UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Riscrittura di interrogazioni XML: un approccio basato sull'analisi semantica degli schemi

Daniele Miselli

Tesi di Laurea

Anno Accademico 2001/2002

Relatore: Chiar.mo Prof. Paolo Tiberio

Controrelatore: Chiar.mo Prof. Sonia Bergamaschi

Correlatore: Dott. Federica Mandreoli

RINGRAZIAMENTI

Ringrazio il Professor Paolo Tiberio e
la Professoressa Sonia Bergamaschi
per la disponibilità dimostrata.
Un ringraziamento particolare alla
Dottoressa Federica Mandreoli per
l'aiuto costante fornito nello sviluppo
di questa tesi di laurea.
Ringrazio anche la mia famiglia
per l'appoggio fornito in questi anni.

PAROLE CHIAVE

Ontologie
MOMIS
Biblioteche Digitali
Xml-Schema
Interrogazioni XML

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Negli ultimi anni il linguaggio *eXtensible Markup Language (XML)* ha ottenuto una sempre maggiore importanza nel campo delle Biblioteche Digitali e nello scambio di dati via Web. Il motivo di questo successo risiede nella assenza di una struttura specifica dei dati descritti: si parla, infatti, di dati semistrutturati. La forma di un documento XML viene decisa dall'autore del documento stesso, rendendo tale standard molto pratico ed in grado di descrivere praticamente qualsiasi cosa. Chiunque abbia dimestichezza con lo standard HTML (attualmente il più popolare linguaggio per la descrizione e la visualizzazione di documenti sul Web) non avrà alcun problema nel comprendere la sintassi di XML. XML si basa su marcatori (*tag*) i cui nomi sono decisi da chi crea il documento ed hanno il duplice compito di descrivere e contenere i dati. Per questi motivi XML risulta adatto per l'implementazione di archivi di dati di grandi dimensioni e concernenti gli argomenti più disparati.

L'interrogazione di un grande archivio di documenti XML, come può essere il *repository* di una biblioteca digitale, non è cosa semplice, proprio per l'assenza di una struttura fissa che ne contraddistingue i documenti stessi. L'assenza di una struttura fissa rappresenta quindi le due facce di una stessa medaglia: la flessibilità e la facilità d'uso da una parte e la difficoltà a recuperare le informazioni volute (su archivi di notevoli dimensioni) dall'altra.

Un'interrogazione posta da un utente su un archivio XML, non può essere risolta efficacemente in maniera *esatta* (cioè recuperando solamente quei documenti descritti dai marcatori presenti nella richiesta); difficilmente, infatti, un utente portà conoscere esattamente la struttura di tutti i documenti presenti nella totalità del repository. Senza contare che persone differenti tendono a descrivere la realtà che li circonda in maniera diversa: documenti XML che parlano degli stessi argomenti potranno avere una struttura ed una terminologia di marcatori molto differente, se creati da due persone distinte. Proprio per questi motivi si parla di *interrogazioni approssimate*

Numerose sono state negli ultimi anni le proposte per la riscrittura di interrogazioni in interrogazioni approssimate, basate sulla struttura dei documenti effettivamente presenti in archivio. Tali metodi sono, per la maggior parte, basati su algoritmi, generalmente computazionalmente pesanti, di *Tree Matching*, ovvero di rilevazione di somiglianze fra due strutture ad albero. Sia un documento formato da dati semistrutturati (XML in particolare), che una interrogazione possono, infatti, essere rappresentati come alberi la cui struttura è determinata dai percorsi, o *path*, composti dagli elementi in essi presenti.

Nella presente tesi è proposto un innovativo metodo per la riscrittura approssimata di query, basato sull'impiego di un sistema mediatore (capace di integrare schemi rappresentanti le strutture di sorgenti di informazione eterogenee) e l'utilizzo di ontologie. L'impiego di ontologie permette di assegnare agli schemi che descrivono i documenti XML, significati che non dipendono dal termine specifico impiegato, ma dal vero e proprio concetto rappresentato. In questa tesi si assume che gli schemi siano espressi nel nuovo linguaggio Xml-Schema per la rappresentazione e la validazione di documenti XML, proposto e consigliato dal W3C (*World Wide Web Consortium*). Proprio in base alle ontologie, ed ai significati assegnati agli schemi, sarà possibile effettuare una corretta ed efficiente opera di riscrittura delle richieste: sarà possibile, infatti, reperire relazioni semantiche fra gli schemi (in particolare fra i percorsi, o *path* contenuti in essi) per mezzo delle quali trovare i percorsi più simili a quelli espressi nella richiesta dell'utente.

Il sistema mediatore impiegato in questo progetto è MOMIS, sviluppato presso il dipartimento di Ingegneria dell'Università di Modena e Reggio Emilia nell'ambito del progetto MURST INTERDATA 97/98. MOMIS, ed in particolare un suo modulo detto SLIM, è in grado, tramite l'interazione col database lessicale WordNet, sviluppato nel Cognitive Science Laboratory dell'Università di Princeton, di *annotare* semanticamente gli schemi proposti e precedentemente tradotti, per mezzo di wrapper, in schemi ODL_{I^3}, una estensione del linguaggio ODL-ODMG. Annotare uno schema significa assegnare ai termini presenti in esso significati. In base a questi significati, il modulo SLIM di MOMIS è in grado di produrre un insieme di relazioni semantiche interschema che potremo utilizzare al fine di riscrivere correttamente le richieste poste sulla totalità dei documenti di un archivio.

Si mostra ora l'organizzazione della tesi ed il contenuto dei vari capitoli in essa presenti:

- Nel **Capitolo 1** si parla di Biblioteche Digitali, di quale potrà essere il loro futuro nella società moderna e di quale struttura potrebbero avere. Si introduce, inoltre, il problema della riscrittura delle richieste su un archivio di documenti XML ed è descritta una panoramica sullo stato dell'arte a tal proposito

- Nel **Capitolo 2** si presenta il problema della integrazione delle informazioni eterogenee e si descrive la struttura del sistema MOMIS. Il questo capitolo sono introdotte anche le modifiche al sistema MOMIS necessarie al nostro scopo.

- Nel **Capitolo 3** viene presentata la sintassi di Xml-Schema, linguaggio per la descrizione di fogli XML. Il capitolo può essere tralasciato da chiunque sia a conoscenza dell'utilizzo di tale linguaggio

- Nel **Capitolo 4** si parla della sintassi del linguaggio ODL_{I^3} utilizzato nel sistema MOMIS, anche questo capitolo può essere tralasciato da chiunque conosca già tale sintassi.

- Nel **Capitolo 5** si parla della specifica di un wrapper per Xml-Schema. Allo stato attuale delle cose, infatti, tale wrapper non è stato ancora progettato ed implementato; il sistema MOMIS, quindi non è ancora in grado di trattare schemi Xml-Schema. In questo capitolo si propone, in particolare, una traduzione fra i linguaggi Xml-Schema ed ODL_{I^3} (descritti nei due capitoli precedenti), che potrà essere impiegata nella realizzazione di tale wrapper.

- Nel **Capitolo 6** viene descritto il database lessicale WordNet, usato da MOMIS, ed un metodo per estenderlo.

- Nel **Capitolo 7** si presenta la nuova versione del modulo SLIM di MOMIS, prodotta nell'ambito di questa tesi, che è in grado di sfruttare con successo la versione estensibile di WordNet ed il browser WNEditor.

- Infine, nel capitolo **Capitolo 8** si descrive un metodo, ed il software prodotto, per cercare i percorsi più simili ad un path dato. In particolare questo modulo software è in grado di reperire i percorsi, presenti all'interno degli schemi ottenuti da un archivio di dati ed impiegati da MOMIS, maggiormente simili a quelli contenuti in una richiesta espressa da un utente. Tale modulo, che impiega un algoritmo basato sull'edit distance unitamente alle relazioni semantiche individuate da SLIM, potrà essere impiegato con successo in un software per la gestione di richieste XQuery su documenti XML.

Capitolo 1

Biblioteche Digitali Xml

1.1 Introduzione

Cos'è, esattamente, una biblioteca digitale?

Le biblioteche digitali sono organizzazioni che forniscono le risorse per selezionare, strutturare, offrire accesso intellettuale, interpretare, distribuire, preservare l'integrità ed assicurare la conservazione nel tempo di collezioni di lavori digitali, in modo che siano facilmente ed economicamente disponibili da una definita comunità o da un insieme di comunità

In questo modo la Federazione Americana per le biblioteche digitali definisce il concetto di *Biblioteca Digitale*, enfatizzando il ruolo che può assumere nel selezionare, conservare nel tempo e rendere immediatamente disponibili le conoscenze condivise da una comunità. Le biblioteche digitali nascono come estensione naturale delle biblioteche cartacee tradizionali; lo scopo è il medesimo: facilitare l'accesso sia fisico che intellettuale alla conoscenza, ai documenti ed alle informazioni ad esse contenute. Le tecnologie necessarie per la creazione di una biblioteca digitale sono molto differenti da quelle utilizzate per la costruzione di una biblioteca tradizionale: stamperie e tipografie sono sostituite da computers, files, monitors e fili elettrici in grado di trasportare un'informazione da un punto all'altro in un tempo infinitesimale. Proprio la velocità di condivisione delle informazioni rappresenta un fondamentale punto di forza delle biblioteche digitali rispetto quelle tradizionali.

Si può ritenere, e credo non a torto, che le biblioteche digitali possano infuenzare in maniera profonda e vasta alcuni settori collegati al mondo dell'informazione e della

divulgazione. Un settore, ad esempio, in cui l'impatto apportato da queste tecnologie potrebbe essere molto rilevante è quello dell'informazione scientifica, un settore sempre in evoluzione e, spesso, in modo rapido.

Le biblioteche digitali nascono e si sviluppano sull'onda del successo planetario, e tuttora in aumento, di Internet e del Web in generale. Proprio da queste tecnologie nasce la consapevolezza che un documento può non esistere solamente in formato cartaceo, ma astrarsi da esso ed avere un'identità anche, e solamente, in formato digitale.

Gli sforzi che, durante gli ultimi, anni sono stati fatti per creare standard in grado di permettere scambi di informazioni tra sistemi differenti (si pensi ad esempio a standard come l'ODBC, il TCP/IP oppure CORBA) hanno permesso di compiere passi da giganti anche nel settore delle biblioteche digitali. Proprio in questo senso uno standard che avrà un impatto sempre crescente sulla tecnologia e sul modo di implementare biblioteche digitali è XML(eXtensible Markup Language).

XML è un linguaggio standard, proposto dal World Wide Web Consortium (W3C), per la rappresentazione e lo scambio di dati sul Web. Questo linguaggio, la cui influenza sta rapidamente aumentando negli ultimi tempi, è la naturale evoluzione dello standard HTML che rappresenta, tuttora, il metodo di rappresentazione più utilizzato delle pagine Web visibili da Internet in tutto il mondo. XML non si occupa, a differenza di HTML, di come apparirà un documento, ma solamente del suo contenuto. Questo standard è infatti un linguaggio per la marcatura di documenti ideato al fine di rendere le informazioni contenute in essi *self-describing*. I marcatori (o *tags*) usati per produrre un documento XML vengono, infatti, decisi dal creatore del documento stesso e possono, quindi, essere usati per spiegare il contenuto delle informazioni, ovvero fornirgli una semantica.

Risulta comprensibile, quindi, come le biblioteche digitali possano trarre vantaggio dall'utilizzo di uno standard (appunto XML) ampiamente utilizzato in Internet ed in grado di fornire ogni genere di informazione digitale(informazioni testuali, immagini, video, audio, ecc...).

Si può così parlare di *biblioteche digitali XML* nei casi in cui lo standard XML viene utilizzato come strumento per la definizione dei metadati ed eventualmente dei documenti digitali, nonchè per la specifica delle caratteristiche della biblioteca digitale e della sua interfaccia con il mondo esterno, in modo da garantire interoperabilità fra le diverse realizzazioni di biblioteche.

1.2 Un'architettura aperta per biblioteche digitali XML

Un'architettura per biblioteche digitali viene detta aperta quando l'insieme delle funzionalità complessive viene partizionato in un gruppo di servizi autonomi ben definiti ed in grado di cooperare fra loro. Tali servizi possono essere distribuiti oppure replicati. Un'architettura di questo tipo può essere fondamentale nello sviluppo di biblioteche digitali il cui funzionamento non si limiti alle semplici funzionalità di ricerca remota e distribuita delle informazioni; al giorno d'oggi, infatti, le caratteristiche che possono essere richieste ad oggetti di questo tipo sono molteplici: si pensi, ad esempio, a funzionalità per il controllo e l'aggiornamento delle versioni dei documenti presenti, al controllo del copyright o a richieste di mediazione ed organizzazione degli oggetti digitali. Risulta essere dunque chiaro come sia necessario lo sviluppo di architetture aperte che permettano la facile estensione delle biblioteche digitali implementate con nuovi servizi distribuiti in rete e, fatto anch'esso molto importante, facilitino la cooperazione e lo scambio di informazioni fra diverse biblioteche. Proprio nell'ottica di un'architettura aperta può essere risolutivo l'utilizzo dello standard XML che, oltre alla definizione dei documenti digitali e dei metadati utilizzati, fornisca le specifiche per la sua interfaccia con il mondo esterno, premessa fondamentale per l'interoperabilità fra le diverse biblioteche.

Consideriamo ora come deve essere implementato un servizio e quali sono i basilari servizi in una biblioteca digitale in una architettura aperta. Ogni servizio è implementato attraverso l'utilizzo di uno specifico modulo software, detto server, accessibile tramite un prestabilito protocollo che ne definisce l'interfaccia pubblica. Tale protocollo deve essere in grado di gestire l'insieme delle richieste di servizio definendone il formato delle richiesta, il formato della riposta e le eccezioni. Il nucleo di servizi fondamentali per la realizzazione di una biblioteca digitale sono:

- *Servizio di archiviazione*: servizio che fornisce i meccanismi e le funzionalità per l'accesso agli oggetti digitali. Tale servizio deve essere in grado di fornire anche una rappresentazione della struttura degli oggetti contenuti. L'archiviazione delle informazioni risulta essere, ovviamente, un servizio imprescindibile e deve essere organizzato in maniera robusta e funzionale.

- *Servizio di naming*: servizio che ha lo scopo di gestire la creazione e l'assegnazione di nomi unici agli oggetti presenti nell'archivio. Ogni oggetto digi-

tale deve possedere, infatti, un nome persistente ed unico all'interno dell'intera biblioteca, in modo da permetterne l'identificazione ed il recupero.

- *Servizio di indicizzazione*: servizio che consente la ricerca di oggetti digitali (testuali o multimediali) attraverso l'impiego di indici costruiti in modo automatico sugli oggetti contenuti in archivio.

- *Servizio di collezione*: servizio in grado di fornire i meccanismi per l'aggregazione dinamica di insiemi di oggetti digitali contenuti nella biblioteca. Tali collezioni devono essere significative nell'ottica della comunità che impiega la biblioteca.

- *Interfaccia utente*: servizio necessario per fornire un punto d'accesso alla biblioteca digitale agli utenti che vogliono farne uso.

1.2.1 La Biblioteca Digitale OPEN-DLIB

Un prototipo di sistema aperto per biblioteche digitali basate sullo standard XML è rappresentato dal sistema OPEN-DLIB[16]. OPEN-DLIB è stato sviluppato dall'istituto ISTI-CNR di Pisa ed è un sistema in grado di creare facilmente una biblioteca digitale in accordo con le richieste e le preferenze di una certa comunità di utenti. OPEN-DLIB consiste in un complesso di servizi distribuiti in rete e fra loro interoperanti, che implementano le funzionalità che possono essere richieste ad una biblioteca digitale. La versione attuale di OPEN-DLIB fornisce una selezione di servizi interconnessi che forniscono solamente le attività basilari, quali l'acquisizione di dati, l'archiviazione degli stessi, l'esplorazione e la descrizione delle informazioni, insieme a servizi quali l'autenticazione e l'autorizzazione degli utenti. L'insieme di servizi forniti, seguendo l'ottica dei sistemi aperti, non è fissato a priori, ma può essere facilmente esteso per fornire funzionalità aggiuntive. I diversi servizi implementati da OPEN-DLIB possono essere:

- *Centralizzati*: un servizio centralizzato ha sempre una sola istanza all'interno della biblioteca digitale. I servizi centralizzati sono soprattutto quelli riguardanti la sicurezza, servizi di autorizzazione ed autenticazione degli utenti, in cui la replicazione e la distribuzione può rappresentare un rischio.

- *Distribuiti*: i servizi distribuiti possono essere implementati attraverso molteplici istanze. Ogni istanza del servizio distribuito si occupa di lavorare su un insieme di dati allocati su di un particolare server. I servizi distribuiti vengono

impiegati soprattutto per la gestione di grandi quantità di dati. Ogni server, quindi, possiede una certa quantità di dati ed è sfruttato da un'istanza dei servizi distribuiti in grado di lavorare su di essi.

- *Replicati*: i servizi replicati sono quei servizi, possibilmente situati su differenti servers, in grado ,ognuno, di gestire l'intera quantità di dati (nell'intero sistema) per cui è stato progettato. La replicazione di un servizio può peggiorarne le prestazioni ma fornire una maggiore robustezza e resistenza ai guasti.

Le diverse istanze dei servizi comunicano fra loro tramite un protocollo denominato OLP (OPEN-DLIB Protocol). Le richieste, attraverso il protocollo OLP vengono espresse attraverso URLs inclusi in richieste http. Tutte le richieste o le risposte strutturate sono basate sullo standard XML.

1.3 Il progetto ECD

Il progetto ECD (*Enhanced Content Delivery*), progetto nell'ambito del quale è stata sviluppata la presente tesi, si concentra sullo sviluppo di tecnologia e strumenti per offrire contenuti arricchiti (da cui il nome del progetto) agli utenti finali (per informazioni sul progetto ECD, finanziato dal MIUR ai fini della legge per i Progetti Strategici 449/97, si veda il sito http://www.ecd.nuce.cnr.it). A questo progetto collaborano numerosi enti di ricerca come l'istituto ISTI-CNR di Pisa, il politecnico di Milano e le università di Modena e Reggio Emilia, Padova e Roma 3. Il programma ECD si propone di offrire, agli utenti che ne fanno richiesta, materiale digitale proveniente da fonti diverse ed eterogenee, trasformato, organizzato, con l'aggiunta di metadati e di informazioni utili la fine della sua qualificazione. Il progetto ECD punta, per raggiungere il suo scopo, su tecnologie, oggi possibili per il grande sviluppo che reti di telecomunicazioni e sistemi digitali hanno avuto, di ricerca sul Web e, soprattutto, di *biblioteche digitali*. Tramite la ricerca sul Web e l'impiego di biblioteche digitali agli utenti può essere offerta una infinita quantità di dati ed informazioni non necessariamente solo sul piano testuale, ma comprensive di audio, filmati video, immagini, ecc.

I principali obiettivi che questo progetto di ricerca si propone di raggiungere sono:

- Sviluppo di algoritmi per indicizzare e ricercare documenti in formato compresso

- Sfruttare tecniche di High Performance Computing per fronteggiare le moli dei dati e il numero di utenti dei servizi
- Sviluppo di tecniche di Web Mining per determinare:
  1. rank o autorevolezza delle fonti
  2. come migliorare le prestazioni di spidering e caching
  3. come classificare i documenti
- Sviluppo di algoritmi per indicizzare e ricercare documenti in formato compresso
- Sviluppo di servizi di ricerca partecipativa e decentralizzata
- Sviluppo di un'architettura aperta per Biblioteche Digitali distribuite
- Utilizzo di XML per strutturare documenti ed esprimere metadati
- Fornire accesso a documenti multimediali nelle Digital Libraries
- Formulare e rispondere a interrogazioni su schemi XML
- Sviluppare servizi avanzati per gli utenti quali: annotazioni di documenti,notifica, supporto al lavoro di gruppo

All'interno di questa tesi ci si occuperà in particolar modo dei seguenti due aspetti del programma:

- Utilizzo di XML per strutturare documenti ed esprimere metadati
- Formulare e rispondere a interrogazioni su schemi XML

1.4 Interrogare un archivio di documenti XML

Un grande problema nel gestire una biblioteca digitale è recuperare in maniera efficiente ed efficace i documenti e le informazioni desiderate. In pratica l'utente deve poter porre, tramite un'apposita interfaccia grafica in grado di aiutarlo nel compito, una richiesta, o *query*, sull'archivio (o *repository*) in cui sono mantenuti i dati in possesso del sistema. Occupandoci di biblioteche digitali XML il problema viene ristretto al compito di reperire informazioni su un insieme di documenti XML. XML, come già

descritto in precedenza, è un linguaggio *self-describing* in cui i marcatori che racchiudono i dati sono scelti dall'autore del documento, e non imposti dall'ideatore dello standard. I documenti XML, inoltre, sono semistrutturati: non è presente, cioè, per ciascun oggetto o attributo una struttura ben definita, ma può variare anche all'interno dello stesso documento XML.

Numerose sono le proposte fatte in questi ultimi anni per linguaggi per l'interrogazione di documenti semistrutturati, ed in particolare XML, ma nessuno standard effettivo è ancora stato trovato. Il W3C, ad esempio, ha proposto alcuni linguaggi come XML-QL e XQuery (forse proprio quest'ultimo è destinato a prendere il sopravvento sugli altri, data la sua flessibilità). Un altro interessante metodo per l'interrogazione di dati XML è quello fornito dall'università di Stanford nel progetto *Lore*. Lore rappresenta un completo e multi-utente database system in grado di trattare dati semistrutturati. All'interno progetto Lore è stato sviluppato un linguaggio per interrogazioni, detto *Lorel*[35], in grado di gestire richieste per dati semistrutturati e quindi anche per XML.

Un grande aiuto nel campo dell'interrogazione di un archivio di documenti digitali può essere dato dall'impiego di *metadati*, cioè di schemi che descrivono la struttura dei documenti limitando così notevolmente il numero di dati da valutare per rispondere ad un interrogazione. Come vedremo in seguito, inoltre, i metadati possono essere analizzati ed 'allineati' semanticamente per facilitare la risoluzione di query su più schemi . Nel prossimo paragrafo si pone l'accento sul concetto di metadati e sul loro impiego all'interno delle biblioteche digitali XML.

1.4.1 I metadati

I metadati sono un concetto di grande importanza nello sviluppo di biblioteche digitali. I metadati sono i dati che hanno come compito la descrizione del contenuto e degli attributi di ogni oggetto presente all'interno di una biblioteca digitale. Il procedimento che permette la rappresentazione del contenuto di un insieme di documenti, non è sfruttato solamente dai costrutori di biblioteche in formato digitale, ma anche da chi si occupa di quelle tradizionali: basta, infatti, pensare ai cataloghi in cui i libri e le riviste sono descritte e schedate in tutte le comuni biblioteche. L'impiego di metadati è molto importante all'interno di una biblioteca digitale, può, infatti, essere considerato la chiave per ritrovare ed usare ogni documento desiderato. Questa affermazione può essere ben compresa se pensiamo agli attuali motori di ricerca per pagine Web,

utilizzabili via Internet (Google, Altavista,Yahoo): essi, dopo aver ricevuto un richiesta da parte di un utente, non svolgono una ricerca *full text* (cioè sull'intero corpo del testo di tutti i documenti), poichè una ricerca del genere risulta essere poco performante e fornire, inoltre, all'utente migliaia di documenti utili solamente in minima parte. L'impiego di metadati appare così utilissimo per rispondere in maniera appropriata alle richieste poste dagli utenti, specialmente se il numero di documenti che formano la biblioteca digitale è molto elevato.

Ma quali metadati si possono utilizzare per una biblioteca digitale XML?

Risulta piuttosto evidente, che per una biblioteca digitale basata su tecnologia XML, i metadati debbano fornire descrizioni e strutture dei documenti XML che trattano. Un ottimo modo per rappresentare i metadati è, quindi, attraverso l'impiego di linguaggi atti alla costruzione di schemi XML. Allo stato attuale dei fatti, circa una dozzina di linguaggi per schemi XML sono stati proposti. Lo standard attuale *de facto* è fornito da DTD XML (DTD significa Document Type Definition) ma le sue possibilità espressive sono leggermente ridotte rispetto a quelle degli altri linguaggi proposti. Il linguaggio che, a parere di molti, prenderà il posto delle DTD è Xml-Schema[], nuovo standard per la validazione di documenti XML proposto dal W3C. Xml-Schema, oltre a risultare più flessibile rispetto alle DTDs, presenta l'indiscusso vantaggio di sfruttare lo stesso linguaggio XML. Questo è un vantaggio enorme in ambiente come le quello dell biblioteche digitali XML in cui, sfruttando Xml-Schema, sia i dati che i metadati possono essere trattati dagli stessi strumenti. Per un'analisi comparativa dei vari standard di rappresentazione di schemi XML si rimanda a [29]. Nell'ambito di questa tesi, e nell'ottica del progetto ECD, i metadati saranno da considerarsi in formato Xml-Schema.

1.4.2 XQuery: un linguaggio per interrogare documenti XML

Il linguaggio di interrogazione XQuery 1.0 [18, 17] è stato disegnato per soddisfare i requisiti identificati, al fine di interrogare documenti XML, dal W3C XML Query Working Group. XQuery deriva da un linguaggio di interrogazione chiamato Quilt che, a sua volta, prende alcune caratteristiche da numerosi linguaggi quali: XPath1.0, XQL, XML-QL, SQL e OQL. Xquery è molto flessibile e permette sia interrogazioni facilmente comprensibili da un utente, che interrogazioni basate più specificatamente sulla sintassi di XML e Xml-Schema. XQuery, inoltre, risulta essere fortemente

tipizzato ed il sistema di tipi presenti al suo interno è basato su quello fornito da Xml-Schema (si veda il capitolo 3).

Nell'ambito di questa tesi non ci interessa tutta la sintassi di questo linguaggio di interrogazione, che risulta essere molto ampia e, tal volta, complessa, bensì un sottoinsieme di essa data dalle cosiddette espressioni *FLWR* (si pronuncia 'flower') e da una parte delle *path expressions* permesse. Le espressioni FLWR permettono, tramite l'impiego di parole chiave quali *for*, *where* e *return*, di percorrere ricorsivamente l'albero definito dal documento XML e restituire i valori richiesti nel formato voluto. Tutto ciò è possibile per il semplice fatto che anche una espressione XQuery rappresenta, come un documento XML, un albero.

Per comprendere più a fondo quest'ultima affermazione consideriamo il seguente foglio XML:

```
...
<Libro>
    <Titolo>Xml 1.0 per  tutti </Titolo>
    <Autore>Marco Rossi</Autore>
        <Capitolo>
            <Argomento>Introduzione</Argomento>
            <Titolo>Introduzione</Titolo>
        </Capitolo>
        <Capitolo>
            <Argomento>XML</Argomento>
            <Titolo>Linguaggio</Titolo>
        </Capitolo>
     ...
</Libro>
...
```

Si supponga, ora, di voler porre una query, in formato XQuery, per poter trovare il titolo di ogni capitolo di ogni libro, il cui argomento sia 'XML'.

Tale richiesta assume la forma della seguente espressione:

```
for $a in /Libro/Capitolo
   where $a/Argomento = XML
   return $a/Titolo
```

Si può notare come attraverso l'utilizzo delle parole chiave *for* e *where*, e l'impiego di espressioni e variabili ($a) che indichino i percorsi (*path*) desiderati, si esprimano le condizioni da soddisfare per il ritrovamento dell'informazione, mentre, attraverso l'uso della parola chiave *return*, si esprima il risultato da ottenere ed il formato in cui il tale risultato deve comparire. La query considerata può essere visualizzata graficamente tramite una rappresentazione ad albero in cui i nodi intermedi rappresentano gli elementi XML citati, mentre le foglie sono i contenuti degli stessi. Si consideri la Figura(1.1):

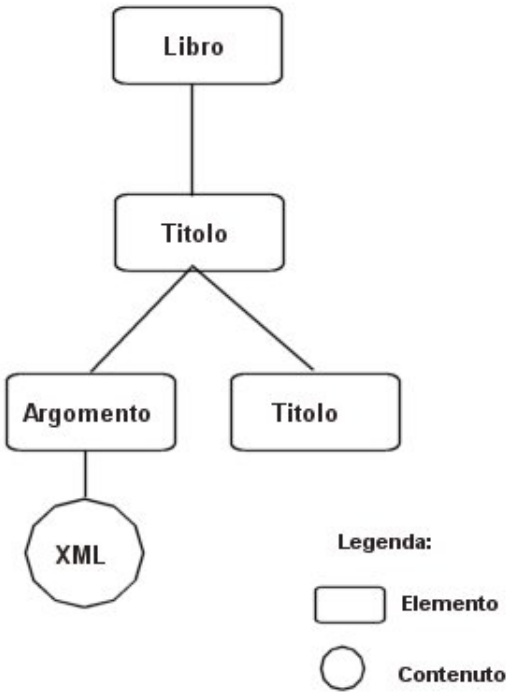

Figura 1.1: Albero rappresentativo della XQuery

Dato che, proprio per la sua struttura, anche i documenti XML sono rappresentabili tramite l'utilizzo di alberi, la risoluzione di una XQuery diviene, in realtà, un problema di confronto fra alberi e fra i cammini che li compongono.

1.4.3 Il problema della riscrittura delle query

La scelta e l'implementazione di un linguaggio per interrogare documenti o schemi XML non può bastare per risolvere tutti i problemi riguardanti il reperimento dei dati, da parte di un utente, all'interno dell'archivio di una biblioteca digitale: supponendo, infatti, che l'archivio considerato sia molto esteso, non è possibile, per l'utente, conoscere la struttura dell'intero repository (o di tutti gli schemi dei suoi documenti) e porre in maniera precisa ed efficace la query voluta. L'utente deve poter porre la query sull'**insieme degli dei documenti dell'archivio**. La query deve poi essere riscritta, in modo automatico, per ogni documento in grado fornire informazioni interessanti. L'unione dei dati reperiti in questa maniera rappresenta la risposta alla richiesta dell'utente.

Al fine di capire meglio questo procedimento consideriamo il seguente esempio:

Supponiamo che un archivio appartenente ed una biblioteca digitale contenga alcuni documenti XML la cui struttura sia conforme agli schemi (espressi ad esempio in Xml-Schema) rappresentati in Figura (1.2). Osservando la figura si può notare come i due schemi siano fra loro molto simili e descrivano, con termini leggermente differenti, lo stesso argomento. Si supponga, ora, che un utente, utilizzando la biblioteca digitale, sia interessato ad ottenere tutti i titoli di ogni capitolo che parli di dati semistrutturati, appartenete ad ogni trattato memorizzato in formato digitale in archivio. Si supponga, anche, che l'interfaccia utente permetta di formulare richieste utilizzando il nuovo linguaggio per interrogazioni XQuery proposto dal W3C.

La richiesta posta dall'utente, si rammenti che l'utente non è a conoscenza della struttura di tutti i documenti nel repository, è la seguente:

```
for $a in /Trattato/Capitolo
   where $a/Argomento = XML
   return $a/Titolo
```

evidentemente la richiesta non può essere soddisfatta se non applicando alla query sottomessa alcune modifiche da parte del sistema: il percorso (*path*) */Trattato/Capitolo* infatti, può essere ritrovato nel secondo documento XML, mentre i termini *Argomento* e *Titolo* appartengono al primo XML. In pratica, sostituendo alla variabile $a il suo valore (*/Trattato/Capitolo*) all'interno della clausola where e del costrutto

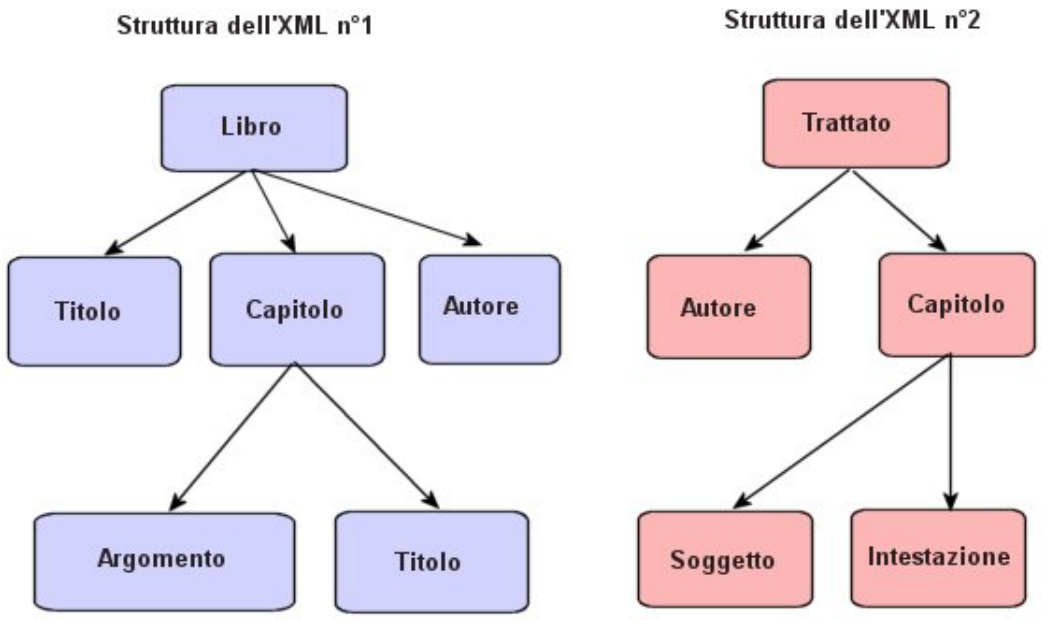

Figura 1.2: Schemi di due documenti XML in archivio

return, si ottengono i path */Trattato/Capitolo/Argomento* e */Trattato/Capitolo/Titolo* che non esistono all'interno dell'archivio. L'operazione da fare è, quindi, riscrivere automaticamente la query originaria in due richieste che possano effettivamente dare un risultato (si deve riscrivere una query per ogni documento ritenuto di interesse). Tali richieste risultano essere le seguenti:

La query da porre sul primo schema:

```
for $a in /Libro/Capitolo
   where $a/Argomento = XML
   return $a/Titolo
```

La query che deve, invece, essere posta sul secondo schema è:

```
for $a in /Trattato/Capitolo
   where $a/Soggetto = XML
   return $a/Intestazione
```

Appare così necessario l'impiego, all'interno della biblioteca digitale, di un modulo mediatore il cui compito sia quello di risolvere le differenze superficiali degli schemi dei vari documenti, permettendo di riscrivere le query in modo corretto su ogni documento di interesse.

