nel capitolo introduttivo, quello di utilizzare le potenzialità strutturali che l'XML possiede per poterlo utilizzare come una base di dati (meglio ancora come OODB) da cui prelevare le informazioni da trasformare in formato ODL_{I^3} , tutto quanto riguarda la descrizione dei tag di rappresentazione assume una importanza minore e quindi saranno in questo paragrafo illustrati solamente i criteri guida, lasciando al lettore il compito dell'approfondimento.

Siccome le specifiche rilasciate dal W3C non davano nessun cenno di come deve essere gestita la rappresentazione dei dati si è pensato a quale potesse essere il modo migliore per la rappresentazione dei dati, considerando anche che tale rappresentazione dovesse mantenere anche l'assoluta indipendenza del dispositivo scelto per la visualizzazione. Questa "vacanza" di specifiche ufficiali a fatto si che si che venissero prese in considerazioni diverse ipotesi:

- Cascading Style Sheet (**CSS**)
- Document Style Semantics and Specification Language (**DSSSL**)
- Extensible Style Language (**XSL**)

Il **CSS** [33] è un linguaggio oramai noto agli sviluppatori di pagine Web e le cui specifiche sono state rilasciate dal W3C per poterle applicare all'HTML, ma ha il grosso limite di non riuscire a modificare radicalmente un documento e quindi rimane lontano dalla filosofia dei creatori di XML.

DSSSL [37] invece viene utilizzato per i documenti SGML e quindi potrebbe essere il candidato ufficiale, ma, come il linguaggio a cui viene applicato, mantiene anche un elevato grado di complessita che ricordiamo essere stato il punto focale per cui si è pensato di creare XML. Il passo successivo quindi potrebbe essere quello di prevedere una semplificazione anche di questo linguaggio.

Il linguaggio **XSL** [34], infine, si candida ad essere il linguaggio "proprietario" di XML proprio dalla sua nascita, ma attualmente non esistono delle specifiche chiare, bensì solamente delle bozze il che fanno si' che siano relativamente pochi i software che lo utilizzano. I concetti su cui nasce dovrebbero essere quelli di prevedere una semplicità d'uso caratteristica dei CSS, ma allo stesso tempo di mantenere la potenza garantita da DSSSSL.

Ognuna di queste tecniche rappresenta quindi dei vantaggi e degli svantaggi che non consentono di dire con sicurezza quale può essere la migliore. È ragionevole quindi supporre che prenderà piede la prima che riuscirà, per i motivi più svariati, a penetrare nella mentalità degli sviluppatori. Per dover di cronaca occorre però sottolineare che il linguaggio XSL, proprio perchè nato per questo preciso scopo, si candida a diventare la specifica ufficiale e quindi preferita dagli sviluppatori.

Attualmente il consorzio si sta' muovendo su tre livelli che sono lo sviluppo di XSL, il momentaneo mantenimento di una buona compatibilità con l'HTML e il mantenimento di uno stretto rapporto con DSSL.

Con un documento del 29 Giugno 1999 [38] si è iniziato a dare una certa normativa riguardante la rappresentazione definendo solamente come devone essere incluse le istruzioni all'interno di un documento, ed in particolare si è scelto di includere tra le istruzioni di processazione una istruzione particolare denominata *xml-stylesheet* e che rispecchi la seguente grammatica riportata in BNF:

```
StyleSheetPI ::= '<?xml-stylesheet' (S PseudoAtt)* S? '?>'
PseudoAtt ::= Name S? '=' S? PseudoAttValue
PseudoAttValue ::= ('"' ([^"<&] | CharRef | PredefEntityRef)* '"'
                   | "'" ([^'<&] | CharRef | PredefEntityRef)* "'")
                   - (Char* '?>' Char*)
PredefEntityRef ::= '&amp;' | '&lt;' | '&gt;' | '&quot;' | '&apos;'
```

4.7.1 XSL - Il prologo

Andiamo ora a descrivere come dovrebbe essere il linguaggio XSL. I suoi componenti principali sono:

- Construction Rules
- Style Rules
- Named Styles
- Macros
- Scripts

L'associazione tra un documento XML ed il relativo foglio di stile deve avvenire nel prologo del file XML stesso e deve rispettare la sintassi:

```
<?xml-stylesheet href type [title] [media] [charset]?>
```

Se volessimo inserire un foglio di stile nominato "style.xsl" dovremmo utilizzare la seguente sintassi:

```
<?xml-stylesheet href="stile.xsl" title="Compact" type="text/xsl"?>
```

Nel caso al documento XML si vogliano associare dei fogli di stile alternativi abbiamo a disposizione l'istruzione `xml:alternate-stylesheet` che mantiene la stessa sintassi illustrata precedentemente.

4.7.2 XSL - Construction Rules

L'otuput formattato viene creato in due passi distinti: prima di tutto viene letto l'intero documento e generata la struttura ad albero ed in seguito vengono attribuiti gli stili. L'associazione tra gli elementi e gli stili che li andranno a rappresentare è svolta dalle "Contruction Rules" che constano di due elementi logici: il Pattern che identifica gli elementi su cui applicare gli stili, e le Action che specifica l'albero da costruire quando il processor incontra quell'elemento o uno di quegli elementi.

XSL - I Pattern

Ogni Construction Rules contiene sempre almeno un pattern, cioè esiste sempre almeno una identificazione dell'elemento al quel deve essere applicato. L'identificazione avviene semplicamente come valore dell'attributo type del tag <`target-element`>, ad esempio:

```
<target-element type="titolo"/>
```

dove l'elemento `titolo` viene identificato come unico destinatario della rules.

In realtà il metodo di assegnazione della regola può anche essere più complesso a seconda dei rapporti di ascendenza o discendenza di un elemento con altri, tramite "wildcard" ed altri metodi ancora che tralasciamo.
Giusto per portare un semplice esempio le seguenti linee di codice:

```
<element type="capitolo">
<target-element type="titolo"/>
</element>
```

le quali applicano la construction rule solamente agli elementi `titolo` che hanno come elemento superiore l'elemento `capitolo`.

Tralascio eventuali dettagli sulle specifiche sia perchè non sono attulamente codificate da documenti ufficiali e sia perchè, come già detto, di interesse secondario ai fini di questa tesi.

XSL - Le Action

Una volta identificati gli elementi ai quali andranno applicate le construction rules diventa necessario specificare anche come dovrà essere generata la struttura del "Flow object", ossia la struttura del documento comprensiva delle informazioni per la formattazione.

Ogni Action comprende due tipi di elementi distinti: i **Flow object** derivati quasi direttamenta dal DSSSL e che indicano la formattazione, e gli **XSL processing element** che sono gli elementi di controllo sugli elementi identificati tramite i pattern.

Per quanto riguarda i Flow object possiamo dire che sono molto numerosi per dare una maggior possibilità di scelta di formattazione di un testo, tra i principali possiamo citare:

- <`paragraph`> che descrive un blocco di testo
- <`display-group`> che descrive un gruppo di blocchi
- <`table`> che descrive una tabella
- <`external-graphic`> che descrive un link ad un ogetto grafico
- <`link`> che descrive un link

Per quanto riguarda i processing element che contegono le informazioni sulla modalità di applicazione delle formattazione dei pattern possiamo indicare, tra gli elementi più comuni, le seguenti parole chiave:

- <`children/`> per processare tutti i figli dell'elemento
- <`select-element`> per processare un elemento
- <`literal`> per aggiungere testo
- <`eval`> per aggiungere il risultato di uno script EMACS

In conclusione possiamo dire che per poter definire un elemento `titolo` che abbia un determinato spazio sia prima che dopo il testo, il cui testo sia in grassetto e con una dimensione di 24 punti dovremmo inserire le seguenti linee di codice:

```
<titolo
space-before="12pt"
space-after="36pt"
font-weight="bold"
font-size="24pt">
<children/>
</titolo>
```

4.7.3 XSL - Style Rules

Per definire per un elemento più di una regola di formattazione dobbiamo avvalerci degll Style Rules, identificati dal tag <`style-rule`>, che sono composte anch'esse da pattern e da action, ma che non generano alcun Flow object ovvero si applicano in cascata su tutti gli elmenti figli di quello indicato.

Quindi, ad esempio, per poter attribuire il colore rosso a tutti gli elementi nel tag `titolo` senza doverlo definire anche nelle successive construction rules dovrò inserire le seguenti linee di codice:

```
<style-rule>
<target-element type="titolo"/>
<apply color="=red"/>
</style-rule>
```

4.7.4 XSL - Named Style, Macros e Scripts

È possibile raggruppare anche diverse regole sotto un unico nome, i **Named Style** appunto, che possono essere richiamate tutte assieme tramite il comando `use` in una action ed applicate all'elemento. Definendo cioè un Named style "gruppo" nel seguente modo:

```
<define-style name="gruppo"
font-weight="bold"
font-size="18pt"
line-spacing="24pt"/>
```

possiamo utilizzarlo per tutti gli elementi, es. i paragrafi, che devono avere queste caratteristiche tramite il semplice comando:

```
<paragraph use="gruppo">
```

Per permettere la costruzione di Flow object complessi ci si può avvalere delle **Macros** tramite l'elemento <define-macro>, mentre per associare il valore tornato da uno **Script** creato per effettuare, ad esempio, dei controlli dobbiamo servirci dell'elemento <define-script>.

4.8 XML Linking Language

Il discorso introduttivo del paragrafo 4.7 può essere riportato inalterato anche per i contenuti del presente paragrafo, quindi anche qui verranno illustrati solamente i criteri guida del linguaggio XLink, lasciando al lettore il compito dell'approfondimento.

La prima bozza porta la data dell'Aprile 1997 ed attualmente l'ultimo documento ufficiale risale al 21 febbraio 2000 [35], ma siamo ancora lontani dall'avere qualcosa di definitivo ed il tutto è ancora dichiarato "Working Draft" cioè in fase di definizione.

L'idea principale attualmente viene ripresa da quanto già visto ed utilizzato per l'HTML, Hytime e TEI; ma lo scopo finale è quello di partire dai link unidirezionali previsti dall'HTML fino ad arrivare all'implementazione di tutti i tipi di link che attualmente sono disponibili per le pubblicazioni multimediali e cho sono state sviluppate nel tempo.

In realtà il meccanismo di link per un file XML si compone di due parti separate con degli scopi diversi e che portano al risultato di mantenere la ipertestuale dei dati:

- XPointer
- XLink

Xpointer è linguaggio anch'esso ancora in fase di Working Draft ed il cui ultimo documento risale al 6 dicembre 1999 [39]. Un punto di forza di questo linguaggio consiste nel fatto che esso è in grado di referenziare una porzione di documento, funzionalità che permette di sorpassare il grosso limite degli anchor di HTML. Infatti mentre con HTML per poter saltare direttamente

ad una parte di un documento lungo era necessario che l'autore avesse previsto un anchor, adesso invece è possibile indirizzare parti di documento scritte da altri senza preoccuparsi degli anchor che questi hanno previsto.

XLink da' la possibilita di controllare gli eventi che associati ad un hyperlink, senza dover passare da elementi esterni come attualemente si deve fare con HTML (basti pensare che per aprire un documento in una nuova finestra diventa necessario utilizzare altri linguaggi come JavaScript). Questa funzionalità risulta utile nel momento in cui ad un hyperlink si vogliono associare dei menù, dei messaggi oppure semplicemente dei controlli in grado di verificare i collegamenti più delicati.

Per fare un esempio possiamo pensare di utilizzare questo strumento per creare una lista di link possibili diversi per ogni utente e per ogni grado di sicurezza che si vuole mantenere, oppure ancora per garantire un navigazione casuale correlata per argomento.

4.9 Applicazioni a cui si presta

Dovrebbe essere a questo punto chiaro che XML è nato con il preciso intento di prendere il posto dell'HTML per quanto riguarda la pubblicazione di pagine Web, di conseguenza tutto quello che attualmente viene scritto e pubblicato sulla Rete è verosimile credere che in futuro sarà convertito in modo da poter superare i limiti attuali di pubblicazione di pagine Web. Ma anche le capacità di poter rappresentare anche una base di dati ne fa sicuramente anche uno strumento potente per l'archiviazione di dati in un formato leggibile a tutti in modo da poter essere utilizzato in diversi ambiti e tramite diverse piattaforme o diversi strumenti.

La sua separazione tra struttura e rappresentazione dovrebbe inoltre portare presto, o almeno questo vorrebbe essere l'intento dei suoi creatori, all'utilizzo di questo linguaggio da parte di qualsiasi strumento (quindi non solo computer) che possa avere una accesso alla rete di Internet. Un esempio vicino a noi ed attualemente in fase si studio è la telefonia cellualare che si sta' orientando verso queste capacità ed, al limite, arrivare anche ad una rappresentazione vocale dei dati.

Il problema della mancanza di specifiche chiare e definitive per la rappresentazione ne limita però molto l'attuale diffusione nell'ambito del Web se non come un formato di una base di dati consultabili con strumenti già presenti sul mercato (pensiamo ad un browser SGML o browser specifici per

XML) oppure come base di partenza per generare documenti HTML "al volo" con applicazioni apposite che quindi rendono più chiari a tutti e consultabili da una maggior quantià di utenti i documenti scritti originariamente in XML.

Occorre però anche fare una piccola cosiderazione riguardante il confronto delle dimensioni dei dati generati in XML con quelle dei dati binari. Si può notare come questi ultimi assumano una dimensione molto inferiore rispetto alla ridondante struttura a tag di un linguaggio di markup. Occorre però considerare che il protocollo HTTP prevede dei meccanismi di compressione per la spedizione dei pacchetti, quindi il confronto corretto dovrebbe essere fatto tra dati binari compressi e dati XML compressi dal quale emerge, secondo alcune prove che ho svolto personalemente, che la differenza di dimensione diventa pressochè trascurabile il che fa cadere anche una eventuale limitazione riguardante l'aumento di dimensione dei dati che transiteranno sulla Rete quando questo lingaggio prenderà piede.

4.10 Il software attualmente disponibile

Andiamo ora ad analizzare brevemente cosa attualemente la produzione di software ci mette a disposizione riguardo a questo linguaggio. Prima di tutto è necessarrio fare alcune distinzioni sulle strade che fino ad oggi sono state intraprese dalle diverse case produttrici di software. Infatti lo stato dell'arte attuale ci porta a distingure principalmente in quattro gruppi il software disponibile per XML:

1. Software per la creazione/editazione/modifica di documenti XML
2. Parser per la lettura di sorgenti XML per i vari linguaggi di programmazione
3. Browsers per la rappresentazione dei dati
4. Prodotti che si avvalgono della tecnologia XML per l'archiviazione dei dati o che comunque si basano su questa tecnologia per i vantaggi che essa porta.

Di seguito sono riportate alcune segnalazioni dei prodotti più diffusi nella comunità degli utilizzatori di XML suddivisi per ogni gruppo. Il criterio di ciatazione riguarda prima di tutto il grado di penetrazione presso gli utilizzatori di documenti XML che può anche essere letto come un segnale della

bontà del prodotto, oltre che un'analisi di cosa le grandi software house (Microsoft, IBM, Oracle, SUN, ecc.) mettono a disposizione.

Per avere un maggiore dettaglio delle funzionalità di ogni prodotto citato e per una lista più ampia è a disposizione sulla rete un sito abbastanza esauriente che raccoglie la maggior parte del software attualmente disponibile all'indirizzo http://www.xmlsoftware.com.

4.10.1 Editors

Fanno parte della prima categoria tutti quegli strumenti che sono attualmente a disposizione degli utenti per la creazione ed editazione dei documenti in formato XML e che, nelle diversità dell'offerta di funzionalità, spaziano da prodotti quali possono essere MS XML Notepad (scaricabile liberamente dal sito della Microsfot all'indirizzo http://msdn.microsoft.com/workshop/c-frame.htm#/worshop/xml/default.asp) che si occupa della semplice editazione dei dati XML non facendo alcun controllo sulla validità derivante dall'eventuale DTD associato fino ad arrivare al prodotto commerciale della Icon chiamato XML Spy 2.5 (http://www.xmplspy.com) che permette l'editazione dei file XML, il controllo di validita e di Well-formedness dei dati oltre ad avere, anche attraverso un forte legame nella versione attuale con Internet Explorer 5.0, la possibilità di avere il preview della rappresentazione dei dati compatibile sia con il formato CSS che con il formato XSL.

Entrambe questi prodotti girano su piattaforme Windows 95/98/NT e si avvalgono del parser della Microsoft per C++, mentre per altre piattaforme vale la pena segnalare il prodotto della Alphaworks, società della IBM, denominato Xeena (http://www.alphaworks.ibm.com/tech/xeena) che gira sia in ambiente Unix che in ambiente Windows 95/98/NT. Anche la Corel è entrata in questo settore introducendo delle nuove funzionalità per la versione 9 del suo WordPerfect (http://www.corel.com/office2000/feature_guide.htm) incluso nel pacchetto commerciale Wordperfect Office 2000 e che è stato sviluppato per girare sia su piattaforme Windows 95/98/NT che su piattaforme Linux.

Degno di segnalazione è anche il prodotto della Vervet Logic chiamato XML Pro (http://www.vervet.com/product-index.html) ed attualmente arrivato alla versione 2.0. Infatti questo prodotto, oltre ad essere un ottimo editor per la creazione di documenti Well-formed ha dalla sua la grande gamma di piattoforma su cui puoi girare che va da Windows 95/98/NT a Solaris

e Linux. Infatti il suo motore si basa sul parser della IBM XML4J e richiede per funzionare Sun JRE 1.2.

Ultimo prodotto da segnalare è quello della Media Design in Progress chiamato Emile (http://www.in-progress.com/emile/index) in quanto è l'unico attualemente che gira su piattaforme Macintosh.

Esistono infine altri prodotti "minori" scritti interamente in Java che si possono prestare ad essere utilizzati su diverse piattaforme proprio per le caratteristiche di questo linguaggio di programmazione.

4.10.2 Parsers

La distinzione tra Parser ed Editor consiste essenzialmente nel fatto che un parser legge un documento XML e ne mette a disposizione di una applicazione sia la struttura che i contenuti, quasi sempre utilizzando una interfaccia standard quale SAX o DOM, mentre l'editor si occupa di una semplice editazione del file a livello di utente.

Attualmente la casa produttrice che sembra essere più attiva in questo settore è la Alphaworks che ha creato diversi Parser per quasi tutte le piattaforme esistenti e per i linguaggi attualmente più utilizzati dai programmatori. Infatti è possibile avere un parser per il linguaggio C++ (XML4C reperibile all'indirizzo http://www.alphaworks.ibm.com/tech/xml4c) che può essere supportato dalle piattaforme AIX, Linux, Solaris, Windows 95/98/NT, HP-UX 1.2, HP-UX 11 ed un prodotto analogo per Java (XML4J reperibile all'indirizzo http://alphaworks.ibm.com/formula/xml) da utilizzare sia su Linux che su tutte quelle piattaforme che supportano Java.

Dal canto suo la SUN ha messo a disposizione degli sviluppatori un prodotto chiamato Java Project X (http://developer.java.sun.com/developer/earlyAccess/xml/index.html) che altro non è che un pacchetto di API in cui è integrato un parser che permette la descrizione di un modello di alberi ad oggetti in memoria conforme con le specifiche del W3C DOM Level 1. Questo pacchetto è scaricabile gratuitamente ed è costruito per operare con il linguaggio Java, quindi su ogni piattaforma in grado di supportare tale linguaggio.