Nel prossimo paragrafo sarà descritto lo stato dell'arte attuale per i problemi di riscrittura delle query, ed in quello successivo si parlerà di come il progetto ECD, con la collaborazione dell'Università di Modena e Reggio Emilia, si sta muovendo per risolvere tale problema. Nell'ottica di questo progetto è stata prodotta la presente tesi.

1.4.4 Lo stato dell'arte

In questo paragrafo sarà presentato un breve excursus sullo stato dell'arte per quanto riguarda l'interrogazione approssimata dei documenti XML e la riscrittura delle query, problema accennato all'interno del paragrafo precedente.

Abbiamo visto, parlando del linguaggio di interrogazione per documenti XML XQuery, come la risoluzione di un interrogazione su un archivio di dati semistrutturati possa essere ricondotta al confronto fra due struttura ad albero: sia le query che i documenti XML (o i loro schemi), infatti, possono essere rappresentati per mezzo di gerarchie ad albero. A partire da questa assunzione sono stati sviluppati una serie di lavori per permettere il confronto approssimato fra due differenti strutture ad albero, per poter trovare tutti i documenti di un archivio che sono in grado soddisfare sufficientemente 'l'albero' della query. In [43] viene proposto un metodo per il confronto di due alberi per mezzo di un algoritmo basato sul concetto di *edit distance*. In questo lavoro si afferma che la distanza fra due alberi (di cui uno può essere l'espressione della struttura richiesta da una query ed il secondo della struttura di un documento formato da dati semistrutturati) può essere misurata in base al costo della sequenza di operazioni necessarie per tramutare il primo albero nel secondo. Le operazioni identificate a questo scopo sono tre:

- Inserimento di un nodo
- Cancellazione di un nodo
- Modifica della *label* (ovvero del nome) di un nodo

In base a queste tre operazioni siamo in grado di trasformare la struttura di un albero (qualunque essa sia) in quella di qualsiasi altro. Ad ognuna delle tre operazioni citate viene assegnato un costo (espresso da un valore reale positivo). Dalla somma di tutti i costi di una sequenza di operazioni in grado di cambiare la struttura dell'albero T_1 in quella dell'albero T_2, si evince la distanza che intercorre fra i due. Sfortunatamente, però, gli stessi autori del metodo hanno dimostrato che confrontare in questo modo due strutture ad albero è un problema NP-completo. La complessità computazionale dell'algoritmo impiegato è, quindi, notevolmente elevata e diviene, per i mezzi attuali, proibitiva se il confronto viene effettuato su un archivio di documenti molto ampio.

Il lavoro che viene presentato in [41] tratta la ricerca approssimata in alberi non ordinati in maniera nettamente differente. L'approccio discusso può essere utilizzato al fine di ritrovare informazioni interessanti in strutture che possono essere rappresentate tramite alberi (eg. XML e dati semistrutturati in generale). La query viene vista anch'essa come un albero formato dai *path* elencati all'interno della richiesta stessa. IL problema, chiamato in gergo *approximate nearest neighbor search* (ANN), è il seguente: dato un numero intero *DIFF* un'albero descritto dalla query, detto *Q*, ed un database di alberi di dati(estratti, ad esempio, da un archivio di documenti XML) *D*, trovare tutti gli alberi appartenenti a *D* che contengono un sottoalbero *D'* la cui distanza da *Q* risulti essere inferiore a *DIFF*. Si consideri la Figura ():

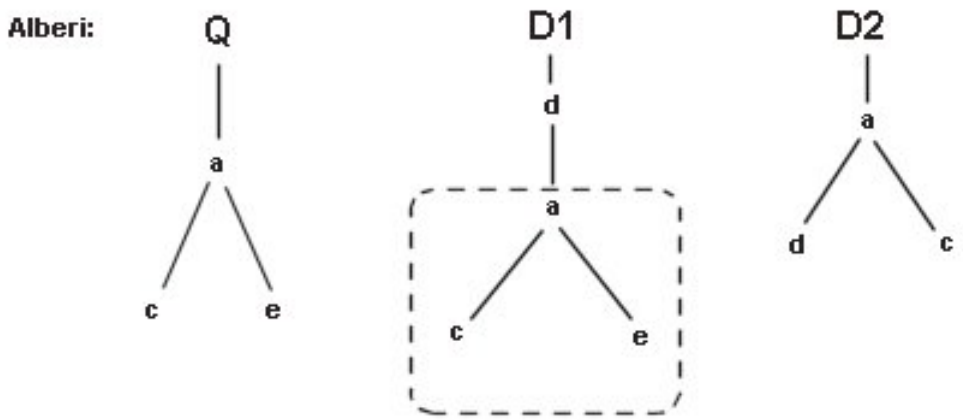

Figura 1.3: Alberi d'esempio

L'albero rappresentato dalla query, *Q*, è contenuto completamente all'interno di *D1* (l'area tratteggiata rappresenta *D'*), ma solo il percorso dall'elemento *a* all'elemento *c* si ritrova sia in *Q* che in *D2*. La distanza fra la query ed il documento la cui struttura è rappresentata da *D1* sarà nulla (pari a 0), la distanza, invece, fra la query e l'altro albero è pari a 1. Tale distanza, infatti, viene calcolata contando i percorsi (*path*) presenti in *Q* e non nell'albero dei dati; se questa distanza supera l'intero *DIFF*, l'albero dei dati selezionato non è ritenuto interessante per la query. Questo metodo, in pratica, definisce la distanza fra una query ed uno schema come il numero di percorsi presenti all'interno della richiesta ma non all'interno dello schema. Questo metodo, pur essendo computazionalmente molto più leggero del precedente, non tratta la semantica delle query e dei documenti, non potranno, quindi, essere identificati come simili documenti che trattano gli stessi concetti, ma con termini differenti.

Un'altro interessante lavoro, più simile a quello che verrà sviluppato in questa tesi, è quello descritto ad Torsten Schlieder in [38]. Le fondamenta su cui si basa tale lavoro sono riassunte perfettamente dalla frase -*Un motore per risolvere query XML dovrebbe trovare il miglior risultato possibile: se non si ritrova un matching esatto con i documenti, i risultati simili dovrebbero essere cercati e presentati in ordine di importanza all'utente* -. L'utente, per poter formulare in modo corretto una query, deve conoscere perfettamente la struttura di tutti i documenti presenti in archivio, compito quanto mai complesso. Per facilitare il compito dell'utente la query deve essere trasformata e adattata agli alberi rappresentanti i documenti nel repository (si tratta esattamente del problema di riscrittura delle query citato in precedenza). A tal fine, Schlieder propone un nuovo linguaggio per l'interrogazione di documenti XML chiamato *ApproXQL*[37]. ApproXQL è un semplice linguaggio di *pattern-matching* che non distingue fra elementi ed attributi, ed un esempio è fornito da:

```
cd[ title ["piano" and "concerto" ] and composer["rachmaninov"]]
```

in cui si richiedono tutti i cd di Rachmaninov il cui titolo sia *concerto* o *piano*. ApproXQL non tratta in modo differente attributi ed elementi XML, rendendo la sintassi delle interrogazioni notevolmente più semplice. Non basta, però, semplicemente cercare una corrispondenza diretta fra l'albero fornito dalla query e gli alberi dei dati: per trovare risultati similari (ad esempio titoli come 'pianoforte') si usano le *basic query trasformations*. Queste trasformazioni, che vengono effettuate sull'albero della query, possono inserire, eliminare nodi o modificarne la *label* (cioè il nome) per adattare lo

schema della query a quelli dei documenti dell'archivio. Ad ogni documento ritrovato in questo modo viene associato, quindi, un costo detto di *trasformazione*. Sono ritenuti interessanti solamente i documenti per i quali il costo di trasformazione della query non supera una certa soglia. Un costo di trasformazione basso rappresenta una buona similarità fra la query ed il documento considerato. Il procedimento di trasformazione di una query è computazionalmente pesante (il numero di trasformazioni possibili, soprattutto se l'archivio è grande, può essere elevatissimo) e può essere aiutato dalla presenza di schemi rappresentanti la struttura dei documenti XML.

Un approccio differente è quello descritto in [9] dai professori P.Ciaccia e W.Penzo dell'Università di Bologna. In questo lavoro si parla del rilassamento dei requisiti posti in una query come un *'must'*, nel caso si stia lavorando con un insieme di documenti molto grande e vario, al fine di evitare risultati inconsistenti. Il risultato ottenuto , allo scopo di trovare e classificare similitudini fra l'albero della query con quelli dei documenti, consiste in una funzione di similarità detta *SATES* (*Scored Approximate Tree Embedding Set*). SATES fornisce come risultato, per ogni query, un insieme di alberi di dati (classificati a seconda della similarità) che approssimano quello della query. Il risultato è ottenuto sfruttando eventuali affinità semantiche presenti fra i nomi delle foglie e dei nodi degli alberi della query e dei documenti. Sfruttando questo approccio, anche se computazionalemnte pesante, si possono ritrovare informazioni desiderate ma immagazzinate in archivio tramite termini simili, ma non uguali, a quelli espressi dalla query.

Un importante lavoro che, invece, sfrutta l'utilizzo di un sistema mediatore per la riscrittura delle query viene offerto dal progetto MIX [1] (Mediation of Information using XML) sviluppato presso il Supercomputer Center di San Diego (SDSC). La tecnologia MIX è stata impiegata per lo sviluppo di una parte della California Digital Library i cui documenti possono essere richiesti ponendo una query usando un'interfaccia grafica (chiamata BBQ). La BBQ fornisce le risposte utilizzando il query processor MIX in grado di processare le richieste e fornire le risposte basandosi su viste madiate in XMAS (un linguaggio di interrogazione basato su XML) degli oggetti in archivio.

L'ultimo lavoro accennato in questa sede al fine di risolvere il problema della riscrittura della query, prima di descrivere l'approccio utilizzato nell'ambito della tesi, è quello presentato da S.Melnik, H.Garcia-Molina ed E.Rahm in [32]. Gli autori presentano un algoritmo di *matching* in grado di trovare le corrispondenze fra i nodi di

due alberi (che possono essere schemi di documenti o altre strutture di questo tipo). L'algoritmo descritto prende il nome di *Similarity Flooding Algorithm* ed usa un approccio basato sulla considerazione che il concetto di similarità si propaga da un nodo a quelli vicini (eg. se un nodo appartenente ad un albero detto T_1 risulta essere simile ad un nodo appartenente ad un albero T_2 è probabile che anche fra i nodi ad essi vicini esista un certo grado di somiglianza). La prima cosa da fare è ottenere valori di similarità iniziali fra i nodi dei due alberi (in [32] si parla, ad esempio, di operatori che utilizzino procedimenti di confronto fra stringhe); nel passo successivo entra in gioco il vero e proprio algoritmo di *Similarity Flooding* che iterativamente modifica il grado di silarità fra i nodi (alterando quindi il *mapping* iniziale fra la due strutture ad albero) *propagando* i più alti gradi di similitudine dai nodi che li posseggono a quelli strettamente connessi con loro. L'algoritmo viene eseguito fino a quando i valori di similarità fra i nodi (visti come numeri compresi fra zero e uno) si stabilizzano o, in caso contrario, un numero prefissato di volte.

1.5 Ontologie Xml

In questo paragrafo si introduce il metodo sviluppato in questa tesi per la risoluzione di query poste su un archivio di documenti XML. Un simile archivio può essere rappresentato dal *repository* di una biblioteca digitale.

Un aspetto fondamentale nella realizzazione di una biblioteca digitale è la modalità di organizzazione dell'enorme quantità di informazione relativa al suo contenuto. Importantissimo, a questo proposito, è l'impiego di schemi (per documenti XML si propone di usare Xml-Schema) che descrivono il contenuto dell'archivio. Per rappresentare in maniera caratterizzante la conoscenza offerta dai metadati presenti negli schemi, è possibile impiegare ontologie e tecniche di ragionamento basate su di esse. Una ontologia può essere vista come un insieme di termini (vocaboli) in grado di definire in modo univoco un determinato concetto (per una definizione più completa di ontologia ed una classificazione dei vari livelli di ontologie si veda il capitolo trattante WordNet). Tramite l'utilizzo di ontologie, dunque, è possibile associare un concetto ad ogni elemento espresso dallo schema (o vista) rappresentante un insieme di documenti in un archivio. Risulta quindi evidente come l'impiego di ontologie, e di tecniche di ragionamento basate su di esse, possa fornire uno strumento efficace per un accesso selettivo ed efficiente alla enorme quantità di informazioni che pos-

sono essere immagazzinate all'interno di una biblioteca digitale. Inoltre, utilizzando ontologie assieme ai metadati ed agli schemi degli oggetti contenuti in archivio è possibile, esprimendo i concetti collegati alle viste, risolvere il problema della riscrittura delle query su schemi differenti (problema descritto nel paragrafo precedente).

Il punto di partenza del progetto ECD su questo tema di ricerca è rappresentato dal sistema MOMIS. MOMIS[] (Mediating system Environment for Multiple Information Sources) è un sistema progettato e realizzato presso l'Università di Modena e Reggio Emilia nell'ambito del progetto MURST INTERDATA 97/98. MOMIS è un mediatore che permette la costruzione di una vista (schema) globale ed integrata su un insieme di sorgenti di informazione eterogenee e distribuite. La vista globale virtuale (GVV) è ottenuta a partire dalla rappresentazione, tramite l'impiego di ontologie, dei metadati, che descrivono lo schema di ogni fonte locale. Ogni schema può essere, tramite l'impiego di un interfaccia grafica, *annotato*, cioè, ad ogni oggetto rappresentato in uno schema può essere associato un significato. Questo procedimento viene eseguito da MOMIS tramite l'impiego di WordNet, un database lessicale che rappresenta la più importante risorsa lessicale utilizzabile sia nel campo della linguistica computazionale che in quello dell'analisi testuale. WordNet è stato sviluppato dal Cognitive Science Laboratory della Università di Princeton dal professor George A. Miller. Tramite l'interazione con questo sistema, MOMIS è in grado di scoprire relazioni fra gli oggetti, basandosi sui concetti che esprimono, descritti nei vari metadati rappresentanti gli schemi delle sorgenti locali. Tali relazioni, trovate oltre che per mezzo di WordNet anche tramite l'impiego di tecniche basate sulle logiche descrittive, vengono raccolte in un thesaurus (dizionario) per mezzo del quale è possibile produrre la GVV. Il thesaurus di relazioni potrà essere impiegato con successo, come sarà mostrato in questa tesi, anche per la riscrittura di query su più schemi rappresentanti concetti compatibili fra loro.

Tramite l'impiego di MOMIS, unitamente a biblioteche digitali aventi una struttura aperta, come ad esempio OPEN-DLIB descritta in precedenza, sarà possibile gestire le interrogazioni poste dagli utenti su di un archivio di documenti digitali. In pratica, ciò che si vuole ottenere è una **Ontologia di Biblioteca Digitale XML**, che rappresenta l'insieme delle relazioni semantiche fra i concetti degli schemi XML. L'ontologia costituirà la strumento principale per mezzo del quale interrogare documenti XML con schemi differenti.

Osservando la Figura 1.4 e la Figura 1.5 si può notare il procedimento proposto

su cui si basa la riscrittura delle query per differenti documenti di un archivio XML: come prima cosa gli schemi dei documenti devono essere tradotti (tramite appositi wrapper), in schemi ODL_{I^3} (un linguaggio che verrà descritto nel proseguo della tesi), che possono essere usati da MOMIS ed *annotati* tramite l'impiego di WordNet. Il prodotto dei questa prima fase (Figura 1.4) è un insieme, o *thesaurus*, di relazioni semantiche che costituiscono l'ontologia di biblioteca digitale.

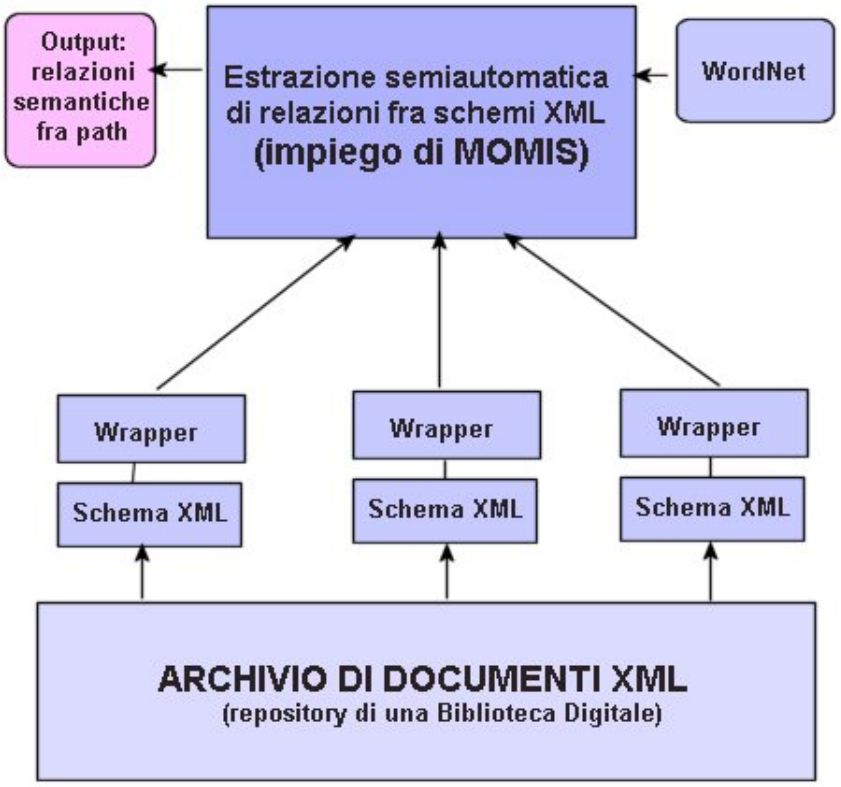

Figura 1.4: Estrazione di relazioni semantiche da schemi XML

Tramite il thesaurus di relazioni sarà possibile confrontare semanticamente i path degli schemi dell'archivio ed anche i path delle query con essi. Una volta individuati i path degli schemi maggiormente simili a quelli della query, essi potranno essere impiegati per la riscrittura della richiesta su ogni singolo schema di interesse(e quindi anche sull'insieme di documenti XML che rappresenta)(Figura 1.5).

In Figura 1.6 è rappresentata graficamente la parte del progetto ECD concernente l'impiego di MOMIS e al sua relazione con OPEN-DLIB.

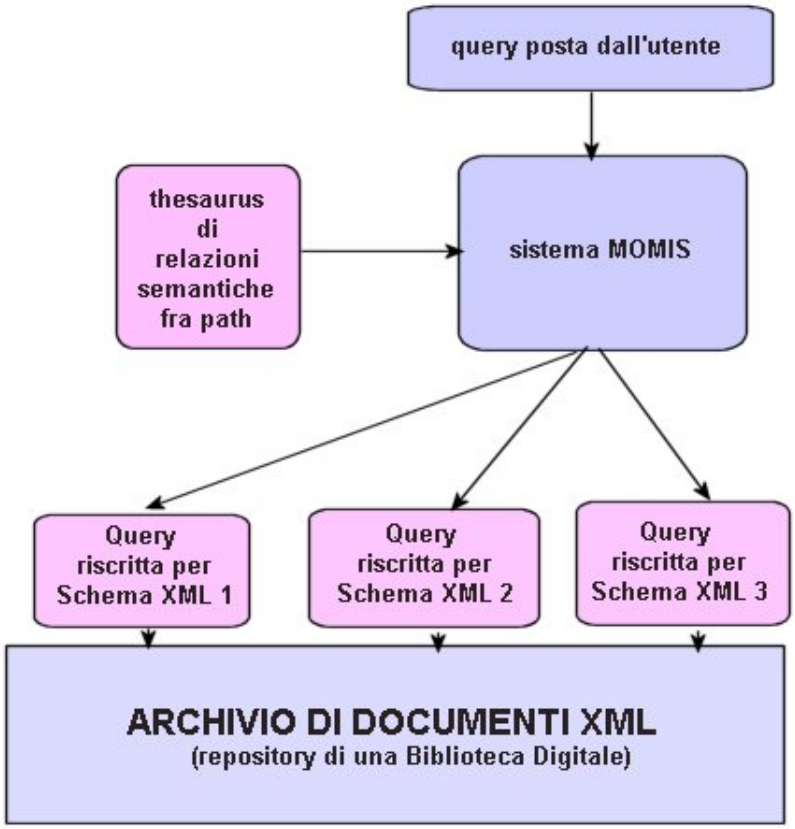

Figura 1.5: Riscrittura di query basandosi sul thesaurus di relazioni

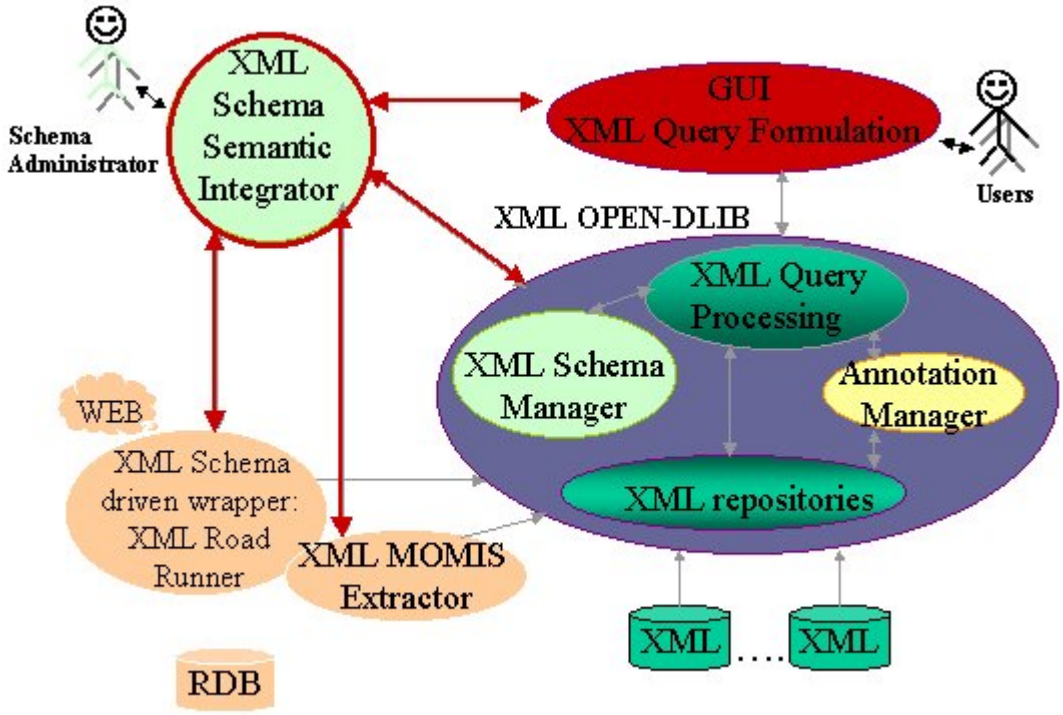

Figura 1.6: Parte del progetto ECD concernente l'impiego di MOMIS

Capitolo 2

Il sistema MOMIS

2.1 Introduzione all'integrazione delle informazioni

Al giorno d'oggi un problema cui devono far fronte numerose imprese ed organizzazioni è quello della dispersione del loro patrimonio informativo. Si pensi ai numerosissimi metodi di immagazzinamento di informazioni presenti sul mercato o utilizzabili gratuitamente: DBMS, pagine HTML, pagine XML, sistemi di Information Retrieval, File Systems e così via. Nel caso in cui un utente voglia reperire informazioni da sorgenti diverse, fatto che accade sempre più frequentemente oggigiorno, si trova di fronte a problemi di non facile soluzione: le sorgenti di conoscenza, infatti, sfrutteranno tecnologie differenti, difficilmente uniformabili, senza contare le possibili contraddizioni ed inconsistenze fra i dati ottenuti da diverse fonti. Un grande aiuto , per quanto concerne il problema dello sfruttamento di tecnologie differenti, viene dato dagli standard oggi esistenti (come l'ODBC, CORBA ed il TCP/IP) che risolvono il problema delle comunicazioni fra moduli diversi. Ciò che rimane irrisolta è la questione della modellazione delle informazioni: i modelli dei dati (e gli schemi in cui vengono sfruttati) possono differenziarsi gli uni dagli altri a tal punto da fornire, ognuno, una propria struttura logica di rappresentazione dei dati da immagazzinare. Tutto ciò crea una eterogeneità semantica non risolvibile dagli attuali standard. Altri problemi, di tipologia differente ma sempre concernenti lo stesso campo, sono l'information overload (sovraccarico di informazioni) dovuto all'utilizzo di un numero sempre maggiore di fonti, l'incremento del tempo di accesso e gli elevati costi di mantenimento per eliminare o aggiungere una nuova sorgente. Dalle problematiche sopra elencate si evincono le difficoltà che sorgono nel creare un sistema di integrazione e mediazione di informazioni eterogenee che sia affidabile, flessibile, modulare (in

modo da permettere il riuso di diverse parti all'evolvere delle tecnologie) e capace di interagire altri sistemi esistenti. Vi sono numerosi approcci all'integrazione descritti in letteratura o realizzati nella realtà come la reingegnerizzazione delle sorgenti mediante la standardizzazione degli schemi o l'utilizza di datawarehouse (sistemi in grado di fornire all'utente finale viste consistenti di porzioni di dati eterogenei, tali tecnologie sfruttano la replicazione fisica dei dati e pesanti algoritmi di riallineamento nel caso di modifiche nelle informazioni originali). Nel seguito verrà descritta una proposta dell'ARPA (Advanced Reserch Project Agency) per una architettura di integrazione di informazioni flessibile e riusabile. L'approccio descritto dall'ARPA[25] è stato seguito anche nel progetto MOMIS, ambito di lavoro per questa tesi.

2.2 L'integrazione intelligente delle informazioni

Come viene citato in[19], l'integrazione delle informazioni (I_2) si distingue da quella dei dati e dei database in quanto non cerca di collegare semplicemente alcune sorgenti, quanto risultati opportunamente selezionati da esse. Lo scopo dell'integrazione delle informazioni è, quindi, quello di ottenere una selezione ragionata dei dati prelevati dalle varie sorgenti, e proporre una fusione intelligente ed una seguente sintesi degli stessi. Proprio a questo scopo è stato sviluppato dall'ARPA un progetto di ricerca atto a fornire una architettura di riferimento che realizzi l'integrazione di sorgenti eterogenee in maniera automatica; il nome di questo progetto è, appunto, I3 (Integrazione Intelligente delle Informazioni). I risultati ottenuti in questo ambito sono molto importanti poiche danno una concreta indicazione di come costruire un sistema di mediazione che sia riusabile e la cui parti non presentino eccessivi costi di sviluppo. L'integrazione delle informazioni, inoltre, ne aumenta il valore, ma non è semplice gestire gli aggiornamenti, le eliminazione e le sostituzioni fra la varie sorgenti, le loro ontologie e semantiche. L'ARPA ritiene che una grossa mano a tal proposito possa essere data dall'utilizzo dell'*intelligenza artificiale* che, essendo in grado di dedurre dagli schemi delle sorgenti informazioni utili, può essere considerata uno strumento prezioso ed in grado di fornire soluzioni flessibili e riusabili. Secondo il programma I3 è opportuno costruire architetture modulari, in grado di abbassare i costi di sviluppo e di mantenimento, eseguite seguendo uno standard che ponga le basi dei servizi necessari all'integrazione, piuttosto che supersistemi complessi in grado di trattare una certa quantità di sorgenti differenti. Il paradigma impiegato nel progetto I3 per la

suddivisione dei servizi e delle risorse fra i vari moduli si basa su due partizionamenti fondamentali:

- Il partizionamento orizzontale che fornisce tre sezioni: database sorgenti, basi di conoscenza intermedie e applicazioni utente
- Il partizionamento verticale che distingue i dimini in cui raggruppare le sorgenti (il cui numero si deve aggirare fra cinque e nove).

I domini non sono strettamente interconnessi fra loro all'interno di un certo livello, ma di scambiano informazioni e dati. Per facilitare la flessibilità e migliorare le prestazioni del sistema il punto importantissimo della combinazione delle informazioni avviene a livello dell'utilizzatore. Nel seguito verra illustrata l'architettura di riferimento per i sistemi I3. Si consideri che una parte fondamentale dell'intero sistema è quella centrale che gestisce gli accessi alle varie sorgenti eterogenee di dati e fornisce le informazioni richieste alle applicazioni eseguendo combinazioni e, in seguito, sintesi sugli stessi.

2.3 L'architettura dei sistemi I_3

L'architettura del programma I_3 definita dall'ARPA deriva la sua forma dal tentativo di far fronte a problemi complessi come quelli citati nei paragrafi precedenti e che vengono qui, brevemente, riproposti in un elenco (per comprendere, infatti, questa particolare architettura è necessario comprendere i problemi a fronte dei quali è sorta).

- *Eterogeneità delle sorgenti*: differenze fra tipi di dati, schemi logici, interfacce per accedere ai dati...
- *L'evoluzione delle sorgenti di dati*: si possono aggiungere nuove fonti o modificare o eliminare quelle vecchie
- *Le dimensioni delle fonti*: bisogna far fronte all'aumento dei dati in una singola sorgente (ed all'aumento delle sorgenti) ed all'aumento dei tempi di risposta che ne deriva
- *La semantica nascosta*: bisogna dedurre regole, dai differenti schemi, per elaborare ed interpretare i dati da integrare

- *La necessità di sviluppare sistemi modulari riusabili*: questo punto è fondamentale per ridurre i tempi ed i costi di sviluppo delle varie applicazioni e far fronte ai mutamenti tecnologici che inevitabilmente si susseguono nel tempo

Una volta compresi i problemi fondamentali si può passare all'analisi della architettura di riferimento per sistemi I_3 proposta dall'ARPA. L'architettura del progetto I_3 si propone di evidenziare (separando in più moduli) i vari servizi che devono essere svolti ai fini dell'integrazione intelligenti delle informazioni. I servizi evidenziati a questo proposito sono cinque:

- Servizi ci coordinamento
- Servizi di amministrazione
- Servizi di integrazione e trasformazione semantica
- Servizi di wrapping
- Servizi ausiliari

Fra tutti quelli elencati in precedenza, i servizi principali sono quelli di coordinamento il cui scopo è, appunto, quello di coordinare le operazioni attuate dai vari servizi sia in fase di progettazione dei vari link di integrazione fra le sorgenti, che in fase di esecuzione (in tempo reale) su specifiche richieste degli utenti. Verranno esaminati ora, in maniera più approfondita, i diversi servizi citati.