Anche la Microsoft ha messo a disposizione un suo parser gratuitamente, esclusivamente per piattaforme Windows (http://msdn.microsoft.com/ down-

loads/tools/xmlparser/xmlparser.asp), ed è disponibile come un componte di COM. Include anche un modulo di programmazione in grado di supportare ECMAScript, Java, Perl, Python, SQL, Visual Basic, Visual C++ e VB-Script. Assieme al prodotto della Alphaworks questo prodotto fa' la parte del leone nel mercato per quanto riguarda questa piattaforma anche grazie al fatto di essere ovvimente utilizzato da Internet Explorer 5.

Per chiudere il cerchio delle grandi case produttrici va' citato il prodotto della Oracle (XML Parser for Java reperibile all'indirizzo http://technet.oracle.com/tech/xml/section.htm) che è supportato per l'utilizzo con JDK 1.1.x su piattaforme Windows e Unix.

Come detto prima il panaroma attuale comprende anche parser, di cui non diamo i dettagli in questa sede, per moltri altri linguaggi quali il C, Phyton, ActiveX, Tcl, Delphy, Perl ed anche una beta per JavaScript. La parte preponderante del mercato, però, sembra essersi orientata verso lo sviluppo per Java, probabilmente anche perché questo è il linguaggio più rispondente alle caratterisitiche di multipiattaforma che il W3C XML Working Group ha voluto includere nella filosofia delle specifiche per l'XML.

4.10.3 Browsers

Prima di tutto occorre evidenziare che un browser compatibile è una applicazione in grado di poter leggere e rappresentare, ove richiesto con i metodi di rappresentazione richiesti, i documenti scritti con il linguaggio XML. Occore inoltre ricordare che attulamente non ci sono delle specifiche chiare e precise per quanto riguarda il metodo di rappresentazione dei dati e questo probabilmente ha portato ad una certa resistenza da parte delle case produttrici a creare del software apposito o ad integrare il software esistente.

Al momento della scrittura di questa tesi l'unico browser, tra quelli più diffusi sul mercato, in grado di poter leggere dei documenti XML e di rappresntarli correttamente è il prodotto della Microsoft Internet Explorer 5. Infatti esso è in grado di rappresentare i documenti che hanno dei moduli di rappresentazione in CSS ed in XSL (perlomeno per quanto attualmente disponibile dal working draft della W3C), oltre che ad una rappresentazione tag-coloured e con elementi collassabili per tutti qui documenti che non hanno fogli di stile associati. Gli altri concorrenti, e mi riferisco soprattuto al concorrente diretto Netscape Navigator, sono in grado solamente di rappre-

sentare il sorgente di tali documenti come un semplice file di testo.

La Netscape ha però messo sul mercato un prodotto denominato Mozilla (http://www.mozilla.org) addiruttura con la licenza Open Source e che è in grado di rappresentare anche i documenti XML che hanno il modulo di rappresentazione CSS. Sembra anche che con la versione 5.0, o comunaue con una versione succesiva alla attuale, del proprio Navigator tali funzionalità verranno integrate, ma non esiste ancora nessun documento ufficiale a riguardo.

Anche la IBM si è occupata di creare un browser, anche se in forma molto più ridotta rispetto alla concorrenza (http://www.alphaworks.ibm.com/tech/xmlviewer) tanto è vero che non è supportato nessun metodo di rappresentazione dei dati.

Da questo punto di vista il discorso potrebbe considerarsi chiuso se non per qualche altro browser esclusivamente per XML e per un prodotto della Fujitsu (HyBrick reperibile all'indirizzo http://www.fujitsu.co.jp/hypertext/free/HyBrick/en/index.html) che merita una citazione in quanto è l'unico prodotto che utilizza la tecnologia DSSSL per la rappresentazione dei dati.

4.10.4 Applicazioni

Infine appartengono a questo gruppo tutte quelle applicazioni che si appoggiano a documenti XML come sistema di archiviazione ed elaborazione dei dati oppure per lo scambio degli stessi. Chiaramente non è facile in questo campo fare una analisi precisa di cosa attualmente il mercato metta a disposizione, anche perchè l'utilizzo di tali tecnologie non è altrettanto pubblicizzata dalle case produttrici quanto possono essere dei prodotti specifici per l'utilizzo di tale linguaggio.

Tra le applicazioni disponibili possiamo racchiudere in un sottogruppo particolare diversi strumenti per la conversione dei dati da ed in lingauggio XML. Citiamo tra queste alcuni che possono essere di supporto o di esempio al lavoro che sto' svolgendo o più in generale allo scopo che MOMIS sta' perseguendo: in particolare XML-DBMS (http://www.informatik.tu-darmstadt.de/DVS1/staff/bourret/xmldbms/readme.html) che è in grado di trasferire dei dati tra documenti XML e dei database relazionali, oppure ancora DB2XML (http://www.informatik.fh-wiesbaden.de/~turau/DB2XML/ index.html) per la trasformazione di database relazionali in documenti XML

tramite l'utilizzo di driver (JDBC oppure ODBC utilizzando il ponte JDBC-ODBC).

Altri strumenti di conversione di cui vale la pena citare l'esistenza sono dei convertitori "al volo" di documenti XML in documenti HTML che quindi coprono la attuale lacuna in certi settori di browser.

Non bisogna infine dimenticare che XML deriva direttamente da SGML e le stesse specifiche hanno imposto la completè compatibilità con il suo progenitore. Questo fatto fa' si che i prodotti studiati e realizzati per la rappresentazione di dati SGML dovrebbero essere utilizzati altrettanto agevolmente anche per i documenti XML.

Capitolo 5

Il wrapper XML/ODL$_{I^3}$: specifiche funzionali

Verranno illustrate le differenze fondamentali ed i punti in comune che esistono tra la struttura del linguaggio XML e quella del linguaggio ODL$_{I^3}$ esaminate rispettivamente nei capitoli 4 e 3. Tramite questa analisi ci si propone l'obiettivo di evidenziare quale dovrebbe essere il lavoro che deve svolgere il wrapper ed in particolare illustrare quali sono i problemi teorici alla base del software che deve essere svuluppato per tradurre da XML ad ODL$_{I^3}$.

5.1 Due linguaggi diversi, due filosofie diverse

Prima di tutto bisogna considerare che i due linguaggi presi in considerazione sono nati con degli obiettivi differenti e sicuramente anche finalizzati ad un target differente anche dal punto di vista dell'utilizzo. Infatti mentre ODL$_{I^3}$ è nato come linguaggio in grado di descrivere una base di dati archiviata in sorgenti dal formato differente, XML nasce con l'intento di descrivere (per utilizzare lo stesso termine per "dare un formato") ai dati che dovranno essere successivamente veicolati con una certa facilità in particolare tramite la rete Internet.

Chiaramente lo scopo che risiede dietro alla definzione di ODL$_{I^3}$, appunto quella di descrivere basi di dati in formato differenti, non può non fare pensare che esistano dei punti in comune tra i due linguaggi. Infatti sempre in ODL$_{I^3}$ è prevista la descrizione di dati semistrutturati ed un file XML è

appunto un formato che risiede in questa categoria.

Malgrado infatti la diversa filosofia ha portato, come ovvia conseguenza, alla creazione di sintassi differenti in grado di perseguire ognuna il proprio scopo, i punti in comune tra le due strutture che le loro sintassi descrivono sono sufficientemente numerosi. Questi punti di contatto ci permettono di affermare che un lavoro di traduzione non sia solo possibile da un punto di vista teorico, ma anche in grado di produrre un interscambio di dati tra un linguaggio e l'altro sufficientemente completo.

Passiamo ora ad analizzare in particolare quali sono gli elementi di convergenza tra i due linguaggi e di quali sono gli algoritimi che si possono utilizzare per effettuare la traduzione che sta' alla base del funzionamento del wrapper.

5.1.1 La descrizione della struttura di un documento XML

Primo passo da realizzare è quello di estrapolare la struttura di un file XML che è quella che serve per definire l'equivalente struttura ODL$_{I^3}$. Come già detto nel paragrafo 4.1.4 un documento XML può essere *Valid* oppure semplicemente *Well-Formed* a seconda che sia presente o meno un DTD, il quale ricordo ha il compito appunto di descrivere la struttura dei dati contenuti nel file stesso.

Chiaramente il caso di documenti Validi è il più semplice da trattare in quanto la struttura è intrinsicamente definita dal DTD. Il caso dei documenti semplicemente Well-formed presenta invece delle difficoltà abbastanza evidenti a cominciare dal fatto che si possono individuare ulteriori due casistiche differenti:

- non è previsto nessun DTD;
- pur essendo previsto un DTD, questo non viene rispettato nella descrizione dei file.

Prima di tutto si sottolinea che questa seconda caratteristica è quella meno verosimile per non dire che potrebbe anche non verificarsi mai oltre ad essere vivamente sconsigliata dal W3C. Infatti oramai quasi tutti i browser, gli editor ed i parser di ultima generazione, o che comunque sono sufficientemente completi dal punto di vista delle specifiche del W3C, includono il

controllo di validità e quindi un meccanismo di blocco/avvertimento nel caso si cerchi di introdurre delle discrepanze tra schema del DTD e schema dei dati. Chiaramente questo controllo/blocco potrebbe non essere sufficiente a garantire tale coerenza poiché, anche se sconsigliata, è la sintassi stessa che consente di introdurre delle discrepanze.

Passando nel particolare dell'approccio che occorre tenere nel caso di file *Well-formed*, possiamo dire che il procedimento è fondamentalmente simile per entrambe le casisitiche illustrate sopra. Infatti per entrambe i casi lo schema dovrebbe sempre essere ricavato dall'albero generato dai dati, ma la differenza consiste nel capire quando i dati XML hanno un DTD che poi non viene rispettatto e quando, invece, non l'hanno proprio.

Ammettendo quindi di aver capito quando il DTD non viene rispettato o quando invece non esiste proprio, l'approccio prosegue con il leggere e costruire, eventualmente attraverso un parser qualsiasi tra quelli disponibili sul mercato, il DOM (**D**ocument **O**bject **M**odel) relativo. Da tale modello è necessario successivamente estrapolare gli elementi con i rispettivi attributi assegnati che dovranno essere utilizzati come base per la traduzione in ODL$_{I^3}$. Certamente si tratta di una operazione laboriosa e rischiosa in quanto potrebbe non essere stato nelle intenzioni del progettista di tale file stabilire una struttura ben definita e quindi ci si potrebbe trovare di fronte ad un insieme di dati difficilmente racchiudibili in un modello ben definito.

Bisogna però sottolineare come lo stesso W3C abbia caldamente invitato gli sviluppatori all'utilizzo del DTD per ogni file XML che viene creato e quindi alla progressiva eliminazione, almeno sempre nelle intenzioni del consorzio, di file semplicmente Well-formed o comunque con una struttura particolare non descrivibile tramite un DTD. Oltre a questo, sempre nella stessa raccomandazione, vengono invitati gli sviluppatori a creare la definizione del DTD all'interno dei documenti XML stessi per limitare il pericolo di veicolare dei dati incompleti.

5.2 Le operazioni del wrapper XML/ODL$_{I^3}$

Premettendo che, d'ora in poi, la base di partenza per il lavoro di traduzione è solamente il DTD, generato o letto dal file, in quanto è l'unico che ci consente di avere chiarezza sulla struttura dei dati. Le operazioni che deve svolgere il wrapper per ogni struttura chiave che incontra nel DTD hanno come scopo

finale quello di riprodurre, chiaramente dove possibile, una analoga struttra ODL$_{I^3}$ che sia in grado di descrivere senza errori ed ambiguità i dati in formato XML.

Prima operazione da fare, è quella di prevedere il nome del file il quale può essere riportato inalterato, chiaramente con le necessarie modifiche nella estensione, dalla sorgente XML al nuovo file di struttura ODL$_{I^3}$.

Considerando invece il contenuto ricordo che l'intera struttura è prevista sia descritta tramite un modello DOM che, come più volte detto, è caratterizzato da una radice di origine da cui si diramano i vari nodi. Tale radice è la base di partenza del lavoro del wrapper in quanto è possibile tradurla nella classe (o interfaccia che dir si voglia) principale di ODL$_{I^3}$ e tutti i nodi successivi che si incontrano possono essere tradotti come dei tipi definiti dall'utente, in cascata, fino al raggiungimento dei vari tipi atomici che non sono altro che le foglie dell'albero associato al DOM.

Andiamo ora a vedere il dettaglio degli algoritmi di traduzione.

5.3 La traduzione degli Elementi XML

La struttura fondamentale dei dati conservati nel formato XML si basa essenzialmente sugli elementi (la cui dichiarazione è stato illustrata nel paragrafo 4.4.1) ed è quindi da qui che deve partire il processo di analisi che deve svolgere il wrapper.

Prima di tutto occorre prendere in considerazione l'elemento radice che ci consente di creare l'interfaccia principale in ODL$_{I^3}$ sulla cui base derivare tutto il resto della struttura. La traduzione da un elemento ad interfaccia è pressoche automatica in quanto sono distintivi di entrambe le struttura un nome identificativo ed un insieme di attributi caratteristici.

Come esempio di riferimento consideriamo un file XML che potrebbe aver generato la struttura ODL$_{I^3}$ riportata nel paragrafo 3.6 e che è stato riscostruito in 4.6. Il DTD corrispondente (che assume lo stesso nome del file dati Eating_Source) potrebbe cominciare, chiaramente ignorando tutte quelle righe che poi servono ai parser XML per elaborare correttamente i dati, con:

```
<!ELEMENT Fast-Food (name, address, phone?, speciality*,
category, nearby?, midprice?, owner?)*>
```

la cui corrispondente in ODL_{I^3} è:

```
interface Fast-Food
(source semistructured Eating_Source)
{
   [...]
};
```

Analizzando le operazioni che il wrapper ha dovuto svolgere per generare tali righe notiamo che denominare l'interfaccia è stato preso il nome dell'elemento radice del DTD, mentre per la descrizione della sorgente abbiamo assegnato la parola chiave `semistrucutred` in automatico poiché è insito il fatto che un file XML sia di tipo semistrutturato.

Procediamo con l'analisi per vedere come vengono tradotti ed inseriti gli attributi dell'interfaccia appena creata.

5.3.1 Tipi di dati

Premettiamo che in XML non esiste una dettagliata specifica del tipo dei dati che possono essere contenuti in un elemento, ma il massimo del dettaglio che si può raggiungere (come poi è caratteristica di un linguaggio semistrutturato) è dato da tipi stringa, che nel caso di XML sono caratterizzate dalla parola chiave `#PCDATA`.

Evidenziando ancora il tipo degli attributi di XML abbiamo detto che le particelle atomiche possono essere al massimo di tipo stringa, ma la stessa struttura permette di essere ulteriormente dettagliata procedendo con la lettura del file DTD in cui possono essere presenti successive dichiarazione di elementi.

Proseguendo nell'esempio si incontrano quindi nel DTD le seguenti righe di codice:

```
<!ELEMENT nearby (r_code, name, category, street, zip_code,
pers_id, tourist_menu_price, special_dish>
<!ELEMENT owner (name, address, job)>
```

il che ci permette di inserire due nuove interfacce il ODL_{I^3} semplicemente effettuando la traduzione come fino a qui è stato illustrato:

```
interface nearby
(source semistructured Eating_Source)
{
   [...]
};

interface owner
(source semistructured Eating_Source)
{
   [...]
};
```

Analizzando queste righe si visualizza subito che esiste si' la possibilita di creare dei tipi definiti dall'utente, ma non è possibile dare un nome. Questo potrebbe sembrare una limitazione del formato XML rispetto a quello di ODL$_{I^3}$, ma in realtà non è così. Infatti le informazioni necessarie per il dettaglio dell'interfaccia possono essere completamente riportate in ODL$_{I^3}$ ed inoltre anche il collegamento del nuovo tipo avviene senza particolari problemi e seguendo le regole grammaticali previste.

Dopo tale elaborazione l'interfaccia principale, già parzialmente tradotta sopra, viene ulteriormente completata e trasformata nelle seguenti caratteristiche:

```
interface Fast-Food
(source semistructured Eating_Source)
{
   [...]
   attribute nearby nearby*;
   [...]
   attribute owner owner*;
};
```

Da una ulteriore lettura del file DTD si possono ancora trovare i dettagli di altri attributi (per tutte e tre le interfacce finora analizzate) che non hanno una struttura supplementare, ma che possono essere considerate come le foglie dell'albero generato dal DTD:

```
<!ELEMENT name #PCDATA>
<!ELEMENT phone #PCDATA>
<!ELEMENT speciality #PCDATA>
```

```
<!ELEMENT category #PCDATA>
<!ELEMENT midprice #PCDATA>

<!ELEMENT job #PCDATA>

<!ELEMENT r_code #PCDATA>
<!ELEMENT street #PCDATA>
<!ELEMENT zip_code #PCDATA>
<!ELEMENT pers_id #PCDATA>
<!ELEMENT tourist_menu_price #PCDATA>
<!ELEMENT special_dish #PCDATA>

[...]
```

il che autorizza la traduzione di tutti questi elementi, come già detto in precendenza, in attributi di tipo stringa di ODL_{I^3} e quindi a completare le interfacce finora introdotte tramite il seguente codice:

```
interface Fast-Food
(source semistructured Eating_Source)
{
   attribute string name;
   [...]
   attribute string address;
   attribute string phone*;
   attribute set<string> speciality;
   attribute string category;
   attribute nearby nearby*;
   attribute string midprice*;
   attribute owner owner*;
}

interface nearby
(source semistructured Eating_Source)
{
   attribute string r_code;
   attribute string name;
   attribute string category;
   attribute string street;
   attribute string zip_code;
```

```
   attribute string pers_id;
   attribute string toursit_menu_price;
   attribute string special_dish;
}

interface owner
(source semistructured Eating_Source)
{
   attribute string name;
   [...]
   attribute string job;
}
```

La sintassi XML prevede oltre ad i tipi `#PCDATA` anche i tipi `ANY` ed `EMPTY` che, come già illustrato nel paragrafo 4.4.1, permettono, nel primo caso, l'inserimento di dati in qualsiasi formato, mentre nel secondo caso implica la presenza di un elemento vuoto.

Analizzando separatamente i due casi sottolineamo che nel caso di un elemento di tipo `ANY` i dati contenuti possono essere sia semplicemente di tipo stringa sia di un tipo più complesso come una struttura XML comprensiva di markup. Siccome questo non è assolutamente prevedibile a livello di descrizione della struttura e può essere sempre diverso nel corso dell'intero file XML si è deciso di creare un nuovo tipo di dati ODL$_{I^3}$ in modo da identificare immediatamente che si sta' trattando un elemento di questo tipo, ma che allo stesso lascia il compito di trattare i dati corrispondenti agli astrumenti atti a fare le query.

Questo tipo che in un DTD XML corrisponde alla dichiarazione

```
<!ELEMENT nome ANY>
```

necessita di una descrizione del tipo:

```
typedef any
{
   string strvalue;
}
```

che corrisponde alla traduzione dell'attributo ODL$_{I^3}$ come

```
[...]
attribute any nome;
[...]
```

Passando invece ad analizzare gli elementi di tipo `EMPTY` occorre sottolineare che essi non portano nessuna informazione aggiuntiva se non quella degli eventuali attributi XML associati. Il fatto, però, che essi possano essere specificati nei dati è da considerarsi come condizione sufficiente per assegnarne importanza e quindi avere la necessità di prevedere nella struttura ODL_{I^3} qualcosa che mi permetta di specificare se l'elemento esiste effettivamente nei dati oppure no.