2.3.1 I servizi di coordinamento

I servizi di coordinamento svolgono lavori di supporto sia in fase di progettazione di nuove configurazioni che a tempo di esecuzione delle richieste di utenti. Questi servizi sono di alto livello e, oltre ad individuare quali sorgenti possono essere utili per soddisfare una certa richiesta, presentano all'utente finale l'intero sistema, diviso fra i suoi vari moduli, come un blocco unico ed omogeneo. Grazie ai servizi de coordinamento, quindi, le divisioni interne di un sistema I_3 sono trasparenti all'usufruitore. I principali moduli appartenenti a questa sezione sono:

- *Broker*: il suo compito è quello di reperire gli strumenti in grado di trattare le richieste dell'utente. Il broker si occupa di contattare un modulo alla volta.

- *Facilitator*: il compito del facilitator è molto simile a quello del broker, ma si occupa di gestire richieste più complesse contattando anche più di un modulo per volta. A partire da un'unica richiesta ne vengono generate diverse da inviare ai moduli che gestiscono sorgenti distinte. Tramite servizi di Query Decomposition e tecniche di inferenza (ottenute tramite l'utilizzo dell'intelligenza artificiale) si decompone una query originale in un insieme di sottoquery e si trovano dinamicamente le sorgenti da interrogare. Per tali propositi il facilitator collabora strettamente con i servizi Ausiliari, con quelli di Amministrazione e quelli di Integrazione e trasformazione semantica.

- *Matchmaker*: serve per definire staticamente configurazioni dedicate ad alcuni servizi. I moduli e gli strumenti in grado di soddisfare determinate richieste sono determinati una volta per tutte e non vengono calcolati dinamicamente (a differenza di quanto fanno i moduli precedentemente descritti).

2.3.2 I servizi di amministrazione

I servizi di amministrazione sono in stretto contatto con quelli di coordinamento (in pratica vengono chiamati dai servizi di coordinamento). I loro scopo è quello di fornire informazioni sulle sorgenti permettendo di reperire quelle utili per soddisfare una certa richiesta. Altro scopo a cui sono dediti è quello, insieme ai servizi di coordinamento, della creazione di *template* (un template è uno schema di comportamento: costituisce la codifica di una configurazione specificando le sorgenti usate, gli strumenti e lo specifico ordine di interazione). I moduli principali connessi a questo tipo di servizi sono:

- *Resource Discovery*: moduli in grado di riconoscere e ritrovare gli strumenti in grado di rispondere positivamente ad una data domanda a run time. Il loro compito è quello di acquisire informazioni sugli strumenti che si possono utilizzare e sul loro stato attuale. Inoltre acquisiscono e mantengono nozioni sui domini informativi.

- *Browser*: modulo per mostrare agli utenti gli schemi delle sorgenti iniziali da sfruttare per l'integrazione e lo schema integrato a partire da essi. Indispensabile a questi propositi è l'interazione con i servizi di Wrapping atti ad estrarre informazioni dalle fonti.

- *Iterative Query Formulation*: si tratta di un modulo di ausilio nella espressione di una query che ha come oggetto lo schema integrato (non uno schema ritraente una fonte originaria). In particolare è di aiuto se si ha già espresso una query che non ha prodotto risultati interessanti (suggerisce quali possono essere le condizioni da rilasciare o come la query può essere resa più specifica).

- *Primitive di costruzione delle configurazioni*: servono a scegliere quali servizi e quali strumenti possono essere utilizzati per la costruzione di una configurazione e come collegarli fra loro

2.3.3 I servizi di integrazione e trasformazione semantica

Lo scopo di questi servizi è la manipolazione semantica necessaria al processo di integrazione. I servizi di integrazione e manipolazione sempantica sono strettamente connessi a quelli di wrapping: infatti le informazioni estratte dai wrappers e le viste sulle singole sorgenti fornite da essi sono proprio l'input per i servizi di integrazione. L'output che viene fornito, invece, è una vista integrata usando i significati semantici estratti delle fonti. Alcuni servizi di integrazione e trasformazione semantica sono:

- *Servizi di Integrazione degli Schemi*: questi servizi servono per integrare gli schemi provenienti da più fonti in un unico schema globale rappresentante l'integrazione delle precedenti viste. Il mappaggio fra lo schema globale e quelli locali (appartenenti alle fonti originali) deve essere mantenuto. Per ottenere questo risultato ci deve essere una trasformazione ed una integrazione dei vari vocabolari e delle varie ontologie usate dalle fonti; in pratica si deve giungere ad una ontologia unica che combini gli aspetti comuni delle singole ontologie. Questo tipo di servizi sono uno dei nodi fondamentali fra quelli offerti da un sistema di mediazione.

- *Servizi di Integrazione delle Informazioni*: sono servizi che provvedono alla traduzione dei termini da una ontologia di partenza ad una di arrivo. Forniscono inoltre servizi per uniformare eventuali discrepanze nelle unità di misura o nelle date.

- *Servizi di supporto al processo di integrazione*: sono i servizi che servono per la scomposizione di una query di partenza su di uno schema integrato in diverse query da eseguire su quelli locali; inoltre sono essenziali per l'aggregazione dei

risultati ottenuti dalle query singole. Fra questi servizi sono annoverati anche quelli tipicamente di aiuto alle prestazione del software, come l'indexing, il caching e l'ordinamento di certe informazioni.

2.3.4 I servizi di wrapping

Il primo passo compiuto nel processo di integrazione di informazioni provenienti da fonti eterogenee è, per i sistemi descritti nell'ottica I_3, quello fatto dai wrappers. Un wrapper è uno strumento che serve ad estrarre le informazioni volute da una sorgente ed a presentarle uniformate ad uno standard scelto. I wrappers dovrebbere essere il più riusabili possibile (in questo modo sarebbe possibile per più di un modulo mediatore sfruttare gli stessi wrappers), quindi dovrebbero uniformarsi agli standard contemporanei di comunicazione più diffusi (come ad esempio CORBA per lo scambio remoto di oggetti). I wrappers rappresentano l'interfaccia fra una sorgente di dati ed il resto del programma, senza questi moduli non è possibile eseguire il processo di integrazione.

2.3.5 I servizi di ausiliari

I servizi ausiliari aumentano le funzionalità dei servizi descritti in precedenza. In pratica in questa branca possiamo inserire tutti i servizi di aiuto all'utente ed al processo di integrazione ma che non sono presenti nelle classi precedenti (come ad esempio i vari servizi di ottimizzazione e di monitoraggio del processo aggiuntivi).

2.4 Il mediatore

Un mediatore è il modulo intermedio della architettura citata in precedenza (I_3) che collega l'utente alle sorgenti di informazione. Il compito di questo modulo è, come descritto da Wiederhold[19], reperire ed integrare informazioni da una molteplicità di sorgenti eterogenee. In letteratura possiamo ritrovare due tipologie differenti di mediatori: quelli che seguono un approccio definito *strutturale* e quelli che optano per un approccio detto semantico. L'approccio strutturale, seguito ad esempio nel progetto TSIMMIS[8, 20], presenta i seguenti punti caratteristici:

- Utilizzo di un modello self-describing per trattare tutti gli oggetti presenti nel sistema

- Utilizzo di un linguaggio self-describing che facilita l'integrazione ed è in grado di trattare anche dati semistrutturati
- Assenza di metadati (mancano gli schemi concettuali delle diverse sorgenti)
- Inserimento delle informazioni semantiche in modo esplicito attraverso l'introduzione di regole dichiarative

Altri progetti, tra cui MOMIS, utilizzato in questa tesi, preferiscono attuare un approccio semantico all'integrazione delle informazioni. I principali punti che caratterizzano tale metodo sono:

- Sono presenti metadati (sotto forma di schemi concettuali e viste) per descrivere ogni sorgente a disposizione
- Nello schema concettuale sono presenti anche informazioni semantiche che possono essere sfruttate nella fase di integrazione delle informazioni (ed anche in quella di ottimizzazione delle interrogazioni).
- Deve essere presente un modello di dati comune
- Deve essere realizzato uno schema globale rappresentante l'unificazione degli schemi concettuali

L'approccio semantico può comportare notevoli vantaggi: l'utilizzo di metadati, ed in particolare dello schema globale, permette di sfruttare le informazioni semantiche contenute in essi al fine di migliorare le ottimizzare le richiesta svolte dall'utente. Molto interessante risulta, inoltre, l'utilizzo di modelli ad oggetti unitamente agli schemi concettuali: tramite l'uso delle primitive di generalizzazione e di aggregazione si può ottenere una riorganizzazione delle conoscenze estensionali.

2.4.1 Principali problematiche da affrontare

Le problematiche dovute alle differenze dei sistemi fisici da cui si estraggono le informazioni devono essere risolte dai Wrappers. Ogni Wrapper, infatti, analizza una singola sorgente offrendo al sistema mediatore una vista, o schema concettuale, uniformabile con quelli estratti dagli altri traduttori. Uno dei compiti del modulo mediatore è, a questo punto, l'identificazione dei concetti comuni presenti nei metadati. Tale compito non è assolutamente banale, come non è semplice realizzare una integrazione

coerente e non ridondante degli schemi locali. Le problematiche identificate a livello di mediatore (quindi a livello di integrazione delle informazione) sono classificate essenzialmente in due tipologie:

- Problemi *ontologici*
- Problemi *semantici*

Nel seguito saranno brevemente trattate queste due tipologie.

2.4.2 Problemi ontologici

Per prima cosa è necessario fornire una definizione di ontologia. Nell'ambito della ricerca di cui fa parte questa tesi il concetto di ontologia può essere espresso come: -l'insieme dei termini e delle relazioni usate in un dominio, che denotano concetti ed oggetti- . In pratica una ontologia può essere vista come un insieme di termini (vocaboli) in grado di definire in modo univoco un determinato concetto. In questo procedimento non esiste ambiguità (ed il concetto risulta effettivamente definito in maniera univoca) se tutti i termini sono padroneggiati dall'intera comunità di utenti. Una classificazione di ontologie è stata effettuata da Guarino[23, 22] in questi termini:

- *Top-level ontology*: questo tipo di ontologia descrive i concetti più generali, quali, ad esempio, lo spazio, il tempo, l'azione... tali nozioni sono ritenute indipendenti da un particolare ambito o dominio di conoscenza. Si considera, quindi, che anche comunità separate di utenti condividano le top-level ontology.
- *Domain e task ontology*: si tratta di ontologie riguardanti concetti più specifici di quelli trattati nelle top-level ontology. Vengono descritti i vocabolari riguardanti particolari domini, come ad esempio quello medico o quello chimico, o particolari obiettivi, come la diagnostica.
- *Application ontology*: i concetti descritti da tali ontologie dipendono sia da un particolare dominio che da un particolare obiettivo. Si tratta dei termini e dei concetti più specifici di tutti (ad esempio la cura di una particolare malattia in ambito medico).

All'interno di questo progetto si considera che tutte la fonti informative (sorgenti di dati) condividano almeno le top-level ontology, quindi i concetti fondamentali. Si pensa di muoversi, invece, all'interno delle domain ontology, il problema, quindi, è quello di trovare i concetti comuni all'interno delle varie fonti.

2.4.3 Problemi semantici

Il problema semantico può essere ben compreso se si comprende che persone differenti possono dare descrizioni anche molto diverse tra loro della stessa porzione di mondo. Anche se si possiede un insieme di conoscenze comuni (come, ad esempio, quelle descritte da alcune domain ontology) non è possibile affermare che tali concetti saranno rappresentati, nelle diverse sorgenti, tramite gli stessi vocaboli o tramite le medesime strutture dati. Si pensi, ad esempio, a due persone distinte che hanno il compito di creare ciascuno un database, quindi anche uno schema relazionale, rappresentante gli articoli presenti in un magazzino e le promozione ad essi associate. Le possibilità per produrre un database di questo tipo sono innumerevoli e gli schemi relazionali prodotti dalle due persone (quindi anche i database finali) presenteranno indubbiamente qualche differenza nella rappresentazione dei medesimi concetti. Il problema citato in precedenza, comunque, non è l'unico in questo ambito, infatti, anche le differenze fra i DBMS che è possibile usare possono portare all'uso di modelli diversi per la rappresentazione degli stessi concetti (si pensi, ad esempio, alla realizzazione dello stesso concetto tramite un database relazionale ed un modello ad oggetti). Obiettivo del sistema mediatore è, oltre che trovare i concetti comuni all'interno delle varie fonti, anche quello di risolvere le differenze semantiche fra le diverse rappresentazioni. Le contraddizioni semantiche possono essere classificate in tre gruppi principali:

- *Eterogeneità tra le classi di oggetti*: classi rappresentanti lo stesso concetto situate in sorgenti differenti possono presentare gli stessi attributi con nomi differenti. E' anche possibile che gli attributi presentino domini di valori differenti oppure abbiano regole differenti su tali domini di valori.

- *Eterogeneità tra le strutture delle classi*: si tratta di differenze nei criteri di specializzazione e di aggregazione. Comprendono, inoltre, anche le *discrepanze semantiche*: valori di attributi sono implicitamente presenti nei metadati du un altro schema (ad esempio se l'attributo SESSO presente in uno schema è implicitamente presente in un altro tramite la divisione di PERSONE in MASCHI e FEMMINE).

- *Eterogeneità nelle istanze delle classi*: sotto questo punto si catalogano le differenze fra le unità di misura dei domini di un attributo e l'assenza o la presenza di valori nulli.

Comunque bisogna tener conto del fatto che è possibile arricchire la conoscenza fornita dal nostro sistema analizzando a fondo tali differenze semantiche e comprendendo il perch della loro presenza. Si ottiene in questo modo un *arricchimento semantico* che rende in maniera esplicita (ed interrogabile) tutte quelle informazioni che erano implicite nei metadati degli schemi.

2.5 MOMIS

Il progetto MOMIS (Mediator envirOnment for Multiple information Sources)[5, 4, 6] è basato sulla proposta del progetti I_3 per l'integrazione intelligente delli informazioni. MOMIS è, dunque, un progetto rivolto all'integrazione di sorgenti di dati di tipo strutturato e semistrutturato, con l'obiettivo di fornire un accesso il più possibile integrato a sorgenti informative eterogenee. Il generico utente deve essere in grado, ponendo una singola query, di ottenere informazioni significative provenienti da diverse fonti. L'approccio architetturale scelto per questo progetto è, essendo orientato sull'idea dell'ARPA-I_3, formato da tre livelli principali:

- *Utente*: l'utente deve essere in grado, tramite l'aiuto di un'interfaccia grafica, di porre query sul sistema globale ricevendo un'unica risposta. Questa risposta deve contenere informazioni integrate appartenenti a tutte le sorgenti locali ritenute interessanti. L'intero procedimento deve essere, possibilmente, trasparente agli occhi dell'utente ed il più automatico possibile.

- *Mediatore*: il mediatore, come già descritto in precedenza, è il modulo che si occupa, data una query da parte di un utente, di combinare ed integrare le informazioni estratte dai Wrappers (e poste in un modello comune) per fornire la risposta desiderata.

- *Wrapper*: ad ogni wrapper è collegata una singola fonte. Questo modulo si occupa di fornire i metadati richiesti per la specifica sorgente, di tradurre le query ottenute dal mediatore in modo opportuno per interrogarla e di fornire le informazioni reperite.

All'interno del sistema MOMIS si è scelto di adottare un modello comune di rappresentazione (il modello ODMi3, Object Data Model) ed un linguaggio object-oriented di descrizione comune per i dati ottenuti dalle varie sorgenti (il linguaggio ODL$_{I^3}$, Object Data Language). Lo strumento (*o tool*) grafico in grado di supportare

il progettista durante l'integrazione semi-automatica di schemi apparteneti a sorgenti eterogenee (di tipo relazionale, ad oggetti o semistrutturato) si chiama *SIDesigner*[2, 42] (Source Integrator Designer). SIDesigner esegue l'integrazione basandosi su un approccio semantico (quindi tramite l'apporto di metadati) che sfrutta tecniche intelligenti basate sulla logica descrittiva OLCD[3] (Object Language with Complements allowing Descriptive cycles). Il punto di partenza dell'intero processo di integrazione sono gli schemi, ottenuti dai wrappers e convertiti in formato ODL_{I^3}, rappresentanti le sorgenti locali. A partire da queste informazioni SIDesigner, tramite il supporto attivo di un progettista, crea una vista integrata dell'intero sistema (comprendente quindi più sorgenti di dati) sempre in formato ODL_{I^3}. Tale vista viene definita uno *schema globale*. Lo schema integrato viene ottenuto per fasi successive tramite la creazione di un dizionario comune (*common thesaurus*) indicante le possibili relazioni inter ed intra-schema. Le relazioni che formano il common thesaurus vengono ricavate tramite tecniche di inferenza basate sulla logica OLCD ed arricchite per mezzo dei sistemi ARTEMIS[7] e WordNet[33]. Sarà ora descritta più a fondo l'architettura del sistema MOMIS per l'integrazione dei dati provenienti da sorgenti eterogenee.

2.5.1 Architettura del sistema MOMIS

Il sistema MOMIS permette di integrare informazioni provenienti da fonti del tipo di database tradizionali (sia relazionali che object-oriented), provenienti oltremodo da file systems e da fonti semistrutturate come, ad esempio, l'HTML o l'XML. Come si è già citato in precedenza, MOMIS segue l'architettura di riferimento per sistemi I_3 proposta dall'ARPA, ente di difesa americano.

Come rappresentato in Figura(2.1) si possono distinguere cinque componenti principali, eccone un elenco:

1. *Wrapper*: i wrappers sono i moduli che forniscono l'interfaccia fra le sorgenti di dati locali ed il modulo mediatore. Ad ogni fonte diversa è associato un diverso wrapper. Questi moduli hanno due distinte funzioni, una svolta durante la fase di integrazione del sistema ed una durante la fase di risposta alle richieste (query) dell'utente. La prima di queste due funzione comporta la creazione dello schema ODL_{I^3} della fonte associata al wrapper, mentre la seconda prevede la traduzione della richiesta ottenuta dal mediatore (in formato OQL, Object

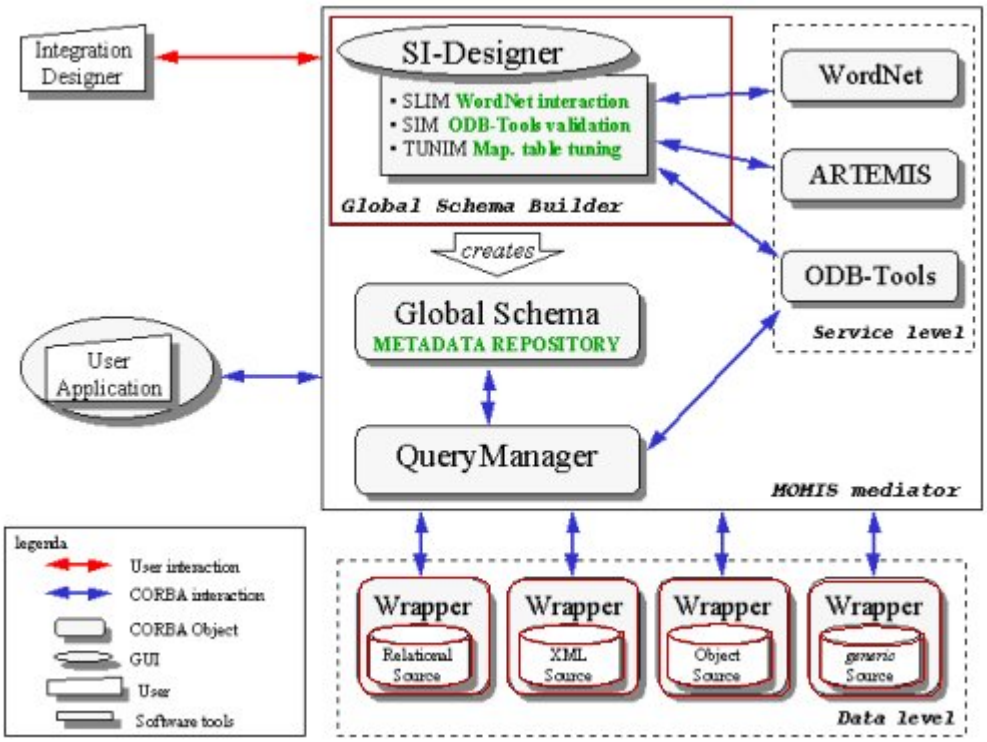

Figura 2.1: Schemi di due documenti XML in archivio

Query Language, compatibile con gli schemi ODL_{I^3}) in un formato che permetta la risoluzione della query sulla specifica sorgente (ad esempio, se un wrapper è collegato ad un database relazionale e riceve una query in OQL dal mediatore, dovrà tradurre ed eseguire la query in SQL). Ulteriore compito di un wrapper è quello di raccogliere i dati ricevuti in risposta ad una query (posta su una sorgente locale) e presentarli nel modello di dati comune al mediatore.

2. *Mediatore*: si tratta indubbiamente del modulo fondamentale dell'intero sistema di integrazione. Il mediatore è composto da due moduli distinti: il Global Schema Builder (GSB) ed il Query Manager (QM). Il Global Schema Builder è responsabile per l'integrazione degli schemi locali presentati dai wrappers. Il prodotto del GSB è uno schema globale integrato sempre in formato ODL_{I^3}. Il secondo modulo componente il mediatore è il Query Manager che ha il compito di gestire le interrogazioni poste dall'utente: ogni singola query sullo schema globale deve essere espressa (o tradotta) in OQL e divisa nelle query da inviare ai diversi wrappers. Servendosi delle tecniche di Logica Descrittiva, il QM

genera automaticamente le query da inviare alle singole sorgenti locali.

3. *ODB-Tools Engine*: sviluppato al dipartimento di Scienze dell'Ingegneria dell'Università di Modena[15]. ODB-Tools è un software che permette la validazione di schemi e l'ottimizzazione di interrogazioni. ODB-Tools è dotato di un nucleo che interpreta e supporta la logica descrittiva OLCD ed è in grado, oltre che di controllare la coerenza delle descrizioni OLCD, anche di effettuare il calcolo della sussunzione e dell'espansione semantica di un tipo[14]. Un utente può inserire in ingresso schemi in formato ODL_{I^3} o query in formato OQL ed ottiene come risultato la validazione dello schema (verifica dell'assenza di classi incoerenti e riclassificazione di gerarchie di specializzazione implicite nello schema) o l'ottimizzazione semantica della query .

4. *ARTEMIS-Tool Environment*: il modulo ARTEMIS[7] compie analisi sugli schemi ODL_{I^3} per verificare il grado di affinità delle classi contenute al loro interno.

5. *WordNet*: si tratta di un database lessicale, in lingua inglese, che è anche in grado di individuare relazioni fra termini. WordNet viene sfruttato nel processo di creazione del thesaurus di relazioni e sarà trattato in maniera più approfondita nel prossimo capitolo.

Fra i cinque moduli presentati in precedenza il 'corpo' centrale è costituito dal mediatore che prende le informazioni iniziali dalla sezione wrapper e sfrutta la collaborazione di ODB-Tool, ARTEMIS e WordNet durante il processo di integrazione delle fonti spiegato nel prossimo paragrafo.

2.5.2 Il processo di integrazione delle fonti

La prima operazione da compiere come atto preliminare la processo di integrazione delle fonti di informazione eterogenee, è l'acqusizione delle sorgenti. Durante questa fase l'utente può, tramite l'interazione con SIDesigner, ed in particolare col modulo SAM (Source Acquisition Module), indicare quali sorgenti desidera integrare. Per ogni sorgente locale così selezionata, un wrapper ne fornisce la traduzione dello schema nel modello ODL_{I^3}. A questo punto inizia il vero è proprio processo di integrazione, suddiviso nei seguenti punti fondamentali:

1. *Creazione di un Common Thesaurus*: Il common thesaurus è un insieme di relazioni terminologiche intensionali ed estensionali. Le *relazioni intensionali* sono relazioni terminologiche che possono esprimere una conoscenza sia

intra che inter-schema (all'interno dello stesso schema o fra schemi differenti) tra attributi o classi ODL_{I^3}. Vi sono quattro differenti tipologie di relazioni intensionali:

- SYN (Synonym-of): definita fra due termini considerati sinonimo in ogni singola fonte
- BT (Broader Term): definita fra due termini in cui il primo è ritenuto possedere un significato più generico del secondo
- NT (Narrow Term): relazione che rappresenta l'opposto rispetto ad una relazione di tipo BT; il primo termine risulta avere un significato più specifico rispetto al primo
- RT (Related Term): relazione fra due termini usati di solito insieme nello stesso contesto

Le *relazioni estensionali*, invece, pur avendo le quattro medesime tipologie di quelle intensionali, sono riferite fra due classi *C1* e *C2* e ne rappresentano un rafforzamento. Consideriamo, ad esempio, una relazione di tipo SYN fra due classi, essa afferma che tutte le istanze della prima classe sono anche istanze della seconda, e viceversa. Una relazione estensionale di tipo BT, invece, afferma che tutte le istanze della seconda classe sono anche istanze della prima, ma non è vero il contrario. Le relazioni intensionali di tipo intra-schema vengono ricavate dal modulo SIM di SIDesigner (per mezzo dell'ausilio di ODB-Tools) ed ampliate tramite le relazioni inter-schema trovate dal modulo SLIM (coadiuvato dal programma esterno wordNet, database lessicale in grado anche di fornire relazioni fra le parole).

2. *Analisi dell'affinità fra le classi*: in questa seconda fase le relazioni del common thesaurus sono utilizzate per calcolare un grado di *affinità* fra le classi. Anche in questo caso le classi possono appartenere allo stesso schema o a schemi differenti. Il concetto di affinità è stato introdotto in modo da formalizzare il tipo di relazione che esiste fra due classi, dal punto di vista dell'integrazione. Il grado di affinità tra classi viene stabilito per mezzo di coefficienti basati sulla similarità fra i nomi delle classi, sulle similitudini fra le strutture delle due classi (come ad esempio la presenza di attributi uguali) ed, ovviamente, sulle relazioni presenti nel common thesaurus.

3. *Clustering delle classi*: in base al grado di reciproca affinità ritrovato al punto precedente, le classi vengono raggruppate in clusters usando tecniche di clusterizzazione gerarchica (questa fase del processo è coadiuvata dal software ARTEMIS). Lo scopo di questo procedimento è individuare le classi che possono essere integrate perch ritenute concernenti il medesimo concetto.

4. *Generazione dello schema globale*: Partendo dai risultati ottenuti durante la fase di clustering, viene ora generata, per ogni cluster, una *classe globale*. Ogni classe globale rappresenta una vista mediata di tutte le classi comprendenti il cluster originario e possiede un insieme di attributi globali ottenuti tramite la fusione degli attributi simili appartenenti alle classi locali. Per ogni classe globale, inoltre, è mantenuta una *mapping-table* il cui scopo è quello di permettere il mapping fra gli attributi globali della classe e gli attributi appartenenti alle classi locali. Lo schema globale, rappresentante il risultato finale del processo di mediazione, è formato dall'unione di tutte le classi globali così ottenute.

2.6 MOMIS ed il progetto ECD

All'interno di questo capitolo abbiamo visto la struttura del sistema mediatore MOMIS, le sue funzionalità ed il procedimento che viene effettuato per produrre lo schema globale (risultato terminale del processo di integrazione delle fonti) a partire da quelli locali. Concentriamoci ora sull'utilizzo che tale sistema può avere nell'ambito del progetto ECD (*enhanced contents delivery*) di cui fa parte la presente tesi.

Ricordiamo che lo scopo di tale progetto è quello di sviluppare tecnologie e strumenti in grado di offrire, agli utenti finali, contenuti arricchiti. Ciò può essere fatto attraverso l'identificazione di materiale digitale presente su fonti diverse ed eterogenee, attraverso la sua trasformazione, organizzazione ed aggiunta di metadati ed informazioni utili a qualificarlo e a far giungere agli utenti materiale più rilevante ai loro fini.

Risulta, quindi, molto importante l'impiego, unitamente a tecnologie per la gestione di archivi di documenti digitali(eg.biblioteche digitali XML), di un modulo mediatore come MOMIS, in particolare se utilizzato per raccogliere schemi di sorgenti eterogenee e permetterne l'*annotazione*, l'arricchimento semantico. Fondamentale risulta, quindi, l'interazione con il database lessicale WordNet, descritto nel capitolo

6, che fornisce le conoscenze necessarie alla fase di annotazione per associare ad ogni termine (e quindi ad ogni elemento degli schemi) un concetto. WordNet è in grado anche di individuare relazioni fra i concetti selezionati, fatto importantissimo per il problema che ci accingiamo a risolvere: identificare quali schemi XML (quindi quali documenti XML apparteneti ad un archivio) sono in grado di fornire informazioni utili ad una richiesta dell'utente, e riformulare tale richiesta per questi schemi.

Cerchiamo ora di capire quali parti del mediatore MOMIS ci interessano, e saranno quindi utilizzate in seguito. Lo scopo di MOMIS è quello di creare una vista globale integrata (GVV) di un insieme di schemi rappresentanti sorgenti di dati eterogenee. Un metodo possibile per risolvere il nostro problema è quello di far porre, all'utente, le richieste, o *query*, direttamente sullo schema globale, per poi riformularle (per mezzo di apposite tabelle di mapping inglobate nelle classi globali ODL_{I^3}) direttamente in query per gli schemi locali. Questo metodo, benchè fornisca i risultati voluti, costringerebbe l'utente a studiare preventivamente la composizione dello schema globale, che può non trovare, nella terminologia, riscontro diretto con gli schemi locali. Non è questo il nostro intento. L'utente, infatti, deve essere libero di esprimere la richiesta con i termini che ritiene possano fornire le informazioni desiderate, un po' come avviene nei motori di ricerca utilizzabili sul Web (eg. Google, Altavista, ecc...), senza bisogno di una lettura preventiva dello schema globale. Deve essere compito del sistema, successivamente, reperire similarità semantiche fra i termini ed i path espressi dalla query con quelli situati negli schemi, in modo da poterla riscrivere sugli schemi identificati come interessanti. A tale scopo, quindi, bastano i seguenti moduli appartenenti al sistema MOMIS:

- *Il modulo SAM* di acquisizione degli schemi dalle sorgenti. Tale modulo impiega i wrapper (uno differente per ogni tipo di fonte da utilizzare) per tradurre gli schemi originali in ODL_{I^3}. Questa rappresenta una fase indispensabile per tutte le altre.

- *Il modulo SLIM*, in grado di interagire col database lessicale WordNet e permettere la annotazione (assegnazione dei significati) agli schemi tradotti in ODL_{I^3}. Questo modulo software è, inoltre, in grado, a partire dagli schemi annotati, di produrre un thesaurus di relazioni semantiche fra gli elementi contenuti in essi. Sarà proprio questo thesaurus (insieme agli schemi arricchiti semanticamente) il punto di partenza per una corretta riscrittura delle richieste degli utenti finali.

Oltre ai due moduli citati nell'elenco precedente, sarebbe possibile impiegare, per il nostro scopo, anche il modulo SIM. Tale modulo si occupa sia di reperire relazioni intra-schema dovute alla struttura degli stessi (ereditarietà, foreign-key, ecc...) che, basandosi sul software ODB-Tools, di inferire nuove relazioni a partire da quelle trovate da SLIM o aggiunte dal progettista. Per quanto riguarda le relazioni intra-schema, esse non interessano molto al nostro fine: ciò che importa sono invece le relazioni inter-schema che legano fra loro termini appartenenti a schemi differenti. Potrebbero essere utili, invece, le relazioni inferite tramite l'utilizzo delle logiche descrittive. Nello sviluppo del software della presente tesi sono stati impiegati solamente i moduli SAM e SLIM. L'impiego del modulo SIM, comunque, non varierebbe il funzionamento del procedimento per la riscrittura di una richiesta basandosi sugli schemi annotati, si impiegherebbe solamente il thesaurus prodotto da SIM invece che quello prodotto dal modulo SLIM.

Vediamo ora, brevemente, le modifiche che è necessario apportare ai moduli di MOMIS, impiegati nell'ambito della presente tesi, al fine di risolvere il problema della riscrittura delle query poste da un utente su un archivio di documenti XML.

1. Assumendo che sia impiegato il linguaggio Xml-Schema (descritto approfonditamente all'interno del capitolo 3) per descrivere la struttura dei documenti che possono comparire all'interno di un archivio XML, è necessario procettare ed implementare un wrapper che traduca da Xml-Schema ad ODL_{I^3}. Tale wrapper non è ancora stato implementato e, all'interno del capitolo 5, si esprimono le specifiche richieste per una corretta traduzione da Xml-Schema ad ODL_{I^3}.

2. Il modulo SLIM, per evitare possibili perdite di conoscenza, deve poter usufruire di una versione estensibile di WordNet, in cui possano essere aggiunti nuovi termini, significati e relazioni fra di essi. Per mezzo della tesi di laurea di Veronica Guidetti[24] WordNet è stato reso facilmente estensibile, riportando la conoscenza da esso espressa in un database relazionale chiamato MOMISWN. Nell'ambito della presente tesi è stata prodotta una nuova versione di SLIM in grado di reperire le informazioni necessarie da questo database.

Capitolo 3

Il linguaggio Xml-Schema

3.1 Introduzione

Lo scopo di un Xml-Schema[11, 13, 12] è quello di definire una classe di documenti Xml, permettendone la validazione. I documenti Xml che corrispondono alla descrizione fornita da un Xml-Schema vengono chiamati 'documenti istanza' di quel particolare schema.

L'utilizzo di questo strumento permette di definire in modo molto preciso la struttura che deve avere un documento Xml, supportando, a differenza delle DTD (Document Type Definition), l'utilizzo di un grande numero di tipi di dato primitivi, quali Stringhe, Interi, Date, Reali L'invenzione di tale linguaggio è stata resa necessaria anche dal fatto che le DTD sono fornite con una sintassi a se stante, differente da quella dell'Xml vero e proprio, mentre gli schemi si basano su descrizioni fornire in Xml, permettendo di lavorare su di essi con strumenti già creati per tale standard e risultando molto più comodi per i programmatori.

Una ulteriore novità rispetto alle DTD è data dalla possibilità di definire nuovi tipi di dato, sia complessi (contenenti dei sottoelementi) che semplici (non contenenti alcun sottoelemento), partendo da elementi gia definiti in questo o in altri schemi. L'ultima caratteristica citata aggiunge ad Xml-Schema un approccio che si può definire Object-Oriented assomigliando moltissimo, appunto, al concetto di ereditarietà presente in numerosi linguaggi di programmazione. Questo fatto, insieme alla puntuale tipizzazione dei dati, rende Xml-Schema un linguaggio più facilmente utilizzabile, rispetto alle DTD, congiuntamente a linguaggi di programmazione veri e propri.

3.2 Documenti validi e documenti ben formati

Un documento Xml è detto ben formato (well formed) se soddisfa tutte le specifiche elencate nel linguaggio Xml 1.0. Esempi di tali specifiche sono: deve esistere un elemento radice che non è contenuto in nessun altro elemento; l'innestamento dei tag deve essere corretto (non è valida, ad esempio, la seguente struttura:$<$ $Nome1 >< Nome2 >< /Nome1 >< /Nome2 >$); ogni entità referenziata direttamente o indirettamente all'interno del documento deve essere anch'essa well formed, ecc...

Un documento Xml viene detto valido (valid) se è ben formato ed inoltre rispetta tutti i vincoli strutturati definiti nello schema (DTD o Xml-Schema) cui si riferisce.

3.3 Un primo esempio di Xml-Schema

Per capire la struttura di un Xml-Schema credo sia logico basarsi inizialmente sull' analisi di un esempio:

Esempio 1:

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

<xsd:element name="Ordine" type="OrdineType"/>

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

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
    < xsd: attribute   name ="Data" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="IndirizzoType">
    < xsd:element  name ="Stato"  type="xsd: string "/>
    < xsd:element  name ="Città"  type="xsd: string "/>
```

```
    < xsd:element  name =”Via” type=”xsd: string ”/>
    < xsd:element  name=”Numero” type=”xsd:int”/>
</xsd:complexType>

<xsd:complexType name=”OggettoType”>
    < xsd:element  name =”Nome” type=”xsd:string”/>
    < xsd:element  name =”Quantità” type=”QuantitàType”/>
    < xsd:element  name =”PrezzoEuro” type=”xsd:decimal”/>
    < xsd:element  ref =”Commento” minOccurs=”0”/>
</xsd:complexType>

<xsd:simpleType name=”QuantitàType”>
    <xsd: restriction   base=”xsd: positiveInteger ”>
        <xsd:maxExclusive value=”100”/>
    </xsd: restriction >
</xsd:simpleType>

</xsd:schema>
```

L’Xml-Schema precedente descrive un semplice ordine d’acquisto. L’elemento principale è ’Ordine’ del tipo ’OrdineType ed è formato da un ’Indirizzo’, uno o più elementi ’Oggetto’, da un attributo ’Data’ e da zero o più elementi ’Commento’. Per meglio capire l’utilizzo di questo schema viene di seguito presentata un possibile documento Xml istanza (quindi rispondente ai requisiti dell’Xml-Schema dell’esempio 1) che rappresenta il nostro ordine d’acquisto:

```
<?xml version=”1.0”?>

<Ordine Data=”2002−11−01”>
    <Indirizzo>
        <Stato>Italia</Stato>
        <Città>Modena</Città>
        <Via>NomeVia</Via>
        <Numero>56</Numero>
    </Indirizzo>
    <Oggetto>
```

```
        <Nome>Monitor</Nome>
        <Quantità>10</Quantità>
        <PrezzoEuro>255.25</PrezzoEuro>
        <Commento>Monitor LCD 15 pollici</Commento>
    </Oggetto>
    <Oggetto>
        <Nome>Tastiera</Nome>
        <Quantità>10</Quantità>
        <PrezzoEuro15.25</PrezzoEuro>
    </Oggetto>
</Ordine>
```

Come si può notare anche nel documento Xml l'elemento 'Ordine' ha un attributo 'Data' e contiene un elemento 'Indirizzo', più elementi 'Oggetto' ed, in questo caso, nessun elemento 'Commento', tutte caratteristiche richieste nell'Xml-Schema dell'esempio 1. E' quindi chiaro come uno schema possa definire strutturalmente e, quindi, validare una pagina Xml. Vediamo ora, in dettaglio, la struttura e la sintassi degli Xml-Schema

3.4 Sintassi di Xml-Schema

Considerando l'esempio 1 presentato in precedenza si può notare che l'intero schema è contenuto all'interno del tag *Schema* (NB: il termine tag ha qui la stessa accezione che possiede per i documenti in Xml, non bisogna dimenticare infatti che un testo Xml-Schema è scritto proprio in Xml); iniziamo quindi la descrizione della sintassi del linguaggio proprio da questo tag.

3.4.1 Il tag Schema

Ogni Xml-Schema deve essere completamente contenuto all'interno del tag *Schema*. Un generico documento si presenterà quindi con questa struttura:

```
<schema attributoSchema1=" ..."  attributoSchema2=" ..."  ... >
    Dichiarazione  degli  elementi
</schema>
```

Gli attributi che possono comparire in questo elemento servono, come ad esempio *xmlns* che compare nell'esempio 1, per definire i namespaces di riferimento per gli elementi dello schema. Il documento può essere visto come una collezione di definizioni di tipo e di dichiarazioni di elementi i cui nomi appartengono ad un particolare namespace, chiamato target namespace; in questo modo possiamo distinguere dichiarazioni di elementi appartenenti a differenti 'vocabolari'. Per una trattazione più approfondita si veda il successivo paragrafo sui namespaces.

3.4.2 Definizioni di tipo e Dichiarazioni di elementi

Per dichiarare un elemento la sintassi da seguire in Xml-Schema è rappresentata dalla stringa Xml:

```
<element name="nomeElemento" type="tipoElemento"/>
```

l'attributo *type* indica il tipo a cui l'elemento nomeElemento appartiene. Un tipo può essere:

- *Semplice*: non può contenere al suo interno elementi ne attributi
- *Complesso*: può contenere al suo interno sia elementi sia attributi

Consideriamo ancora l'esempio 1 sull'ordine d'acquisto e più precisamente la dichiarazione dell'elemento Ordine e la definizione del tipo OrdineType:

```
<xsd:element name="Ordine" type="OrdineType"/>

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0"/>
    < xsd: attribute   name ="Data" type="xsd:date"/>
</xsd:complexType>
```

Come si può notare il tipo OrdineType è un tipo complesso: sono infatti presenti al suo interno sia altri elementi sia un attributo. La definizione di un nuovo tipo complesso avviene sempre tramite il tag

```
<complexType name="NuovoTipoComplesso">
    ...
</complexType>
```

Naturalmente gli elementi presenti all'interno di un complexType possono essere sia semplici che complessi; si possono così creare gerarchie di elementi a diversi livelli di profondità.

Consideriamo ora le seguenti righe (tratte naturalmente sempre dallo stesso esempio):

```
<xsd:complexType name="OggettoType">
    < xsd:element name ="Nome" type="xsd:string"/>
    < xsd:element name ="Quantità" type="Quantit'aType"/>
    < xsd:element name ="PrezzoEuro" type="xsd:decimal"/>
    < xsd:element ref ="Commento" minOccurs="0"/>
</xsd:complexType>
```

Viene presentata un'altra definizione di tipo complesso (OggettoType) che contiene tre elementi semplici ed un solo elemento complesso (Quantità appartenente al tipo QuantitàType). Consideriamo l'elemento Nome, esso è stato definito come appartenente al tipo *string*. Il tipo string è un tipo semplice primitivo definito direttamente come parte del linguaggio Xml-Schema e non può contenere elementi o attributi. Il contenuto dell'elemento Nome sarà quindi solamente una stringa di caratteri.

Si noti inoltre che l'elemento semplice,di tipo string, Commento viene solamente riferito, non dichiarato all'interno del tipo OggettoType. Riferire un oggetto in questo modo permette, in alcuni casi, di risparmiare tempo fancendo una sola volta la dichiarazione dell'elemento in questione. Per potersi riferire con l'attributo *ref='...'* ad uno specifico oggetto esso deve essere dichiarato come elemento globale, deve cioè comparire direttamente come figlio del tag Schema.

Esaminiamo ora più attentamente la dichiarazione:

< xsd:element name =”Oggetto” type=”OggettoType” minOccurs=”1”
maxOccurs=”unbounded”/>

Questa riga compare all’interno della definizione del tipo complesso OrdineType e, essendo Ordine proprio del tipo OrdineType, possiamo considerare Oggetto come figlio dell’elemento Ordine. Gli attributi *nimOccurs=’n intero positivo’* e *maxOccurs=’n intero positivo’* specificano il numero massimo e minimo di occorrenze che l’elemento Oggetto può assumere (all’interno di un documento Xml istanza dell’Xml-Schema dell’esempio 1) dentro l’ente Ordine. *Unbounded* ha il significato di infinito. Nel caso una dichiarazione di elemento non presenti l’attributo minOccurs o l’attributo maxOccurs si assume, per entrambi, il valore di default 1.

3.4.3 Dichiarazione di attributi

La dichiarazione di un attributo può avvenire solamente all’interno della definizione di un tipo complesso (si ricordi che un tipo semplice non può contenere ne elementi ne attributi). La sintassi da seguire è:

< attribute name=”nomeAttributo” type=” tipoAttributo ”/>

Il tipoAttributo può riferirsi solamente ad un tipo semplice, mai ad uno complesso. Normalmente in un documento Xml istanza un attributo può comparire una volta o non comparire affatto e può avere un valore qualsiasi fra quelli permessi dal tipo specificato in *type*.

Oltre alla sintassi base l’elemento *attribute* può avere ulteriori attributi: *use*, *fixed* e *default* che servono per specificare meglio il funzionamento dell’attributo nello schema istanza. *Use* può assumere i seguenti valori:

- *Required*: il valore può essere uno qualsiasi fra quelli permessi dal tipo type, l’attributo deve essere presente nel ducumento istanza.
- *Optional*: l’attributo può essere presente nel documento istanza ma può anche essere omesso. Non ci sono vincoli sul suo valore.

- *Prohibited*: l'attributo non deve comparire.

Fixed permette di specificare un valore fisso per l'attributo.

Es:< attribute name="Stato" type=" string " fixed =" Italia "/>

Default sagnala che, se l'attributo non compare nel documento Xml istanza, il suo valore deve essere quello specificato nell'Xml-Schema dall'attributo default dell'elemento *attribute*.

3.4.4 I tipi semplici

I tipi semplici si differenziano in due sottocategorie:

- *Tipi semplici primitivi*
- *Tipi semplici derivati*

I tipi semplici di primitivi,come *string* o *decimal*, sono definiti direttamente all'interno del linguaggio proprio di Xml-Schema, mentre i tipi semplici derivati sono ottenuti partendo da quelli di base tramite costrutti quali l'enumerazione, l'unione, le liste o la restrizione di tipo. Nei prossimi paragrafi forniremo una descrizione dettagliata di tutti questi sotto-tipi.

3.4.5 I tipi semplici primitivi

I tipi semplici primitivi sono stati definiti direttamente all'interno di Xml-Schema che, essendo un linguaggio fortemente tipizzato, ne presenta numerosi. Viene riportata di seguito una tabella che li riassume tutti fornendo anche, per ciascuno, un esempio ed alcune note:

Tipo	**Esempio**	**Note**
string	Stringa di prova	Stringa di caratteri
normalizedString	Stringa di prova	Stringa non contenente Carriage Return, Line Feed o Tab

Tipo	Esempio	Note
token	Stringa di caratteri	Come normalizedString, non può avere due spazi consecutivi
byte	-1, 126	Intero compreso fra 127 e -128
unsignedByte	0, 126	Intero con il valore compreso fra 0 e 255
Base64Binary	GpM7	Valore codificato tramite Base64 Content-Transfer-Encoding defined in Section 6.8 of [RFC 2045].
hexBinary	0FB7	Numero binario in formato esadecimale
integer	-126789, -1, 0, 1, 126789	Intero positivo o negativo
positiveInteger	1, 126789	Come Intero solo positivo, zero escluso
negativeInteger	-126789, -1	Intero solo negativo, zero escluso
nonNegativeInteger	0, 1, 126789	Intero solo positivo, zero incluso
nonPositiveInteger	-126789, -1, 0	Intero solo negativo, zero incluso
int	-1, 126789675	Intero compreso fra 2147483647 e -2147483648
unsignedInt	0, 1267896754	Come unsignedLong, valore massimo posto a 4294967295
long	-1, 12678967543233	Intero compreso fra 9223372036854775807 e -9223372036854775808
unsignegLong	0, 12678967543233	Come nonNegativeInteger, valore massimo posto a 18446744073709551615

Tipo	Esempio	Note
short	-1, 12678	Intero compreso fra 32767 e -32768
unsignedShort	0, 12678	Come unsignedInt, valore massimo posto a 65535
decimal	-1.23, 0, 123.4, 1000.00	Rappresenta un numero decimale di precisione arbitraria
float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	Corrisponde all'IEEE single-precision 32-bit floating point, NaN è 'not a number'
double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	Corrisponde all'IEEE double-precision 64-bit floating point
boolean	true, false	Classico valore booleano
time	13:20:00.000, 13:20:00.000-05:00	Un istante del tempo che ricorre ogni giorno
dataTime	1999-05-31T13:20:00.000-05:00	Maggio 31 1999 alle ore 1.20pm
duration	P1Y2M3DT10H30M12.3S	1 anno, 2 masi, 3 giorni, 10 ore, 30 minuti e 12.3 secondi
date	1999-05-31	Semplice data
gMonth	–05–	Maggio
gYear	1999	Anno 1999
gYearMonth	1999-02	Il mese di Febbraio 1999
gDay	—31	Il giorno 31
Name	shipTo	Come XML 1.0 Name
QName	po:USAddress	XML Namespace QName
NCName	USAddress	'non-colonized' namesXML Namespace NCName, i.e. un QName senza prefisso e :
anyURI	http://www.example.com/	Valore URI
language	en-GB, en-US, fr	Valori validi per xml:lang definiti in XML 1.0

Oltre ai numerosi tipi di dato rappresentati nella tabella precedente, l'Xml-Schema, per una questione di completezza e di compatibilità con lo standard Xml 1.0, mantiene anche le seguenti tipologie:

Tipo	Esempio	Note
ID		XML 1.0 ID
IDREF		XML 1.0 IDREF
IDREFS		XML 1.0 IDREFS
ENTITY		XML 1.0 ENTITY
ENTITIES		XML 1.0 ENTITIES
NOTATION	US,Brsil	XML 1.0 NOTATION
NMTOKEN	US UK,Brsil Canada Mexique	XML 1.0 NMTOKEN
NMTOKENS		XML 1.0 NMTOKENS

3.4.6 I tipi semplici derivati

Possiamo raggruppare i tipi semplici derivati in tre diverse categorie:

- *Liste*
- *Unioni*
- *Tipi semplici derivati per restrizione*

Ogni volta che bisogna definire un nuovo tipo semplice (quindi un tipo semplice derivato) si ricorre all'uso del tag

<simpleType name="NomeDelNuovoTipoSemplice"> ... </simpleType>

Per poter meglio capire questi nuovi tipi di dato riproporremo l'esempio sull'ordine d'acquisto modificato che diventerà il nostro secondo esempio:

Esempio 2:

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

```
<xsd:element name="Ordine" type="OrdineType"/>

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

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
    < xsd: attribute   name ="Data" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="IndirizzoType">
    < xsd:element  name ="Stato"  type="StatoType"/>
    < xsd:element  name ="Città"  type="xsd: string "/>
    < xsd:element  name ="Via" type="xsd: string "/>
    < xsd:element  name="Numero" type="NumeroType"/>
</xsd:complexType>

<xsd:complexType name="OggettoType">
    < xsd:element  name ="Nome" type="xsd:string"/>
    < xsd:element  name ="Quantità" type="QuantitàType"/>
    < xsd:element  name ="PrezzoEuro" type="xsd:decimal"/>
    < xsd:element  name =" ColoriDisponibili "  type=" ListaColori "/>
    < xsd:element  ref  ="Commento" minOccurs="0"/>
</xsd:complexType>

<xsd:simpleType name="QuantitàType">
    <xsd: restriction   base="xsd: positiveInteger ">
        <xsd:maxExclusive value="100"/>
    </xsd: restriction >
</xsd:simpleType>

<xsd:simpleType name="StatoType">
    <xsd: restriction   base=" string ">
        <xsd:enumeration value=" Italia "/>
```

```
        <xsd:enumeration value="USA"/>
    </xsd: restriction >
</xsd:simpleType>

<xsd:simpleType name="ListaColori">
    <xsd: list  itemType="ColoreType"/>
</xsd:simpleType>

<xsd:simpleType name="ColoreType">
    <xsd: restriction  base=" string ">
        <xsd:enumeration value="Rosso"/>
        <xsd:enumeration value="Nero"/>
        <xsd:enumeration value="Verde"/>
    </xsd: restriction >
</xsd:simpleType>

<xsd:simpleType name="NumeroType">
    <xsd:union memberTypes="xsd:string xsd: positiveInteger "/>
</xsd:simpleType>

</xsd:schema>
```

Tramite la definizione di nuovi tipi semplici il nostro schema sull'ordine d'acquisto è stato reso più particolareggiato: si specifica, ad esempio, tramite un costrutto di enumerazione, che i soli valori possibili per l'elemento Stato figlio dell'elemento Indirizzo sono Italia e USA; viene aggiunta anche la possibilità di fornire una lista di colori alla descrizione dell'elemento Oggetto, colori che, a loro volta, possono appartenere solamente alla enumerazione Rosso, Verde e Nero.

3.4.7 Il tipo semplice derivato unione

Consideriamo le seguenti righe di testo tratte dall'esempio 2:

```
<xsd:simpleType name="NumeroType">
    <xsd:union memberTypes="xsd:string xsd: positiveInteger "
</xsd:simpleType>
```

normalmente il valore di un tipo semplice è definito dal suo attributo *type*; nel caso del tipo semplice unione, invece, il valore dell'elemento può essere uno a scelta fra quelli contenuti nell'attributo *memberType* dell'elemento *union*. In una stessa istanza Xml potremmo così ritrovare elementi di tipo NumeroType il cui valore è fornito da un numero positivo intero ed altri rappresentati da una stringa di caratteri.

ES:

```
...
<Indirizzo>
        <Stato>Italia</Stato>
        <Città>Modena</Città>
        <Via>NomeVia</Via>
        <Numero>56</Numero>
</Indirizzo>
...
<Indirizzo>
        <Stato>Italia</Stato>
        <Città>Roma</Città>
        <Via>NomeVia</Via>
        <Numero>Cinquantasei</Numero>
</Indirizzo>
```

L'attributo *memberType* può contenere solamente tipi semplici (primitivi, liste, derivati tramite restrizioni ed altre unioni) separati fra loro da un carattere di spazio.

3.4.8 Il tipo semplice derivato lista

Per fare in modo che il contenuto di un elemento di tipo semplice non sia solamente composto da un valore atomico, ma da una lista di valori di tipo atomico separati da spazi bianchi viene utilizzato l'elemento *list*; eccone un esempio:

```
<xsd:element name="Numeri" type="ListaDiNumeri"/>

<xsd:simpleType name="ListaDiNumeri">
    <xsd:list itemType="xsd:integer"/>
</xsd:simpleType>
```

In un documento Xml 1.0 istanza potrebbe comparire l'elemento

<Numeri>1 234 −3435 0 23 980 −4 </Numeri>

Osserviamo che l'attributo *itemType* dell'elemento *list* contiene il tipo atomico (che deve essere, proprio come nel caso dell'attributo *memberTypes* per l'unione, di tipo semplice) da cui si ottiene la lista.

Anche nel nostro esempio 2 compare la definizione di un tipo semplice lista:

```
<xsd:simpleType name="ListaColori">
    <xsd: list  itemType="ColoreType"/>
</xsd:simpleType>
```

ColoreType è un tipo semplice ottenuto da una restrizione enumerazione del tipo primitivo string. Per una descrizione del concetto di restrizione si rimanda al paragrafo successivo.

3.4.9 Il tipo semplice derivato per restrizione

In Xml-Schema è possibile definire nuovi tipi di dati semplici tramite il concetto di restrizione. In questo modo, partendo da un tipo semplice primitivo, si ottiene un nuovo tipo il cui valore può essere considerato come un sottoinsieme proprio del valore del tipo primitivo. Consideriamo, per fare chiarezza, le seguenti righe tratte dall'esempio 2:

```
<xsd:simpleType name="StatoType">
    <xsd: restriction  base=" string ">
        <xsd:enumeration value=" Italia "/>
        <xsd:enumeration value="USA"/>
    </xsd: restriction >
</xsd:simpleType>
```

Il concetto di restrizione è esplicitato dal tag restriction ed il tipo primitivo su cui viene svolta l'operazione è specificato come valore dell'attributo base. All'interno del tag *restriction* deve comparire il tipo di restrizione (che viene chiamato in gergo

facet) da applicare. Nell'esempio precedente viene usato il facet *enumeration* (enumerazione), che fornisce un elenco di tutti i possibili valori (appartenenti al tipo di base, quindi stringhe) che possono comparire nel tipo StatoType; in un documento Xml istanza, quindi, ogni elemento appartenente al tipo StatoType potrà contenere solamente il valore stringa Italia oppure il valore stringa USA.

... < Stato > Italia </Stato > ... < Stato >USA</Stato> ...

Vi sono numerosi *facets* che possono essere utilizzati nelle restrizioni, eccone un elenco.

- *Length*: per specificare il numero di caratteri che deve avere una stringa
- *MaxLength*: specifica il numero massimo di caratteri che può avere una stringa
- *MinLength*: specifica il numero minimo di caratteri che può avere una stringa
- *Pattern*: si fornisce un'espressione regolare che deve essere soddisfatta dal valore dell'elemento ottenuto per restrizione
- *Enumeration*: serve per fornire tutti e soli i valori che possono essere adottati dell'elemento ottenuto per restrizione
- *WhiteSpace*: specifica che tipo di comportamento bisogna seguire nei confronti dei caratteri spazio in una stringa di caratteri
- *MaxInclusive*: fornisce il massimo valore inclusivo che può contenere il valore di un nuovo tipo semplice ottenuto per restrizione a partire da un tipo primitivo specificante un numero (int, integer, long, decimal...)
- *MinInclusive*: esattamente come maxInclusive, ma si specifica il valore minimo
- *MaxExclusive*: come maxInclusive, ma il valore fornito è compreso
- *MinExclusive*: come minInclusive, ma il valore minimo è compreso
- *TotalDigits*: serve per specificare il numero di cifre che può contenere il valore di un nuovo tipo semplice ottenuto per restrizione a partire da un tipo primitivo specificante un numero (int, integer, long, decimal...)

- *FractionalDigits*: serve per specificare il numero di cifre della parte frazionaria (quella dopo il punto) che può contenere il valore di un nuovo tipo semplice ottenuto per restrizione a partire da un tipo primitivo specificante un numero frazionario (es: decimal...)

Ovviamente non tutti i facets possono essere applicati nelle restrizioni di tutti i tipi primitivi elencati nelle tabelle 1 e 2: ad esempio non si può usare il facet *minInclusive* associato ad un tipo di base *string*, ne usare *length* applicato ad un tipo che rappresenta un numero. Più facets possono comparire assieme nella stessa restrizione; consideriamo le seguenti righe:

```
<xsd:element name=”Numero” type=”NumeroType”/>

<xsd:simpleType name=”NumeroType”>
    <xsd: restriction   base=”xsd: integer ”>
        <xsd:minInclusive  value=”10”/>
        <xsd:maxInclusive value=”999”/>
    </xsd: restriction >
</xsd:simpleType>
```

In un ipotetico documento Xml istanza dell'Xml-Schema in cui compare la definizione del tipo NumeroType potrebbe comparire il tag

```
<Numero>54</Numero>
```

Infatti un elemento appartenete al tipo NumeroType appena definito deve contenere un valore di tipo integer (integer è infatti il valore dell'attributo *base* dell'elemento *restriction*) compreso fra 10 e 999 inclusi (se si fossero usati minExclusive e maxExclusive i valori 10 e 999 sarebbero esclusi).

Nell'attributo *base* può comparire, oltre che ad un tipo primitivo (es:*string*), un tipo semplice ottenuto da un tipo primitivo proprio tramite restrizione:

```
<xsd:simpleType name=”Numero1Type”>
    <xsd: restriction   base=”NumeroType”>
        <xsd:minInclusive  value=”50”/>
    </xsd: restriction >
</xsd:simpleType>
```

Gli elementi appartenenti al tipo Numero1Type potranno avere un valore compreso fra 50 e 999, valore che è stato ulteriormente ristretto rispetto all'intervallo compreso fra i valori 10 e 999 permessi dal tipo semplice NumeroType definito poco sopra.

NOTA: i facets *length*, *maxLength*, *minLength* ed *enumeration* possono essere applicati anche a restrizioni del tipo semplice derivato *List*. In questo particolare caso length, maxLength e minLength assumono un significato diverso da quello consueto (cioè indicare il numero di caratteri di un campo stringa); prendiamo in considerazione il seguente esempio:

```
<xsd:element name="Numeri" type="ListaD6iNumeri"/>

<xsd:simpleType name="ListaDiNumeri">
    <xsd: list  itemType="xsd: integer"/>
</xsd:simpleType>

<xsd:simpleType name="ListaDi6Numeri">
    <xsd: restriction  base="ListaDiNumeri">
        <xsd:length  value="6"/>
    </xsd: restriction >
</xsd:simpleType>
```

L'elemento Numeri appartenente al tipo ListaDi6Numeri rappresenta praticamente un array di 6 valori di tipo integer. Una possibile istanza potrebbe essere:

```
<Numeri>1 2 −3 234 5 6</Numeri>
```

3.4.10 Il tipo semplice anyType

Esiste, in verità, anche un altro tipo semplice, oltre a tutti quelli già citati: il tipo *anyType*. AnyType rappresenta il tipo base da cui derivano tutti gli altri tipi semplici e complessi e non pone nessun vincolo sul proprio contenuto (all'interno di un anyType possiamo quindi, per esempio, ritrovare altri elementi semplici o complessi oppure solamente un valore rappresentato da un tipo primitivo). La dichiarazione di un tipo anyType viene svolta in questo modo:

<xsd:element name=”QualunqueCosa” type=”anyType”/>

o più semplicemente con la forma abbreviata:

<xsd:element name=”QualunqueCosa”/>

AnyType rappresenta infatti il tipo di default nel caso il tipo non venga direttamente specificato nella clausola *type*.

3.4.11 Le definizioni di tipo anomino

In Xml-Schema si ha una definizione di tipo detta anonima quando nella dichiarazione di un elemento non è presente l'attributo *type* ma l'elemento *element* contiene al suo interno direttamente l'elemento *complexType* (oppure *simpleType* nel caso di tipo semplice). Riprendiamo la definizione del tipo IndirizzoType presente nell'Xml-Schema rappresentante l'ordine d'acquisto (Esempio 2) e riscriviamola in modo che contenga una definizione anonima:

```
<xsd:complexType name=”IndirizzoType”>
    < xsd:element name =”Stato” type=”StatoType”/>
    < xsd:element name =”Città” type=”xsd: string ”/>
    < xsd:element name =”Via” type=”xsd: string ”/>
    < xsd:element name=”Numero”>
        < xsd:simpleType>
            <xsd:union memberTypes=”xsd:string␣xsd: positiveInteger ”
        </ xsd:simpleType>
    </ xsd:element>
</xsd:complexType>
```

Si può notare come la definizione del tipo dell'elemento Numero sia contenuta nell'elemento stesso, e come manchi l'attributo *type* e lo stesso nome del nuovo tipo semplice unione sopra definito. La definizione è perciò anonima. Definizioni di questo tipo presentano comunque uno svantaggio abbastanza evidente: non avendo nome non possono essere riutilizzate per più elementi.

3.4.12 Elementi a contenuto misto

E' possibile definire elementi complessi il cui contenuto non sia formato solamente da sottoelementi, ma sia formato da sottolementi e da stringhe di caratteri. Tali elementi vangono chiamati a contenuto misto. Il fatto che un elemento sia a contenuto misto lo si nota dal fatto che il tipo complesso *complexType* cui si riferisce presenta un attributo *mixed='true'*. Ecco un breve esempio:

```
<xsd:element name="Lettera" type="LetteraType"/>

<xsd:complexType name="LetteraType" mixed="true">
    < xsd:element name ="Nome" type="xsd:string"/>
    < xsd:element name ="Data" type="xsd:date"/>
    < xsd:element name ="Firma" type="xsd: string "/>
</xsd:complexType>
```

un istanza di questo schema potrebbe essere:

```
<Lettera>
    Caro<Nome>Marco Rossi</Nome>ti scrivo in
data<Data>2002-11-13</Data>per informarti che...