Quanto detto è praticamente identificativo di un attributo di tipo booleano e come tale prevede che ogni volta che si incontra nel DTD di XML una dichiarazione del tipo:

```
<!ELEMENT sino EMPTY>
```

debba essere tradotto come

```
attribute boolean sino;
```

5.3.2 Opzionalità e tipi di collezione

Una traduzione che è stata fatta riguarda l'opzionalità di un elemento: infatti nella sintassi di XML è stato previsto un carattere di suffisso (`?`) che indica se un elemento o "particella di contenuto" (come vengono denominati nelle specifiche riportate nell'appendice A) possa comparire zero o una volta. Questa opzionalità esiste inalterata anche in ODL_{I^3} e come tale dovrà essere utilizzata per descrivere l'equivalente possibilità fornita dalla struttura XML originaria, l'unica differenza riguarda il suffisso che per tale linguaggio di destinazione è il carattere "`*`".

Oltre al discorso dell'opzionalità sono presenti sempre nella sintassi di XML anche altri due caratteri di suffisso per una particella di contenuto con un significato differente:

- il "`*`" per indicare la presenza di zero o più ricorrenze
- il "`+`" per indicare la presenza di almeno una ricorrenza, ma ne consente anche la collezione

Queste caratteristiche trovano riscontro anche in ODL$_{I^3}$ tramite la definizione dei tipi di collezione, in particolare tramite la dichiarazione dei *set*. Una differenza sostanziale consiste nel fatto che tale linguaggio non è così dettagliato come l'XML e quindi non è possibile distinguere tra ricorrenze di cui è obbligatorio specificare almeno un valore oppure ricorrenza in cui tale specificazione non è obbligatoria.

5.3.3 Elementi di tipo misto

Caso particolare e non ancora trattato finora riguarda l'attributo `address` che sempre in 3.6 assume due sintassi differenti, cioè viene utilizzato sia come una semplice stringa che come un tipo complesso formato da diversi tipi atomici.

In XML la sintassi consente una possibilità simile tramite una dichiarazione mista, ed in particolare indica che *"[...] gli elementi di quel tipo possono contenere 'character data', opzionalmente mescolati con elementi figlio [...]"* separati dal carattere "|". In realtà la forma che viene assunta non consente una descrizione propriamente identica a quella di ODL$_{I^3}$ in quanto nella versione attuale di XML non è prevista la dichiarazione diretta di una lista di attributi, ma occorre definire preventivamente un elemento padre che poi potrà essere successivamente dettagliato.

Vediamo ora quale potrebbe essere una forma in cui sono state scritte le righe che hanno generato il codice ODL$_{I^3}$:

```
<!ELEMENT address (#PCDATA|address_type)*>
<!ELEMENT address_type (city,street,zipcode)>

<!ELEMENT city #PCDATA>
<!ELEMENT street #PCDATA>
<!ELEMENT zipcode #PCDATA>
```

che ci permettono appunto, tramite una serie di semplici operazioni di creare l'interfaccia ODL$_{I^3}$ di address:

```
interface address
(source semistructured Eating_Source)
{
   attribute string city;
   attribute string street;
```

```
   attribute string zipcode;
};
union
{
   string
};
```

Tale interfaccia consente di applicare alle precedenti interfacce `Fats-Food` e `owner` anche tale attributo tramite la seguente riga di codice:

```
[...]
attribute address address;
[...]
```

5.3.4 Elementi di tipo choice

Oltre al caso precedente, che potrebbe essere considerato relativamente semplice da un punto di vista dell'algoritmo di traduzione, la sintassi di un DTD XML prevede anche degli elementi i cui figli (o più correttamente alcuni di essi) possono essere di tipo *choice*, cioè separati dal carattere pipe ("|").

Tale opzione della grammatica permette quindi di avere nei dati la possibilità di inserire delle informazioni le quali potrebbero assumere differenti significati oltre dovuti anche da una diversa struttura in cui queste informazioni sono rappresentate.

Riprendendo a grandi linee l'esempio fatto nel paragrafo 5.3.3 ci si potrebbe trovare ad esaminare delle righe di un DTD come le seguenti:

```
<!ELEMENT person (name, (address | address_comp), date_of_birth)>

<!ELEMENT address #PCDATA>
<!ELEMENT address_comp (street, city, country)>
```

che vorrebbero significare che l'indirizzo di una persona può essere scritto sia in una forma semplice caratterizzata da una stringa oppure in una forma più complessa che prevede struttura diversamente articolata.

Osservando questa struttura nell'ottica della sintassi ODL_{I^3} si nota subito che anche questo caso potrebbe essere risolto tramite il costrutto `union`. Da un punto di vista teorico l'ideale sarebbe una sintassi che prevedesse la possibilità di definire semplicemente per l'attributo quali possono essere le forme

che esso può assumere, ma siccome la sintassi non prevede questa possibilità diventa quindi necessario adottare una strada leggermente più laboriosa che consiste nel definire un nuovo tipo, tramite il costrutto `typedef` il quale contempla l'utilizzo di una union.

Le righe ODL$_{I^3}$ che verrebbero generate potrebbe quindi essere:

```
interface person
{
   attribute string name;
   attribute person_union_2 person_union_2;
   attribute string date_of_birth;
}

typedef person_union_2
{
   string address;
}
union
{
   string street;
   string city;
   string country;
}
```

dove per tutti gli elementi per cui non era specificato niente si è assunto per semplicità che siano stati dichiarati di tipo `#PCDATA`.

Occorre però a questo punto chiarire quale è il criterio di definizione sia del nome dell'attributo di tipo *choiche* che del nuovo tipo creato. Tale nome è la composizione dei tre elementi distinti che servono per identificare univocamente il problema che si sta affrontando, cioè il nome dell'attributo padre, nel nostro esempio `person`, la parola chiave `_union` che sta' ad indicare che si sta trattando un elemento di tipo *choice* ed infine da un numero il quale rappresenta la posizione di tale elemento nella lista, nel nostro caso `_2` sta' a significare che si tratta del secondo elemento della lista dei figli.

La sintassi XML prevede però anche dei casi ancora più articolati rispetto a questo consentendo diversi livelli di dettaglio di un elemento di questo tipo. Un esempio potrebbe essere dato dalle seguenti righe:

```
<!ELEMENT person (name, (address | (address1 | address2)),
```

```
          date_of_birth)>

<!ELEMENT address #PCDATA>
<!ELEMENT address1 (street, city)>
<!ELEMENT address2 (street, city, country)>
```

dove, in questo esempio, si è arrivati fino ad un terzo livello di profondità.

A questo punto l'algortimo di traduzione rimane pressoché identico a quanto già visto, con l'unica differenza che nel nuovo tipo dichiarato si ha la necessità di introdurre un ulteriore `typedef`.

Le righe che rappresentano una possibile traduzione sarebbero quindi:

```
[...]

typedef person_union_2
{
   string address;
}
union
{
   person_union_2_2 person_union_2_2;
}

typedef person_union_2_2
{
   string street;
   string city;
}
union
{
   string street;
   string city;
   string country;
}
```

dove il nome del nuovo tipo inserito segue l'algoritmo di costruzione già visto in precedenza a cui è stato aggiunto un suffiso (in questo caso ancora _2) che identifica quale elemento *child* necessità di un ulteriore dettaglio.

Seguendo quindi queste regole diventa possibile tradurre tanti livelli di dettaglio quanti previsti dalla sintassi XML (teoricamente infiniti) tramite l'introduzione di volta in volta un nuovo tipo ODL$_{I^3}$.

5.4 La traduzione degli Attributi

Nel linguaggio XML sono previsti per gli elementi anche una serie di attributi caratteristici (dichiarati con la sintassi rappresentata in 4.4.2) e non previsti in ODL$_{I^3}$ anche se la parola chiave `attribute` è presente in ODL$_{I^3}$ stesso, ma con un significato differente.

NOTA: per non fare confusione nel seguito del capitolo denomineremo *attlist* la lista di attributi di XML, mentre con *attribute* gli attributi delle interfacce di ODL$_{I^3}$.

La mancanza di un corrispondente di *attlist* in ODL$_{I^3}$ rende necessario l'utilizzo di qualche artificio o stratagemma per poter descrivere tali informazioni. È infatti necessario che la relazione esistente in XML tra elemento ed *attlist* sia riportata inalterata anche per l'*attribute* di ODL$_{I^3}$ che dovrà essere creato. Questa relazione è possibile mantenerla attraverso il nome che gli verrà assegnato. Oltre a questo occorre però ricordare che è possibile dichiarare più *attlist* per uno stesso elemento e quindi si ha la necessità di gestire anche questa evenienza.

Passiamo ore ad analizzare i meccanismi per la traduzione. Nel caso un elemento abbia un solo *attlist* collegato la traduzione può essere effettuata tramite una semplice creazione di un nuovo *attribute* come nell'esempio:

```
<!ELEMENT city (#PCDATA)>
<!ATTLIST city zipcode CDATA>
```

che può essere tradotta in ODL$_{I^3}$ come

```
{
   attribute string city;
   attribute string city_zipcode*;
}
```

Analizzando le righe ODL$_{I^3}$ che sono state generate si nota che si è denominato l'*attribute* utilizzando come prefisso il nome dell'elemento a cui è riferito l'*attlist* seguito dal carattere underscore "_" (in questo caso "`city_`") e dal nome stesso dell'*attlist*. Questo meccanismo è appunto quello che mantiene la relazione elemento/*attlist* altrimenti non gestibile in ODL$_{I^3}$.

Nel caso invece ci si trovi a dover trattare una serie di attributi per un unico elemento si utilizza sempre lo stesso criterio di costruzione del nome e ripetuto per ogni elemento dichiarato nel DTD.

Ad esempio un parte di DTD che presente le seguenti righe:

```
<!ELEMENT city (#PCDATA)>
<!ATTLIST city
          zip_code CDATA
          phone_prefix CDATA>
```

verrebbe tradotto in ODL$_{I^3}$ come:

```
attribute string city;
attribute string city_zip_code*;
attribute string city_phone_prefix*;
```

Negli esempi fino qui illustrati gli *attlist* sono stati inseriti come *attribute* opzionali per i motivi che verranno meglio illustrati nel successifo paragrafo 5.4.1.

La mancanza di un meccanismo di traduzione preciso ed univoco per gli *attlist* potrebbe essere considerato come un problema, ma andando più a fondo nell'analisi non ci si può non accorgere come questo non è del tutto vero poiché è proprio grazie alla sintassi prevista per la dichiarazione degli *attlist* che diventa lecito, tra le altre cose, pensare alla creazione di un meccanismo di *primary key* e di *foreign key* concettualmente lontano dalla filosofia di un file semistrutturato, ma consentito dalla sintassi XML.

Prima di analizzare in dettaglio le caratteristiche peculiari degli *attlist* occorre ricordare che per essi sono previste dalla definizione anche degli "Attribute Defaults" che andiamo subito a studiare.

5.4.1 Default di attributo

Gli"Attribute Defaults" sono delle informazioni supplementari che indicano se la presenza di un attributo è richiesta e, se non lo è, come un processore XML dovrebbe reagire se un attributo dichiarato è assente in un documento.

Si tratta fondamentalmente di caratteristiche che riguardano più i dati che la struttura, ma in alcuni casi hanno delle implicanze anche con la struttura e che quindi rientrano in questa fase di analisi della traduzione.

Le possibilità previste per un default di attributo sono:

- `#REQUIRED`
- `#IMPLIED`
- `#FIXED`

Nel caso si incontri un *attlist* **#REQUIRED** significa che è necessario fornire sempre l'*attlist* nei dati e quindi, riportando il discorso sulla nostra applicazione, che la traduzione deve corrispondere ad un *attribute* obbligatorio. Ad esempio una dichiarazione del tipo:

```
<!ELEMENT city (#PCDATA)>
<!ATTLIST city zipcode CDATA #REQUIRED>
```

può essere agevolmente tradotto in *attribute* del tipo:

```
{
   attribute string city;
   attribute string city_zipcode;
}
```

Negli altri casi in cui non si tratta di un *attlist* richiesto deve essere considerato come opzionale e quindi come tale deve essere tradotto:

```
<!ELEMENT city (#PCDATA)>
<!ATTLIST city zipcode CDATA>
```

dovrà essere tradotto in *attribute* del tipo:

```
{
   attribute string city;
   attribute string city_zipcode*;
}
```

Passando ad una dichiarazione di tipo **#IMPLIED** le specifiche riportano che *"...significa che nessun valore di default è fornito"*. Quindi a livello di traduzione in ODL_{I^3} questo non comporta nessuna attenzione particolare in quanto è una informazione che si riferisce ai dati e non a livello di struttura. Il comportamento del wrapper è quindi quello già illustrato nell'esempio precedente.

Altra casistica già trattata nello stesso esempio è quella in cui un *attlist* non sia dichiarato ne' `#REQUIRED` e neppure `#IMPLIED` che da un punto di vista sintattico implica che il valore da attribuire deve essere il valore di default dichiarato, mentre dal punto di vista della traduzione non implica nessuna differenza in quanto è un'altra caratteristica riferita ai dati e non alla struttura.

Ultimo caso è quello di una dichiarazione di tipo **#FIXED** che ha delle implicanze principalmente dal punto di vista della struttura. Infatti in questo caso si stabilisce che l'*attlist* deve sempre avere il valore di default. Ma questo, visto nell'ottica di ODL_{I^3} , non è altro che una dichiarazione di una costante e quindi come tale deve essere tradotta. Ad esempio trovandosi il wrapper di fronte ad una riga di sorgente XML del tipo:

```
[...]
<!ATTLIST zip_code state CDATA #FIXED "ITALY">
[...]
```

dovrà provvedere alla generazione di una corrispondente linea di codice XML come:

```
const string state="ITALY";
```

Procedendo per passi ed andiamo ad analizzare nel particolare i tipi di *attlist* previsti e quali possono essere le corrispondenze in ODL_{I^3} . Prima di tutto distinguiamo gli *attlist*, come già detto nel paragrafo 4.4.2, in tre gruppi fondamentali:

- **String**
- **Tokenized**
- **Enumerated**

5.4.2 String

Nella sintassi tutti gli *attlist* che contengono la parola chiave `CDATA` indicano che sono di tipo stringa e quindi i valori che possono essere archiviati dovranno essere delle stringhe.

Questo caso è già stato trattato, essendo il più semplice ed immediato, come base per gli esempi precedenti e quindi il discroso va riportato inalterato per questa sezione.

5.4.3 Tokenized

Nel caso di *attlist* di tipo *Tokenized*, che è anche il più interessante viste le caratteristiche che racchiude, sono previsti diversi "Vincoli di validità" identificati dalle seguenti parole chiavi:

- ID
- ID/IDREFS
- ENTITY/ENTITIES
- NMTOKEN/NMTOKENS

Andiamo adesso ad analizzarne le particolarità evidenzione le potenzialità che hanno e gli algoritmi di traduzione in altrettante strutture ODL$_{I^3}$.

Vincolo di validità ID

Le specifiche riportate nell'appendice A citano testualmente per questo vincolo di validità:

"[...] Un nome non deve apparire più di una volta in un documento XML come un valore di questo tipo: cioè i valori ID devono identificare univocamente gli elementi che li portano."

il che ci rende lecito pensare ad un meccanismo di *Primary key* da tradurre nell'analogo costrutto di ODL$_{I^3}$ come descritto in 3.3.2.

Anche questa traduzione, però, non può essere fatta in modo automatico, ma occorre l'intervento del progettista che autorizzi a consiederare tale *attlist*

appunto come una chiave primaria.

Prima di procedere con l'analisi dettagliata bisogna ancora sottolineare che la forma BNF che descrive i vincoli che devono rispettare i valori di tipo ID è data da:

```
Name ::= (Letter | '_' | ':')  (NameChar)*
```

quindi non sono previsti dei codici esclusivamente numerici, ma piuttosto dei codici letterari, il che comunque non crea delle problematiche dal punto di vista della traduzione.

Riprendendo sempre l'esempio di 3.6 le righe di codice che possono aver creato l'interfaccia Restaurant, limitatamente al codice ed al nome, potrebbero essere le seguenti:

```
<!ELEMENT Restaurant (name,[...])*>

<!ELEMENT name (#PCDATA)>
<!ATTLIST name r_code ID #REQUIRED>
```

che ha l'equivalente in ODL_{I^3} data dal seguente pezzo di sorgente:

```
interface Restaurant
(source semistructured Eating_Source
key name_r_code)
{
   attribute string name_r_code;
   attribute string name;
   [...]
}
```

dove si è assunto che l'utente abbia confermato che effettivamente questo *attlist* debba essere considerato come una chiave primaria.

Vincolo di validità IDREF/IDREFS

Questo costrutto, caratterizzato dalle parole chiave `IDREF` ed `IDREFS`, consente, sempre riportando quando definito nelle specifiche del W3C, di creare un *attlist* al quale:

"[...] deve corrispondere il valore di un attributo ID di qualche elemento del documento XML; cioè i valori IDREF devono essere uguali al valore di qualche attributo ID."

Da tale definizione si può notare immediatamente come questo possa essere interpratato come un meccanismo di *foreign key* e come tale traducibile nell'analoga struttura definita in ODL$_{I^3}$.

Prima di passare all'algoritmo vero e proprio di traduzione analizziamo un attimo la sintassi XML dalla quale si cercheranno di estrapolare le informazioni che andranno poi analizzate dal wrapper. Riprendendo la struttura generale del caso precedente ed immaginando le righe di codice di un DTD, il quale prevede questo costrutto, ci si potrebbe trovare ad una serie di righe di codice come le seguenti:

```
<!-- Dichiarazione di Restaurant -->
<!ELEMENT Restaurant (name,...,pers_id,...)>

<!ELEMENT name (#PCDATA)>
<!ATTLIST name r_code ID #REQUIRED>
[...]
<!ELEMENT pers_id EMPTY>
<!ATTLIST pers_id person IDREF #IMPLIED>
[...]


<!-- Dichiarazione di Person -->
<!ELEMENT Person (pers_name,[...])>

<!ELEMENT pers_name EMPTY>
<!ATTLIST pers_name id ID #REQUIRED>
[...]
```

Tralasciando il discorso della *primary key*, peraltro già trattato nel paragrafo precedente e riportato qui per un discorso di completezza, cominciamo con l'evidenziare come per `pers_id` il DTD prevede che non contenga nessun valore in quanto dichiarato dalla parola chiave `EMPTY`. Questa scelta non è detto che debba sempre verificata, ma in questo caso ci aiuta nella comprensione del meccanismo in quanto si evita di introdurre dei valori diversi da quelli che semplicemente servono per la definizione del legame implicato da una *foreign key*.

Passando invece ad analizzare la struttura nella sua generalità si nota subito come non sia evidenziato quale sia la struttura o la primary key a cui l'eventuale foreign key referenzia. Infatti la dichiarazione prevede esclusivamente, come per le tutte le dichiarazioni di *attlist*, la dichiarazione dell'elemento a cui assegnare l'*attlist* stesso, il nome che lo identifica ed una dichiarazione di default caratterizzata dalla struttura BNF:

```
AttDef ::= Name AttType DefaultDecl
```

Questo problema potrebbe essere intuitivamente risolto nel caso venga creato un DTD in cui si tenga presente di questa peculiarità, come fatto nell'esempio riportato sopra, ed in cui il nome di *attlist* è il medesimo della struttura a cui ci si riferisce; diversamente potrebbe diventare problematica l'identificazione. Nel caso poi si conservino queste accortezze, comunque assolutamente ignorate dal punto di vista del W3C e dagli scopi che vuole perseguire per tale linguaggio, non ci si mette ancora dalla parte del sicuro in quanto non è detto che nella struttura o elemento riferito sia presente un solo *attlist* di tipo ID e quindi potrebbe permanere il dubbio di come costruire il legame.

Soluzione a queste problematiche potrebbe venirci dai dati in quanto è chiaro che una volta inserite le informazioni riguardanti le primary key, ed essendo tali univoche, sarà sufficiente ricercare quale è l'*attlist* in cui la mia foreign key va a mappare per potere risolvere la relazione. Ma anche questo potrebbe rivelarsi un metodo non sufficiente in quanto non è detto che un *attlist* di tipo IDREF vada a mappare sempre su di un *attlist* di tipo ID univoco e quindi ci si ritrova ancora al punto di partenza rischiando di creare delle relazioni che poi non sono verificate da tutti i dati.

Permanendo quindi ancora l'incertezza non rimane che far ricadere la scelta al progettista al quale rimane il compito di risolvere la relazione di mapping, sempre che esista, indicando a quale *attribute* di quale interfaccia la forign key è legata.

Ritornando al nostro semplice esempio il quale lo immaginiamo tale da garantirci che i riferimenti di `pers_id` mappino sempre su `id` dell'elemento `pers_name` il wrapper potrebbe procedere con la generazione delle seguenti righe di sorgente ODL_{I^3} :

```
interface Restaurant
```

```
(source semistructured Eating_Source
key name_r_code
foreign_key(person_pers_id) references Person)
{
   attribute string name;
   attribute string name_r_code;
   [...]
   attribute string person;
   attribute string person_pers_id;
   [...]
}

interface Person
(source semistructured Eating_Source
key pers_name_id)
{
   attribute string pers_name;
   attribute string pers_name_id;
   [...]
}
```

dove segnaliamo ancora che la traduzione completa è stata possibile in quanto il progettista è intervenuto a risolvere l'ambiguità.

L'algoritmo fin qui riportato è riferito al solo vincolo `IDREF`, ma siccome per `IDREFS` il concetto di fondo non cambia può quindi essere riportato inalterato. L'unica differenza riguarda i dati che nel primo caso sono formati da una sola stringa, mentre adesso possono essere caratterizzate anche da diverse stringhe (o per la precisione da più nomi) separate da spazi bianchi.

Altri vincoli di validità

I rimanenti due vincoli di validità, caratterizzati dalla parole chiave `ENTITY` ed `ENTITIES` il primo e `NMTOKEN` ed `NMTOKENS` il secondo non comportano delle rilevanze particolari dal punto di vista della traduzione della struttura in quanto si tratta di vincoli che devono essere applicati esclusivamente sui dati e come tali si applica l'algoritmo già visto per i normali *attlist*.

Infatti il primo implica che per l'*attlist* a cui viene assegnato tale vincolo è obbligatorio specificare solamente il nome (o i nomi nel caso plurale) di

entità dichiarate nel DTD.

Il secondo caso invece implica che per ogni *attlist* con tale vincolo debba essere assegnata un token (o più token sempre considerando il caso plurale). Per token in XML prevede un formato riconosciuto anche per altri linguaggi che può essere descritto tramite la forma BNF come:

```
Nmtoken ::= (NameChar)+
```

dove fondamentalemnte il `NameChar` non è altro che una stringa.

5.4.4 Enumerated

Ultimo tipo previsto per gli *attlist* è quello per i dati di tipo enumerated, cioè una lista chiusa di valori i quali possone essere gli unici che possono essere assunti dai dati veri e propri.

Dal punto di vista della traduzione non ci sono delle differenze fondamentali tra XML e ODL_{I^3} in quanto anche nella sintassi di questo secondo linguaggio è prevista una struttra di questo tipo. Trovandosi quindi nel DTD di fronte ad una dichiarazione del tipo:

```
<!ELEMENT person ([...],recapito,[...])>

[...]
<!ELEMENT recapito (#PCDATA)>
<!ATTLIST recapito type (telefono|cellualre|fax|email) 'telefono'>
[...]
```

il wrapper potrebbe quindi generare delle righe di codce come le seguenti:

```
enum recapito_type
{
   'telefono',
   'cellualre',
   'fax',
   'email'
}

interface persona
{
```

```
    [...]
    attribute string recapito;
    attribute recapito_type recapito_type;
}
```

Da segnalare in questo caso che l'artificio adottato, come detto in 5.4, è stato quello di mantenere il legame tra *attlist* ed elemento tramite la costruzione di un nome che riportasse questa informazione.

Nella dichiarazione in XML è prevista anche una informazione supplementare riguardante l'attributo di default nel caso nei dati non venga riportato nessuno dei valori della lista. Questa peculiarità è, come appena detto, caratteristica esclusivametne dei dati, quindi non significativa dal punto di vista della struttura e quindi non traducibile a questo livello.

Oltre ad un tipo enumerato come quello appena analizzato la sintassi XML prevede anche una gli "Attributi di notazione" i cui valori devono

"[...] essere uguali a uno dei nomi di notazione inclusi nella dichiarazione; tutti i nomi di notazione presenti nella dichiarazione devono essere dichiarati."

Un *attlist* di tipo `NOTATION` identifica una notazione dichiarata nel DTD, come specificato nel paragrafo 4.4.4, e che verrà analizzata in dettaglio in 5.6.

5.5 La traduzione delle Entità

Come detto in 4.4.3 tramite queste strutture è possibile il riferimento a porzioni di dati che, per vari motivi, possono venire rappresentati da una parola chiave definita appunto tramite la definizione di una entità.

Dal punto di vista della traduzione ci si potrebbe trovare di fornte a due casi distinti, ma fondamentalmente non trattatti dal wrapper per motivi diversi. Primo di questo e il caso in cui l'entità descriva una porzioni di dati il che implica che non esite la necessita di riportare alcuna informazione supplementare per la struttura.

Si ricade in un caso leggermente più complicato quando nella definizione dell'entità viene riportata una parte della struttura dei dati che quindi dovrà essere tradotto ed utilizzate in ODL$_{I^3}$. Un esempio di questo tipo è dato

dalle seguenti righe di codice prese da tre file separati e filtrate in modo da riportare solamente la struttura degli elementi e non tutte le informazioni necessarie per la completa descrizione della struttura prevista dall'autore:

```
// catalogo_stellare23.xml

<!DOCTYPE cielo SYSTEM "catalogo_star3.dtd"
 [
  <!ENTITY % STELLA2 SYSTEM "catalogo_star2.dtd" >
  <!ELEMENT cielo (costellazione*, galassia*)+ >
  %STELLA2;
]>
[...]


// catalogo_star2.dtd

<!ELEMENT costellazione (stella+, segmento?)*>

<!ELEMENT stella ( coordinate, magnitudine ) >
<!ELEMENT coordinate EMPTY >
<!ELEMENT magnitudine EMPTY >
[...]

// catalogo_star3.dtd
<!ELEMENT segmento EMPTY >
<!ELEMENT galassia (#PCDATA) >
[...]
```

Un parser che andasse a leggere il file "catalogo_stellare23.xml" ricostruirebbe il DTD e l'albero associato presentando come risultato una struttura come quella rappresentata nella figura 5.1 che è stata generata da un browser per XML.

In realtà anche in questo caso il wrapper non dovrebbe fare nulla, ma per una motivazione fondamentalmente diversa. Infatti il peso dell'operzione di ricostruzione ricade totalmente sul parser che legge il DTD ed il quale si dovrebbe poi occuparsi della creazione dell'albero completo da passare all'applicazione client, nel nostro caso al wrapper, la quale non deve a svolgere nessuna operazioni particolari consntendo di operare allo stesso modo di quanto già visto in precedenza.

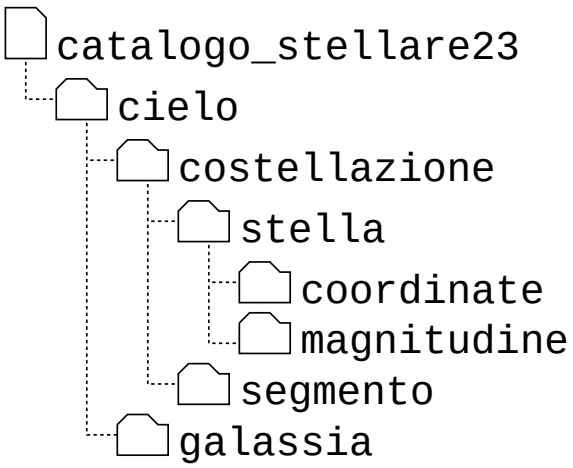

Figura 5.1: Rappresentazione della struttura del DTD del file catalgo_stellare23.xml

5.6 La traduzione delle Notazioni

Ultima dichiarazione possibile per un DTD è quella delle notazioni, caratterizzate come detto in 4.4.4, dalla parola chiave `NOTATION`.

In questo caso ci si trova di fronte ad un costruttuto applicabile principalmente, per non dire esclusivametne, ai dati e non alla struttura in quanto è tramite questa dichiarazione che XML identifica un eventuale formato degli elementi oppure permette l'aggancio con l'applicazione a cui una istruzione di elaborazione è indirizzata.

Dal punto di vista del wrapper l'unica cosa di cui ci si potrebbe occupare sembra essere il formato degli elementi, ma in realtà le informazioni che vengono fornite non sono sufficienti per garantire una identificazione univoca del tipo come avviene per ODL$_{I^3}$. Infatti per tipo di dati non si intendono i tipi atomici riconosciuti in diversi linguaggi (stringhe, interi, boolean, ecc.), piuttosto ci si riferisice ai formati nella quale l'informazione è conservata. Ad esempio un caso classico sono le definzioni dei formati in cui è conservata una immagine, quindi viene riporteta una notazione nel quale si definisce che sono archiviate in un formato specfico (classici sono i formati GIF o JPEG) e viene anche indicata l'applicazione client (in questo caso il lettore di immagini) che permette la visualizzaizone dell'immagine stessa. Come si vede questo, oltre ad essere caratteristico dei dati e della struttura, non ha nes-

suna corrsipondenza con delle strutture analoghe in ODL_{I^3} e quindi non è possibile pensare alcun algoritmo di traduzione.

5.7 Tabella riassuntiva

Riassumiamo gli algoritmi di traduzione illustrati finora in una tabella conclusiva che mostra nella colonna a sinistra tutte le possibili combinazioni di righe che si possono incontrare in un DTD XML, mentre sulla destra viene riportato, ove possibile, la corrispondente traduzione il ODL_{I^3} .

Tale tabella sarà quella sulla quale si baseranno gli algoritmi del parser.

DTD XML	**ODL_{I^3}**

<!ELEMENT [name] [list]>	interface [name]
	(source semistructured ...)
	{
	[...]
	}

<!ELEMENT [name] (#PCDATA)>	attribute string [name];

<!ELEMENT [name] ANY>	typedef any
	{
	string strvalue;
	}
	
	attribute any [name];

<!ELEMENT [name] EMPTY>	attribute boolean [name];

<!ELEMENT [name]
([l_1],...,[l_n])>
<!ELEMENT [l_x]
([sl_1],...,[sl_m])>
	interface name
	(source semistructured ...)
	{
	[...]
	attribute [l_x] [l_x];
	[...]
	}
	
	interface [l_x]
	(source semistructured ...)
	{
	[...]
	}

<!ELEMENT [name] ([l$_1$],...,[l$_x$]?,...,[l$_n$])>	attribute string [l$_x$]*;
<!ELEMENT [name] ([l$_1$],...,[l$_x$]*,...,[l$_n$])> <!ELEMENT [l$_x$] (#PCDATA)>	attribute set<string> [l$_x$];
<!ELEMENT [name] ([l$_1$],...,[l$_x$]+,...,[l$_n$])> <!ELEMENT [l$_x$] (#PCDATA)>	attribute set<string> [l$_x$];

<!ELEMENT [name]
(#PCDATA|[other])*>
<!ELEMENT [other] [list]>
	interface [name]
	(source semistructured ...)
	{
	[...]
	};
	union
	{
	string
	};

<!ELEMENT [father]
([childA]|[childB])>
<!ELEMENT [childA] #PCDATA>
<!ELEMENT [childB] (...)>
	interface [father]
	(source semistructured ...)
	{
	attribute [father]_union_1
	[father]_union_1;
	}
	
	typedef [father]_union_1
	{
	string [childA];
	}
	union
	{
	[...]
	}

<!ELEMENT [name] (#PCDATA)> <!ATTLIST [name] [att$_1$] CDATA '[val1]' [att$_2$] CDATA '[val2]'>	attribute string [name]; attribute string [name]_[att$_1$]*; attribute string [name]_[att$_2$]*;
<!ATTLIST [name] [att$_1$] CDATA #REQUIRED>	attribute string [name]_[att$_1$];
<!ATTLIST [name] [att$_1$] CDATA #FIXED [value]>	const string [name]_[att$_1$] = '[value]';

`<!ATTLIST [name] [att`$_1$`] CDATA #IMPLIED>`	Questo tipo di attributo ha delle implicanze solamente a livello di dati e non di struttura e come tale viene considerato come un attributo "normale" cioè sarà tradotto come `attribute string [name]_[att`$_1$`]*;`

`<!ATTLIST [name] [att`$_1$`] ID #REQUIRED>`	In questo caso è necessario l'intervento del progettista che indichi se questo attributo debba essere effetivamente considerato come una *primary key* oppure no
	
	`interface ...`
	`key[name]_[att`$_1$`]`
	`)`
	`{`
	`attribute string [name]_[att`$_1$`];`
	`}`
	
	Questa traduzione è stata possibile dal momneto che il progettista ha confermato che si tratta effettivamente di una *primary key*.

`<!ATTLIST [name] [att`$_1$`] IDREF #IMPLIED>`
`<!ATTLIST [name] [att`$_1$`] IDREFS #IMPLIED>`
	`interface ...`
	`(...`
	`foreign key ([name]_[att`$_1$`])`
	`references [interface])`
	`{`
	`[...]`
	`attribute string [name]_[att`$_1$`]`
	`[...]`
	`}`
	
	Dove `[interface]` rappresenta il nome dell'interfaccia ODL$_{I^3}$ a cui il progettista ha indicato che tale *foreign key* debba referenziare.

<!ATTLIST [name] [att$_1$] ENTITY> <!ATTLIST [name] [att$_1$] ENTITIES>	Non necessita di una conversione differente a quanto già visto in quanto riguarda esclusivamente i dati. attribute string [name]_[att$_1$]*;
<!ATTLIST [name] [att$_1$] NMTOKEN> <!ATTLIST [name] [att$_1$] NMTOKENS>	Non necessita di una conversione differente a quanto già visto in quanto riguarda esclusivamente i dati. attribute string [name]_[att$_1$]*;

<!ATTLIST [name] [att$_1$]
([l$_1$]|...|[l$_n$]) '[l$_x$]'>
	enum [name]_[att$_1$]
	{
	'[l$_1$]',
	[...]
	'[l$_n$]'
	}
	
	[...]
	
	attribute [name]_[att$_1$]
	[name]_[att$_1$];

<!ENTITY [name] [unparsed]>	Non viene tradotta in quanto riguarda esclusivamete i dati.
<!ENTITY [name] [parsed]>	Non ha una traduzione diretta in ODL$_{I^3}$, ma il parser deve occuparsi di ricostruire l'eventuale DTD esterno che contiene l'entità e tradurlo con le regole finora illustrate.
<!NOTATION ...>	Non ha un corrispondente in ODL$_{I^3}$ in quanto serve esclusivamente per la gestione dei dati in XML.

Capitolo 6

Il wrapper XML/ODL$_{I^3}$: il prototipo software

In questo capitolo andiamo ad analizzare nel dettaglio come funziona il Wrapper XML/ODL$_{I^3}$ nella sua implementazione pratica illustrando quali sono le fasi nelle quali il prototipo software si può scomporre e come queste sono state implementate.

Fondamentalmente il prototipo è costituito da 4 componenti ben identificati e con loro funzioni precise:

1. Estrazione del DTD da un file XML
2. Parsing del DTD e generazione della corrispondente struttura in memoria
3. Definizione delle eventuali chiavi (primary e foreign)
4. Generazione del corrispondente codice ODL$_{I^3}$

Il tutto può essere agevolemente rappresentato nella figura 6.1 che da' una visione di insieme di queste fasi necessarie per implementare l'algortimo di traduzione.

Durante l'intero processo il wrapper deve svolgere un duplice lavoro: da una parte si occupa di fare tutte le operazioni necessarie per la generazione di uno schema ODL$_{I^3}$ seguendo le regole anplimante illustrate nel capitolo 5, contemporanemanete deve però prevedere una interfaccia nella quale il progettista possa essere messo a conoscenza di eventuali problemi che vengono riscontrati al momento del parsing della sorgente oltre che ad essere

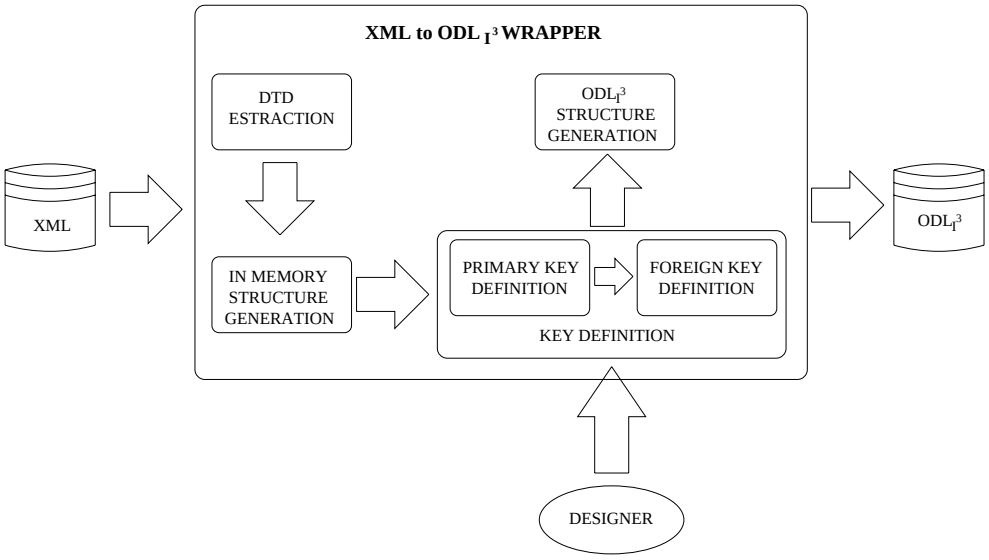

Figura 6.1: Il wrapper XML

nella condizione di poter intervenire dove richiesto per risolvere le ambiguità ampliamente descritte nei paragrafi 5.4.3 e 5.4.3.

6.1 Estrazione DTD e generazione della struttura in memoria

Come evidenziato nella figura 6.1 il wrapper necessita in ingresso un file XML il quale, come detto nel paragrafo 4.1.4, può essere *Valid* o semplicemente *Well-formed*.

Nel primo caso, per i motivi illustrati dettagliatamente nel paragrafo 5.1.1, non è possibile effettuare la traduzione e quindi non vengono iniziate le operazioni di parsing sulla sorgente, ma viene piuttosto generato un messaggio di errore in modo che il progettista sia nella condizione di operare sulla sorgente in modo da renderla Valida oppure di procedere con altre operazioni caratteristiche del sistema MOMIS.

Nel caso invece il file XML sia *Valid* oppure venga fornito in ingresso direttamente il DTD è possibile proseguire con le operazioni di parsing. In particolare si effettuerà la lettura del file di input richiamando l'apposita

funzione `startParse` che, nel caso siano presenti degli eventuali riferimenti esterni descritti dalle entità parsed, sarà iterata tante volte quante necessario. Il risultato finale è quello di restituire quindi l'intera struttura del DTD come l'autore l'ha effettivamente pensata e realizzata.