<Firma>Daniele Miselli</Firma>
</Lettera>
```

Appare evidente come il contenuto dell'elemento Lettera sia misto in quanto composto sia da sottoelementi che da stringhe di caratteri (che rappresenterebbero il contenuto di un elemento semplice).

3.4.13 Elementi a contenuto vuoto

Xml-Schema prevede la possibilità di dichiarare elementi il cui contenuto deve essere nullo. Un modello di contenuto vuoto significa non solo che non possono essere presenti elementi figli ma che l'oggetto non possiede neppure un contenuto di tipo semplice. Una dichiarazione di questo tipo può essere svolta in due modi differenti: un metodo esteso che comprende la derivazione dall'elemento di base *anyType* ed uno semplificato in cui il tipo di contenuto viene semplicemente omesso. Vediamo i due casi:

```
<xsd:element name=”Indirizzo”/>
    <xsd:complexType >
        <xsd:complexContent>
            <xsd: restriction   base=”anyType”>
                < xsd: attribute   name =”Stato”  type=”xsd: string ”/>
                < xsd: attribute   name =”Città”  type=”xsd: string ”/>
                < xsd: attribute   name =”Via” type=”xsd: string ”/>
                < xsd: attribute   name=”Numero” type=”xsd:int”/>
            <xsd: restriction />
        <xsd:complexContent/>
    </xsd:complexType>
<xsd:element/>
```

L’esempio precedente rappresenta la dichiarazione completa di un elemento Indirizzo dal contenuto vuoto, la dichiarazione semplificata è la seguente:

```
<xsd:element name=”Indirizzo”/>
    <xsd:complexType >
        < xsd: attribute   name =”Stato”  type=”xsd: string ”/>
        < xsd: attribute   name =”Città”  type=”xsd: string ”/>
        < xsd: attribute   name =”Via” type=”xsd: string ”/>
        < xsd: attribute   name ”Numero” type=”xsd:int”/>
    </xsd:complexType>
<xsd:element/>
```

Queste due dichiarazioni possono dar luogo ad elementi identici in un eventuale documento Xml istanza; eccone un esempio:

```
<Indirizzo  Stato=” Italia ”  Città=”Modena” Via=”Verdi”
    Numero=”34”/>
```

3.4.14 I namespaces

L’utilizzo dei namespaces permette di evitare collisione fra i nomi usati nei documenti Xml ed in quelli rappresentanti degli Xml-Schema tramite l’aggiunta di prefissi anteposti ai nomi locali. In questo modo $< ns1 : nome >$ e $< ns2 : nome >$ risultano

due elementi distinti (appartenenti a namespaces differenti) anche se il loro nome è identico. I prefissi anteposti ai vari elementi fanno in realtà riferimento a degli URI (Uniform Resource Identificator) associati ai prefissi stessi tramite l'attributo *xmlns*. In generale i vari *xmlns* si trovano come attributi dello stesso elemento *Schema* che contiene l'intero documento Xml-Schema, ma possono essere attributi anche di qualsiasi altro elemento all'interno dello stesso. Gli esempi 1 e 2 riguardanti gli ordini d'acquisto si aprono direttamente con la dichiarazione:

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

che associa al prefisso xsd l'URI rappresentato dal valore dell'attributo xmlns; in questo modo, dato che l'URI citato si riferisce proprio allo schema in cui è contenuta la sintassi base di Xml-Schema, possiamo distinguere fra gli elementi appartenenti a questa sintassi base (come i vari *element*, *attribute*, *schema*, *string*, *integer*, *int*, ...) ed i nuovi elementi definiti nello schema. Un concetto da tenere presente quando si considerano i namespaces è che ad un URI di riferimento non deve necessariamente corrispondere l'esistenza di un file contenente dichiarazioni di elementi, ma serve semplicemente per indicare un certo elemento come appartenenete ad un ben definito dominio. L'URI valore di un attributo xmlns può, quindi, anche non riferirsi ad un reale oggetto.

L'elemento *schema* può inoltre possedere altri elementi concernenti i nemespaces:

- *TargetNamespace*: attributo che serve per definire il namespace del documento da validare
- *ElementFormDefault*: serve per indicare se un elemento non globale deve o no avere il prefisso
- *AttributeFormDefaul*: serve per indicare se un attributo deve o no avere il prefisso

Ad esempio si potrebbe ritrovare una dichiarazione di schema come la seguente:

```
<xsd: schema xmlns:xsd="http :// www.w3.org/2001/XMLSchema"
   xmlns:mio="http ://  ferrari  .mo.it /SchemiXml/"
   targetNamespace="http ://  ferrari  .mo.it /SchemiXml/"
   elementFormDefault="unqualified"
    attributeFormDefault =" unqualified ">
```

Il valore *unqualified* serve per specificare che il prefisso non deve essere apposto ai nomi degli attributi e degli elementi non globali. Il contrario di *unqualified* è *qualified*.

3.4.15 I gruppi Sequence, Choice ed All

I tag *sequence*, *choice* ed *all* permettono di applicare alcune caratteristiche a gruppi di attributi presenti in un modello di contenuto complesso. Vediamo, tramite esempi, il loro utilizzo:

- *Sequence*: il gruppo sequence forza gli elementi contenuti a essere rappresentati in un documento Xml istanza nello stesso ordine in cui vengono presentati null'Xml-Schema

```
<xsd:element name="Indirizzo" type="IndirizzoType"/>

<xsd:complexType name="IndirizzoType">
    < xsd: sequence>
        < xsd:element name ="Stato" type="StatoType"/>
        < xsd:element name ="Città" type="xsd: string "/>
        < xsd:element name ="Via" type="xsd: string "/>
        < xsd:element name="Numero" type="NumeroType"/>
    </xsd: sequence>
</xsd:complexType>
```

  in un documento Xml un elemento Indirizzo seguente questa sintassi deve presentare i sottoelementi Stato, Città, Via e Numero nello stesso ordine in cui appaiono all'interno del gruppo *sequence*.

  ES:

```
...
<Indirizzo>
        <Stato>Italia</Stato>
        <Città>Modena</Città>
        <Via>NomeVia</Via>
        <Numero>56</Numero>
</Indirizzo>
...
```

- *Choice*: il costrutto choice permette di produrre elementi con una struttura variabile propria dei documenti formati da dati semistrutturati: in una istanza può apparire un solo elemento figlio del gruppo choice.

```
<xsd:element name="Indirizzo" type="IndirizzoType"/>

<xsd:complexType name="IndirizzoType">
    < xsd: sequence>
        < xsd:element name ="Stato" type="StatoType"/>
        <xsd: choice>
            < xsd:element name ="Città" type="xsd: string "/>
            < xsd:element name ="CAP" type="xsd: positiveInteger "/>
        </xsd: choice>
        < xsd:element name ="Via" type="xsd: string "/>
        < xsd:element name="Numero" type="NumeroType"/>
    </xsd: sequence>
</xsd:complexType>
```

in questa versione di Indirizzo vediamo che la città di riferimento può essere rappresentata tramite una stringa di caratteri (appunto il suo nome) *oppure* tramite il suo CAP. Vediamo come tutto ciò può risultare in alcune possibili istanze dell'elemento Indirizzo (NB: le seguenti istanze possono trovarsi all'interno dello stesse documento Xml)

ES:

```
...
<Indirizzo>
        <Stato>Italia</Stato>
        <Città>Modena</Città>
        <Via>NomeVia</Via>
        <Numero>56</Numero>
</Indirizzo>
...
<Indirizzo>
        <Stato>Italia</Stato>
        <CAP>41100</CAP>
        <Via>NomeVia</Via>
```

```
        <Numero>96</Numero>
</Indirizzo>
 ...
```

- *All*: gli elementi che compaiono all'interno di un gruppo *all* possono apparire, in una istanza, in qualsiasi ordine si desideri (a differenza di quanto accade con il gruppo *sequence*), e gli unici valori permessi per gli attributi *minOccurs* e *maxOccurs* sono 0 e 1. Gli elementi all'interno di un gruppo *all* possono quindi apparire in ogni ordine ed al massimo una sola volta. Ogni elemento contenuto in un gruppo di questo tipo deve essere un elemento singolo e non un gruppo a sua volta.

```
<xsd:element name="Indirizzo" type="IndirizzoType"/>

<xsd:complexType name="IndirizzoType">
    < xsd: all >
        < xsd:element name ="Stato" type="StatoType"/>
        < xsd:element name ="Città" type="xsd: string "/>
        < xsd:element name ="Via" type="xsd: string "/>
        < xsd:element name="Numero" type="NumeroType"/>
    </xsd: all >
</xsd:complexType>
```

3.4.16 Gruppi di elementi e gruppi di attributi

Xml-Schema permette, per rendere più leggibile uno schema e più riusabili i suoi componenti, di definire gruppi di elementi e di attributi esternamente alle definizioni di tipo complesso. Un gruppo di elementi è definito dal tag *group* e deve possedere un attributo *name* in modo da poter essere referenziato.

```
<xsd:element name="Indirizzo" type="IndirizzoType"/>

<xsd:complexType name="IndirizzoType">
    < xsd:sequence>
        < xsd:element name ="Stato" type="StatoType"/>
```

```
        <xsd: choice>
            < xsd:element name =”Città” type=”xsd: string ”/>
            < xsd:group ref =”GruppoCittà”/>
        </xsd: choice>
        < xsd:element name =”Via” type=”xsd: string ”/>
        < xsd:element name=”Numero” type=”NumeroType”/>
    </ xsd:sequence>
</xsd:complexType>

<xsd: group name=”GruppoCittà”>
    <xsd: sequence>
        < xsd:element name =”Città” type=”xsd: string ”/>
        < xsd:element name =”CAP” type=”xsd: positiveInteger”/>
    </xsd: sequence>
</xsd: group>
```

Come si può notare nell'esempio precedente è stato definito, tramite l'elemento *group*, un gruppo di elementi chiamato GruppoCittà e contenente due oggetti: l'elemento Città di tipo *string* e quello CAP di tipo *positiveInteger*. All'interno del gruppo gli oggetti devono apparire nell'ordine in cui vengono dichiarati (si trovano infatti all'interno di un elemento *sequence*). Tutto ciò permette di definire indirizzi in cui sia presente o l'elemento stringa città oppure gli elementi Città e CAP, cosa che senza l'aiuto del costrutto group non sarebbe stata possibile.

Per definire un gruppo di attributi si ricorre all'elemento *attributeGroup*, l'uso del quale è molto simile a quello di *group*, vediamo come:

```
<xsd:element name=”Ordine” type=”OrdineType”/>

<xsd:complexType name=”OrdineType”>
    < xsd:element name =”Indirizzo” type=”IndirizzoType”/>
    < xsd:element name =”Oggetto” type=”OggettoType” minOccurs=”1”
        maxOccurs=”unbounded”/>
    < xsd:element ref =”Commento” minOccurs=”0” />
    < xsd: attributeGroup ref =” AttributiOrdine ” />
</xsd:complexType>
```

```
<xsd: attributeGroup name="AttributiOrdine">
    < xsd: attribute name ="Data" type="xsd:date" use="required"/>
    < xsd: attribute name ="TipoConsegna" >
        <xsd: simpleType>
            <xsd: restriction base="xsd: string ">
                <enumeration value="Treno"/>
                <enumeration value="Nave"/>
                <enumeration value="Aereo"/>
                <enumeration value="Strada"/>
            </xsd: restriction >
        </xsd: simpleType>
    < /xsd: attribute >
</xsd: attributeGroup >
```

in questo caso si è definito un gruppo di attributi per l'elemento OrdineType contenente al suo interno addirittura una nuova definizione di tipo semplice anonima. Ricorrere in questo caso al costrutto attributeGroup rende lo schema più leggibile ed inoltre c'è la possibilità di sfruttare nuovamente questo gruppo in un nuovo elemento.

3.4.17 Ereditarietà in Xml-Schema

Xml-Schema fornisce due metodi per derivare nuovi oggetti a partire da oggetti già definiti:

- *Derivazione per Restrizione*
- *Derivazione per Estensione*

Alcuni esempi e metodologie di derivazione per restrizione sono già state fornite quando si è parlato di creazione di nuovi tipi semplici per restrizione; è possibile creare anche nuovi tipi complessi in questa maniera, mentre non è possibile creare nuovi tipi semplici per estensione di contenuto. L'estensione di contenuto, infatti, espande un tipo di dato precedente aggiungendo i nuovi elementi o attributi definiti nell'espansione (un tipo semplice non può avere elementi o attributi). Quindi, ricapitolando, è possibile derivare un nuovo tipo di dato complesso sia tramite la restrizione che tramite l'espansione, mentre per quanto riguarda i dati semplici, è possibile usare solamente la restrizione. Dato che la parte riguardante l'ereditarietà dei tipi semplici

è già stata descritta nel capitolo intitolato -I TIPI SEMPLICI DERIVATI- verranno di seguito spiegate solamente le regole riguardanti quelli complessi.

- *Nuovi tipi complessi ottenuti per Estensione*

  Ovviamente un nuovo tipo di dato complesso deve essere dichiarato a partire dall'elemento *complexType*, subito dopo, però, bisogna indicare se vogliamo derivare da un tipo semplice (utilizzando l'elemento *simpleContent*) o complesso (utilizzando *complexContent*) e poi usare l'elemento *extension*. Nell'attributo base di extension bisogna inserire il nome del tipo da cui si deriva e, all'interno dello stesso tag *extension*, si inseriscono i nuovi elementi e attributi che si desidera aggiungere all'elemento padre. E' importante notare che se si specifica il modello di contenuto come *simpleContent* si possano aggiungere solamente attributi e non elementi; un contenuto semplice, infatti, è previsto essere solamente testuale e privo di elementi innestati. Il nuovo modello di contenuto risulta essere quindi l'unione del contenuto del tipo specificato nell'attributo *base* con il nuovo contenuto compreso nel tag *extension*. Se si deriva a partire da un tipo semplice si possono aggiungere solamente attributi al suo contenuto. Supponiamo di volere estendere il contenuto del tipo complesso OggettoType (esempio 1) per poter considerare novità che devono ancora uscire sul mercato.

```
<xsd:complexType name="OggettoType">
    < xsd:element  name ="Nome" type="xsd:string"/>
    < xsd:element  name ="Quantità" type="QuantitàType"/>
    < xsd:element  name ="PrezzoEuro" type="xsd:decimal"/>
    < xsd:element  ref ="Commento" minOccurs="0"/>
</xsd:complexType>

<xsd:complexType name="OggettoNovità">
    <xsd:complexContent>
        <xsd:extension  base="OggettoType">
            < xsd:element  name ="DataDiUscita" type="xsd:data"/>
            < xsd: attribute   name ="Novità" type="xsd:boolean"  fixed ="true"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

All'interno di un documento Xml istanza un elemento del tipo OggettoNovità puo essere dichiarato in modo normale oppure comparire al posto di un elemento di tipo OggettoType, vediamo come:

```
<Ordine>
    ...
    <Oggetto>
        <Nome>Tastiera</Nome>
        <Quantità>10</Quantità>
        <PrezzoEuro>15.25</PrezzoEuro>
    </Oggetto>
    ...
    <Oggetto xsi:type="OggettoNovità" Novità="true">
        <Nome>TastieraNuova</Nome>
        <Quantità>5</Quantità>
        <PrezzoEuro>26.25</PrezzoEuro>
        <DataDiUscita>2002-11-15</DataDiUscita>
    </Oggetto>
    ...
</Ordine>
```

il tipo derivato usato deve essere comunque specificato tramite l'utilizzo dell'attributo *xsi:type* che fa riferimento al namespace http://www.w3.org/2001/XMLSchema-instance. Si è parlato precedentemente di nuovi tipi complessi ottenuti per estensione di contenuto a partire da un tipo semplice, un esempio di questo processo può essere il seguente:

```
<xsd:complexType name="PrezzoType">
    <xsd:simpleContent>
        <xsd:extension base="xsd:decimal">
            < xsd: attribute name ="Tipo" type="xsd: string " required ="true"/>
        </xsd: extension >
    </xsd:simpleContent>
</xsd:complexType>
```

le definizioni di questo tipo servono per creare elementi con un contenuto sostanzialmente semplice ma che possono avere attributi. Per capire meglio il funziona-

mento di questo concetto viene di seguito presentata una possibile istanza Xml (supponendo che l'elemento prezzo sia di tipo Prezzo Type):

<Prezzo Tipo="Euro">34.65<Prezzo/>

- *Nuovi tipi complessi ottenuti per Restrizione*

  Oltre che derivare nuovi tipi complessi estendendo un modello di contenuto è possibile anche effettuare l'operazione opposta, derivando restringendo il modello di contenuto. Il procedimento è concettualmente simile alla creazione di nuovi tipi semplici per restrizione. Un oggetto complesso ottenuto in questa maniera è molto simile al suo tipo di base ma le dichiarazioni presenti al suo interno sono maggiormente limitate. Consideriamo as esempio il seguente tipo complesso OrdineType:

```
<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
    < xsd: attribute   name ="Data" type="xsd:date"/>
</xsd:complexType>
```

  una possibile restrizione è:

```
<xsd:complexType name="OrdineStock">
    <xsd:complexContent>
        <xsd: restriction   base="OrdineType">
            < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
            < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="50"
                maxOccurs="100"/>
            < xsd:element  ref  ="Commento" minOccurs="0" />
            < xsd: attribute   name ="Data" type="xsd:date"/>
        </xsd: restriction >
    </xsd:complexContent>
</xsd:complexType>
```

possiamo notare come le dichiarazioni di elementi presenti all'interno del nuovo tipo OrdineStock sono più vincolate rispetto a quelle di OrdineType, infatti è possibile avere un numero di elementi Oggetto fra 50 e 100, range più ristretto rispetto a quello presente in OrdineType che spazia da 1 a infinito. Tramite questo procedimento è possibile eliminare gli elementi opzionali, come Commento, eliminando la loro dichiarazione all'interno della definizione del tipo derivato.

3.4.18 I substitution groups

Se nello schema sono presenti elementi globali (figli direttamente dell'elemento *Schema*) è possibile fare in modo che essi possano essere sostituiti nel documento istanza da altri elementi; questo meccanismo è chiamato substitution groups. Per implementare suddetto procedimento si devono dichiarare gli elemento globali che vogliamo come sostituti ed aggiungere ad essi l'attributo *substitutionGroup* il cui valore deve essere il nome dell'elemento che può essere rimpiazzato. Ovviamente tutte le entità appartenenti al medesimo substitution group devono essere dello stesse tipo. Supponiamo di voler dichiarare come possibile sostituto per l'elemento globale Commento l'oggetto MioCommento:

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

<xsd:element name="MioCommento" type="xsd:string"
    substitutionGroup ="Commento"/>

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
    < xsd: attribute   name ="Data" type="xsd:date"/>
</xsd:complexType>
```

ed in una eventuale istanza potrebbe comparire:

```
<Ordine Data="2002-11-01">
    <Indirizzo>
```

```
        ...
    </Indirizzo>
    <Oggetto>
        ...
    </Oggetto>
    <MioCommento>Ordine già arrivato</MioCommento>
</Ordine>
```

3.4.19 Elementi e tipi astratti

In Xml-Schema è possibile definire come astratti sia elementi globali (figli direttamente dell'elemento *Schema*) che nuovi tipi di dato. Definire un'entità come astratta serve per forzare il meccanismo della sostituzione: nei documenti istanza non potranno comparire ne elementi ne tipi astratti ma solamente elementi appartenenti allo stesso substitution group oppure, per quanto riguarda le definizioni di tipo, si sfrutta il meccanismo dello *xsi:type* già descritto nel paragrafo concernente l'ereditarietà. Per quanto riguarda la sintassi bisogna aggiungere alla dichiarazione dell'elemento globale o alla definizione del nuovo tipo di dato l'attributo *abstract='true'*. Ad esempio:

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

<xsd:element name="MioCommento" type="xsd:string"
    substitutionGroup ="Commento"/>
```

In questo caso in una generica istanza non potrà comparire mai l'oggetto Commento che sarà sempre sostituito da MioCommento.

3.4.20 L'elemento Any e l'attributo anyAttribute

Per poter estendere ulteriormente il modello di contenuto di un elemento possiamo usare l'elemento *Any*. Si consideri il seguente esempio:

```
<xsd:element name="ContenutoHTML">
    <xsd:complexType>
        <xsd:any namespace="http://www.w3.org/1999/xhtml"
```

```
            minOccurs="1" maxOccurs="unbounded"/>
    </xsd:complexType>
</xsd:element>
```

L'elemento ContenutoHTML può avere un numero imprecisato (da uno a infinito) di elementi figli di tipo *Any*. L'attributo *namespace=http://www.w3.org/1999/xhtml* indica che il generico elemento Any in questione deve essere un qualunque elemento well-formed appartenente al namespace indicato, in questo caso un qualunque elemento HTML. Potranno quindi comparire all'interno dell'elemento ContenutoHTML (in una istanza Xml) elementi quali $< Table >, < BR >, < Body >$...

Un discorso molto simile può essere fatto per l'attributo *anyAttribute*; se all'interno di un elemento compare, ad esempio:

```
<xsd:anyAttribute namespace="http://www.w3.org/1999/xhtml"/>
```

nel documento Xml istanza potremmo ritrovare un qualsiasi attributo descritto in HTML. Il valore dell'attributo *namespace* non deve essere per forza definito da un URI, ma può assumere i seguenti valori:

Valore dell'attributo namespace	**Contenuti permessi**
##any	Ogni well-formed XML da ogni namespace (default)
##local	Ogni well-formed XML che non sia qualificato, i.e. non dichiarato come appartenente ad un namespace
##other	Ogni well-formed XML che non sia del target namespace
'http://www.w3.org/1999/xhtml ##targetNamespace'	Ogni well-formed XML che appartenga ad un namespace della lista

3.4.21 Chiavi, riferimenti a chiave a valori unici

Xml-Schema permette, tramite gli elementi *unique*, *key* e *keyref* di specificare valori di chiave, riferimenti a chiavi e chiavi candidate (valori unici). Questi elementi rappresentano un netto miglioramento rispetto agli attributi ID ed IDREF usati nelle DTD, infatti, mentre con IDREF si indicava solamente la presenza di un riferimanto ad una chiave senza specificare esattamente a quale, con la coppia key e keyref si conosce esattamente l'abbinamento voluto. Per spiegare meglio questi concetti ed il loro utilizzo ripropongo l'esempio completo sull'ordine d'acquisto con l'aggiunta di chiavi ed elementi unici.

Esempio 3:

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

<xsd:element name="OrdiniEPromozioni"
    type="OrdiniEPromozioniType"/>

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

<xsd:complexType name="OrdiniEPromozioniType">
    < xsd:element  name ="Ordine" type="OrdineType" minOccurs="0"
        maxOccurs="unbounded"/>
    < xsd:element  name ="Promozione" type="PromozioneType"
        minOccurs="0" maxOccurs="unbounded"/>
</xsd:complexType>

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
    < xsd: attribute   name ="Data" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="IndirizzoType">
    < xsd:element  name ="Stato"  type="xsd: string "/>
    < xsd:element  name ="Città"  type="xsd: string "/>
```

```
    < xsd:element  name =”Via” type=”xsd: string ”/>
    < xsd:element  name=”Numero” type=”xsd:int”/>
</xsd:complexType>

<xsd:complexType name=”OggettoType”>
    < xsd:element  name =”CodiceOggetto” type=”xsd: positiveInteger ”/>
    < xsd:element  name =”Nome” type=”xsd:string”/>
    < xsd:element  name =”Quantità” type=”QuantitàType”/>
    < xsd:element  name =”PrezzoEuro” type=”xsd:decimal”/>
    < xsd:element  name =”InPromozione” type=”xsd: string ”>
    < xsd:element  ref =”Commento” minOccurs=”0”/>
</xsd:complexType>

<xsd:simpleType name=”QuantitàType”>
    <xsd: restriction
        base=”xsd: positiveInteger ”>
    <xsd:maxExclusive value=”100”/>
    </xsd: restriction >
</xsd:simpleType>

<xsd:complexType name=”PromozioneType”>
    < xsd:element  name =”Nome” type=”xsd:string”/>
    < xsd:element  name =”ScontoPercentuale”
        type=”xsd: postiveInteger ”/>
    < xsd:element  name =”DataDiInizio” type=”xsd:date”/>
    < xsd:element  name=”Durata” type=”xsd:duration”/>
</xsd:complexType>

<xsd:unique name=”NomeUnico”>
    < xsd: selector  xpath= ”/OrdiniEPromozioni/Ordine/Oggetto”/>
    < xsd: field  xpath=”CodiceOggetto”/>
<xsd:unique>

<xsd:key name=”NomePromozione”>
    < xsd: selector  xpath= ”/OrdiniEPromozioni/Promozione”/>
    < xsd: field  xpath=”Nome”/>
<xsd:key>
```

```
<xsd:keyref name="RifPromozione refer="NomePromozione ">
    < xsd: selector   xpath= "/OrdiniEPromozioni/Ordine/Oggetto"/>
    < xsd: field   xpath="InPromozione"/>
<xsd:keyref>

</xsd:schema>
```

Nell'esempio è stato aggiunto un elemento Promozione ed un elemento PromozioneE-Ordini che serve solamente a contenere i vari oggetti dello schema. Sono state inserite anche la definizione di un nome unico (chiave candidata), una chiave vera e propria ed un riferimento alla chiave per creare una relazione fra gli elemento Oggetto e Promozione. Vediamo che sono stati usati gli elementi *unique*, *key* e *keyref* composti tutti da un campo *selector* ed uno *field*. Il sottoelemento selector serve per specificare il campo di validità della chiave, del riferimento o della chiave candidata, mentre *field* fa riferimento all'elemento o all'attributo che, insieme al percorso dato dall'attributo *xpath* di *selector*, rappresenta il nome unico (o il riferimento ad esso). Entrambi gli elementi *selector* e *field* devono possedere un attributo *xpath* che rappresenta, appunto, una espressione nell'apposito linguaggio Xpath 1.0[10] atto a navigare fra gli elementi di un documento Xml come fra i nodi di un albero. Per capire meglio consideriamo la dichiarazione di NomeUnico:

```
 <xsd:unique name="NomeUnico">
    < xsd: selector   xpath= "/OrdiniEPromozioni/Ordine/Oggetto"/ >
    < xsd: field   xpath="Codice"/>
<xsd:unique>
```

La chiave candidata è rappresentata dall'elemento Codice, il cui valore che deve essere unico fra tutti quelli contenuti negli elementi Oggetto (il campo validità è dato dalla stringa xpath /OrdiniEPromozioni/Ordine/Oggetto). La definizione dell'elemento chiave è identica a quella dell'elemento unico mentre il riferimento alla chiave RifPromozione presenta un attributo *refer* che contiene il nome della chiave cui si fa riferimento, NomePromozione nel caso specifico. Vediamo come può risultare il documenti Xml istanza dello schema in esempio 3:

```
<xml version=1.0> <OrdiniEPromozioni>
```

```
<Ordine Data="2002-11-01">
    <Indirizzo>
        <Stato>Italia</Stato>
        <Città>Modena</Città>
        <Via>NomeVia</Via>
        <Numero>56</Numero>
    </Indirizzo>
    <Oggetto>
        <Codice>244</Codice>
        <Nome>Monitor</Nome>
        <Quantità>10</Quantità>
        <PrezzoEuro>255.25</PrezzoEuro>
        <Commento>Monitor LCD 15 pollici</Commento>
        <InPromozione>Promozione A</InPromozione>
    </Oggetto>
    <Oggetto>
        <Codice>59</Codice>
        <Nome>Tastiera</Nome>
        <Quantità>10</Quantità>
        <PrezzoEuro15.25</PrezzoEuro>
        <InPromozione>Promozione B</InPromozione>
    </Oggetto>
</Ordine>

<Promozione>
    <Nome>Promozione A</Nome>
    <ScontoPercentuale>10</ScontoPercentuale>
    <DataDiInizio>2002-11-20</DataDiInizio>
    <Durata> P1Y2M3DT10H30M12.0S </Durata>
</Promozione>

<Promozione>
    <Nome>Promozione B</Nome>
    <ScontoPercentuale>7</ScontoPercentuale>
    <DataDiInizio>2002-11-20</DataDiInizio>
    <Durata> P1Y2M3DT10H30M12.0S </Durata>
```

</Promozione>

<OrdiniEPromozioni>

Da questa istanza si nota bene il funzionamento delle chiavi e dei riferimenti ad essi, si può infatti osservare come il contenuto degli elementi InPromozione (riferimento e chiave) presenti solo valori contenuti negli elementi Nome di Promozione (chiave). Si possono così mettere in relazione le due entità Promozione e Oggetto.

3.4.22 Le notazioni

Ad un Xml-Schema è possibile aggiungere commenti che possono essere utili per la comprensione dello stesso a lettori umani o a programmi applicativi. Questi commenti sono interni ad un elemento *Annotation*. In un documento istanza non saranno presenti questi oggetti che si servono solamente a rendere più leggibile lo schema in cui si trovano. L'elemento *Annotation* può avere al suo interno due sottoelementi figli:

- *Documentation*
- *AppInfo*

Documentation contiene al suo interno eventuali commenti per gli utenti, mentre *appInfo* contiene informazione che possono essere utili ad applicazioni esecutive che leggono la struttura dello schema. L'elemento Documentation può presentare l'attributo *xml:lang* che specifica in quale lingua è scritto il commento interno. Riscriviamo la definizione di Ordine presentando al suo interno una Annotation:

```
<xsd:element name="Ordine" type="OrdineType"/>

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

<xsd:complexType name="OrdineType">
    < xsd: annotation >
        < xsd:documentation xml:lang="it">
         definizione del tipo OrdineType, questo commento è scritto in italiano !
        </xsd:documantation>
    < /xsd: annotation >
    < xsd:element name ="Indirizzo" type="IndirizzoType"/>
```

```
    < xsd:element  name =”Oggetto” type=”OggettoType” minOccurs=”1”
        maxOccurs=”unbounded”/>
    < xsd:element  ref  =”Commento” minOccurs=”0” />
    < xsd: attribute   name =”Data” type=”xsd:date”/>
</xsd:complexType>
```

3.4.23 Include ed Import

Gli elementi *include* e *import* servono per poter utilizzare all'interno dell'Xml-Schema definizioni fornite su file separati da quello contenente lo schema stesso. Il funzionamento dei due elementi è leggermente diverso: *include* serve per importare definizioni esterne al documento provenienti da uno schema che si riferisce allo stesso target namespace (è quindi come avere uno schema diviso fra più file), mentre *import* viene usato per importare elementi da differenti target namespaces. Se, per esempio, vogliamo aggiungere ad uno schema le dichiarazioni fornite su di un altro documento Xml-Schema con lo stesso target namespace basta aggiungere la riga di codice:

```
<xsd:include
schemalocation=”http ://  ferrari  .mo.it /SchemiXml/schema1.xsd/”/>
```

in cui il valore dell'attributo *schemalocation* rappresenta l'URI cui ritrovare il file con le dichiarazione desiderate. Gli elementi importati dal file schema1.xsd possono essere usati apponendo loro lo stesso prefisso impiegato per indicare le entità appartenenti al target namespace (come già citato, infatti, i due target namespaces devono essere coincidenti).

Per usare un elemento definito in un documento referente un diverso namespace, invece, si usa l'espressione:

```
<xsd:import
namespace=”http ://  ferrari  .mo.it /SchemiXml/namespace1/”/>
```

e si aggiunge alla dichiarazione dell'elemento schema un'attributo xmlns per indicare un prefisso da apporre agli elementi importati.

Capitolo 4

Il linguaggio ODL$_{I^3}$

4.1 Introduzione

Il linguaggio ODL$_{I^3}$[36] rappresenta una estensione al precedente ODL[26, 27] (Object Definition Language). L'ODL è stato proposto dal gruppo di standardizzazione ODMG-93 al fine di rappresentare, in un formato standard, la conoscenza relativa ad un singolo schema ad oggetti. L'ODL può essere considerato un punto di partenza in un'opera di integrazione di informazioni: può essere sufficiente nel caso si debbano mediare solamente i dati relativi a schemi ad oggetti. Nel caso, come il progetto MOMIS si propone di fare, si vogliano integrare informazioni provenienti da sorgenti eterogenee, comprendenti eventualmente anche sorgenti di dati semistrutturati, questo linguaggio non è più soddisfacente. Si è dunque deciso di apportare alcune estensioni all'ODL in accordo con le indicazioni del programma I3 (Integrazione Intelligente delle Informazioni) dell'ARPA, creando il linguaggio ODL$_{I^3}$. Il programma I3, formato presso l'università del Maryland, si propone di creare una architettura di riferimento per integrare sorgenti eterogenee in maniera automatica. Nei prossimi paragrafi sarà presentato brevemente il linguaggio ODL e, successivamente, saranno descritte le modifiche ad esso apportate per ottenere l'ODL$_{I^3}$.

4.2 Il linguaggio ODL

Di seguito sarà brevemente trattata la struttura dati del linguaggio ODL; come si potrà notare tramite questa struttura possono essere rappresentati schemi ad oggetti ma non schemi referenti dati semistrutturati.

4.2.1 Tipi classe e classi valore

L'ODL prevede una fondamentale distinzione dei suoi tipi di dati un due macro-categorie:

- *Tipi Valore*
- *Tipi Classe*

I tipi semplici vengono impiegati per la dichiarazione di variabili ed attributi semplici. La principale distinzione fra questi tipi ed i tipi classe (chiamati più semplicemente classi) è che lo loro non possiedono un identificatore unico ma l' identità è definita direttamente dal proprio valore. Le istanze dei tipi classe, invece, possiedono tutte un proprio OID (Object IDentifier) che deve essere unico. In pratica l'identità di un oggetto complesso è data dal proprio OID.

4.2.2 I tipi valore

I tipi valore (o *valueType*) si differenziamo in due distinte categorie:

- *SimpleType*
- *ConstrType*

4.2.3 I tipi semplici

I *simpleType* (o tipi semplici) rappresentano i tipi atomici di base. Vi sono diversi tipi semplici predefiniti nel linguaggio ODL; eccone un elenco:

- *Boolean*
- *String*
- *Octet*
- *Int*
- *Short*
- *Long*

- *Float*
- *Double*
- *Char*

I tipi Int, Short e Long possono anche essere dichiarati *unsigned*. Oltre quelli già citati esiste un ulteriore tipo semplice: *any*. Il tipo any è il tipo da cui derivano tutti gli altri presenti nel linguaggio ODL$_{I^3}$, sia i tipi valore che quelli classe. Il contenuto di un oggetto *any* può essere in pratica qualsiasi cosa.

4.2.4 I constrType

La seconda categoria di tipi valore è quella dei *constrType* (tipi costrutti) che contiene tre sottocategorie:

- *StructType*
- *EnumType*
- *UnionType*

Un esempio di definizione di tipo struttura (structType) è il seguente:

```
typedef struct nometag {
    int a;
    string b;
    unsigned long c;
    } tipostruct ;
```

gli elementi contenuti in un tipo struttura devono essere tipi valore. *Nometag* è opzionale ed una variabile strutturata come nell'esempio precedente può essere dichiarata indipendentemente in due modi distinti:

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

si noti che tramite l'utilizzo del termine opzionale *nometag* le dichiarazioni devono sempre essere accompagnate dall'utilizzo della parola chiave *struct*. I tipi enumerazione (*EnumType*) servono per restringere il campo dei valori possibili di un *simpleType* a quelli presenti in un elenco. Consideriamo il seguente esempio esplicativo:

```
enum colore { rosso , nero , bianco , verde , blu };
```

una variabile di tipo colore potrà assumere solamente uno dei valori (di tipo stringa) contenuti all'interno dell'elenco fra le parentesi graffe. Il tipo unione, invece, serve per scegliere, in base al valore di una variabile in una clausola *switch*, il tipo di appartenenza di una variabile.