Per questa fase di inserimento del nome della sorgente è stata prevista una maschera apposita nella quale il progettista è messo nella condizione di indicare il file da mandare in input nella fase di parsing. Il tutto è rappresentato nella figura 6.2

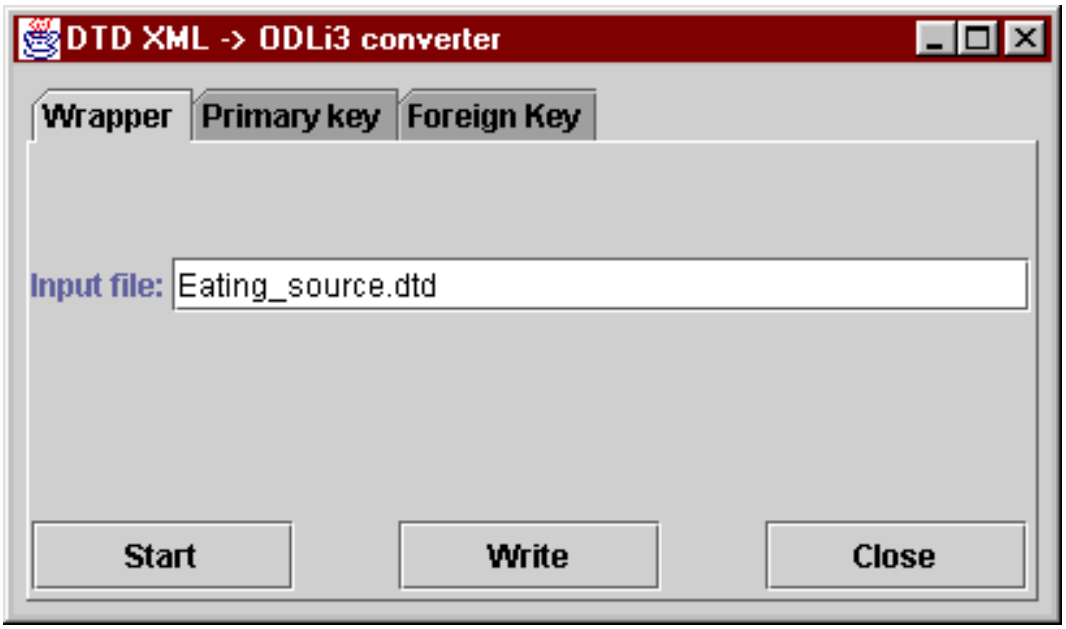

Figura 6.2: Screenshot del primo pannello dell'applicazione

nella quale la barra dei bottoni in basso permette al progettista di comandare l'inizio della fase di parsing vera e propria (bottone *Start*) e di scrivere, quando possibile, la corrispondente traduzione in ODL_{I^3} non appena tutte le informazioni riguardanti le eventuali primary key e foreign key sono state inserite (bottone *Write*).

Le funzioni di parsing si basano su di un presupposto di Well-formedness del DTD che ci si appresta a leggere, questo anche perché lo scopo di tali funzioni non è quello di controllore la sintassi XML (funzionalità prevista tra l'altro dai più recenti editors e parser attualmente reperibili, come indicato nei paragrafi 4.10.1 e 4.10.2) quanto piuttosto quello di operare su dei dati la

cui struttura è definita ed univoca. Nel caso quindi l'input non sia un Well-formed ci si limita a segnalare un errore nella grammatica e si sospendono tutte le operazioni di lettura.

I dati letti dovranno essere contemporanemente archiviati in una struttura in memoria appositamente creata e che verrà descritta nel dettaglio nel paragrafo 6.1.2. Tale struttura sarà poi utilizzata come base dalla quale reperire le informazioni da riportare il ODL$_{I^3}$.

Il W3C ha messo a disposizione un documento nel quale viene specificato il DOM (**D**ocument **O**bject **M**odel) [40], cioè un insieme di specifiche indipendenti dalla piattaforma e dal linguaggio utilizzati che permettono ai programmi ad agli scripts di accedere e modificare in maniera dinamica i contenuti, lo stile e la struttura di un documento. Allo stato attuale, denominato *Level 1*, è possibile definire un set di oggetti per rappresentare documenti sia HTML che XML, definire il modo di combinare i vari oggetti e provvedere alla definizione di una interfaccia per l'inserimento e la manipolazione dei documenti stessi.

In tale documento non è però prevista nessuna definizione riguardante la struttura di un file XML e quindi non viene fornita nessuna specifica formale di come rappresentare un DTD XML. Questa lacuna si protrae, con l'eccessione di alcune indicazioni di massima, anche nella versione successiva (denominata *DOM Level 2* [41] ad attualmente ancora in fase di "Working-Draft"), mentre dovrebbe essere definitivamente risolta con l'avvento del DOM Level 3 [42].

Partendo da questo pressupposto di e considerando che per il futuro non è possibile avere la certezza che le versioni successive di XML si basino incondizionatamente su quanto stabilito nella raccomandazione del 10 febbraio 1998 [3] si è deciso di colmare questa lacuna creando ex-novo una struttura sufficientemente semplice e snella la quale si possa prestare esclusivamente agli scopi del presente parser e non abbia l'intento di rappresentare interamente un DTD in tutte le sue particolarità.

Quanto detto a ancor più valido considerando che la direzione attuale verso la quale il W3C, o comunque l'intera comunità degli utilizzatori di XML, si sta' orientiando è quella della definizione anche di uno *Schema* [43, 44, 45] che dovrebbe integrare il DTD ed apportare nuove informazioni sulla struttura, quali, ad esempio, quelle riguardanti il tipo di un dato.

Occorre infine sottolineare che il DOM Level 3 ha, come detto, tra i suoi obiettivi anche quello di permettere la descrizione, oltre che dei dati, anche della struttura nel quale questi saranno contenuti. Conseguenza diretta di questo riguarda i software futuri, già nelle versioni successive dei parser attualmenti più diffusi visto che esistono le prime avvisaglie, conterranno delle funzioni e delle API in grado di getire e modificare la struttura stessa. Questo sviluppo degli attuali parser farà probabilmente si' che tutta la prima parte del presente Wrapper che si occupa del parsing diventi obsoleta o comunque sia conveniente rimpiazzarla con conseguenze a cascata anche sul resto del prototipo.

6.1.1 Le operazioni di lettura

Per quanto riguarda gli algortimi di parsing vero e prorio del DTD ci si è avvalsi della tecnologia fornita dal BYacc, le cui caratteristiche fondamentali sono brevemente descritte nel paragrafo 7.3, la quale consente, data una grammatica BNF che descrive una sintassi, di procedere con una certa agilità nell'acquisizione dei dati.

Nelle specifiche del W3C per il linguaggio XML, infatti, oltre alla descrizione della grammatica utilizzata dalla versione 1.0 del linguaggio stesso erano incluse anche diverse sezioni che descrivono, appunto in forma BNF, le regole di costruzione di un DTD. Raggruppando, quindi, tutte queste informazioni (riportate anche nell'appendice A) è stato possibile creare il file sorgente per BYacc.

Siccome, però, per i nostri scopi di wrapping non è necessario controllare ed includere l'intera grammtica, tutte quelle parti non necessarie ai nostri fini sono stata ignorate il che ha consentito di semplificare notevolemente la forma e procedere correttamente con il resto delle operazioni di parsing.

Ovviamente la parte di parsing è stata integrata, come previsto dal Byacc, con tutte quelle funzioni necessarie per l'archiviazione delle informazioni lette in una struttura in memoria apposita da utilizzare per le successive operazioni di wrapping.

6.1.2 La struttura in memoria

Andiamo adesso a descrivere brevemente la struttura nel quale il parser archivia le informazioni lette dal DTD illustrando le caratterisitiche peculiari e come queste potranno essere integrate dal progettista al momento della definizione delle eventuali *primary key* e *foreign key* che si desidera inserire.

L'idea di fondo è quella di costruire una struttura ad albero nel quale riconoscere immediatamente le posizioni dei vari nodi e quindi identificare con una certà facilità se si tratta di un nodo intermedio (caratterizzato dal fatto di possedere dei nodi figlio) oppure di un nodo terminale.

Premettiamo prima di tutto che nell'implementazione di tale struttura ci si è avvalsi esclusivamente degli strumenti standard che il linguaggio utilizzato, Java ver 1.2.2 [46, 47] (descritto meglio in 7.2) fornisce nel pacchetto di implementazione standard.

Passando all'analisi tecnica sappiamo che l'intera struttura di un file DTD si basa sul concetto di *element* e di conseguenza era logico prevedere la creazione di una classe apposita, denominata appunto `element`, in grado di essere considerata anch'essa come la nostra base per l'archivazione dei dati in memoria, e la quale è stata predisposta per contenere le caratteristiche nencessarie di questo oggetto XML.

Come già detto in precedenza la necessità di non avere a disposizione tutte le informazioni caratteristiche di un elemento XML per l'algoritmo di traduzione in linguaggio ODL$_{I^3}$ ha fatto si' che questa classe possa risultare incopleta da un punto di vista sintattico, ma esauriente da un punto di vista operativo. Infatti il parser stesso, nel momento in cui incotra delle informazioni non rilevanti agli scopi preposti dal wrapper, non prevede alcun algortimo per l'archiviazione di tali informazioni ed a maggior ragione non sono necessari gli appositi spazi in memoria in cui archiviarle.

Altro oggetto presente in XML è l'*attlist* e quindi anche nella nostra struttura è stata prevista una classe apposita in cui archiviare le informazioni ricavate da tali oggetti eventuali presenti. Come nel caso degli elementi, anche per questa classe non è stato ritenuto necessario descrivere completamente tutte le peculiartià previste dalla grammatica XML, ma piuttosto sono state riportate solamente le informazioni ancora una volta necessarie per lo svolgiemento delle operazioni di wrapping.

A differenza di quanto visto, però, per gli elementi in questa classe sono stati introdotti alcune informazioni avulse alla grammatica XML, ma che servono per la definizione da parte del progettista delle eventuali *primary key* e *foreign key* con il relativo referenziamento. Queste informazioni potranno essere acquisite solamente successivamente alla fase di parsing tramite una interfaccia che metta a conoscenza il progettista stesso delle alternative possibili e lo guidi nella definizione di tali informazioni.

Come già visto per gli elementi, anche per gli *attlist* il procedimento di creazione dell'oggetto è simile in quanto consiste sempre con la definizione di un nuovo oggetto per ogni attributo letto. La differenza consiste invece nel fatto che in questo caso non si necessita di archiviare queste informazioni in un apposito contenititore, ma è sufficiente aggiungere i riferimenti agli oggetti creati ad ogni elemento a cui l'attributo referenzia completando quindi l'albero in memoria.

Passiamo ora a vedere come è stata implementata la struttura stessa e come sono reperibili le informazioni in essa contenuta.

Riprendendo, per quanto possibile, l'idea di fondo che il documento sul DOM Level 1 [40] suggerisce abbiamo creato un contenitore nel quale raggruppare tutti i Nodi della struttura, e precisamente è stata prevista una `HashTable` nella quale per ogni elemento letto viene generato un nuovo nodo a cui si assegnano di volta in volta le informazioni che necessitano e la cui descrizione si incontra procedendo nella fase di parsing. Chiaramente nel caso di Nodi padre si specifica anche i riferimenti ai relativi figli fino al raggiungimento della fine del DTD che corrisponde anche al termine della definizione della struttura. Il risultato finale è quindi quello di aver definito l'albero in memoria il quale è pronto per le succisave operazioni.

6.2 Definizione delle chiavi

Questa fase può essere a suo volta essere suddivisa in altre due parti distinte e conseguenti:

- Individuazione delle eventuali *primary key*
- Individuazione delle eventuali *foreign key* e traduzione dei rispettivi

reference per sviluppare la relazione implicata in questo tipo di struttura

Per quanto già detto nel paragrafo 5.4.3 a questo punto è necessario l'intervento del progettista che dovrà indicare, selezionando in un elenco generato dal wrapper, quali sono le eventuali *primary key* tra quelle possibili lette dal DTD sorgente.

Per quanto riguarda invece la definizione delle *foreign key* diciamo che l'algoritmo applicato è molto simile a quanto visto in precedenza, infatti verrà nuovamente proposto al progettista un elenco generato dal Wrapper stesso con le informazioni lette dal DTD riguardante questa volta gli oggetti con i vincoli di validità IDREF/IDREFS nel quale selezionare quali *attlist* fanno parte di una relazione. Una informazione aggiuntiva rispetto a prima è quella riguardante gli estremi della relazione che dovranno essere anch'essi scelti esclusivamente tra quelli presente nella lista delle eventuali *primary key* definite precedentemente.

A questa seconda sottofase chiaramente si potrà accedere solamente nel caso esistano nel DTD dei vincoli di validità di tipo IDREF/IDREFS ed il progettista abbia definito delle chiavi primarie. Diversamente non avrebbe alcun senso pensare al meccanismo delle relazioni, nel primo caso poiché si andrebbe a trattare su una lista vuota, nel secondo caso invece perchè mancando le chiavi primarie viene a mancare un estremo necessario per la definizione della relazione stessa.

Entrambe le informazioni acquisite dovranno essere memorizzate dal wrapper per poter essere successivamente utilizzate e la via scelta è quella di archiviarle nella struttura in memoria che rispecchia il DTD ed illustrata brevemente in 6.1.2. Da un punto di vista puramente formale si potrebbe obiettare che queste informazioni non sono caratteristiche di un DTD e quindi dovrebbero trovare posto altrove, ma da un punto di vista dell'ottimizzazione questa scelta non limita assolutamente i nostri scopi, piuttosto potrebbe essere considerato come un completamento delle informazioni lette dal DTD stesso.

Tale memorizzazzione avviene nella forma di appositi variabili private della classe *attribute*. In particolare nel caso di una primary key è sufficiente un flag che indichi le intenzione del progettista, mentre nel caso di una foreign key diventa necessario fare riferimento anche all'attributo a cui si referenzia.

Da un punto di vista implementativo si è scelta la strada di proporre, nel caso siano presenti queste ambiguità, al progettista una finestra costituita da due pannelli (figure 6.3 6.4) e rappresentanti le due sottofasi. Chiaramente il passaggio al secondo pannello dovrà essere abilitata solamente nel caso le condizione citate precedentemente siano verificate.

Il primo pannello, riportato in figura 6.3, è composto da due campi lista che rappresentano rispettivamente quello di destra (*Candidate Primary key*) tutte le possibili chiavi primarie individuate durante la fase di parsing, mentre quello di destra (*Selected Primary key*) riporta i valori che il progettista ha effettivamente selezionato. La fase di selezione/deslezione è svolta tramite la pressione dei due bottoni (*Add* e *Remove*) posti tra le due liste.

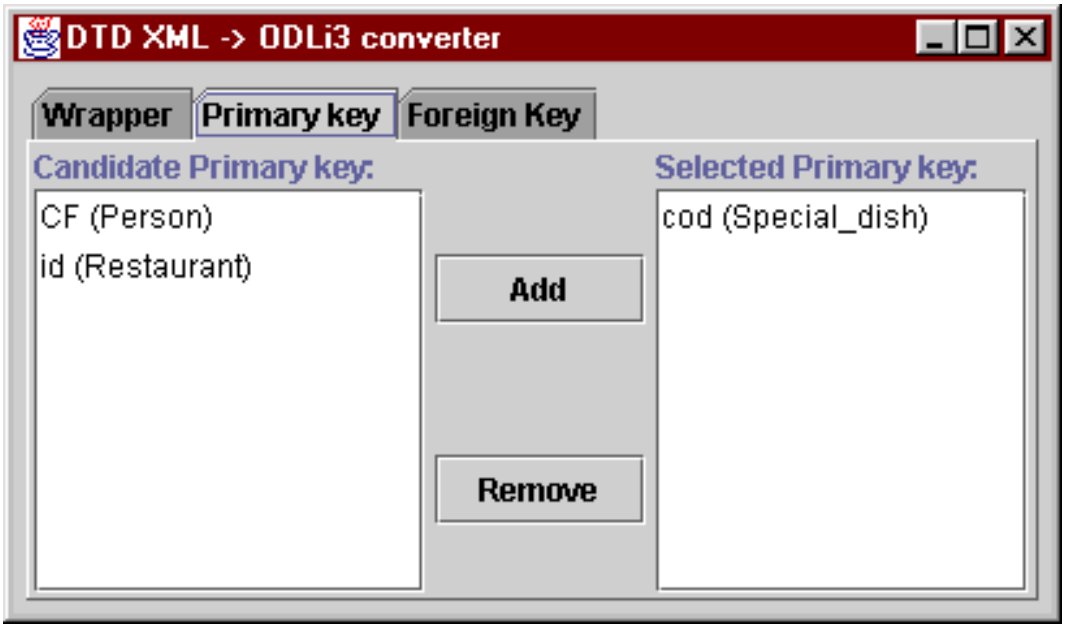

Figura 6.3: Screenshot del secondo pannello dell'applicazione

Il secondo pannello, riportato in figura 6.4, è anch'esso formato da due campi lista che rappresentano rispettivamente a sinistra (campo *Candidate foreing key*) le possibili foreign key presenti nella sorgente ed a destra (campo *Primary key*) le chiavi primarie selezionate nel pannello precedente.

Tramite questo caso il meccanismo viene data la possibilità al progettista di evidenziare due valori i quali saranno messi in relazione tramite la pressione

del bottone *Reference to*, il che significa dichiarare il valore di sinistra come foreign key effettivamente presente e referenziarla con la chiave primaria evidenziata a sinistra. Nel caso a sinistra sia stata evidenziata la chiave `<none>`, che è una chiave fittizia utilizzata dal wrapper, significa dichiarare il valore di destra come non identificante nessuna foreign key e come tale viene tradotto come semplice *attribute* in ODL$_{I^3}$.

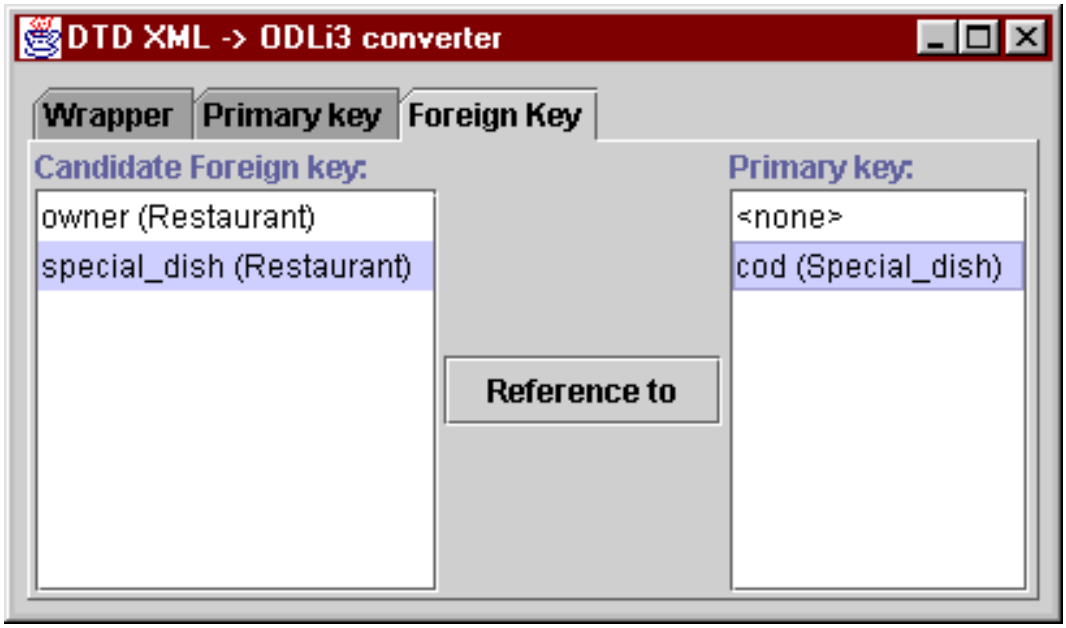

Figura 6.4: Screenshot del terzo pannello dell'applicazione

6.3 Generazione del codice ODL$_{I3}$

Ultima fase prevista per il wrapper è quella della traduzione in formato ODL$_{I^3}$ delle informazioni lette ed acquisite. Gli algoritmi di traduzione sono stati ampliamente discussi nel capitolo 5 a cui si rimanda per ottenere eventuali dettagli a riguardo.