```
union NumeroType switch(int) {
    case 1: string prova;
    case 2: int prova;
};
```

Nel precedente esempio NumeroType può essere una stringa di caratteri oppure un intero. Come per il caso del tipo struttura, la variabili che compaiono in un'unione devono essere *valueType* e non tipi classe.

4.2.5 I tipi collezione

I tipi semplici sono tutti tipi atomici. Per rappresentare collezioni di dati semplici si sfruttano, appunto, i tipi collezione messi a disposizione dal linguaggio ODL: *set*, *bag*, *list* e *array*. I tipi collezione sono anch'essi un sottoinsieme dei tipi valore. Se si desidera, ad esempio, creare un insieme ordinato di elementi di genere stringa la formulazione è la seguente:

```
list < string > ListaDiStringhe ;
```

Se si desidera, invece, formare una collezione non ordinata di tali elementi si una il termine *set*:

```
set < string > SetDiStringhe;
```

4.2.6 I tipi classe

In ODL le classi vengono chiamate anche interfacce (*interface*) ed ognuna di esse è costituita da due parti fondamentali:

- *Interface header*
- *Interface body*

Nell'interface header si specifica il nome della classe, le classi da cui eredita (se presenti) ed una lista di proprietà chiamata *typePropertyList*. In ODL è ammissibile l'ereditarietà multipla: ogni classe può avere, cioè, una o più superclassi. Per ogni superclasse vengono ereditati tutti gli attributi e tutti i metodi. Nella *typePropertyList* possono essere specificate proprietà come l'*extent*. L'extent rappresenta l'insieme di tutte la istanze di una particolare classe all'interno di un Database. Sempre all'interno di questa lista di proprietà (che deve essere racchiusa da parentesi tonde) è possibile definire una chiave (*key*). Ogni chiave può mappare un singolo attributo presente nel corpo dell'interfaccia (chiave semplice) oppure può mappare un insieme di attributi (chiave composita). La seconda parte di una classe ODL è data dall'*interface body* (il corpo dell'interfaccia) che ne descrive la struttura interna (l'insieme di attributi e di metodi che ne fanno parte). Un interface body può essere composto dai seguenti elementi:

- *Attributi*: gli attributi presenti in una classe (interfaccia) ODL possono essere sia semplici, sia complessi; nel primo caso il tipo dell'attributo apparterrà ad un tipo valore (*valueType*), nel secondo, invece, sarà un tipo classe
- *Relationships*: la relazioni possono fare riferimento solamente ad interfacce, mai a tipi valore. E' possibile aggiungere inoltre informazioni sulla relazione inversa. In pratica la relationship si può descrivere come un'attributo complesso per cui può esistere la relazione inversa.
- *Operazioni*: la operazioni rappresentano i metodi della classe, quindi descrivono il comportamento della classe stessa.
- *Costanti*: la sintassi della costanti in ODL è praticamente questa dell'ANSI C: si deve specificare il nome della costante, il tipo della stessa (che deve essere un tipo semplice) ed il suo valore.

Un esempio di interfaccia ODL è il seguente:

```
interface  Indirizzo  ( extent  Indirizzo )
{
```

```
    attribute   string  Stato ;

    attribute   string  Città ;

    attribute   string  Via;

    attribute   int  Numero;
  };
```

Il corpo dell'interfaccia indirizzo è formato da quattro attributi (tutti semplici), e nella sua lista di proprietà (fra parentesi tonde) è presente l'elemento *extent*.

4.3 L'estensione di ODL: il linguaggio ODL$_{I^3}$

Esamineremo di seguito le aggiunte e le modifiche che sono state apportate al linguaggio ODL seguendo il lavoro dell'I3 workgroup. Si ricorda che il motivo di tali cambiameni è dovuto al fatto che l'ODL pur essendo ben progettato per la rappresentazione della conoscenza relativa ad un singolo schema ad oggetti, risulta insufficiente per la descrizione relativa ad un insieme di sorgenti di basi di dati eterogenee. Nel progetto MOMIS, quindi, il linguaggio ODL può risultare solamente una buona base di partenza.

4.3.1 Estensione ai tipi valore

Ad i vari tipi valore presenti in ODL è stato aggiunto il tipo *range*. *Range* è un tipo di dato semplice che serve per rappresentare un valore intero compreso fra due estremi. Consideriamo l'esempio:

```
range 10,100 Numero;
```

la variabile Numero può assumere qualunque valore intero compreso compreso fra i valori 10 e 100. Per rappresentare il concetto di infinito si usano i termini *+INF* e *-INF*; in questo modo una variabile Numero che può assumere i valori da 10 ad infinito viene dichiarata nella seguente maniera:

```
range 10,+INF Numero;
```

4.3.2 Estensioni ai tipi classe

Le estensioni apportate alle classi (interfacce) ODL nel linguaggio ODL$_{I^3}$ sono diverse:

- Per ogni classe è data la possibilità di indicare il nome ed il tipo della sorgente di appartenenza (relazionale, ad oggetti, semistrutturato, file). Questa estensione è resa necessaria dal fatto che, trattando diverse sorgenti di dati, non è esclusa la possibilità di avere due classi con lo stesso nome. La presenza del nome della sorgente rende unica una classe all'interno di un documento ODL$_{I^3}$. Il nome ed il tipo della sorgente si trovano all'interno della lista di proprietà della classe (assieme all'*extent* e alla dichiarazione delle chiavi).

- Poichè il linguaggio ODL$_{I^3}$ deve essere in grado di descrivere anche schemi relazionali, è stata inserita la possibilità di definire (sempre all'interno della sezione riguardante le proprietà della classe) delle foreign keys. In una dichiarazione di chiave estera si deve riportare il nome degli attributi che compongono la chiave e , preceduto dalla parola *references*, il nome della classe cui si riferisce la foreign key.

- È stato aggiunto il costrutto *union* tramite il quale si possono assegnare più *interface body* ad una stessa classe. La possibilità di avere più strutture dati associate ad una stessa entità serve per poter rappresentare in ODL$_{I^3}$ schemi di dati semistrutturati, in cui, appunto, un elemento può comparire nello stesso documento con rappresentazioni differenti.

- Sempre per rappresentare dati semistrutturati è stato inserito il costrutto *optional*. Postponendo un asterisco (*) alla dichiarazione di un attributo (all'interno del corpo di un'interfaccia) si afferma che in una istanza della classe stessa l'apparizione dell'attributo in questione è opzionale.

- Oltre ai normali attributi locali (normalmente usati in ODL) è possibile dichiarare anche attributi globali. Gli attributi globali (*globalAttribute*) hanno le stesse caratteristiche dei normali attributi ODL ma, in più, presentano un collegamento ad un oggetto *MappingRule*.

4.3.3 Gli oggetti mappingRule

Qualora, all'interno di un corpo di interfaccia, un attributo globale presenti un riferimento ad una mappingRule (un insieme di regole di mapping) esso è considerato automaticamente un attributo globale. Gli attributi globali si trovano all'interno di classi generate, dal software del progetto MOMIS, dal raggruppamento di più classi ritenute simili fra loro. Le mappingRule sono, appunto, regole di mapping fra gli attributi di una classe generata con il suddetto procedimento e quelli delle classi originarie. In un mapping di questo tipo si possono presentare i seguenti casi:

- Corrispondenza fra un attributo globale ed un solo attributo locale (di una sola classe); è questo il caso più semplice.
- Corrispondenza fra un attributo globale ed una serie di attributi locali in *and* fra loro (un insieme di attributi appartenenti a classi differenti ma in fusione tra loro)
- Corrispondenza fra un attributo globale ed una serie di attributi locali in *union* fra loro. In questo caso l'attributo globale può corrispondere solamente ad uno per volta fra gli attributi in union.
- Un valore di default. Un valore di default può servire per esprimere un concetto se non c'è corrispondenza fra un attributo globale e quelli locali.

4.3.4 Relazioni terminologiche

Un'altra aggiunta alla sintassi base del linguaggio ODL è la possibilità di inserire relazioni terminologiche che possono intercorrere fra classi, attributi o miste fra classi ed attributi. Le relazioni fra attributi sono memorizzate in un oggetto *InterfaceRel*, quella fra attributi in un oggetto *AttributeRel* ed invece, quelle miste in uno *AttrIntRel*. Vi sono quattro tipi di relazioni considerate:

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

4.3.5 Regole if-then

ODL$_{I^3}$ offre la possibilità di definire regole di integrità di tipo *if-then* in grado di fornire informazioni non specificabili altrimenti. Le condizioni citate in queste regole devono essere rispettate dalle istanze delle classi ODL$_{I^3}$ memorizzate in un Database. La generica sintassi di una regola di integrità è la seguente:

rule nomerule forall iteratore in collezione : antecedente then conseguente

oppure:

rule nomerule [{case of indentifier : caselist }]

I vari termini che compaiono nelle due sintassi hanno I seguenti significati:

- *Nomerule*: è il nome fornito dall'utente alla regola d'integrità
- *Iteratore*: rappresenta una istanza di un elemento fra quelli appartenenti a collezione
- *Collezione*: un insieme di istanze. Di solito è rappresentata da una classe o parte di essa.
- *Antecedente*: è la parte *if* della regola, è formata da una serie di predicati booleani in AND fra loro
- *Conseguente* : è la parte *then* della regola; definisce il comportamento delle entità che soddisfano il predicati *if*

Capitolo 5

Specifica di un wrapper per schemi XML

5.1 Introduzione

In questo capitolo si tratterà una possibile traduzione dal linguaggio Xml-Schema ad ODL_{I^3}. Tale traduzione potrà essere sfruttata nella realizzazione di un wrapper da impiegare all'interno del sistema MOMIS. Naturalmente le traduzioni possibili fra un linguaggio e l'altro possono essere molte ed anche discretamente differenti fra loro. La trasposizione presentata di seguito è stata influenzata dalle seguenti scelte:

- Si vuole avere una traduzione il più possibile bidirezionale (con la possibilità, quindi, di poter tornare dalla versione ODL_{I^3} a quella Xml-Schema)
- Nella trasposizione dei concetti non si devono MAI perdere informazioni fondamentali come il nome degli elementi e la struttura degli stessi

Diciamo subito che, purtroppo, il primo punto sarà, per quanto riguarda alcuni costrutti dei due linguaggi, irrealizzabile; Xml-Schema ed ODL_{I^3} sono stati creati per scopi molto differenti tra loro e, proprio per questo motivo, molti concetti non riusciranno ad essere tradotti efficacemente. Il secondo punto, invece, è dato dal fatto che la traduzione è presentata nell'ambito del progetto MOMIS. L'intenzione è quella di tradurre in una rappresentazione comune (il linguaggio descrittivo ODL_{I^3}) schemi relazionali, ad oggetti, e semistrutturati (è il nostro caso) per poter trovare similarità fra le varie classi ODL_{I^3}. Come si può ben comprendere è pertanto necessario mantenere intatti tutti i nomi delle classi, degli elementi e degli attributi contenuti in esse che

ci possono fornire preziose indicazioni sul contenuto della classi stesse. Il progetto MOMIS non intende solamente trovare similarità fra le classi, ma si propone anche di poter reperire le informazioni ed i dati contenuti nei differenti documenti originari tramite l'esecuzione di query. Le query saranno prima di tutto formulate in OQL (Object Query Language), un linguaggio di interrogazione per ODL ed ODL_{I^3}, e successivamente tradotte nei linguaggi di interrogazione specifici per i vari documenti le cui viste sono state tradotte in classi ODL_{I^3}. E' importante comprendere quindi che Xml-Schema descrive la struttura di documenti Xml e che le informazioni reperibili da una Query (scritta ad esempio nel linguaggio XQuery 1.0 per l'interrogazione di fogli Xml) si trovano proprio in tali documenti; la traduzione di un Xml-Schema in uno ODL_{I^3} non dovrà assolutamente compromettere la reperibilità di tali informazioni. Volendo essere più specifici possiamo dire che, per ottenere lo stesso dato, una eventuale query XQuery 1.0 e la sua corrispondente in OQL devono attraversare lo stesso percorso (path) all'interno del documento Xml (istanza dell'Xml-Schema) ed attraverso le classi dello schema OLDi3.

5.2 Traduzione dei tipi semplici primitivi

Iniziamo la traduzione da un documento Xml-Schema ad uno ODL_{I^3} considerando i tipi semplici primitivi. ODL_{I^3} fornisce solamente un numero limitato di questi tipi, che vengono chiamati tipi semplici di base, per cui non è possibile ottenere una trasposizione uno ad uno di simili concetti. Ciò che invece si può conseguire è una corrispondenza uno a molti fra tipi di base ODL_{I^3} ed i tipi semplici primitivi di Xml-Schema. La traduzione proposta viene riportata nella tabella seguente:

Tipo	**Esempio**	**Traduzione**
string	Stringa di prova	string
normalizedString	Stringa di prova	string
token	Stringa di caratteri	string
byte	-1, 126	char
unsignedByte	0, 126	char
Base64Binary	GpM7	string
hexBinary	0FB7	String
integer	-126789, -1, 0, 1, 126789	int
positiveInteger	1, 126789	int

negativeInteger	-126789, -1	int
nonNegativeInteger	0, 1, 126789	int
nonPositiveInteger	-126789, -1, 0	int
int	-1, 126789675	int
unsignedInt	0, 1267896754	unsigned int
long	-1, 12678967543233	long
unsignegLong	0, 12678967543233	unsigned long
short	-1, 12678	short
unsignedShort	0, 12678	unsigned short
decimal	-1.23, 0, 123.4, 1000.00	float
float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	float
double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	double
boolean	true, false	boolean
time	13:20:00.000, 13:20:00.000-05:00	string
dataTime	1999-05-31T13:20:00.000-05:00	string
duration	P1Y2M3DT10H30M12.3S	string
date	1999-05-31	string
gMonth	–05–	string
gYear	1999	short
gYearMonth	1999-02	string
gDay	—31	string
Name	shipTo	string
QName	po:USAddress	string
NCName	USAddress	string
anyURI	http://www.example.com/	string
language	en-GB, en-US, fr	string

Come si può notare dalla tabella i tipi di base ODL_{I^3}, come ad esempio *string*, mappano numerosi tipi di dato appartenenti ad Xml-Schema.

Vediamo ora quali sono le traduzioni proposte per i tipi importati in Xml-Schema direttamente da XML 1.0:

Tipo	Esempio	Note
ID		string
IDREF		string
IDREFS		string
ENTITY		dipende dall'entity
ENTITIES		dipende dalle entities
NOTATION	US,Brsil	string
NMTOKEN	US UK,Brsil Canada Mexique	string
NMTOKENS		string

5.3 Traduzione di elementi e tipi complessi

La traduzione degli elementi, semplici e complessi, e delle definizioni di nuovi tipi di dato complesso rappresenta il cuore dell'intero processo di trasposizione; sono proprio le gerarchie formate dagli elementi innestati che formano la struttura portante di un documento Xml, ed è proprio seguendo questa struttura che siamo in grado di ottenere l'informazione (il dato) che ci interessa. Abbiamo già visto come un Xml-Schema fornisca una rappresentazione (vista) dettagliata dell'insieme di elementi componenti un documento Xml e quindi è facile comprendere come una buona traduzione dei concetti di *element* e *complexType* rappresenti un'ottima base per tutto il lavoro.

Praticamente tutti gli esempi di schemi Xml che verranno utilizzati e tradotti in questo capitolo sono già stati presentati e descritti durante la sezione riguardante la struttura di Xml-Schema.

La scelta che è stata fatta è quella di tradurre un elemento complesso come una classe ODL$_{I^3}$ *interface* derivante per ereditarietà dall'*interface* che rappresenta il suo tipo complesso di appartenenza. Sia dunque gli element complessi che le definizioni *complexType* saranno tradotte come *interface* e per distinguere questi due costrutti si è deciso di aggiungere alla definizione dell'interface di un elemento complesso la dichiarazione extent seguita dal nome della classe stessa. In ODL$_{I^3}$, infatti, l'*extent* di una classe rappresenta la collezione di tutte le sue istanze e solamente gli oggetti

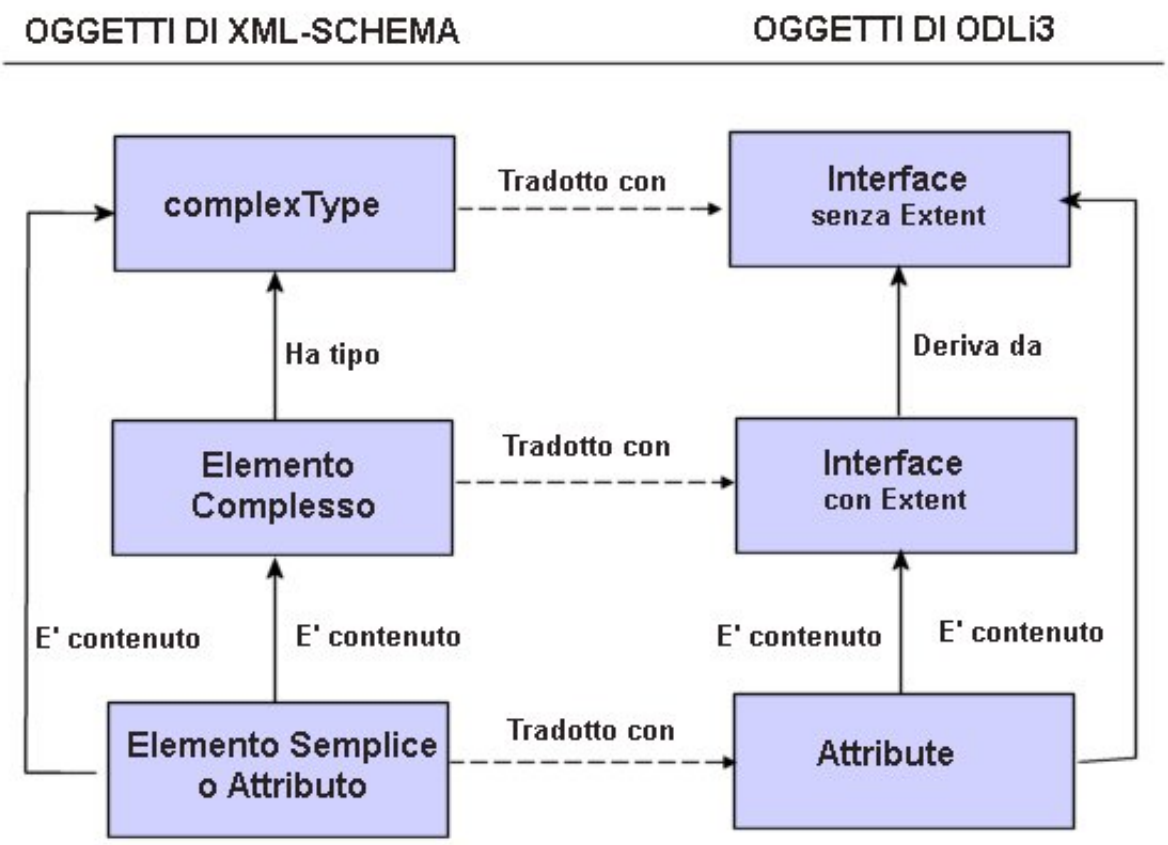

Figura 5.1: Schema della traduzione dei concetti più importanti

e le classi possono avere istanze, non i tipi di dato. Un elemento avente un contenuto semplice e sprovvisto di sottoelementi è, invece, tradotto come un attributo *attribute* contenuto all'interno dell'interface che rappresenta il *complexType* in cui si trova nell'Xml-Schema. Vediamo come possiamo quindi tradurre l'elemento Indirizzo appartenente al tipo complesso IndirizzoType:

< xsd:element name =”Indirizzo” type=”IndirizzoType”/>

<xsd:complexType name=”IndirizzoType”>
< xsd:element name =”Stato” type=”xsd: string ”/>
< xsd:element name =”Città” type=”xsd: string ”/>
< xsd:element name =”Via” type=”xsd: string ”/>

```
    < xsd:element name=”Numero” type=”xsd:int”/>
</xsd:complexType>
```

traduzione in ODL_{I^3}:

```
interface Indirizzo : IndirizzoType ( source semistructured
Ordine.xml extent Indirizzo )
{};

interface IndirizzoType ( source semistructured Ordine.xml)
{
    attribute string Stato ;

    attribute string Città ;

    attribute string Via;

    attribute int Numero;
   };
```

Una traduzione di questo tipo può in alcuni casi causare una perdita di informazione: supponiamo, ad esempio, che l'elemento semplice Numero sia definito non come *int* ma come *positiveInteger*: la traduzione in ODL_{I^3} sarebbe comunque la medesima, perdiamo quindi l'informazione del tipo di dato di base originario. Questo problema potrebbe essere ovviato tramite l'inserimento dell'informazione carente dall'interno del tipo di appartenenza dell'*attribute*.

```
interface IndirizzoType ( source semistructured Ordine.xml)
{
   typedef string elem_Stato_string ;
    attribute elem_Stato_string Stato ;

   typedef string elem_Città _string ;
    attribute elem_Città _string Città ;

   typedef string elem_Via_string ;
    attribute elem_Via_string Via;
```

```
typedef int elem_Numero_positiveInteger ;
 attribute elem_Numero_positiveInteger Numero;
};
```

in questo caso ogni elemento semplice è tradotto come un *attribute* appartenente ad un tipo definito all'interno della classe medesima il cui nome esprime: il fatto che si stia trattando un elemento, il nome dell'elemento (per eliminare possibili duplicati) ed il tipo di dato originario di Xml-Schema. A parer mio la traduzione migliore rimane comunque la prima (senza i *typedef*) poich in uno schema Xml è possibile definire nuovi tipi di dato semplice (tradotti in ODL_{I^3} anch'essi con il costrutto *typedef*) e si potrebbe ottenere una traduzione decisamente troppo ridondante per un singolo elemento.

Un secondo metodo per evitare la perdita di informazione del tipo di dato semplice è fornire, per ogni tipo di base non mappato direttamente in ODL_{I^3} (come, ad esempio, string oppure int), la traduzione all'inizio dello schema nel corrispondente tipo ODL_{I^3} tramite il costrutto *typedef*. Riprendiamo l'esempio appena trattato riguardante il tipo complesso IndirizzoType (e consideriamo che l'elemento semplice Numero contenuto al suo interno sia, ancora una volta, definito *positiveInteger*, e dunque non mappato direttamente in ODL_{I^3}):

```
<xsd:complexType name="IndirizzoType">
    < xsd:element name ="Stato" type="xsd: string "/>
    < xsd:element name ="Città" type="xsd: string "/>
    < xsd:element name ="Via" type="xsd: string "/>
    < xsd:element name="Numero" type="xsd:positiveInteger"/>
</xsd:complexType>
```

una traduzione possibile è, dunque:

```
... typedef int positiveInteger ; typedef int negativeInteger ;
typedef string ID; typedef string Date; typedef string Name; ...

interface IndirizzoType ( source semistructured Ordine.xml)
{
    attribute string Stato ;
```

```
attribute  string  Città ;

attribute  string  Via;

attribute  positiveInteger  Numero;
};
```

Le traduzioni, tramite typedef, dei tipi di base non mappati direttamente in ODL_{I^3} seguono le tabelle di trasposizione 3 e 4 descritte nel paragrafo precedente.

Nel caso un elemento complesso non sia globale (figlio direttamente dell'elemento *schema*) ma si trovi all'interno di un tipo complesso, la traduzione rimane quella precedente con l'aggiunta, nell'interfaccia che rappresenta il *complexType* contenente l'elemento, di un *attribute* il cui nome è quello del nostro oggetto. Si consideri il seguente esempio:

```
<xsd:element name="Ordine" type="OrdineType"/>

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    ...
</xsd:complexType>

<xsd:complexType name="IndirizzoType">
    < xsd:element  name ="Stato"  type="xsd: string "/>
    < xsd:element  name ="Città"  type="xsd: string "/>
    < xsd:element  name ="Via" type="xsd: string "/>
    < xsd:element  name="Numero" type="xsd:decimal"/>
</xsd:complexType>
```

traduzione in ODL_{I^3}:

```
interface  Ordine  :  OrdineType (source  semistructured  Ordine.xml
extent  Ordine ) {};

interface  OrdineType (source  semistructured  Ordine.xml)
```

```
{
    attribute  Indirizzo  Indirizzo ;
    ...
   };

interface  Indirizzo  :  IndirizzoType  ( source  semistructured
Ordine.xml  extent  Indirizzo )
 {};

interface  IndirizzoType  ( source  semistructured  Ordine.xml)
 {
    attribute  string  Stato ;

    attribute  string  Città ;

    attribute  string  Via;

    attribute  int  Numero;
   };
```

5.4 Traduzione di attributi

ODL_{I^3} non è in grado di distinguere, come Xml-Schema, fra elementi semplici ed attributi, perciò anche questi ultini saranno tradotti come *attribute*. Vi sono diverse modalità in cui si può presentare un attributo (attributo optional, a valore fisso, required, a valore di default) e verranno trattate separatamante.

5.4.1 Traduzione di attributi opzionali

Un attributo, di default, viene considerato opzionale e può comparire oppure non comparire nel documento Xml istanza. Dato che una query (ottenuta ad esempio con XQuery 1.0) posta su un documento Xml distingue fra elementi ed attributi tale distinzione deve essere resa con qualche artificio anche in ODL_{I^3}. Supponiamo di dover tradurre le seguenti righe:

```
<xsd:complexType name="OrdineType">
    < xsd:element name ="Indirizzo" type="IndirizzoType"/>
    ...
    < xsd: attribute name ="Data" type="xsd:date"/>
</xsd:complexType>
```

il cui corrispondente ODL_{I^3} risulterà:

```
interface OrdineType (source semistructured Ordine.xml)
{
    attribute Indirizzo Indirizzo ;
    ...
    typedef string attr_Data ;
    attribute attr_Data Data*;
    };
```

Il fatto che si sta considerando un attributo e non un elemento viene evidenziato dal nome del tipo dell'*attribute* Data (attr_Data, in cui attr reca l'informazione di attributo). Ovviamente in una implementazione pratica del traduttore non si deve in ogni caso usare esattamente questo tipo di nomi (attr_Data), l'importante è che il nome scelto sia differente da qualsiasi nome di tipo possa essere trovato in un Xml-Schema e conservi l'informazione voluta. L'asterisco dopo la dichiarazione dell'*attribute* Data ci indica che la sua presenza è opzionale. La traduzione risulta la medesima anche in questo caso:

```
<xsd:complexType name="OrdineType">
    < xsd:element name ="Indirizzo" type="IndirizzoType"/>
    ...
    < xsd: attribute name ="Data" type="xsd:date" use="optional"/>
</xsd:complexType>
```

5.4.2 Traduzione di attributi required

La traduzione di attributi required è praticamente identica a quella di attributi di tipo optional, l'unico particolare che cambia è che l'attributo deve essere presente nell'istanza dello schema e quindi verrà tradotto in ODL_{I^3} senza l'asterisco (che indica, appunto, un *attribute* opzionale).

```
<xsd:complexType name="OrdineType">
    < xsd:element name ="Indirizzo" type="IndirizzoType"/>
    ...
    < xsd: attribute name ="Data" type="xsd:date" use="required"/>
</xsd:complexType>
```

viene tradotto in:

```
interface OrdineType (source semistructured Ordine.xml)
{
    attribute Indirizzo Indirizzo ;
    ...
    typedef string attr_Data ;
    attribute attr_Data Data;
    };
```

5.4.3 Traduzione di attributi a valore fisso

Il fatto che un attributo debba comparire all'interno di un elemento complesso esprimendo sempre le stesso valore può essere tradotto in ODL_{I^3} per mezzo di una costante *const*.