A differenza di quanto visto in precedenza questa fase è del tutto trasparente al progettista in quanto non si necessita di nessun'altra informazione supplementare ed il risultato ottenuto dovrà essere passato ai livelli superiori di MOMIS in modo da poter arrivare, passando per tutti i moduli intermedi illustrati in 2.2, ad includere le informazioni acquisite nel Global Schema.

Per quanto riguarda l'aspetto implementativo si parte dalla struttura in memoria, che ricordiamo essere un elenco di tutti gli elementi previsti dal DTD, e la si scorre individuando tutti gli elementi presenti con le loro caratteristiche e traducendoli contemporaneamente nelle corrispondenti strutture ODL_{I^3} .

Siccome tale struttura, come detto in precedenza, rappresenta un albero allora lo scorrimento dei vari elementi (immaginati come i vari nodi) corrisponde allo scorrimento di tutti i rami presenti. Chiaramente l'algoritmo si soffermerà maggiormente sui nodi interni, i quali sono quelli che portano un maggior numero di informazioni oltre che ad essere maggiormente significativi per quanto riguarda l'algoritmo di traduzione.

Le informazioni portate dagli attributi saranno tradotte nel momento in cui si incontra l'elemento a cui si riferiscono, e che refernzia i vari oggetti di questo tipo, e quindi non è necessario ripercorrere nuovamente l'intera struttura.

6.4 Problemi riscontrati

In questo paragrafo saranno illustrate tutte le problematiche che sono intervenute al momento della scrittura sia della fase di parsing che della fase di wrapping vera e propria.

Verranno quindi di seguito illustrati i dettagli di tali problematiche ed eventualmente proposte delle metodologie implementative per aggirare tali problemi in modo da essere in grado di sviluppare, ove non già previsto nella versione attuale del software stesso, le necessarie modifiche o completamenti.

6.4.1 Rischi di duplicazioni dei nomi

Un primo problema deriva dal fatto che i nomi degli *attribute* sono generati tramite un algoritmo e quindi non possono essere ripresi inalterati dal DTD di origine. Nel caso degli *attlist*, infatti, il nome (come ampiamente visto in 5.4) è la composizione tra il nome dell'*attlist* stesso ed il nome dell'elemento a cui si riferisce. Tale composizione potrebbe però portare alla creazione di un nome già esistente e quindi creare una duplicazione indesiderata ed in-

gestibile in ODL$_{I^3}$.

Prendendo infatti ad esempio il codice XML che abbia tra le sue righe le seguenti:

```
<!ELEMENT address #PCDATA>
<!ATTLIST address type #REQUIRED>

<!ELEMENT address_type (street, zipcode?, city)>
```

l'algoritmo comporterebbe la creazione di due *attribute* denominati entrambi `address_type`, ma con significati differenti oltre che, addirittura, di due tipi distinti. Infatti ci si troverebbe a trattare allo stesso modo una stringa ed una struttura complessa con evidenti problemi di risoluzione delle ambiguità.

Questo chiaramente è solamente uno degli esempi possibili di ambiguità, altri casi differenti potrebbero verificarsi riportando però sempre il problema ad una natura comune. D'altro canto occorre dire che questa è una situazione si' possibile, ma sufficientemente remota per poter essere considerata come critica per l'algoritmo di traduzione previsto.

Una possibile soluzione dovrebbe passare da un controllo preventivo di tutti nomi previsti in ODL$_{I^3}$ alla ricerca di eventuali ambiguità. Nel caso se ne individuasse qualcuna è sufficnete prevedere un meccanismo di modifica del nome generato in modo di individuare un nuovo nome non ambiguo.

Questo sistema porterebbe però alla perdita delle univocità di traduzione insite nel nome stesso e quindi dovrebbe essere prevista la creazione di una meccansimo, che potrebbe benissimo essere rappresentato da una Mapping Table, nella quale indicare i nomi presenti nel DTD XML ed i corrispondenti nomi univoci generati per la strututra ODL$_{I^3}$. Chiaramente tale Mapping Table dovrebbe poi essere sempre messa a disposizione dal Wrapper in modo da risolvere le ambiguità durante le interrogazioni fatte al sorgente dai corrispondenti moduli del Query Manager.

6.4.2 Choice element

Come abbiamo visto nel paragrafo 5.3.4 il caso degli elementi di tipo *choice* implica un algoritmo di traduzione abbastanza macchinoso e particolarmente dettagliato il che' fa si che sia stata presa la decisione, per questa versione

del Wrapper, di ignorare tale struttura prevista dalla sintassi di un DTD e di rimandare l'eventuale implementazione nelle eventuali revisioni successive.

Il problema non ricade tanto nella fase di traduzione della struttura, peraltro già illustrato in precedenza, anche se macchinoso e poco naturale, quanto piuttosto nella fase di interrogazione da parte dei componenti del Query Manager proposti alla richiesta di informazioni sulla sorgente XML.

Infatti l'introduzione di alcune strutture intermedie, come visto necessarie per una traduzione completa e non ambigua ma non previste nella sorgente XML, crea dei problemi nella ricostruzione del nome in "dot notation", caratteristica di una rappresentazione ad oggetti quale quella del linguaggio ODL_{I^3}, e ricevute dal Global Schema al momento del passaggio della query stessa.

Le possibili soluzioni proposte potrebbero essere due, operanti su due fronti completamente opposti: la modifica delle regole grammaticali di ODL_{I^3} oppure il mantenimento della corrispondenza tra XML ed ODL_{I^3} con un meccanismo diverso da quello del nome.

La prima strada è sicuramente quella più dispendiosa in quanto implica la modifica di regole che coinvolgono non solo il presente modulo ma anche diverse parti del sistema MOMIS con le ovvie conseguenze dovute a cambiamenti di così ampia portata. Le modifiche necessarie consisterebbero nell'introdurre il costrutto `union` anche per un singolo attributo piuttosto che per un intero tipo oppure una interfaccia.

In questo modo ci si riporta ad una maggiore somiglianza tra le strutture XML e le strutture ODL_{I^3} e quindi il meccanismo di traduzione si automatizzerebbe parecchio e si ridurrebbero al minimo le ambiguità.

Mantenendo invece inalterata la sintassi di ODL_{I^3} occorre allora introdurre un meccanismo che mantenga la corrispondenza corretta ed univoca tra i due linguaggi. Anche in questo caso allora occorrerebbe introdurre un meccanismo di Mapping Table, molto simile a quanto visto in precedenza, a cui il Wrapper dovrà accedere costantemente e che ha il compito di risolvere le ambiguità al momento delle interrogazioni da parte delle query.

A differenza del caso precedente, però, l'ambiguità non dipende semplicemente dal nome e quindi non è più sufficiente mantenere una sorta di alias tra gli oggetti dei due linguaggi. In questo caso il problema consite nel valutare

attentamente la richiesta che viene posta dal Wrapper analizzando la struttura ODL$_{I^3}$ passata, dopo di che occorre ricostruire (o più correttamente si potrebbe dire integrare) la struttura ODL$_{I^3}$ corrispondente al DTD XML ed infine operare sulla sorgente originaria in formato XML.

Ancora una volta, quindi, si tratta di una algoritmo fattibile da un punto di vista teorico, ma allo stempo estramente macchinoso e dispendioso oltre che ad alto rischio di errore.

Una terza soluzione, non considerata in partenza in quanto non ritenuta effettivamente risolutiva del problema, coinvolgeva l'algoritmo di traduzione. L'idea ed era quella di creare tante strutture diverse, messe in alternativa tra loro tramite il costrutto `union`, quanti erano gli elementi che caratterizzavano una *choice*.

Quest'ultima soluzione è stata però scartata da subito in quanto potrebbe rivelarsi ottimale nel caso la scelta coinvolga solamente un numero limitato di elementi, nel caso però il numero di elementi sia considerevole (ed in XML non vi è un limite alle opzioni possibili per un elemento di tipo *choice*) oppure si preveda una struttura su più livelli le combinazioni possibili aumentano in modo esponenziale e quindi anche le eventuali strutture tradotte sarebbero un numero difficilmente gestibile.

Capitolo 7

Strumenti utilizzati

In questo capitolo saranno brevente illustrati gli strumenti che sono stati utilizzati per lo sviluppo del software o comunque per tutte quelle operazioni che riguardano il trattamento dei file sorgenti e dei file XML necessari per i test o semplicemente per il controllo delle sintassi.

Occorre premettere che l'ambiente di sviluppo scelto è quello Microsoft e precisamente il sistema operativo Windows 9x. Questa scelta è stata fatta sia per una maggiore dimestichezza nelle operazioni oltre che per la possibilità di scegliere su un panorama software leggermente più ampio per quanto riguarda i prodotti a disposizione per il supporto allo sviluppo.

Il fatto di scegliere una piattaforma di partenza piuttosto che un'altra non preclude assolutamente la interoperabilità del software proposto poiché tutti gli strumenti di sviluppo a disposizione possiedono delle versioni multipiattaforma che non vincolano l'utilizzo del software creato. Oltre a questo il software stesso è stato creato con un linguaggio multipiattaforma per eccellenza, come puè essere il Java [46], appunto per non vincolare il risultato ad uno specifico sistema operativo.

7.1 XML Reader/Editor

Come detto nel paragrafo 4.10 esistono diverse tipologie di software che servono per il trattamento dei dati archiviati in formato XML. In questo paragrafo descriviamo brevemente gli strumenti che sono stati utilizzati per l'editazione e la lettura dei file XML che sono serviti come definizione dell'input

per il wrapper.

A differenza di quanto detto precedentemente riguardo all'interoperabilità su diverse piattaforme, questi prodotti sono (per la maggior parte di essi) strettamente legati al sistema operativo su cui dovranno girare. Questo non rappresenta però un limite dal punto di vista dello sviluppo, infatti il panorama software attuale copre tutte le piattaforme più diffuse e si tratta comunque di una situazione in espansione man mano che il linguaggio XML si difonde nella comunità informatica.

7.1.1 MS Internet Explorer 5.0

Il prodotto sviluppato dalla Microsoft per la navigazione in Internet è attualmente l'unico a disposizione sul mercato che consenta, oltre che alla rappresentazione dei formati "storici" di internet (HTML, DHTML, ASP, ecc.), anche la gestione e la rappresentazione dei dati in formato XML.

Per quanto riguarda la rappresentazione MSIE5 fornisce sia la possibilità di visualizzare i soli dati (nel caso non abbiano un stile di rappresentazione associato) sia di rappresentare i dati formattatti attraverso i formati XSL e CSS.

Nel primo caso produce una struttura ad albero collassabile nella quale à data la possibilità di visualizzare solamente alcuni rami dell'albero stesso ed il livello di dettaglio che si desidera. Un esempio di questa rappresentazione è data dalla figura 7.1 dove alcuni rami sono stati espansi (caratterizzati dal segno - sulla sinistra), mentre altri sono stati collassati in un nodo intermedio (caratterizzati dal segno +).

Nel caso invece siano associati degli stili il browser rappresenta i dati nel formato definito negli stili stessi. Da segnalare che nel caso sia stato utilizzato uno stile XSL, le cui specifiche non sono state ancora ufficializzate dal W3C come ampliamente detto in 4.7, la Microsoft ha deciso di non rilasciare per le installazioni in locale il supporto per questo stile, ma di consentire le operazioni di rappresentazione tramite delle librerie presenti su degli appositi siti di proprietà MS.

Figura 7.1: Screen shot di una finestra di MS Internet Eplorer

7.1.2 Icon XML Spy 2.5

Questo software della Icon [48], presente solamente in una versione commerciale, è stato ritenuto il migliore attualmente disponibile sul mercato per quanto riguarda l'editazione di file XML, sia da un punto di vista della completezza delle informazioni a disposizione che da un punto di vista dell'utilizzo.

Per quanto riguarda la rappresentazione/editazione dei dati sono previste tre possibilità diverse:

- "as is" solo testo
- tramite una struttura ad albero
- completa

Il primo tipo di rappresentazione visualizza il sorgente del file in modo testo come potrebbe fare un qualsiasi text-editor, con la differenza che uti-

lizza la tecnica del "text colouring" per avere una più immediata percezione delle parole chiave di XML e dei contenuti nel file sorgente.

Una modalità più avanzata è quella della rappresentazione tramite una struttura ad albero nella quale si ha una immediata percezione di come sono rappresentati i dati e della struttura che questi assumono. Tale rappresentazione è suddivisa in due sezioni: in quella di sinistra viene visualizzata la struttura in forma di albero collassabile, come rappresentato nella figura 7.2, mentre nella sezione a destra vi è la possibilità di modificare direttamente i dati contenuti visualizzandoli in forma di tabella strutturata, come riportato nella figura 7.3, che permette di far capire la struttura stessa oltre ad avere la possibilità di un controllo diretto sui dati inseriti e sulla loro posizione.

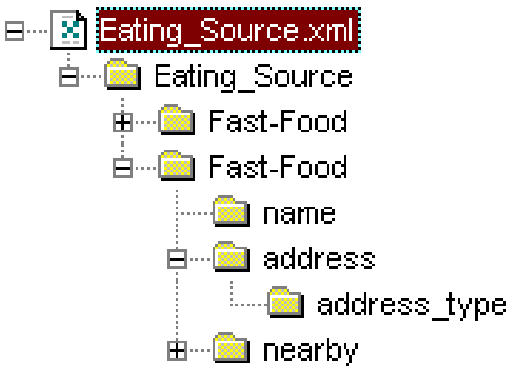

Figura 7.2: Screen shot dell'albero collasabile che rappresenta la struttura

Ultimo tipo di rappresentazione è quello di come effettivamente verranno visualizzati i dati. Occorre premettere che questo prodotto ed il browser della Microsoft sono legati a doppio filo in quanto la Icon ha deciso di utilizzare sia il parser distribuito dalla Microsoft per la lettura di un file XML e sia tutta la loro tecnologia legata alla rappresentazione dei dati.

Infatti la finestra di questo terzo tipo di rappresentazione non solo è del tutto identica a quanto già visto nella figura 7.1, ma necessita anche che sia installato Internet Explorer 5.0 per poter funzionare correttamente. Quindi tutti i discorsi fatti nel paragrafo 7.1.1 possono essere riportati inalterati in questa sezione.

Oltre alle funzioni di editazione di un file di dati XML questo software contiene anche delle procedure di editazione di un file DTD (possibile in for-

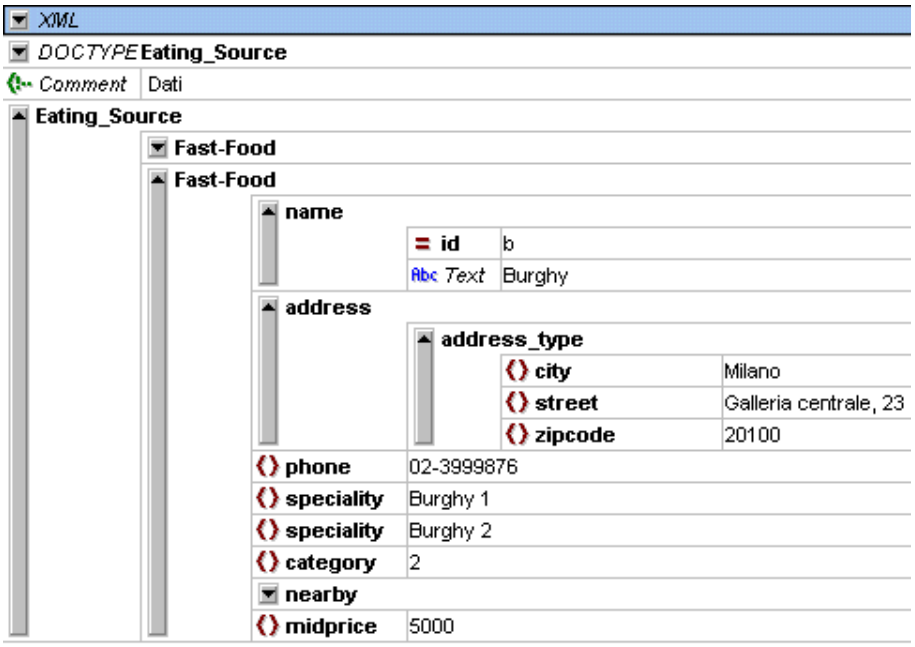

Figura 7.3: Screen shot della sezione di editazione dei dati

mato text-colouring che con una rappresentazione simie a quanto illustrato nella figura 7.3) con un controllo rigoroso di Well-formedness. Per quanto riguarda invece il controllo di Validà dei dati eventualmente associati ad uno specificho DTD esistono delle apposite funzioni a riguardo oltre che una completa gestione degli eventuali incongruenze eventualmente risconrtate.

Questa possibilità di un controllo rigoroso della grammatica sono state ampliamente utilizzate nello sviluppo del presente progetto per riportare tutte le possibilità previste dalla grammatica stessa in modo da avere un panorama completo della sintassi XML e quindi controllare la correttezza di tutte le funzionalità previstea dal Wrapper.

7.2 Java (Java Development Kit 1.2.2)

La definizione ufficiale di Java data dalla Sun Microsystems é la seguente:

"Java: un linguaggio semplice, a oggetti, distribuito, interpretato, robusto, sicuro, indipendente dall'architettura, portabile, cone elevate prestazioni,

multithread e dinamico".

Da questa definizione è sufficientemente chiaro il motivo per cui è stato scelto questo linguaggio per lo sviluppo non solo di del presente Wrapper, ma anche per buona parte dell'intero sistema MOMIS.

Infatti il fatto che sia portabile su diverse piattaforme senza modifiche del codice sorgente consente a MOMIS stesso di poter essere eseguito in diverse realtà operative senza preoccupazioni riguardanti la compatibilità del prodotto con le risorse Hardware e Software a disposizione dell'utente.

Dalla definizione si evince anche cha Java è un linguaggio orientato agli oggetti, strada che tutti i linguaggi di programmazione di ultima generazione hanno intrapreso, il che favorisce la componibilità e la modularità del software sviluppato.

Oltre a questo tale linguaggio, a differenza di altri quali ad esempio il C++, si fa carico della gestione della memoria dinamica, della sincronizzazione dei processi, delle caratteristiche della piattaforma hardware il che consente allo sviluppatore di concentrarsi esclusivamente sulla fase di progettazione e modellazione dell'applicazione tralasciando queste problematiche tecniche che spesso si rivelano come fonte di errori di scrittura o solamente di dispendio di tempo per l'ottimizzazzione.

Inoltre l'intera gestione degli errori è affidata ad un sistema di eccezioni che segnalano il problema e offrono la possibilità al processo in esecuzione di gestirlo nel modo più opportuno.

Punto non trascurabile è anche il fatto che il pacchetto di sviluppo mette a disposizione pacchetti per la gestione della grafica (ovviamente anch'essi indipendenti dalla piattaforma), i quali consentono lo sviluppo di interfacce estremamente efficaci e funzionali. Grazie a tali strumenti viene notevolmente ridotto lo sforzo necessario per la creazione di ambienti di supporto sia al progettista del Mediatore sia all'utente finale.

Non ultimo vale la pena segnalare anche il modulo *Javadoc* con il quale è possibile creare una documentazione completa ed estremamente efficiente. Essa viene presentata mediante documenti HTML con collegamenti ipertestuali, capaci, quindi, di percorrere in modo naturale le gerarchie di aggregazione e specializzazione che raccolgono il software realizzato.

Per le ragione esposte sopra Java è quindi stato scelto come linguaggio "ufficiale" per il progetto MOMIS. Inoltre il fatto che esso è destinato a ricevere il contributo di diversi tesisti e ricercatori per effettaure continue estensione e revisioni propone la necessità di avere un ambiente di sviluppo tale da offrire flessibilità e componibilità e, al contempo, in grado di stimolare la produzione di una documentazione dettagliata e di pratico impiego.

7.3 BYacc/J v0.93

Questo prodotto ha come obiettivo quello di facilitare il processo di creazione di un analizzatore sintattico basato su di una grammatica precisa e quindi ben si presta a tutta la fase di parsing necessaria per poter mettere a disposizione i dati letti da un file DTD di XML.

La versione di BYacc, denominata *Java extension v0.93* [49], utilizzata per lo sviluppo del wrapper consente di generare codice Java e non codice C come previsto nella versione originale del Byacc stesso, il che fa si' che possiamo considerare l'output fornito da questo prodotto come facilmente integrabile con il resto delle funzioni previste per uno sviluppo completo.

Analizzando brevemente le caratteristiche di Byacc occorre dire che base necessaria di partenza per la lettura di un file è quella di prevedere la definizione di grammatica la quale dovrebbe essere molto simile alla forma BNF. Allo stesso tempo occorre però sottolineare che non tutte le grammatiche possono essere analizzate dal presente software, ma solamente quelle che possono essere definite *non contestuali*. Il caso della sintassi prevista per un DTD di XML non rientra nel caso delle grammatiche non contestuali e quindi è possibile l'utilizzo di BYacc senza preoccuparsi di incorrere in problemi di fondo.

L'algoritmo sul quale si basa BYacc è quello del riconoscimento dei vari Token della grammatica i quali, tramite una metodologia *bottom-up*, includono delle regole le quali possono essere riconosciute in regole sempre più generali fino ad arrivare ad un particolare simbolo terminale che include tutti gli altri.

In questo modo il testo sorgente è stato completamente descritto e l'analisi sintattica può essere considerata come portata a termine, manca però tutta la fase di analisi semantica e di generazione del codice.

Per quanto riguarda la generazione del codice BYacc consente al programmatore l'introduzione di segmenti di codice corrispondenti con le azioni da effettuare una volta che si incontra una particolare regola grammaticale.

Una volta interamente definito il file sorgente, il quale permette il completo riconoscimento della grammatica, ed è stato associato, per ogni regola prevista, a tutti i segmenti di codice necessari, allora da riga di comando è possibile generare il corrispondente codice Java puro il quale potrà essere utilizzato come parser per gli scopi prefissati.

7.4 IBM XML Parser for Java (XML4J) Version 2

Prima di scendere nel particolare del software utilizzato ritengo utile fare alcune considerazioni generali valide non solo per questo parser, ma anche per altri prodotti utilizzati al momento dagli sviluppatori di software che si basi sui documenti XML.

In generale possiamo dire che esistono due metodologie di approccio differenti per accedere ai dati ed ai riusultati delle lettura effettuate dal Parser stesso:

- Interfaccia SAX (Simple API for XML)
- Interfaccia DOM (Document Object Model)

Chiaramente entrambe questi approcci hanno scopi diversi per quanto riguarda i risultati che si ottengono. Infatti nel caso l'applicazione richieda di fare delle operazioni di manipolazione dei dati quali possono essere le normali ricerche e modifiche di informazioni oppure la traformazione di elementi l'interfaccia SAX si dimostra essere l'approccio migliore in quanto garantisce allo sviluppatore la possibilità di scrivere del codice abbastanza agevolemnte.

Diversamente nel caso in cui l'applicazioni necessiti di lavorare sulla struttura dei dati piú che sul contenuto vero e proprio allora la struttura ad albero dell'interfaccia DOM si presta maggiormente ad essere utilizzata proprio per la sua caratteristica di mantenere in memoria, e quindi a disposizione dell'applicazione, l'intero albero con tutte le informazioni per i Nodi intermedi e finali.

Entrambe questi approcci possiedono inoltre la caratteristica di essere validanti o non validanti, cioé di controllare anche se i dati rispettano la struttura di un'eventuale DTD associato (vedi capitoloparagrafo 4.1.4) e di gestire di conseguenza le incongruenze in un qualche modo intercettabile dall'applicazione e quindi gestibile dallo sviluppatore del software.

Scendendo nel particolare di XML4J [50] possiamo dire dire che è scritto interamente in Java e viene fornito con il relativo file .jar per per il runtime dell'applicazione scritta anch'essa in Java. Questa caratteristica si sposa a perfezione con l'ambiente di sviluppo scelto per MOMIS e quindi candida il presente parser ad essere utilizzato ovunque nell'intero progetto si necessiti di acquisire dei dati in formato XML.

Per i nostri scopi, in realtà, non necessito di avere un prodotto così completo anche perchè XML4J, nella versione attuale, contiene le specifiche inserite nel DOM Level 1, e quindi mancano tutte le informazioni riguardanti il DTD. Siccome però esistono alcune API che mi consentono di avere quantomeno i riferimenti all'eventuale DTD si è deciso di utilizzarle per avere con sicurezza tali riferimenti da passare alle funzioni di parsing del wrapper.

Da segnalare che attualmente esistono delle versioni che supportano sia il DOM Level 1 che il DOM Level 2. Tutte le API, però, che interessano la lettura di un DTD sono classificate come *Deprecated*, che significa che tali funzioni Java sono attualmente supportate nella versione corrente, ma non si assicura che siano supportate anche per le versione successive.

Questa classificazione delle API che interessano ai nostri scopi hanno fatto si' che la scelta di creare un parser per il DTD internamente invece che utilizzare quelli esistenti garantisca una certa stabilità nel codice sorgente creato e non necessiti invece di revisioni dovute a cambiamenti di versioni di parser.

Appendice A

Sintassi del linguaggio XML

In questa appendice viene riportato la BNF del linguaggio XML come specificato nel documento ufficiale che riporta la Raccomandazione del W3C dal titolo "Extensible Markup Language (XML) 1.0".

Il documento completo può essere trovato all'indirizzo http://www.w3.org /TR/1998/REC-xml-19980210 che riporta la versione originale in inglese. Una traduzione in italiano è stata fatta da Andrea Marchetti (Istituto Applicazioni Telematiche - CNR) ed ha messo a disposizione tale traduzione all'indirizzo http://www.iat.cnr.it/xml/REC-xml-19980210-it.html.

A.1 Extensible Markup Language (XML) 1.0

Documento

document ::= prolog element Misc*

Rango dei caratteri

Char ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]

Spazio bianco

S ::= (#x20 | #x9 | #xD | #xA)+

Nomi e Token

```
NameChar   ::=  Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender
Name       ::=  (Letter | '_' | ':') (NameChar)*
Names      ::=  Name (S Name)*
Nmtoken    ::=  (NameChar)+
Nmtokens   ::=  Nmtoken (S Nmtoken)*
```

Dati letterali

```
EntityValue    ::=  '"' ([^%&"] | PEReference | Reference)* '"'
                  | '"' ([^%&'] | PEReference | Reference)* '"'
AttValue       ::=  '"' ([^ <&"] | Reference)* '"' | "'" ([[^ <&'] | Reference)* "'"
SystemLiteral  ::=  ('"' [^"]* '"') | ("'" [^']* "'")
PubidLiteral   ::=  '"' PubidChar* '"' | "'" (PubidChar - "'")* "'"
PubidChar      ::=  #x20 | #xD | #xA | [a-zA-Z0-9] | [-'()+,./:=?;!*#@$_%]
```

Character Data

```
CharData   ::=  [^ <&]* - ([^ <&]* ']]>' [^ <&]*)
```

Commenti

```
Comment   ::=  '<!–' ((Char - '-') | ('-' (Char - '-')))* '–>
```

Istruzioni di elaborazione

```
PI        ::=  '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
PITarget  ::=  Name - (('X' — 'x') ('M' — 'm') ('L' — 'l'))
```

Sezioni CDATA

```
CDSect   ::=  CDStart CData CDEnd
CDStart  ::=  '<![CDATA['
CData    ::=  (Char* - (Char* ']]>' Char*))
CDEnd    ::=  ']]>'
```

Prologo

```
prolog        ::=  XMLDecl? Misc* (doctypedecl Misc*)?
XMLDecl       ::=  '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
VersionInfo   ::=  S 'version' Eq (' VersionNum ' | " VersionNum ")
Eq            ::=  S? '=' S?
VersionNum    ::=  ([a-zA-Z0-9_.:] | '-')+
Misc          ::=  Comment | PI | S
```

Document Type Definition

```
doctypedecl   ::=  '<!DOCTYPE' S Name (S ExternalID)? S?
                   ('[' (markupdecl | PEReference | S)*']' S?)? '>'
markupdecl    ::=  elementdecl | AttlistDecl | EntityDecl
                   | NotationDecl | PI | Comment
```

Sottoinsieme Esterno

```
extSubset     ::=  TextDecl? extSubsetDecl
extSubsetDecl ::=  ( markupdecl | conditionalSect | PEReference | S )*
```

Dichiarazione di documento "standalone"

```
SDDecl   ::=  S 'standalone' Eq (("'" ('yes' | 'no') "'")
              | ('"' ('yes' | 'no') '"'))
```

Identificazione del linguaggio

```
LanguageID   ::=  Langcode ('-' Subcode)*
Langcode     ::=  ISO639Code | IanaCode | UserCode
ISO639Code   ::=  ([a-z] | [A-Z]) ([a-z] | [A-Z])
IanaCode     ::=  ('i' | 'I') '-' ([a-z] | [A-Z])+
UserCode     ::=  ('x' | 'X') '-' ([a-z] | [A-Z])+
Subcode      ::=  ([a-z] | [A-Z])+
```

Elemento

```
element   ::=  EmptyElemTag | STag content ETag
```

Tag-di-inizio

```
STag      ::=  '<' Name (S Attribute)* S? '>'
Attribute ::=  Name Eq AttValue
```

Tag-di-fine

```
ETag   ::=  '</' Name S? '>'
```

Contenuto degli elementi

```
content   ::=  (element | CharData | Reference | CDSect | PI | Comment)*
```

Tag per elementi vuoti

```
EmptyElemTag   ::=   '<' Name (S Attribute)* S? '/>'
```

Dichiarazione di element type

```
elementdecl   ::=   '<!ELEMENT' S Name S contentspec S? '>'
contentspec   ::=   'EMPTY' | 'ANY' | Mixed | children
```

Modelli di contenuto di soli elementi (Element-content)

```
children   ::=   (choice | seq) ('?' | '*' | '+')?
cp         ::=   (Name | choice | seq) ('?' | '*' | '+')?
choice     ::=   '(' S? cp ( S? '|' S? cp )* S? ')'
seq        ::=   '(' S? cp ( S? ',' S? cp )* S? ')'
```

Dichiarazione "Mixed-content"

```
Mixed   ::=   '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*'
            | '(' S? '#PCDATA' S? ')'
```

Dichiarazione di Attribute-list

```
AttlistDecl   ::=   '<!ATTLIST' S Name AttDef* S? '>'
AttDef        ::=   S Name S AttType S DefaultDecl
```

Attribute Types

```
AttType          ::=   StringType | TokenizedType | EnumeratedType
StringType       ::=   'CDATA'
TokenizedType    ::=   'ID' | 'IDREF' | 'IDREFS' | 'ENTITY'
                       | 'ENTITIES' | 'NMTOKEN' | 'NMTOKENS'
```

Enumerated Attribute Types

```
EnumeratedType   ::=   NotationType | Enumeration
NotationType     ::=   'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'
Enumeration      ::=   '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'
```

Default di attributo

```
DefaultDecl   ::=   '#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)
```

Sezioni condizionali

```
conditionalSect      ::=  includeSect | ignoreSect
includeSect          ::=  '<![' S? 'INCLUDE' S? '[' extSubsetDecl ']]>'
ignoreSect           ::=  '<![' S? 'IGNORE' S? '[' ignoreSectContents* ']]>'
ignoreSectContents   ::=  Ignore ('<![' ignoreSectContents ']]>' Ignore)*
Ignore               ::=  Char* - (Char* ('<![' | ']]>') Char*)
```

Riferimento a carattere

```
CharRef   ::=  '&#' [0-9]+ ';' | '&#x' [0-9a-fA-F]+ ';'
```

Riferimento a entità

```
Reference      ::=  EntityRef | CharRef
EntityRef      ::=  '&' Name ';'
PEReference    ::=  '%' Name ';'
```

Dichiarazione di entità

```
EntityDecl   ::=  GEDecl | PEDecl
GEDecl       ::=  '<!ENTITY' S Name S EntityDef S? '>'
PEDecl       ::=  '<!ENTITY' S '%' S Name S PEDef S? '>'
EntityDef    ::=  EntityValue | (ExternalID NDataDecl?)
PEDef        ::=  EntityValue | ExternalID
```

Dichiarazione di entità esterna

```
ExternalID   ::=  'SYSTEM' S SystemLiteral
                  | 'PUBLIC' S PubidLiteral S SystemLiteral
NDataDecl    ::=  S 'NDATA' S Name
```

Dichiarazione di testo

```
TextDecll   ::=  '<?xml' VersionInfo? EncodingDecl S? '?>'
```

Entità parsed esterne Well-Formed

```
extParsedEntl   ::=  TextDecl? content
extPE           ::=  TextDecl? extSubsetDecl
```

Dichiarazione di codifica

```
EncodingDecl   ::=  S 'encoding' Eq ('"' EncName '"' | "'" EncName "'" )
EncName        ::=  [A-Za-z] ([A-Za-z0-9._] | '-')*
```

Dichiarazione di notazione

```
NotationDecl  ::=  '<!NOTATION' S Name S (ExternalID | PublicID) S? '>'
PublicID      ::=  'PUBLIC' S PubidLiteral
```

Appendice B

Sintassi del linguaggio descrittivo ODL$_{I^3}$

Questa appnedice riporta la descrizione delle BNF del linguaggio ODL$_{I^3}$ derivato come estensione del linguaggio ODL dello standard ODMG-93.

⟨specification⟩ ::= ⟨definition⟩
| ⟨definition⟩ ⟨specification⟩

⟨definition⟩ ::= ⟨type_dcl⟩ ;
| ⟨const_dcl⟩ ;
| ⟨except_dcl⟩ ;
| ⟨interface_dcl⟩ ;
| ⟨relationships_dcl⟩ ;
| ⟨rule_dcl⟩ ;
| ⟨module⟩ ;

⟨relationships_dcl⟩ ::= ⟨local_entity⟩ ⟨relationship_type⟩ ⟨local_entity⟩

⟨local_entity⟩ ::= ⟨local_class_name⟩ | ⟨local_attr_name⟩

⟨local_class_name⟩ ::= ⟨source_name⟩.⟨class_name⟩

⟨relationship_type⟩ ::= **SYN** | **BT** | **NT** | **RT**

⟨rule_dcl⟩ ::= **rule** ⟨identifier⟩ ⟨rule_spec⟩

⟨rule_spec⟩ ::= ⟨rule_pre⟩ **then** ⟨rule_post⟩ | **{** ⟨case_dcl⟩ **}**

⟨rule_pre⟩ ::= ⟨forall⟩ ⟨identifier⟩ **in** ⟨identifier⟩ **:** ⟨rule_body_list⟩

⟨rule_post⟩ ::= ⟨rule_body_list⟩

⟨case_dcl⟩ ::= **case of** ⟨identifier⟩ **:** ⟨case_list⟩

⟨case_list⟩ ::= ⟨case_spec⟩ | ⟨case_spec⟩ ⟨case_list⟩

⟨case_spec⟩ ::= ⟨identifier⟩ **:** ⟨identifier⟩ ;

⟨rule_body_list⟩ ::= **(** ⟨rule_body_list⟩ **)**
| ⟨rule_body⟩
| ⟨rule_body_list⟩ **and** ⟨rule_body⟩
| ⟨rule_body_list⟩ **and (** ⟨rule_body_list⟩ **)**

⟨rule_body⟩ ::= ⟨dotted_name⟩ ⟨rule_const_op⟩ [⟨rule_cast⟩] ⟨literal_value⟩
| ⟨dotted_name⟩ **in** ⟨dotted_name⟩
| ⟨forall⟩ ⟨identifier⟩ **in** ⟨dotted_name⟩ **:** ⟨rule_body_list⟩
| **exists** ⟨identifier⟩ **in** ⟨dotted_name⟩ **:** ⟨rule_body_list⟩

⟨rule_const_op⟩ ::= = | ≥ | ≤ | > | <

⟨rule_cast⟩ ::= **(**⟨simple_type_spec⟩**)**

⟨dotted_name⟩ ::= ⟨identifier⟩ | ⟨identifier⟩.⟨dotted_name⟩

⟨forall⟩ ::= **for all** | **forall**

⟨module⟩ ::= **module** ⟨identifier⟩ **{** ⟨specification⟩ **}**

⟨interface⟩ ::= ⟨interface_dcl⟩
| ⟨forward_dcl⟩

⟨interface_dcl⟩ ::= ⟨interface_header⟩ **{**[⟨interface_body⟩]
[**union** ⟨interface_body⟩]**}**

⟨persistence_dcl⟩ ::= **persistent** | **transient**

⟨forward_dcl⟩ ::= **interface** ⟨identifier⟩

⟨interface_header⟩ ::= **interface** ⟨identifier⟩
[⟨inheritance_spec⟩]
[⟨type_property_list⟩]

⟨type_property_list⟩ ::= **(**[⟨source_spec⟩] [⟨extent_spec⟩] [⟨key_spec⟩] [⟨f_key_list⟩]**)**

⟨source_spec⟩ ::= **source** ⟨source_type⟩ ⟨source_name⟩

⟨source_type⟩ ::= **relational**|**nfrelational**|**object**|**file**|**semistructured**

⟨source_name⟩ ::= ⟨identifier⟩

⟨extent_spec⟩ ::= **extent** ⟨extent_list⟩

⟨extent_list⟩ ::= ⟨string⟩ | ⟨string⟩ **,** ⟨extent_list⟩

⟨key_spec⟩ ::= **key**[**s**] ⟨key_list⟩

⟨f_key_list⟩ ::= ⟨f_key_spec⟩ | ⟨f_key_spec⟩ **,** ⟨f_key_list⟩

⟨f_key_spec⟩ ::= **foreign_key (**⟨key_list⟩**) references** ⟨identifier⟩
[,(⟨key_list⟩)]

⟨key_list⟩ ::= ⟨key⟩ | ⟨key⟩ **,** ⟨key_list⟩

⟨key⟩ ::= ⟨property_name⟩ | **(**⟨property_list⟩**)**

⟨property_list⟩ ::= ⟨property_name⟩
| ⟨property_name⟩ **,** ⟨property_list⟩

⟨property_name⟩ ::= ⟨identifier⟩

⟨interface_body⟩ ::= ⟨export⟩ | ⟨export⟩ ⟨interface_body⟩

⟨export⟩ ::= ⟨type_dcl⟩ ;
| ⟨const_dcl⟩ ;
| ⟨except_dcl⟩ ;
| ⟨attr_dcl⟩ ;
| ⟨rel_dcl⟩ ;
| ⟨op_dcl⟩ ;