```
<xsd:complexType name="OrdineType">
    < xsd:element name ="Indirizzo" type="IndirizzoType"/>
    ...
    < xsd: attribute name ="Data" type="xsd:date" use="fixed" value="2002-12-03"/>
</xsd:complexType>
```

```
interface OrdineType (source semistructured Ordine.xml)
{
    attribute Indirizzo Indirizzo ;
    ...
    typedef string attr_Data ;
    const attr_Data Data = '2002-12-03';
    };
```

Anche in questo caso il nome del tipo cui si riferisce la costante Data viene sfruttato per immagazzinare un'informazione aggiuntiva: la costante è stata ottenuta traducendo un attributo in un Xml-Schema.

5.5 Traduzione di elementi di riferimento ed elementi globali

All'interno di un *complexType* in uno schema Xml possiamo trovare, invece che dichiarazioni di elementi, riferimenti ad elementi globali. Ricordiamo ancora una volta che un elemento globale è innestato direttamente sotto l'elemento *schema* che contiene l'intero Xml-Schema. Si era pensato, in un primo momento, di utilizzare un costrutto differente da *attribute* per specificare in ODL_{I^3} questo concetto, come ad esempio *relationship*. Una relationship, comunque, può essere stabilita solamente fra due *interface* e, visto che un elemento globale può essere semplice, non possiamo adoperarla per questo scopo. La soluzione adottata è dunque ancora quella di tradurre i riferimenti come *attribute* nel cui nome del tipo di appartenenza viene aggiunta l'informazione del riferimento (ad esempio con ref_NomeElemento). All'interno dell'interfaccia in cui si trova il riferimento bisogna anche aggiungere una definizione di tipo *typedef* per specificare cosa in realtà è ref_NomeElemento.

Questo discorso ci obbliga anche a riflettere su come tradurre gli elementi globali: non c'è nessun problema per quelli complessi, che saranno tradotti come interfacce nella maniera già spiegata, il guaio riguarda quelli appartenenti ad un tipo semplice. Si potrebbe pensare di tradurre questi ultimi come una definizione di tipo *typedef* all'esterno di un interfaccia, ma credo sia meglio usare questo costrutto per rappresentare le nuove definizioni di tipo semplice che possono essere presenti nell'Xml-Schema. La mia idea è quella di tradurre solamente gli elementi semplici globali come attributi nell'interfaccia in cui si trova il riferimento, infatti, in un documento Xml istanza, questi elementi si trovano solamente come figli del *complexType* in cui sono definiti. Supponiamo di dover tradurre le seguenti righe di Xml-Schema:

<xsd:element name=”Ordine” type=”OrdineType”/>

<xsd:element name=”Commento” type=”xsd:string”/>

```
<xsd:complexType name="OrdineType">
    < xsd:element name ="Indirizzo" type="IndirizzoType"/>
     ...
    < xsd:element ref ="Commento" />
    < xsd: attribute name ="Data" type="xsd:date"/>
</xsd:complexType>
```

la traduzione che propongo è la seguente:

```
interface Ordine : OrdineType (source semistructured Ordine.xml
extent Ordine)
{};

interface OrdineType (source semistructured Ordine.xml)
{
    attribute Indirizzo Indirizzo ;

    typedef string ref_Commento;
    attribute ref_Commento Commento;

    typedef string attr_Data ;
    const attr_Data Data = '2002-12-03';
    };
```

In tale traduzione l'elemento globale Commento non possiede una trascrizione a se stante ma viene tradotto tramite le due righe del codice ODL_{I^3}:

```
typedef string ref_Commento; attribute ref_Commento Commento;
```

in cui il nome del nuovo tipo definito ref_Commento mantiene l'informazione del riferimento ed è definito appartenere al tipo semplice dell'elemento Commento.

5.6 Ricorrenza degli elementi nei tipi complessi

Tramite gli attributi *minOccurs* e *maxOccurs* è possibile specificare, in Xml-Schema, il numero minimo e massimo di volte in cui un elemento può comparire in un'istanza.

Questo concetto non sempre è perfettamente traducibile in ODL_{I^3}. Una corrispondenza perfetta è possibile solamente in due casi:

- Quando è presente il solo minOccurs ed ha il valore zero; *minOccurs='0'*
- Quando sono presenti sia l'attributo minOccurs con il valore zero che l'attributo maxOccurs con il valore unbounded (infinito); *minOccurs='0' maxOccurs='unbounded'* (si ricorda che il valore di default per minOccurs e maxOccurs è pari a uno)

Nel primo caso (*minOccurs='0'*) si può tradurre l'elemento come un normale *attribute* postponendo ad esso il simbolo asterisco che ne indica l'opzionalità. Nel secondo caso, invece, (*minOccurs='0' maxOccurs='unbounded'*) è possibile tradurre l'elemento tramite in tipo collezione set. In tutti i casi rimanenti (*ad esempio minOccurs='2' maxOccurs='100'*) propongo di tradurre l'elemento ancora con il tipo collezione *set* (esattamente come nel secondo caso); una trascrizione di questo tipo non conserva le esatte informazioni di occorrenza ma ODL_{I^3} non prevede un metodo per indicare che un attributo può essere presente in una classe o interfaccia un certo numero di volte variabile fra un minimo ed un massimo. Consideriamo il seguente esempio:

```
<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
    < xsd: attribute   name ="Data" type="xsd:date"/>
</xsd:complexType>
```

prevede la traduzione:

```
interface  OrdineType (source  semistructured  Ordine.xml)
{
     attribute   Indirizzo   Indirizzo ;

     attribute   set<Oggetto> Oggetto;

    typedef  Commento ref_Commento;
```

attribute ref_Commento Commento∗;

typedef string attr_Data ;
attribute attr_Data Data∗;
};

MinOccurs='0' è un attributo di un elemento riferito, così la traduzione che ne cosegue prevede di mantenere anche questa informazione. Come si può notare, inoltre, l'elemento oggetto è stato tradotto come se il numero minimo di occorrenze fosse pari a zero. La stessa traduzione si applica dunque anche nel caso di :

< xsd:element name =”Oggetto” type=”OggettoType” minOccurs=”1”
maxOccurs=”5”/>

tradotto sempre con:

attribute set<Oggetto> Oggetto;

5.7 Traduzione dei tipi semplici derivati

Un nuovo tipo semplice viene definito in Xml-Schema nella seguente maniera:

<xsd:simpleType name=”Nome” base=”TipoBase”>
...
</xsd:simpleType>

in cui TipoBase rappresenta un tipo semplice definito direttamente all'interno dello stesso linguaggio o tramite un'ulteriore definizione di tipo semplice. Lo stesso concetto può essere espresso in ODL_{I^3} tramite l'uso di *typedef* con cui è possibile definire, appunto, nuovi tipi semplici.

typedef TipoBase Nome;

Sarà presentata ora in maniera particolareggiate la possibile traduzione per ogni particolare tipo semplice derivato di Xml-Schema.

5.7.1 Traduzione delle liste

Xml-Schema permette di definire liste come particolari tipi di dato semplice derivato. La traduzione è piuttosto semplice in questo caso poich anche ODL_{I^3} fornisce un tipo semplice e che rappresenta una collezione di elementi chiamato *list*. Si consideri il seguente esempio di traduzione:

```
<xsd:simpleType name="ListaDiNumeri">
    <xsd: list  itemType="xsd: integer"/>
</xsd:simpleType>

typedef  list <int> ListaDiNumeri;
```

In pratica l'elemento ListaDiNumeri definito in Xml-Schema come una lista di valori interi viene tradotto in ODL_{I^3} tramite una nuova definizione di tipo *list*, il cui valore degli elementi componenti deve essere *int* (il rispettivo ODL_{I^3} del tipo integer in Xml-Schema).

Una considerazione particolare merita il seguente esempio:

```
<xsd:simpleType name="ListaDiNumeri">
    <xsd: list  itemType="xsd: integer"/>
</xsd:simpleType>

<xsd:simpleType name="ListaDi6Numeri">
    <xsd: restriction
        base="ListaDiNumeri">
        <xsd:length  value="6"/>
    </xsd: restriction >
</xsd:simpleType>
```

In un documento Xml istanza l'elemento ListaDi6Numeri presenterà al suo interno esattamente sei valori interi, concetto che può essere ben rappresentato in ODL_{I^3} tramite la seguente traduzione:

```
typedef  list <int> ListaDiNumeri;

typedef  ListaDiNumeri ListaDi6Numeri[6];
```

Mentre si effettua la traduzione da Xml-Schema ad ODL_{I^3} è necessario, nel caso si ritrovi la definizione di un nuovo tipo semplice derivato per restrizione e a cui sia applicato il *facet length*, controllare il tipo semplice padre; nel caso il tipo padre (ottenuto dall'attributo *base* dell'elemento *restriction*) sia una lista, la traduzione ODL_{I^3} risulterà in un vettore(*list*) di k elementi dove k è il valore dell'attributo length.

5.7.2 Traduzione di tipi unione

Per tradurre un elemento di Xml-Schema il cui tipo sia formato da un'unione di diversi tipi di dato, è stato previsto in ODL_{I^3} il costrutto *union* che svolge la medesima funzione.

Ad esempio:

```
<xsd:simpleType name="NumeroType">
    <xsd:union memberTypes="xsd:string␣xsd: positiveInteger "/>
</xsd:simpleType>
```

si dovrebbe tradurre, utilizzando l'UnionType

```
union NumeroType switch(int) {
    case 1: string prova;
    case 2: int prova;
};
```

5.7.3 Traduzione dei tipi derivati per restrizione

Vi sono moltissimi modi per creare un tipo di dato semplice con il metodo della derivazione per restrizione. Xml-Schema, infatti, prevede moltissimi *facets* da poter utilizzare a questo proposito. Sfortunatamente solo pochi di essi sono traducibili correttamente nel linguaggio ODL_{I^3}, ed in particolare il facet *enumeration* (applicabile a qualsiasi tipo semplice definito in un Xml-Schema), ed i facet *minInclusive*, *maxInclusive* , *minExclusive* e *maxExclusive* (applicabili ai tipi semplici ordinati di Xml-Schema). Si consideri un esempio de enumerazione:

```
<xsd:simpleType name=”StatoType”>
    <xsd: restriction  base=” string ”>
        <xsd:enumeration value=” Italia ”/>
        <xsd:enumeration value=”USA”/>
    </xsd: restriction >
</xsd:simpleType>
```

La traduzione in ODL_{I^3} di un tipo semplice ottenuto per enumerazione verrà eseguita tramite l'impiego del costrutto *enum* all'interno de quale saranno elencati i vari valori assumibili:

```
enum StatoType { ’ Italia ’ , ’USA’};
```

In pratica l'unica reale differenza fra l'*enum* ODL_{I^3} ed il facet enumerazione di Xml-Schema è che nel primo caso non è possibile esprimere a quale tipo semplice appartengano i valori elencati (informazione che deve essere desunta dagli stessi valori dell'enumerazione). Considerando ora i facets *minInclusive*, *maxInclusive* , *minExclusive* e *maxExclusive* notiamo che essi sono applicabili, in Xml-Schema, a qualsiasi tipo di dato semplice e ordinato (sia, ad esempio, al tipo *integer*, che al tipo *float*). Esiste in ODL_{I^3} un tipo semplice che esprime il concetto di un intervallo di valori, ed è il tipo *range*. Purtroppo questo tipo di dato permette solamente la definizione di un intervallo di valori interi e quindi può essere utilizzato per la traduzione dei facets di intervallo, solamente se il tipo di dato nell'attributo *base* dell'elemento *restriction* è traducibile in ODL_{I^3} con *int*. L'esempio successivo definisce un nuovo simpleType chiamato NumeroType:

```
<xsd:simpleType name=”NumeroType”>
    <xsd: restriction  base=”xsd: integer ”>
        <xsd:minInclusive  value=”10”/>
        <xsd:maxInclusive value=”999”/>
    </xsd: restriction >
</xsd:simpleType>
```

La traduzione proposta è la seguente:

```
typedef  range  10,999  NumeroType
```

Se, al posto dei facets *minInclusive* e *maxInclusive* si usano *minExclusive* e *maxExclusive* la traduzione risulta leggermente differente (bisogna considerare esclusi gli estremi dell'intervallo definito):

```
<xsd:simpleType name="NumeroType">
    <xsd: restriction   base="xsd: integer ">
        <xsd:minExclusive value="10"/>
        <xsd:maxExclusive value="999"/>
    </xsd: restriction >
</xsd:simpleType>
```

viene infatti tradotto con:

```
typedef range 11,998 NumeroType
```

Nel caso sia definito uno solo degli estremi dell'intervallo si ricorre in ODL_{I^3} ad i valori *+INF* e *-INF* (rappresentanti rispettivamente il concetto di infinito positivo e negativo) che sono compatibili con il tipo *range*. Consideriamo un esempio nel quale sia presente solamente l'elemento *minInclusive*:

```
<xsd:simpleType name="NumeroType">
    <xsd: restriction   base="xsd: integer ">
        <xsd:minInclusive  value="10"/>
    </xsd: restriction >
</xsd:simpleType>
```

La sua trascrizione in ODL_{I^3} risulterà:

```
typedef range 10,+INF NumeroType
```

Si consideri ora il caso in cui si presenti nel documento Xml-Schema una definizione di tipo semplice per restrizione utilizzante un facet differente da quelli trattati in precedenza. Ad esempio:

```
<xsd:simpleType name="StringaType">
    <xsd: restriction  base="xsd: string ">
        <xsd:pattrn  value="\d{3}-[A-Z]{2}"/>
    </xsd: restriction >
</xsd:simpleType>
```

Il valore dell'elemento StringaType, derivato da string, deve essere un insieme di caratteri aventi una certa struttura: prima devono essere presenti tre numeri decimali, poi due caratteri compresi fra A e Z. Il linguaggio ODL_{I^3} non permette assolutamente di mantenere un informazione di questo tipo, per cui ritengo che l'unica traduzione possibile sia utilizzare *typedef* al fine di indicare che in tipo StringaType deriva dal tipo *string*.

```
typedef  string  StringaType;
```

Ricapitolando, una derivazione ottenuta tramite l'elemento *enumeration* viene tradotta in ODL_{I^3} per mezzo di una definizione di tipo *enum* seguita (fra parentesi graffe) dell'elenco di termini dell'enumerazione; una restrizione presentante i facets *minInclusive*, *maxInclusive*, *minExclusive* o *maxExclusive* viene tradotta con una nuova definizione typedef range in cui il valore dell'intervallo è dato dagli attributi *value* presente in questi elementi; tutti gli altri casi di derivazione per restrizione di tipo semplice vengono tradotti in ODL_{I^3} con *typedef* seguito dal nome del tipo padre, contenuto nell'attributo base dell'elemento *restriction* in questione, e dal nome del nuovo tipo semplice derivato.

5.8 Traduzione del tipo semplice anyType

Il tipo anyType rappresenta la base da cui derivano sia gli elementi semplici che quelli complessi, il suo contenuto, inoltre, non è vincolato in alcun modo. ODL_{I^3} permette di create attributi il cui contenuto non è vincolato (può essere quindi qualunque cosa) tramite il tipo *any*. Il tipo *any* di ODL_{I^3} può essere considerato l'esatto corrispettivo di anyType per Xml-Schema. Un elemento anyType può essere definito in due modi:

```
<xsd:element name="QualunqueCosa" type="anyType"/>
```

oppure

<xsd:element name=”QualunqueCosa”/>

per entrambi la traduzione sarà (considerando che gli elementi si trovino all'interno di un *complexType*):

```
attribute any QualunqueCosa;
```

5.9 Traduzione di definizioni di tipo anonimo

Un nuovo elemento può essere definito, in Xml-Schema, tramite una definizione anonima in cui l'elemento *complexType* o *simpleType* che ne rappresenta il tipo è contenuto al suo interno. La traduzione in ODL_{I^3} di elementi definiti in modalità anonima risulterà differente a seconda che si stia trattando un oggetto complesso o un semplice. Consideriamo per prima cosa gli elementi complessi prendendo come esempio l'oggetto Indirizzo:

```
<xsd:element name=”Indirizzo”>
    <xsd:complexType>
        < xsd:element name =”Stato” type=”xsd: string ”/>
        < xsd:element name =”Città” type=”xsd: string ”/>
        < xsd:element name =”Via” type=”xsd: string ”/>
        < xsd:element name=”Numero” type=”xsd:int”/>
    </xsd:complexType>
</xsd:element>
```

Nei casi in cui il *complexType* risulta diviso dall'*element*, ognuno dei due viene tradotto come una *interface*. L'interfaccia che rappresenta l'elemento complesso eredita da quella rappresentante il tipo complesso (e che contiene la dichiarazione degli *attribute*). Nel caso anonimo, invece, non si necessitano di due interfacce, è sufficiente crearne una, chiamata come l'elemento da tradurre, e contenente gli attributi posseduti dall'elemento stesso.

```
interface Indirizzo ( source semistructured Ordine.xml extent
Indirizzo )
{
```

```
    attribute   string  Stato ;

    attribute   string  Città ;

    attribute   string  Via;

    attribute   int  Numero;
   };
```

Consideriamo ora definizioni di tipo anonimo per elementi semplici. Come esempio viene presentato un complexType, di nome IndirizzoType, al cui interno si trova l'elemento Numero definito, appunto, in modo anonimo:

```
<xsd:complexType name="IndirizzoType">
    < xsd:element  name ="Stato"  type="␣xsd: string "/>
    < xsd:element  name ="Città"  type="xsd: string "/>
    < xsd:element  name ="Via" type="xsd: string "/>
    < xsd:element  name="Numero">
        < xsd:simpleType>
            <xsd: restriction   base="xsd: integer ">
                <xsd:minExclusive value="0"/>
                <xsd:maxExclusive value="1001"/>
            </xsd: restriction >
        </ xsd:simpleType>
    </ xsd:element>
</xsd:complexType>
```

In questo caso Numero non può essere tradotto come un'interfaccia (si tratta infatti di un tipo semplice) e sarà rappresentato come un'*attribute* all'interno di IndirizzoType. Per mantenere l'informazione di definizione anonima si potrebbe creare (con il metodo già usato per differenziare la traduzione degli elementi semplici dagli attributi e dai riferimenti) un nome di tipo apposito; ad esempio elemdef_Numero (in cui elemdef significa definizione anonima di elemento). La traduzione risulta quindi:

```
interface   IndirizzoType  ( source   semistructured   Ordine.xml)
 {
    attribute   string  Stato ;
```

```
attribute string Città ;

attribute string Via;

typedef range 1,1000 elemdef_Numero;
attribute elemdef_Numero Numero;
};
```

5.10 Traduzione di elementi a contenuto misto

La traduzione il ODL_{I^3} di questo genere di elementi è problematica. ODL_{I^3} non contempla, infatti, la possibilità di definire oggetti complessi (come le *interface*) aventi anche un contenuto semplice. Data la mancanza di una qualsiasi corrispondenza le traduzione possibili sono numerose, ma tutte con qualche difetto. In questo paragrafo se ne presentano tre. Utilizziamo, al fine di presentare queste possibili traduzione, l'elemento misto Lettera:

```
<xsd:element name="Lettera" type="LetteraType"/>

<xsd:complexType name="LetteraType" mixed="true">
    < xsd:element name ="Nome" type="xsd:string"/>
    < xsd:element name ="Data" type="xsd:date"/>
    < xsd:element name ="Firma" type="xsd: string "/>
</xsd:complexType>
```

La prima traduzione proposta è la seguente:

```
interface Lettera : LetteraType ( source semistructured
Ordine.xml){};
```

oppure

```
interface LetteraType ( source semistructured Ordine.xml extent
Lettera )
```

```
{
    attribute   string  Mixed_Text_1;
    attribute   string  Nome;
    attribute   string  Mixed_Text_2;
    attribute   string  Data;
    attribute   string  Mixed_Text_3;
    attribute   string  Firma;
    attribute   string  Mixed_Text_4;
   };
```

In questo caso sono stati aggiunti svariati attributi all'interno dell'interfaccia che traduce il *complexType* LetteraType. Lo scopo è quello di rappresentare, con un *attribute* di tipo *string*, ogni possibile testo che compare nell'elemento (è necessario inserire un'attribute prima e dopo ogni elemento contenuto nel tipo complesso). Il nome assegnato a questi attributi aggiuntivi non è importante se non per identificare la loro provenienza da un tipo misto. A mio parere questa soluzione ha il difetto di alterare notevolmente la struttura dell'oggetto che traduce il tipo misto; vengono inseriti, infatti, molti elementi inesistenti e che non verranno mai interrogati direttamente in una query sul documento Xml. Una seconda soluzione potrebbe essere la seguente:

```
interface  LetteraType ( source  semistructured  Ordine.xml extent
Lettera )
{
   const  Mixed_Text booean = TRUE;

    attribute   string  Nome;

    attribute   string  Data;

    attribute   string  Firma;
   };
```

in cui la costante booleana Mixed_Text ha, evidentemente, il solo scopo di indicare che il tipo di dato in considerazione è stato ottenuto a partire da un *complexType* misto. Ritengo, comunque, che una soluzione più corretta si possa ottenere definendo

un'interfaccia priva di elementi ed il cui nome esprima il concetto di tipo misto, da cui possa derivare l'interfaccia LetteraType. Un'implementazione potrebbe essere:

```
interface Mixed_Text_Type{};

interface LetteraType:Mixed_Text_Type (source semistructured
Ordine.xml extent Lettera )
 {
    attribute string Nome;

    attribute string Data;

    attribute string Firma;
   };
```

L'interfaccia Mixed_Text_Type non presenta il parametro *source*, che ne indica la provenienza da un testo semistrutturato, poichè non è possibile ritrovarla nell'Xml-Schema; il suo unico ruolo è quello di indicare che che le interfacce derivanti da essa sono state ottenute a partire da un elemento a contenuto misto. L'interfaccia Mixed_Text_Type può essere riutilizzata per ogni altro eventuale elemento misto. Il nome dell'interfaccia aggiuntiva (Mixed_Text_Type) deve distinguersi dai possibili nomi degli elementi tradotti. Utilizzando questo metodo si può pensare di tradurre in modo efficiente le query poste in ODL sulle interfacce derivanti da Mixed_Text_Type.

5.11 Traduzione degli elementi a contenuto vuoto

La traduzione di elementi a contenuto vuoto (cioè aventi solamente attributi e non sottoelementi) è, basandosi sulla trascrizioni da Xml-Schema a ODL_{I^3} svolte finora, praticamenta automatica: non occorre fare altro che tradurre normalmente l'elemento a contenuto vuoto con una *interface*, e tradurre all'interno di essa (o dell'*interface* rappresentante il suo *compleType*) gli attributi con il metodo precedentemente esposto. Si consideri come esempio la seguente definizione anonima di un elemento vuoto Indirizzo:

```
<xsd:element name="Indirizzo"/>
    <xsd:complexType >
```

```
        < xsd: attribute   name =”Stato”  type=”xsd: string ”/>
        < xsd: attribute   name =”Città”  type=”xsd: string ”/>
        < xsd: attribute   name =”Via” type=”xsd: string ”/>
        < xsd: attribute   name=”Numero” type=”xsd:int”/>
      </xsd:complexType>
<xsd:element/>
```

rispettando le regole di traduzione usata finora abbiamo:

```
interface  Indirizzo  ( source  semistructured  Ordine.xml extent
Indirizzo )
 {
     typedef  string   attr_Stato  ;
      attribute    attr_Stato   Stato ;

     typedef  string   attr_Città  ;
      attribute    attr_Città   Città ;

     typedef  string   attr_Via ;
      attribute    attr_Via  Via;

     typedef  int  attr_Numero;
      attribute   attr_Numero  Numero;
     };
```

In pratica è possibile comprendere che l'*interface* Indirizzo è stata ottenuta a partire da un elemento a contenuto vuoto dal fatto che contiene solamente la traduzione di attributi (e non di elementi).

5.12 Traduzione dei namespaces

Xml-Schema offre la possibilità di gestire i namespaces; in particolar modo permette di definire un target namespace per indicare quali documenti Xml è possibile validare tramite lo schema stesso. Inoltre, tramite l'utilizzo di prefissi anteposti ai nomi degli elementi, Xml-Schema permette di indicare a quale namespace appartenga una nuova definizione di tipo. L'utilizzo di tali prefissi non è obbligatorio e, nel caso si decida

di non farne uso, il W3C raccomanda di anteporre comunque agli elementi definiti nel namespace proprio di Xml-Schema il prefisso xsd, in modo da distinguerli da quelli definiti dall'utente. ODL$_{I^3}$ non permette l'utilizzo dei namespace ed è quindi impossibile la traduzione sia dell'indicazione del target namespace che dai prefissi. Per quanto riguarda questi ultimi, si potrebbe pensare di tradurli anteponendo anche nella trascrizione in ODL$_{I^3}$ il prefisso al nome dell'elemento o dell'attributo; questa soluzione ritengo rappresenti un errore poichè cambierebbe il nome degli oggetti renderebbe più difficoltoso (se non impossibile) un procedimento atto a trovare somiglianze fra le varie classi basato sui nomi stessi delle classi e degli elementi contenuti. In conclusione ritengo che nella traduzione verso l'ODL$_{I^3}$ debbano essere eliminati tutti i riferimenti ai namespaces.

5.13 Traduzione dei gruppi Sequence, Choice ed All

Verranno analizzate nei prossimi paragrafi possibili traduzioni dei gruppi di elementi di tipo *sequence*, *choice* ed *all*.

5.13.1 Traduzione del gruppo Sequence

Il primo gruppo ad essere analizzato è il gruppo *sequence*. Si ricorda che tale insieme permette di aggiungere la seguente informazione: gli elementi contenuti in esso devono apparire in una istanza esattamente nell'ordine in cui sono dichiarati nell'Xml-Schema. Anche in questo caso ODL$_{I^3}$ non mette a disposizione una struttura che possa fornire la suddetta informazione: tale linguaggio descrittivo, infatti, non prevede di definire un ordine per gli elementi che compaiono all'interno di un'*interface*. Vista la situazione si può pensare di non tradurre in alcun caso il gruppo sequence. Una mia proposta è quella di mantenere l'informazione di sequenza nel caso in cui tutti gli elementi di un *complexType* siano racchiusi nell'elemento *sequence*: con un meccanismo simile a quello usato per gli elementi a contenuto misto, si potrebbe crare un'*interface* fittizia e vuota cui far derivare le interfacce che racchiudono solo elementi sequenziali. Il nome di questa interfaccia fittizia potrebbe essere per esempio Group_Sequence. Vediamone il funzionamento:

```
<xsd:complexType name="IndirizzoType">
    < xsd: sequence>
        < xsd:element name ="Stato" type="StatoType"/>
```

```
        < xsd:element name =”Città” type=”xsd: string ”/>
        < xsd:element name =”Via” type=”xsd: string ”/>
        < xsd:element name=”Numero” type=”xsd:int”/>
    </xsd: sequence>
</xsd:complexType>
```

sarebbe tradotto con:

```
interface Group_Sequence{};

interface IndirizzoType :Group_Sequence (source semistructured
Ordine.xml)
 {
    attribute StatoType StatoType;

    attribute string Città ;

    attribute string Via;

    attribute int Numero;
   };
```

è chiaro come non si possa tradurre in questo modo un tipo complesso in cui non tutti gli elementi siano contenuti in *sequence*.

5.13.2 Traduzione del gruppo Choice

All'interno di un gruppo choice solamente un elemento fra quelli presenti può comparire in un documento istanza. Questo meccanismo è tipico dei documenti semistrutturati, in cui un oggetto può anche non avere una struttura rigida ed immutabile. Il gruppo choice può essere tradotto in ODL_{I^3} tramite l'utilizzo del termine *union* (aggiunto all'ODL proprio per poter rappresentare dati semistrutturati). Union serve per poter ottenere interfacce con due o più tipi di contenuto (o corpo). In ogni corpo appartenente all'interfaccia traduzione del *complexType* contenente un elemento *choice* deve apparire:

- la trasposizione ODL_{I^3} di tutti gli elementi al di fuori del gruppo *choice*

- uno solo fra gli elementi contenuti in *choice* (deve esserci un elemento differente per ogni corpo).

Si consideri il seguente esempio di indirizzo:

```
<xsd:element name="Indirizzo" type="IndirizzoType"/>

  <xsd:complexType name="IndirizzoType">
    < xsd: sequence>
        < xsd:element name ="Stato" type="StatoType"/>
        <xsd: choice>
            < xsd:element name ="Città" type="xsd: string "/>
            < xsd:element name ="CAP" type="xsd: positiveInteger"/>
        </xsd: choice>
        < xsd:element name ="Via" type="xsd: string "/>
        < xsd:element name="Numero" type="xsd:int"/>
    </xsd: sequence>
  </xsd:complexType>
```

la traduzione ODL$_{I^3}$, per quanto detto prima, risultera:

```
interface Indirizzo : IndirizzoType (source semistructured
Ordine.xml extent Indirizzo )
{};

interface IndirizzoType (source semistructured Ordine.xml)
{
    attribute StatoType StatoType;

    attribute string Città;

    attribute string Via;

    attribute int Numero;
} union {

attribute StatoType StatoType;
```

```
    attribute   int  CAP;

    attribute   string  Via;

    attribute   int  Numero;
};
```

5.13.3 Traduzione del gruppo All

Il gruppo all permette agli elementi contenuti dentro di se di apparire in qualsiasi ordine all'interno di un documento Xml. Ogni entità che si trova all'interno di un gruppo *all* deve, inoltre, essere un elemento singolo (non rappresentare, quindi, un gruppo di elementi) avente come unici valori permessi di *minOccurs* e *maxOccurs* uno e zero. Anche in questo caso, dato che ODL_{I^3} non considera l'ordine degli elementi all'interno di una *interface*, è possibile decidere di non tradurre affatto l'elemento all di Xml-Schema. Vale comunque lo stesso discorso affrontato per il gruppo *sequence*: nel caso in cui tutti gli elementi di un tipo complesso siano contenuti in un gruppo all, propongo di mantenere questa informazione per mezzo di una interfaccia ausiliaria (di nome, per esempio, Group_All). L'unico utilizzo dell'interfaccia suddetta sarà quello di fornire, tramite il suo nome, un'informazione aggiuntiva sul processo di traduzione svolto. Tutti i tipi complessi al cui interno il complesso degli elementi è completamente racchiuso in un gruppo all potranno derivare da questa classe aggiuntiva.

```
<xsd:complexType name="IndirizzoType">
    < xsd: all >
        < xsd:element name ="Stato" type="␣xsd: string "/>
        < xsd:element name ="Città" type="xsd: string "/>
        < xsd:element name ="Via" type="xsd: string "/>
        < xsd:element name="Numero" type="xsd:int"/>
    </xsd: all >
</xsd:complexType>
```

potrebbe essere tradotto tramite:

```
interface Group_All{};

interface IndirizzoType :Group_All (source semistructured
Ordine.xml)
{
    attribute string Stato;

    attribute string Città;

    attribute string Via;

    attribute int Numero;
   };
```

5.14 Traduzione di gruppi di elementi e gruppi di attributi

Per la traduzione di gruppi di elementi e gruppi di attributi si è pensato, in un primo momento, di reificare queste strutture; l'idea era quella di promuovere i *group* e gli *attributeGroup*, tradotti in ODL$_{I^3}$, a livello di interfaccia, quindi a livello di oggetto complesso. Una traduzione di questo tipo presenta un problema fondamentale, da tenere in considerazione: reificando un gruppo di elementi in un'interfaccia si obbliga l'utente a formulare una query OQL in cui compare il nome del suddetto gruppo. Trasponendo, successivamente, la query in un linguaggio di interrogazione per documenti Xml al fine di reperire i dati voluti, il nome del gruppo non deve essere presente: i gruppi di elementi e di attributi si trovano, infatti, solamente negli Xml-Schema (per renderne più leggibile la struttura) e non nelle istanze Xml. Traducendo, inoltre, i gruppi di elementi e di attributo come interfacce si renderebbe più difficoltoso il riconoscimento di affinità fra classi di differenti sorgenti: la struttura di un'interfaccia risulterebbe notevolmente modificata. La mia proposta è, quindi, di tradurre questi concetti riportando direttamente gli elementi o gli attributi situati dentro di essi nell'interfaccia contenente il riferimento al gruppo (sfruttando la normale traduzione vista per elementi ed attributi).

5.14.1 Traduzione di gruppi di elementi

Si consideri, come esempio di Xml-Schema utilizzante un gruppo di elementi, il seguente:

```
<xsd:element name="Indirizzo" type="IndirizzoType"/>

<xsd:complexType name="IndirizzoType">
    < xsd:sequence>
        < xsd:element name ="Stato" type="␣xsd: string "/>
        <xsd: choice>
            < xsd:element name ="Città" type="xsd: string "/>
            <xsd:group ref="GruppoCittà"/>
        </xsd: choice>
        < xsd:element name ="Via" type="xsd: string "/>
        < xsd:element name="Numero" type="NumeroType"/>
    </ xsd:sequence>
</xsd:complexType>

<xsd: group name="GruppoCittà">
    <xsd: sequence>
        < xsd:element name ="Città" type="xsd: string "/>
        < xsd:element name ="CAP" type="xsd:positiveInteger"/>
    </xsd: sequence>
</xsd: group>
```

Traducendo direttamente gli elementi situati in GruppoCittà all'interno di IndirizzoType (cioè sostituendo il riferimento al gruppo con la struttura del gruppo) risulta:

```
interface Indirizzo : IndirizzoType (source semistructured
Ordine.xml extent Indirizzo )
{};

interface IndirizzoType (source semistructured Ordine.xml)
{
    attribute string Stato;

    attribute string Città;
```

```
    attribute   string  Via;

    attribute   int  Numero;
} union {

 attribute   string  Stato ;

    attribute   string  Città ;

    attribute   int  CAP;

    attribute   string  Via;

    attribute   int  Numero;
};
```

Purtroppo con questa traduzione si perde l'informazione sull'esistenza del gruppo di elementi, ma ritengo che, dato lo scopo della trasposizione, sia più utile un procedimento di questo tipo piuttosto di uno che impieghi la reificazione in una interfaccia.

5.14.2 Traduzione di gruppi di attributi

Con lo stesso processo di traduzione applicato al gruppo di elementi è stato tradotto il seguente gruppo di attributi:

```
<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
    < xsd: attributeGroup   ref  =" AttributiOrdine " />
</xsd:complexType>

<xsd:  attributeGroup   name="AttributiOrdine">
    < xsd:  attribute   name ="Data" type="xsd:date"  use="required"/>
    < xsd:  attribute   name ="TipoConsegna" >
```

```
        <xsd: simpleType>
            <xsd: restriction  base="xsd: string ">
                <enumeration value="Treno"/>
                <enumeration value="Nave"/>
                <enumeration value="Aereo"/>
                <enumeration value="Strada"/>
            </xsd: restriction >
        </xsd: simpleType>
</xsd: attributeGroup >
```

traduzione in ODL_{I^3} proposta:

```
interface  OrdineType (source  semistructured  Ordine.xml)
{
    attribute   Indirizzo   Indirizzo ;

    attribute   set<Oggetto> Oggetto;

    typedef  string  ref_Commento;
    attribute   ref_Commento Commento*;

    typedef  string   attr_Data ;
    attribute   attr_Data  Data;

    enum attrdef_TipoConsegna{'Treno', 'Nave', 'Aereo', 'Strada'};
    attribute   attrdef_TipoConsegna  Data*;

    };
```

anche in questo caso si perde l'informazione dell'esistenza del gruppo AttributiOrdine; la struttura dell'interfaccia OrdineType risulta comunque corretta.

5.15 Traduzione dei concetti di ereditarietà

Xml-Schema prevede due differenti modalità di ereditarietà: l'ereditarietà ottenuta tramite l'estensione di contenuto e quella ottenuta tramite la restrizione di contenuto. I precedenti concetti verranno trattati separatamente nei seguenti sottocapitoli.

5.15.1 Derivazione per estensione

Il processo di traduzione del meccanismo di ereditarietà per estensione di differenzia in base al tipo di contenuto esteso. Xml-Schema permette infatti di ottenere nuovi tipi complessi a partire sia da precedenti tipi complessi che semplici. Per quanto riguarda il primo fra questi due casi, la trascrizione ODL_{I^3} non presenta particolari difficoltà: l'ereditarietà fra interfacce è infatti possibile. Quando in Xml-Schema un tipo complesso detto A è definito per estensione a partire da un altro tipo complesso detto B, la traduzione ODL_{I^3} sarà la seguente: il tipo complesso A esteso diverrà un'interfaccia che eredita dall'interfaccia B. Nel seguente esempio OggettoNovità eredita da OggettoType:

```
<xsd:complexType name="OggettoType">
    < xsd:element name ="Nome" type="xsd:string"/>
    < xsd:element name ="Quantità" type="QuantitàType"/>
    < xsd:element name ="PrezzoEuro" type="xsd:decimal"/>
    < xsd:element ref ="Commento" minOccurs="0"/>
</xsd:complexType>

<xsd:complexType name="OggettoNovità">
    <xsd:complexContent>
        <xsd:extension base="OggettoType">
            < xsd:element name ="DataDiUscita" type="xsd:data"/>
            < xsd: attribute name ="Novità" type="xsd:boolean"
                fixed ="true"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

la traduzione sarà semplicemente:

```
interface OggettoType (source semistructured Ordine.xml)
{
    attribute string Nome;

    attribute Quantità Quantità;
```

```
    attribute   float  PrezzoEuro;

    typedef  string  ref_Commento;
    attribute  ref_Commento Commento*;
    };

interface  OggettoNovità:OggettoType (source  semistructured
Ordine.xml)
 {
    attribute   string  DataDiUscita;

    typedef  boolean  attr_Novità ;
    const  attr_Novità  Novità=TRUE;
    };
```

Un nuovo tipo complesso può essere ottenuto, in Xml-Schema, anche tramite ereditarietà per estensione a partire da un tipo semplice. Un procedimento di questo tipo serve per creare elementi dal contenuto semplice ma con l'aggiunta di attributi. ODL_{I^3} non mette a disposizione una struttura in grado di adattarsi perfettamente a questo concetto; la proposta formulata è quella di tradurre il tipo complesso derivato in questo modo con un'*interface* in cui sia il contenuto semplice che gli attributi aggiunti sono tradotti tramite *attribute*. Per chiarire che una interfaccia è stata ottenuta tramite la traduzione di un tipo semplice esteso per ereditarietà si potrebbe utilizzare, come si è già proposto di fare in altre circostanze, una interfaccia ausiliaria; il nome di questa *interface* deve fornire l'informazione voluta e le classi ottenute tramite questo procedimento devono ereditare da essa. Un nome per l'interfaccia ausiliaria potrebbe essere ad esempio: Simple_Content_Extension.

```
<xsd:complexType name="PrezzoType">
    <xsd:simpleContent>
        <xsd:extension  base="xsd:decimal">
            < xsd: attribute   name ="Tipo" type="xsd: string "
                required ="true"/>
        </xsd: extension >
    </xsd:simpleContent>
</xsd:complexType>
```

può essere tradotto nella seguente maniera:

```
interface Simple_Content_Extension{};

interface PrezzoType:Simple_Content_Extension ( source
semistructured Ordine.xml)
{
    attribute int Element_Content;

   typedef string attr_Tipo ;
    attribute attr_Tipo Tipo;
   };
```

Come si può notare all'interno della classe PrezzoType è presente un attributo di nome Element_Content. Questo attributo non esista nell'Xml-Schema ed è in realtà la rappresentazione del contenuto semplice del tipo di dato. Il nome di questo attributo non è fondamentale (può essere scelto a piacere) ma deve essere il medesimo per tutte le interfacce derivanti da Simple_Content_Extension.