⟨inheritance_spec⟩ ::= **:** ⟨scoped_name⟩ [**,** ⟨inheritance_spec⟩]

⟨scoped_name⟩ ::= ⟨identifier⟩
| **::**⟨identifier⟩
| ⟨scoped_name⟩**::**⟨identifier⟩

⟨const_dcl⟩ ::= **const** ⟨const_type⟩ ⟨identifier⟩=⟨const_exp⟩

⟨const_type⟩ ::= ⟨integer_type⟩
| ⟨char_type⟩
| ⟨boolean_type⟩
| ⟨floating_pt_type⟩
| ⟨string_type⟩
| ⟨scoped_name⟩

⟨const_exp⟩ ::= ⟨or_expr⟩

⟨or_expr⟩ ::= ⟨xor_expr⟩
| ⟨or_expr⟩ | ⟨xor_expr⟩

⟨xor_expr⟩ ::= ⟨and_expr⟩
| ⟨xor_expr⟩ ^ ⟨and_expr⟩

⟨and_expr⟩ ::= ⟨shift_expr⟩
| ⟨and_expr⟩ & ⟨shift_expr⟩

⟨shift_expr⟩ ::= ⟨add_expr⟩
| ⟨shift_expr⟩ >> ⟨add_expr⟩
| ⟨shift_expr⟩ << ⟨add_expr⟩

⟨add_expr⟩ ::= ⟨mult_expr⟩
| ⟨add_expr⟩ + ⟨mult_expr⟩
| ⟨add_expr⟩ - ⟨mult_expr⟩

⟨mult_expr⟩ ::= ⟨unary_expr⟩
| ⟨mult_expr⟩ * ⟨unary_expr⟩
| ⟨mult_expr⟩ / ⟨unary_expr⟩
| ⟨mult_expr⟩ % ⟨unary_expr⟩

⟨unary_expr⟩ ::= ⟨primary_expr⟩
| ⟨unary_operator⟩ | ⟨primary_expr⟩

⟨unary_operator⟩ ::= - | + | ~

⟨primary_expr⟩ ::= ⟨scoped_name⟩
| ⟨literal⟩
| (| ⟨const_exp⟩)

⟨literal⟩ ::= ⟨integer_literal⟩
| ⟨string_literal⟩
| ⟨character_literal⟩
| ⟨floating_pt_literal⟩
| ⟨boolean_literal⟩

⟨boolean_literal⟩ ::= **TRUE** | **FALSE**

⟨positive_int_const⟩ ::= ⟨const_exp⟩

⟨type_dcl⟩ ::= **typedef** ⟨type_declarator⟩
| ⟨struct_type⟩
| ⟨union_type⟩
| ⟨enum_type⟩

⟨type_declarator⟩ ::= ⟨type_spec⟩ ⟨declarators⟩

⟨type_spec⟩ ::= ⟨simple_type_spec⟩
| ⟨constr_type_spec⟩

⟨simple_type_spec⟩ ::= ⟨base_type_spec⟩
| ⟨template_type_spec⟩
| ⟨scoped_name⟩

⟨base_type_spec⟩ ::= ⟨floating_pt_type⟩
| ⟨integer_type⟩
| ⟨char_type⟩
| ⟨boolean_type⟩
| ⟨octet_type⟩
| ⟨range_type⟩
| ⟨any_type⟩

⟨template_type_spec⟩ ::= ⟨array_type⟩
| ⟨string_type⟩
| ⟨coll_type⟩

⟨coll_type⟩ ::= ⟨coll_spec⟩ <⟨simple_type_spec⟩>

⟨coll_spec⟩ ::= **set** | **list** | **bag**

⟨constr_type_spec⟩ ::= ⟨struct_type⟩
| ⟨union_type⟩
| ⟨enum_type⟩

⟨declarators⟩ ::= ⟨declarator⟩
| ⟨declarator⟩ **,** ⟨declarators⟩

⟨declarator⟩ ::= ⟨simple_declarator⟩
| ⟨complex_declarator⟩

⟨simple_declarator⟩ ::= ⟨identifier⟩

⟨complex_declarator⟩ ::= ⟨array_declarator⟩

⟨floating_pt_type⟩ ::= **float** | **double**

⟨integer_type⟩ ::= ⟨signed_int⟩
⟨unsigned_int⟩

⟨signed_int⟩ ::= ⟨signed_long_int⟩
⟨signed_short_int⟩
⟨signed_long_int⟩ ::= **long**
⟨signed_short_int⟩ ::= **short**
⟨unsigned_int⟩ ::= ⟨unsigned_long_int⟩
⟨unsigned_short_int⟩
⟨unsigned_long_int⟩ ::= **unsigned long**
⟨unsigned_short_int⟩ ::= **unsigned short**
⟨char_type⟩ ::= **char**
⟨boolean_type⟩ ::= **boolean**
⟨octet_type⟩ ::= **octet**
⟨any_type⟩ ::= **any**
⟨range_type⟩ ::= **range** [⟨range_specifier⟩]
⟨range_specifier⟩ ::= ⟨const_exp⟩ **,** ⟨const_exp⟩
| ⟨const_exp⟩ **, +inf**
| **-inf ,** ⟨const_exp⟩
⟨struct_type⟩ ::= **struct** ⟨identifier⟩ **{**⟨member_list⟩**}**
⟨member_list⟩ ::= ⟨member⟩
| ⟨member⟩ ⟨member_list⟩
⟨member⟩ ::= ⟨type_spec⟩ ⟨declarators⟩ **;**
⟨union_type⟩ ::= **union** ⟨identifier⟩ **switch**
(⟨switch_type_spec⟩**) {**⟨switch_body⟩**}**
⟨switch_type_spec⟩ ::= ⟨integer_type⟩
| ⟨char_type⟩
| ⟨boolean_type⟩
| ⟨enum_type⟩
| ⟨scoped_name⟩
⟨switch_body⟩ ::= ⟨case⟩ | ⟨case⟩ ⟨switch_body⟩
⟨case⟩ ::= ⟨case_label_list⟩ ⟨element_spec⟩ **;**
⟨case_label_list⟩ ::= ⟨case_label⟩
| ⟨case_label⟩ ⟨case_label_list⟩
⟨case_label⟩ ::= **case** ⟨const_exp⟩**:**
| **default:**
⟨element_spec⟩ ::= ⟨type_spec⟩ ⟨declarator⟩
⟨enum_type⟩ ::= **enum** ⟨identifier⟩ **{**⟨enumerator_list⟩**}**
⟨enumerator_list⟩ ::= ⟨enumerator⟩
| ⟨enumerator⟩,⟨enumerator_list⟩
⟨enumerator⟩ ::= ⟨identifier⟩
⟨array_type⟩ ::= ⟨array_spec⟩
<⟨simple_type_spec⟩,⟨positive_int_const⟩>
| ⟨array_spec⟩ <⟨simple_type_spec⟩>

⟨array_spec⟩ ::= **array** | **sequence**
⟨string_type⟩ ::= **string** | **string** ⟨positive_int_const⟩
⟨array_declarator⟩ ::= ⟨identifier⟩ ⟨array_size_list⟩
⟨array_size_list⟩ ::= ⟨fixed_array_size⟩
| ⟨fixed_array_size⟩ ⟨array_size_list⟩
⟨fixed_array_size⟩ ::= [⟨positive_int_const⟩]
⟨attr_dcl⟩ ::= [**readonly**] **attribute**
⟨domain_type⟩ ⟨attribute_name⟩ [*]
[⟨fixed_array_size⟩] [⟨mapping_rule_dcl⟩]
⟨mapping_rule_dcl⟩ ::= **mapping_rule** ⟨rule_list⟩
⟨rule_list⟩ ::= ⟨rule⟩ | ⟨rule⟩ , ⟨rule_list⟩
⟨rule⟩ ::= ⟨local_attr_name⟩ | '⟨identifier⟩'
⟨and_expression⟩ | ⟨union_expression⟩
⟨and_expression⟩ ::= (⟨local_attr_name⟩ **and** ⟨and_list⟩)
⟨and_list⟩ ::= ⟨local_attr_name⟩ | ⟨local_attr_name⟩ **and** ⟨and_list⟩
⟨union_expression⟩ ::= (⟨local_attr_name⟩ **union** ⟨union_list⟩ **on** ⟨identifier⟩)
⟨union_list⟩ ::= ⟨local_attr_name⟩ | ⟨local_attr_name⟩ **union** ⟨union_list⟩
⟨local_attr_name⟩ ::= ⟨source_name⟩.⟨class_name⟩.⟨attribute_name⟩
⟨domain_type⟩ ::= ⟨simple_type_spec⟩
| ⟨struct_type⟩
| ⟨enum_type⟩
⟨rel_dcl⟩ ::= **relationship** ⟨target_of_path⟩ ⟨identifier⟩
inverse ⟨inverse_trasversal_path⟩
⟨target_of_path⟩ ::= ⟨identifier⟩
| ⟨rel_collection_type⟩ <⟨identifier⟩>
⟨inverse_trasversal_path⟩ ::= ⟨identifier⟩ :: ⟨identifier⟩
⟨attribute_list⟩ ::= ⟨scoped_name⟩
| ⟨scoped_name⟩,⟨attribute_list⟩
⟨rel_collection_type⟩ ::= **set** | **list** | **bag** | **array**
⟨except_dcl⟩ ::= **exception** ⟨identifier⟩ {[⟨member_list⟩]}
⟨op_dcl⟩ ::= [⟨op_attribute⟩] ⟨op_type_spec⟩ ⟨identifier⟩
⟨parameter_dcls⟩ [⟨raises_expr⟩] [⟨context_expr⟩]
⟨op_attribute⟩ ::= **oneway**
⟨op_type_spec⟩ ::= ⟨simple_type_spec⟩ | **void**
⟨parameter_dcls⟩ ::= ([⟨param_dcl_list⟩])
⟨param_dcl_list⟩ ::= ⟨param_dcl⟩
| ⟨param_dcl⟩,⟨param_dcl_list⟩
⟨param_dcl⟩ ::= ⟨param_attribute⟩ ⟨simple_type_spec⟩ ⟨declarator⟩
⟨param_attribute⟩ ::= **in** | **out** | **inout**
⟨raises_expr⟩ ::= **raises** (⟨scoped_name_list⟩)

⟨scoped_name_list⟩ ::= ⟨scoped_name⟩
| ⟨scoped_name⟩,⟨scoped_name_list⟩
⟨context_expr⟩ ::= **context** (⟨string_literal_list⟩)
⟨string_literal_list⟩ ::= ⟨string_literal⟩
| ⟨string_literal⟩,⟨string_literal_list⟩

Bibliografia

[1] MOMIS staff. Momis (mediator environment for multiple information sources) project part of the interdata project. Available at http://sparc20.dsi.unimo.it/Momis.

[2] The World Wide Web Consortium (W3C). Extensible markup language (xml). Available at http://www.w3.org/XML/.

[3] The World Wide Web Consortium (W3C). Extensible markup language (xml) 1.0. W3C Recommendation 10-February-1998. Available at http://www.w3.org/TR/1998/REC-xml-19980210.html.

[4] S. Bergamaschi, D. Beneventano, S. Castano, and M. Vincini. Integrazione di informazione: il linguaggio odl_i^3 e la logica descrittiva olcd. Technical Report INTERDATA T3-R03, MURST 40%, 1998. http://bsing.ing.unibs.it/ deantone/interdata_tema3/.

[5] Alberto Zanoli. Si-designer, un tool di ausilio all'integrazione di sorgenti di dati eterogenee distribuite: progetto e realizzazione. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di ingegneria Informatica, 1997-1998. http://sparc20.dsi.unimo.it/tesi/index.html.

[6] Sonia Bergamaschi, Beneventano Domenico, and Maurizio Vincini. Il sistema momis per l'integrazione di sorgenti strutturate e semistrutturate. Technical Report T3–R08, INTERDATA, 1999.

[7] MOMIS staff. The momis project: Articles and technical report. Available at http://sparc20.dsi.unimo.it/Momis/articles.html.

[8] MOMIS staff. Momis pubblications - list of protected pubblications related to the momis prototype. Available at http://sparc20.dsi.unimo.it/Momis/prototipo/publications.html.

[9] S. De Capitani di Vimercati S. Bergamaschi, S. Castano and M. Vincini. Momis: An intelligent system for the integration of semistructured and structured data, 1998.

[10] S. Castano and V. De Antonellis. Semantic dictionary design for database interoperability. In *Proc. of Int. Conf. on Data Engineering, ICDE'97*, Birmingham, UK, 1997.

[11] S. Castano and V. De Antonellis. Deriving global conceptual views from multiple information sources. In *preProc. of ER'97 Preconference Symposium on Conceptual Modeling, Historical Perspectives and Future Directions*, 1997.

[12] D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. Odb-tools: A description logics based tool for schema validation and semantic query optimization in object oriented databases. In *Proc. of Int. Conf. on Data Engineering, ICDE'97*, Birmingham, UK, April 1997.

[13] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. ODB-QOPTIMIZER: A tool for semantic query optimization in oodb. In *Int. Conference on Data Engineering - ICDE97*, 1997. http://sparc20.dsi.unimo.it.

[14] R. G. G. Cattell, editor. *The Object Database Standard: ODMG93*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[15] P. Buneman, L. Raschid, and J. Ullman. Mediator languages - a proposal for a standard. Technical report, University of Maryland, 1996. ftp://ftp.umiacs.umd.edu /pub/ONRrept/ medmodel96.ps.

[16] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Consistency checking in complex object database schemata with integrity constraints. In P. Atzeni and V. Tannen, editors, *5th Int. Workshop on Database Programming Languages*, Lecture Notes in Computer Science, pages 48–57, Gubbio, Italy, September 1995. Springer-Verlag.

[17] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

[18] P. Ciaccia. Block access estimation for clustered data. *IEEE Trans. on Knowledge and Data Engine*, 1993. toappear.

[19] C. Carpineto and G. Romano. Galois: An order-theoretic approach to conceptual clustering. In *Macine Learning Conference*, pages 33–40, 1993.

[20] Andrea Zacaria. Momis: Il componente query manager. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di ingegneria Informatica, 1997-1998. http://sparc20.dsi.unimo.it/tesi/index.html.

[21] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Consistency checking in complex object database schemata with integrity constraints. *IEEE Transactions on Knowledge and Data Engineering*, 10:576–598, July/August 1998.

[22] K. Shoens et al. The rufus system: Information organization for semistructured data. In *Proc. VLDB Conference - Dublin, Ireland*, 1993.

[23] P. Buneman. Semistructured data. In *Proc. of 1997 Symposium on Principles of Database Systems (PODS97)*, pages 117–121, Tucson, Arizona, 1997.

[24] P. Buneman, W. Fan, and S. Weinstein. Path constraints on semistructured and structured data. In *PODS '97. Proceedings of the Seventeenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 129–138. ACM Press, 1998.

[25] S. Bergamaschi, C. Sartori, and P. Tiberio. On taxonomic reasoning in conceptual design. Technical Report 68, CIOC - CNR, Viale Risorgimento, 2 - Bologna, Italy, March 1990.

[26] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A structural data model for objects. In *SIGMOD*, pages 58–67, Portland, Oregon, 1989.

[27] D. Calvanese, G. De Giacomo, and M. Lenzerini. Structured objects: Modeling and reasoning. In *Proc. of Int. Conference on Deductive and Object-Oriented Databases*, 1995.

[28] S. Castano, V. De Antonellis, and S. De Capitani Di Vimercat. Global viewing of heterogeneous data sources. Technical Report 98-08, Dip. Elettronica e Informazione, Politecnico di Milano, Milano, Italy, 1998.

[29] R.G.G. Cattell and others., editors. *The Object Data Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[30] S. Bergamaschi, D. Beneventano, S. Castano, and M. Vincini. Semantic integration and query of heterogeneous information sources. *Journal of Data and Knowledge Engineering*, 1999. Available at http://sparc20.dsi.unimo.it/Momis/prototipo/publications.html.

[31] Information processing – text and office systems – standard generalized markup language (sgml), 1986.

[32] The World Wide Web Consortium (W3C). Html 4.01 specification. W3C Recommendation 24 December 1999. Available at http://www.w3.org/TR/1999/REC-html401-19991224.

[33] The World Wide Web Consortium (W3C). Cascading style sheets, level 2 - css2 specification. W3C Recommendation 12-May-1998. Available at http://www.w3.org/TR/1998/REC-CSS2-19980512.

[34] The World Wide Web Consortium (W3C). Extensible stylesheet language (xsl) version 1.0. W3C Working Draft 27 March 2000. Available at http://www.w3.org/TR/2000/WD-xsl-20000327/.

[35] The World Wide Web Consortium (W3C). Xml linking language (xlink). W3C Working Draft 21-February-2000. Available at http://www.w3.org/TR/2000/WD-xlink-20000221.

[36] The World Wide Web Consortium (W3C). Namespaces in xml. W3C Recommendation 14 January 1999. Available at http://www.w3.org/TR/1999/REC-xml-names-19990114.

[37] Robin Cover. Dsssl - document style semantics and specification language. iso/iec 10179:1996. Last modified: May 03, 2000. Available at http://www.oasis-open.org/cover/dsssl.html.

[38] The World Wide Web Consortium (W3C). Associating style sheets with xml documents version 1.0. W3C Recommendation 29 June 1999. Available at http://www.w3.org/1999/06/REC-xml-stylesheet-19990629.

[39] The World Wide Web Consortium (W3C). Xml pointer language (xpointer). W3C Working Draft 6 December 1999. Available at http://www.w3.org/TR/1999/WD-xptr-19991206.

[40] The World Wide Web Consortium (W3C). Document object model (dom) level 1 specification. W3C Recommendation 1 October 1999. Available at http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001.

[41] The World Wide Web Consortium (W3C). Document object model (dom) level 2 specification. W3C Working Draft 23 September, 1999. Available at http://www.w3.org/TR/1999/WD-DOM-Level-2-19990923.

[42] The World Wide Web Consortium (W3C). Document object model (dom) requirements. W3C Working Draft 12 April, 2000. Available at http://www.w3.org/TR/2000/WD-DOM-Requirements-20000412.

[43] The World Wide Web Consortium (W3C). Xml schema part 0: Primer. W3C Working Draft, 7 April 2000. Available at http://www.w3.org/TR/2000/WD-xmlschema-0-20000407/.

[44] The World Wide Web Consortium (W3C). Xml schema part 1: Structures. W3C Working Draft, 7 April 200. Available at http://www.w3.org/TR/2000/WD-xmlschema-1-20000407/.

[45] The World Wide Web Consortium (W3C). Xml schema part 2: Datatypes. W3C Working Draft 07 April 2000. Available at http://www.w3.org/TR/2000/WD-xmlschema-2-20000407/.

[46] Sun Microsystems. Java sdk v 1.2.2. Available at http://java.sun.com/products/jdk/1.2/index.html.

[47] Kathy Walrath Mary Campione. *The Java Tutorial : Object-Oriented Programming for the Internet (Java Series)*. Addison-Wesley, 1998.

[48] Icon. Xml spy v 2.5. Available at http://xmlspy.com.

[49] Byacc/java. Available at http://sparc20.dsi.unimo.it/Momis.

[50] IBM Alphaworks. Xml4j for java parser v 2.0. Available at http://www.alphaworks.ibm.com.