Dato che sia il contenuto del tipo di dato che gli attributi devono essere oggetti semplici, un tipo complesso derivante per estensione da un tipo semplice, potrebbe essere tradotto in ODL_{I^3} anche con il costrutto *struct*. Questo tipo di struttura ODL_{I^3}, infatti, non permette che i propri elementi siano complessi e si potrebbe adattare al nostro scopo; inoltre non essendo stata, al contrario dell'*interface*, utilizzata allo fine di tradurre altri concetti, sarebbe chiaro il modo in cui è stata ottenuta (favorendo la bidirezionalità della traduzione). Io tipo PrezzoType potrebbe quindi essere rappresentato in ODL_{I^3} nella maniera seguente:

```
struct PrezzoType {
    int Struct_Content ;

    string Tipo;};
```

5.15.2 Derivazione per restrizione

Al contrario di quanto accade per il metodo di derivazione (o ereditarietà) per estensione di contenuto, ODL_{I^3} non implementa il metodo di derivazione per restrizione

di contenuto. Dato che tutti gli attributi e gli elementi di un tipo complesso derivato per restrizione devono essere dichiarati nuovamente (al contrario di quello che accade tramite l'espansione, dove si dichiarano solamente gli attributi e gli elementi aggiuntivi), una buona soluzione è quella di tradurre il *complexType* ristretto come una nuova *interface*. All'interno di tale interfaccia si devono normalmente inserire tutti gli *attribute* e le costanti derivate dalla traduzione degli elementi contenuti nel tipo complesso. Si consideri il seguente esempio in cui OrdineStock deriva per restrizione di contenuto da OrdineType:

```
<xsd:complexType name="OrdineType">
    < xsd:element name ="Indirizzo" type="IndirizzoType"/>
    < xsd:element name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element ref ="Commento" minOccurs="0" />
    < xsd: attribute name ="Data" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="OrdineStock">
    <xsd:complexContent>
        <xsd: restriction base="OrdineType">
            < xsd:element name ="Indirizzo" type="IndirizzoType"/>
            < xsd:element name ="Oggetto" type="OggettoType"
                minOccurs="50" maxOccurs="100"/>
            < xsd: attribute name ="Data" type="xsd:date"/>
        </xsd: restriction >
    </xsd:complexContent>
</xsd:complexType>
```

la traduzione ODL_{I^3} fornita risulta essere:

```
interface OrdineType (source semistructured Ordine.xml)
{
    attribute Indirizzo Indirizzo ;

    attribute set<Oggetto> Oggetto;

    typedef string ref_Commento;
```

```
    attribute  ref_Commento Commento*;

   typedef  string  attr_Data ;
    attribute  attr_Data  Data;
   };

interface  OrdineStock (source  semistructured  Ordine.xml)
{
    attribute  Indirizzo  Indirizzo ;

    attribute  set<Oggetto> Oggetto;

   typedef  string  attr_Data ;
    attribute  attr_Data  Data;
   };
```

Con una traduzione di questo tipo l'interfaccia OrdineStock risulta essere correttamente trasposta: contiene, infatti, al suo interno tutti gli elementi presenti nel tipo complesso di Xml-Schema omonimo. Si perde, comunque, una informazione: l'ereditarietà fra le due classi. Considerando lo scopo della traduzione nell'ambito del progetto MOMIS la suddetta perdita di informazione è trascurabile (ogni interfaccia contiene tutte le informazione necessarie per essere associata ad altre o per essere interrogata); non si ottiene però una traduzione completamente reversibile. Un'idea potrebbe essere quella di far derivare OrdineStock da un'interfaccia ausiliaria contenente nel suo nome l'informazione di derivazione per restrizione dalla classe OrdineType. Tale interfaccia deve essere logicamente priva di contenuto ed un esempio del suo nome potrebbe essere: Restriction_Content_OrdineType.

5.16 Traduzione dei substitution groups

Con il metodo dei substitution groups è possibile, in Xml-Schema, definire delle classi di sostituzione per alcuni elementi. In una istanza Xml un elemento può essere rimpiazzato da un qualunque altro appartenente al suo substitution group. Anche in questo caso ODL_{I^3} non fornisce meccanismi simili ad Xml-Schema. E' perciò necessario effettuare la traduzione tramite qualche artificio. Le trasposizioni proposte in seguito sono due: la prima prevede di elencare come *attribute* opzionali tutti gli

elementi coinvolti nel gruppo di sostituzione; la seconda , invece, fa uso di un termine *union* al fine di creare un corpo di interfaccia per ogni elemento del gruppo. In entrambi i casi l'appartenenza degli elementi ad un substitution group viene evidenziata tramite un particolare tipo di dato; nel nome di questo tipo (definito tramite un *typedef*) viene inserita l'informazione riguardante il nome del gruppo di sostituzione. Consideriamo, all'inizio, la prima delle due trasposizioni:

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

<xsd:element name="MioCommento" type="xsd:string"
substitutionGroup ="Commento"/>

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
    < xsd: attribute   name ="Data" type="xsd:date"/>
</xsd:complexType>
```

traduzione:

```
interface  OrdineType (source  semistructured  Ordine.xml)
{
    attribute   Indirizzo   Indirizzo ;

    attribute   set<Oggetto> Oggetto;

   typedef  string  ref_Commento;
    attribute   ref_Commento Commento*;

   typedef  string  sub_Commento_MioCommento;
    attribute   sub_Commento_MioCommento MioCommento*;

   typedef  string   attr_Data ;
    attribute    attr_Data  Data;
   };
```

Nell'esempio precedente, ogni membro appartenente al gruppo di sostituzione di Commento (MioCommento e, ovviamente, Commento stesso) è stato tradotto come un *attribute* opzionale. La traduzione non è perfetta, infatti, in questo caso, entrambi gli attributi Commento e MioCommento potrebbero essere contemporaneamente presenti nella stessa istanza dell'*interface* OrdineType (nell'istanza Xml ritroviamo, invece, solamente uno di questi due elementi). L'appartenenza di MioCommento al gruppo di sostituzione di Commento è evidenziata dal particolare nome del suo tipo di dato (sub_Commento_MioCommento).

Consideriamo ora la seconda traduzione proposta (quella che prevede l'utilizzo di *union*) che, a mio avviso, è probabilmente più corretta della precedente.

```
interface OrdineType (source semistructured Ordine.xml)
{
    attribute Indirizzo Indirizzo ;

    attribute set<Oggetto> Oggetto;

    typedef string ref_Commento;
    attribute ref_Commento Commento*;

    typedef string attr_Data ;
    attribute attr_Data Data;
    }union{
    attribute Indirizzo Indirizzo ;

    attribute set<Oggetto> Oggetto;

    typedef string sub_Commento_MioCommento;
    attribute sub_Commento_MioCommento MioCommento*;

    typedef string attr_Data ;
    attribute attr_Data Data;
    };
```

Con una trascrizione di questo tipo solamente uno fra gli elementi di un substitution group può essere presente nell'istanza di una *interface*. Il problema è che allo

stesso modo viene tradotto il gruppo *choice*. Anche a questo proposito è opportuno utilizzare un particolare tipo di dato (sub_Commento_MioCommento) che mi ricordi di essere in presenza di un gruppo di sostituzione.

5.17 Traduzione dei tipi e degli elementi astratti

In Xml-Schema definire un elemento astratto significa non permettere che esso compaia direttamente in un documento istanza. Al posto di tale elemento possiamo ritrovarne altri appartenenti al suo subtitution group. Un discorso analogo vale per i tipo definiti astratti: gli elementi dello schema non possono appartenere direttamente ad un tipo astratto ma solamente ad un tipo derivato da esso. Consideriamo come prima cosa una traduzione ODL_{I^3} per gli elementi astratti: essendo questo meccanismo molto simile a quello dei gruppi di sostituzione (in realtà si sfrutta esattamente questo concetto), la traduzione sarà identica a quella già trattata nel paragrafo concernente i substitution groups. Vi è, comunque, una differenza da considerare: l'elemento astratto (che, ricordiamo, deve essere globale) non verrà riportato nella traduzione, ma compariranno solamente gli elementi del suo gruppo di sostituzione. Nel caso, come nell'esempio a seguire, sia presente solamente un elemento con cui sostituire l'oggetto astratto, esso può essere normalmente inserito nel corpo dell'interfaccia contenente l'elemento astratto (in sostituzione di questo). Un mio consiglio, per rendere la traduzione maggiormente invertibile, è mantenere l'informazione dei substitution groups nei nomi dei tipi degli *attribute* (allo stesso modo visto nel paragrafo precedente, con l'unica differenza che, in questo caso, non sarà presente la traduzione dell'elemento astratto che origina il gruppo di sostituzione). Nel seguente esempio l'elemento globale Commento è astratto:

```
<xsd:element name="Commento" type="xsd:string" abstract="true"/>

<xsd:element name="MioCommento" type="xsd:string"
 substitutionGroup ="Commento"/>

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
```

```
    < xsd: attribute   name =”Data” type=”xsd:date”/>
</xsd:complexType>
```

la traduzione proposta è:

```
interface   OrdineType (source   semistructured   Ordine.xml)
{
     attribute    Indirizzo    Indirizzo ;

     attribute    set<Oggetto> Oggetto;

    typedef   string   sub_Commento_MioCommento;
     attribute    sub_Commento_MioCommento MioCommento*;

    typedef   string    attr_Data ;
     attribute    attr_Data   Data;
    };
```

Ovviamente, nel caso fossero stati dafiniti altri elementi come sostitutivi di Commento, essi sarebbero dovuti comparire all'interno dell'*interface* OrdineType nello stesso corpo come opzionali, oppure in corpi separati (a seconda della traduzione che si vuole adottare per i substitution groups).

Prendiamo ora in considerazione un tipo astratto. Dato che nessuna informazione aggiuntiva sulla struttura del tipo (a parte il fatto che non può essere usato direttamente) viene aggiunta dall'astrazione, la proposta è quella di tradurre normalmente il tipo definito astratto. Semplicemente nello schema ODL_{I^3} non ci saranno interfacce che traducono elementi complessi di un Xml-Schema derivanti dalle interfacce che traducono tipi complessi astratti. Un consiglio per esplicitare direttamente l'informazione di astrazione per tipi complessi è usare un'interfaccia aggiuntiva il cui nome sia esplicativo (si farebbe derivare l'interfaccia traduzione del tipo complesso dall'interfaccia generica che indica -tipo astratto-; metodo già sfruttato, ad esempio, per il tipo a contenuto misto). Una possibile interfaccia addizionale potrebbe essere:

```
interface   Abstract_Type{};
```

5.18 Traduzione dell'elemento Any e dell'attributo anyAttribute

L'elemento di Xml-Schema Any viene rappresentato, all'interno di un documento istanza, tramite un qualsiasi elemento appartenente al dominio indicato dall'attributo *namespace*. La traduzione più logica di questo concetto in ODL_{I^3} implica l'utilizzo del tipo semplice *any*. Si ricorda che il tipo *any* può assumere qualsiasi contenuto. Il suddetto tipo semplice è, comunque, stato impiegato anche nella traduzione di elementi dichiarati appartenenti al tipo *anyType* (come si può notare dal paragrafo apposito). Per differenziare la due traduzioni si potrebbe pensare di utilizzare un metodo già sfruttato in precedenza: immagazzinare un'informazione in un nome di tipo ODL_{I^3} utilizzando un *typedef* (ricordiamo che i nomi delle variabili non devono essere assolutamente modificati); l'informazione immagazzinata indicherà che la traduzione deriva da un elemento Xml-Schema Any. Il nome del nuovo tipo definito che verrà usato nel seguente esempio è elemdef_any (sviluppando un software di traduzione si può comunque scegliere un altro nome al posto di elemdef_any o, eventualmente, decidere di perdere la distinzione fra elementi di tipo anyType ed elementi Any trascrivendoli in ODL_{I^3} allo stesso modo).

```
<xsd:complexType Name="Esempio">
    <xsd:element name="ContenutoHTML">
        <xsd:complexType>
          <xsd:any namespace="http://www.w3.org/1999/xhtml"
            minOccurs="1" maxOccurs="unbounded"/>
        </xsd:complexType>
    </xsd:element>
</xsd:complexType>
```

La traduzione proposta è:

```
interface Esempio (source semistructured Ordine.xml)
{
    typedef set<any> elemdef_any;
     attribute elemdef_any any_element;
    };
```

Come si può notare un elemento Any non è tradotto semplicemente tramite un attributo ODL_{I^3} *any*, ma tramite un attributo di tipo elemdef_any definito *any*. Il nome dell'*attribute* è un nome arbitrario, nell'Xml-Schema, infatti, tale nome è assente. Se si desidera mentenere l'informazione del namespace che indica il dominio dal quale si possono attingere gli elementi per in documento Xml istanza (nell'esempio citato il dominio è quello dell'HTML), si può inserire tale namespace all'interno del nome del nuovo tipo ausiliario definito con *typedef* (elemdef_any). La traduzione diviene quindi:

```
interface Esempio (source semistructured Ordine.xml)
{
    typedef set<any> elemdef_any_http:// www.w3.org/1999/xhtml;
     attribute elemdef_any_http :// www.w3.org/1999/xhtml any_element;
    };
```

Consideriamo ora attributi AnyAttribute. La traduzione di tali concetti sarà molto simile a quella proposta per gli elementi Any:

```
<xsd:complexType Name="Esempio">
    <xsd:anyAttribute
        namespace="http:// www.w3.org/1999/xhtml"/>
</xsd:complexType>
```

verrà tradotto tramite:

```
interface Esempio (source semistructured Ordine.xml)
{
    typedef any attr_any ;
     attribute attr_any any_attribute ;
    };
```

L'unica differenza con la traduzione degli elementi Any è rappresentata dal nome del tipo ausiliario definito attr_any (e dal nome dell'*attribute* any_attribute, ma tale nome non ha importanza ai fini della traduzione, poich non è presente nel documento Xml-Schema). Se si desidera mantenere l'informazione sul namespace la trascrizione diviene:

```
interface Esempio (source semistructured Ordine.xml)
{
    typedef any attr_any_http :// www.w3.org/1999/xhtml;
     attribute attr_any_http :// www.w3.org/1999/xhtml any_attribute ;
    };
```

5.19 Traduzione di chiavi, riferimenti a chiave e valori unici

Xml-Schema offre la possibilità di specificare (tramite l'elemento unique) che il valore di un particolare attributo o elemento deva essere unico. Si ricorda che l'unicità di un valore viene effettuata selezionando un set di elementi (tramite l'attributo *path* dell'elemento *selector*) e specificando quale campo, all'interno di tale set, deve essere unico (tramite l'attributo *path* dell'elemento *field*). Il concetto di unicità del valore può essere tradotto in ODL_{I^3} tramite quello di chiave candidata. Tramite gli elementi *selector* e *field* della versione Xml-Schema un software di traduzione è in grado di capire a quale elemento di quale classe è applicato il concetto di unicità: tramite l'elemento selector si riesce ad identificare quale interfaccia possiede un attributo unico; tramite l'elemento field, invece, possiamo definire quale attributo, all'interno di tale interfaccia, è unico. In particolare, un elemento unique viene tradotto in ODL_{I^3} tramite in costrutto *candidate_key* all'interno dell'interfaccia selezionata da selector ed indicante l'attributo selezionato da field. Traduzioni molto simili sono applicabili anche nei casi di chiavi e chiavi candidate. L'elemento di Xml-Schema *key*, che definisce appunto un valore di chiave all'interni di un elemento complesso, funziona esattamente come l'elemento *unique*. La traduzione avverra quindi tramite un costrutto ODL_{I^3} *key* situato all'interno dell'interfaccia selezionata dall'elemento selector ed indicante l'attributo, appartenente all'interfaccia stessa, selezionato dall'elemento field. Per quanto riguarda le foreign keys (chiavi estere) espresse in Xml-Schema dell'elemento *keyref* , bisogna fare un'ulteriore considerazione: oltre ai soliti sottoelementi selector e field, *keyref* possiede anche un importante attributo :*refer*. Il suddetto attributo serve per specificare il nome della chiave (elemento key situato anch'esso nello stesso Xml-Schema) cui si riferisce la chiave estera. Nella tradizione in ODL_{I^3} (che avviene tramite il costrutto *foreign_key* situato al solito nell'interfaccia indicata dall'elemento selector) l'attributo *refer* viene tradotto tramite il nome *references* se-

guito dal nome dell'interfaccia indicata dal campo selector della chiave cui la foreign key si riferisce. Di seguito viene presentato un esempio completo di traduzione da Xml-Schema ad ODL_{I^3} in cui si trovano anche un elemento *unique*, uno *key* ed uno *keyref*.

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

<xsd:element name="OrdiniEPromozioni"
    type="OrdiniEPromozioniType"/>

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

<xsd:complexType name="OrdiniEPromozioniType">
    < xsd:element  name ="Ordine" type="OrdineType"
        minOccurs="0" maxOccurs="unbounded"/>
    < xsd:element  name ="Promozione" type="PromozioneType"
        minOccurs="0" maxOccurs="unbounded"/>
</xsd:complexType>

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType"
        minOccurs="1" maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
    < xsd: attribute   name ="Data" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="IndirizzoType">
    < xsd:element  name ="Stato"  type="xsd: string "/>
    < xsd:element  name ="Città"  type="xsd: string "/>
    < xsd:element  name ="Via" type="xsd: string "/>
    < xsd:element  name="Numero" type="xsd:int"/>
</xsd:complexType>

<xsd:complexType name="OggettoType">
    < xsd:element  name ="CodiceOggetto" type="xsd: positiveInteger "/>
    < xsd:element  name ="Nome" type="xsd:string"/>
    < xsd:element  name ="Quantità" type="QuantitàType"/>
```

```
    < xsd:element  name =”PrezzoEuro” type=”xsd:decimal”/>
    < xsd:element  name =”InPromozione” type=”xsd: string ”>
    < xsd:element  ref  =”Commento” minOccurs=”0”/>
</xsd:complexType>

<xsd:simpleType name=”QuantitàType”>
    <xsd: restriction
        base=”xsd:  positiveInteger  ”>
        <xsd:maxExclusive value=”100”/>
    </xsd:  restriction  >
</xsd:simpleType>

<xsd:complexType name=”PromozioneType”>
    < xsd:element  name =”Nome” type=”xsd:string”/>
    < xsd:element  name =”ScontoPercentuale”
        type=”xsd:  postiveInteger ”/>
    < xsd:element  name =”DataDiInizio”  type=”xsd: date”/>
    < xsd:element  name=”Durata” type=”xsd:duration”/>
</xsd:complexType>

<xsd:unique name=”NomeUnico”>
    < xsd: selector   xpath= ”/OrdiniEPromozioni/Ordine/Oggetto”/>
    < xsd: field   xpath=”CodiceOggetto”/>
<xsd:unique>

<xsd:key name=”NomePromozione”>
    < xsd: selector   xpath= ”/OrdiniEPromozioni/Promozione”/>
    < xsd: field   xpath=”Nome”/>
<xsd:key>

<xsd:keyref  name=”RifPromozione refer=”NomePromozione ”>
    < xsd: selector   xpath= ”/OrdiniEPromozioni/Ordine/Oggetto”/>
    < xsd: field   xpath=”InPromozione”/>
<xsd:keyref>

</xsd:schema>
```

la traduzione dell’intero schema è:

```
interface OrdiniEPromozioni:OrdiniEPromozioniType (source
semistructured Ordine.xml extent OrdiniEPromozioni ) {};

interface OrdiniEPromozioniType (source semistructured Ordine.xml)
{
    attribute set<Ordine> Ordine;

    attribute set<Promozione> Promozione;
   };


interface Ordine:OrdineType (source semistructured Ordine.xml
extent Ordine ) {};

interface OrdineType (source semistructured Ordine.xml)
{
    attribute Indirizzo Indirizzo ;

    attribute set<Oggetto> Oggetto;

   typedef string ref_Commento;
    attribute ref_Commento Commento*;

   typedef string attr_Data ;
    attribute attr_Data Data;
   };

interface Indirizzo : IndirizzoType ( source semistructured
Ordine.xml extent Indirizzo ) {};

interface IndirizzoType ( source semistructured Ordine.xml)
{
    attribute string Stato ;

    attribute string Città ;

    attribute string Via;
```

```
    attribute int Numero;
   };

interface Oggetto:OggettoType (source semistructured Ordine.xml
extent Oggetto candidate_key (CodiceOggetto)
foreign_key (InPromozione) references Promozione) {};

interface OggettoType (source semistructured Ordine.xml)
 {
    attribute int CodiceOggetto;

    attribute string Nome;

    attribute QuantitàType Quantità;

    attribute float PrezzoEuro;

    attribute string InPromozione;

   typedef string ref_Commento;
    attribute ref_Commento Commento*;
   };

typedef range 0,99 QuantitàType;

interface Promozione: PromozioneType (source semistructured
Ordine.xml extent Promozione key(Nome)) {};

interface PromozioneType (source semistructured Ordine.xml)
 {
    attribute string Nome;

    attribute int ScontoPercentuale;

    attribute string DataDiInizio;
```

```
    attribute   string  Durata;
};
```

Dalla traduzione precedente si nota come le chiavi, le chiavi candidate e le foreign-keys si trovano specificate nelle interfacce rappresentanti gli elementi complessi, non in quelle rappresentanti i tipi complessi (*complexType*). Il motivo è semplice: i dati che si possono reperire da uno schema sono istanze proprio delle interfacce che descrivono gli elementi, non i tipi complessi.

Tramite un Xml-Schema esiste un secondo metodo per dichiarare un elemento come chiave ed un altro come chiave estera: si ricorre agli attributi ID ed IDREF mantenuti da Xml 1.0. Consideriamo l'elemento Promozione appartenente al precedente esempio rivisto tramite questo secondo metodo; la sua descrizione in Xml-Schema risulta essere la medesima mentre varia la definizione del suo tipo complesso PromozioneType:

```
<xsd:complexType name="PromozioneType">
    < xsd:element  name ="Nome" type="xsd:ID"/>
    < xsd:element  name ="ScontoPercentuale"
        type="xsd: postiveInteger "/>
    < xsd:element  name ="DataDiInizio"  type="xsd:date"/>
    < xsd:element  name="Durata" type="xsd:duration"/>
</xsd:complexType>
```

Il sottoelemento Nome assume il tipo ID per indicare che il suo valore deve essere unico all'interno del documento. Un riferimento a chiave, invece, viene eseguito nella seguente maniera:

```
<xsd:complexType name="OggettoType">
    < xsd:element  name ="CodiceOggetto" type="xsd: positiveInteger "/>
    < xsd:element  name ="Nome" type="xsd:string"/>
    < xsd:element  name ="Quantità"  type="QuantitàType"/>
    < xsd:element  name ="PrezzoEuro" type="xsd:decimal"/>
    < xsd:element  name ="InPromozione" type="xsd:IDREF">
    < xsd:element  ref  ="Commento" minOccurs="0"/>
</xsd:complexType>
```

L'elemento che svolge il collegamento è InPromozione. Purtroppo questo metodo non consente di specificare a quale altro elemento di tipo ID ci si riferisce; non sorgono problemi se all'interno dello schema si trovamo un solo elemento ID ed un solo elemento IDREF, ma nel caso questo non sia vero occorre per ogni IDREF, nella traduzione in ODL_{I^3}, una interazione con l'utente per stabilire a quale identificatore ci si vuole riferire. Anche in questo caso ritengo che durante la traduzione sia opportuno traslare la dichiarazioni di *key* e di *foreign_*key alle interfacce rappresentanti gli elementi complessi e non i tipi complessi, per cui la traduzione risulta essere identica a quella presentata utilizzando in Xml-Schema gli elementi *key* e *keyref*.

5.20 Traduzione delle notazioni

Le notazione che ritroviamo in Xml-Schema (si trovano racchiuse all'interno dell'elemento *annotation*), forniscono generalmente informazioni sull'Xml-Schema stesso all'utente. In un documento Xml istanza non troveremo elementi derivanti dalle notazioni nello schema. Dato che, quindi, le notazioni non servono per definire la struttura di un documento Xml e che sono inutili al fine del reperimento delle informazioni dal documento stesso, una decisione sensata è quella di omettere completamente la traduzione di questi elementi in ODL_{I^3}. Tralasciando queste informazioni, infatti, non alteriamo la struttura delle interfacce ODL_{I^3} e non compromettiamo il riscontro di similarità fra classi distinte. Al contrario, traducendo in qualche modo le notazioni si rischia di modificare la struttuta di una classe con tutte le conseguenze del caso. Nel caso, invece, si desideri tradurre in ODL_{I^3} anche tali elementi (per ottenere una traduzione il più possibile reversibile) il metodo proposto impiega l'uso di costanti di tipo stringa (gli elementi della notazione, come ad esempio la *documentation*, vengono tradotti ognuno con una *const* di tipo stringa). Il nome di tali costanti non dovrebbe essere simile a nessun altro nome di *attribute* o di interfaccia dello schema e dovrebbe recare in se l'informazione di notazione. Vediamo come potrebbe essere tradotto il seguente esempio:

```
<xsd:complexType name="OrdineType">
    < xsd: annotation >
        < xsd:documentation xml:lang="it">
            definizione del tipo OrdineType,
            questo commento è scritto in italiano !
        </xsd:documentation>
```

```
    < /xsd: annotation >
    < xsd:element name =”Indirizzo” type=”IndirizzoType”/>
    < xsd:element name =”Oggetto” type=”OggettoType”
        minOccurs=”1” maxOccurs=”unbounded”/>
    < xsd:element ref =”Commento” minOccurs=”0” />
    < xsd: attribute name =”Data” type=”xsd:date”/>
</xsd:complexType>
```

La traduzione ODL_{I^3} proposta è:

```
interface OrdineType (source semistructured Ordine.xml)
{
const string Documentation_1=’definizione del tipo
OrdineType, questo commento è scritto in
italiano !’;

const string Documentation_lang_1=’it’;

attribute Indirizzo Indirizzo ;

    attribute set<Oggetto> Oggetto;

    typedef string ref_Commento;
    attribute ref_Commento Commento*;

    typedef string attr_Data ;
    attribute attr_Data Data;
    };
```

Si noti che i due elementi contenuti all’interno dell’elemento *annotation* nell’Xml-Schema sono stati tradotti tramite l’uso di costanti il cui nome (Documentation_1 e Documentation_lang_1) è un’indicazione degli elementi di Xml-Schema da cui sono stati ottenuti.

5.21 Traduzione di schemi in documenti multipli

Si ricordi che in Xml-Schema è possibile creare uno schema componendolo a partire da altri situati su file differenti. Lo scopo di questo procedimento è, sostanzialmente, creare schemi che siano più leggibili e mantenibili di schemi in cui un numero molto elevato di elementi sono presenti nello stesso documento. ODL_{I^3}, allo stato attuale, non prevede costrutti che supportino o che rimandino definizioni di attributi o interfacce su documenti distinti. Tale fatto significa che un eventuale programma di traduzione da Xml-Schema a ODL_{I^3} deve prevedere, al suo interno, funzioni in grado di capire quando si è in presenza di riferimenti a definizioni esterne, reperire queste definizioni ed, infine, inserirle correttamente nella traduzione ODL_{I^3}. Ricordiamo che gli elementi di Xml-Schema che permettono di importare oggetti definiti altrove sono *include* ed *import*. Si ricorda, inoltre, che:

- *include*: serve per importare definizioni di elementi situate su differenti schemi, ma appartenenti allo stesso target namespace
- *import*: serve per importare definizioni di elementi situate su differenti schemi, ma appartenenti a differenti namespaces

Per una migliore comprensione si consideri il seguente esempio, poniamo che lo schema sia situato in un file di nome Schema.xsd (si fa riferimento all'uso di *include*):

```
<xsd: schema xmlns:xsd="http :// www.w3.org/2001/XMLSchema"
    targetNamespace="http :// www.example.it/2001/Schemas">

<xsd:element name="Ordine" type="OrdineType"/>

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

<xsd:include
    schemaLocation="http :// www.example.it/2001/Schema1.xsd">

<xsd:complexType name="OrdineType">
    < xsd:element  name ="Indirizzo"  type="IndirizzoType"/>
    < xsd:element  name ="Oggetto" type="OggettoType" minOccurs="1"
        maxOccurs="unbounded"/>
    < xsd:element  ref  ="Commento" minOccurs="0" />
```

```
    < xsd: attribute   name =”Data” type=”xsd:date”/>
</xsd:complexType>

<xsd:complexType name=”OggettoType”>
    < xsd:element  name =”Nome” type=”xsd:string”/>
    < xsd:element  name =”Quantità” type=”QuantitàType”/>
    < xsd:element  name =”PrezzoEuro” type=”xsd:decimal”/>
    < xsd:element  ref =”Commento” minOccurs=”0”/>
</xsd:complexType>

<xsd:simpleType name=”QuantitàType”>
    <xsd: restriction
        base=”xsd: positiveInteger ”>
        <xsd:maxExclusive value=”100”/>
    </xsd: restriction >
</xsd:simpleType>

</xsd:schema>
```

All’interno del precedente schema vi è un riferimento al file Schema1.xsd appartenente allo stesso namespace. La definizione dell’elemento complesso IndirizzoType si trova, appunto, in tale file e viene importata. Il file Schema1.xsd potrebbe essere come segue:

```
<xsd: schema xmlns:xsd=”http :// www.w3.org/2001/XMLSchema”
    targetNamespace=”http :// www.example.it/2001/Schemas”>

<xsd:complexType name=”IndirizzoType”>
    < xsd:element  name =”Stato”  type=”xsd: string ”/>
    < xsd:element  name =”Città”  type=”xsd: string ”/>
    < xsd:element  name =”Via” type=”xsd: string ”/>
    < xsd:element  name=”Numero” type=”xsd:int”/>
</xsd:complexType>

</xsd:schema>
```

Un software impegnato nella traduzione dello schema precedente deve cercare e leggere il file Schema1.xsd e riportare la sua trascrizione in ODL_{I^3} nello stesso schema

contenente la traduzione del file Schema.xsd.

Capitolo 6

Il database lessicale WordNet e le sue estensioni

6.1 Introduzione

Il database lessicale WordNet [33] è stato sviluppato presso l'Università di Princeton sotto la direzione del professore George A. Miller. WordNet è disponibile gratuitamente presso il sito **http://www.cogsci.princeton.edu/ wn/**, la licenza d'uso permette l'utilizzo gratuito del database anche a fini commerciali ed al di fuori della ricerca, purchè siano citati gli autori ed il sito ufficiale del progetto.

Il database lessicale WordNet non è un semplice dizionario di termini inglesi, è un sistema lessicale di riferimento il cui disegno è inspirato alle teorie psicolinguistiche contemporanee sulla memoria lessicale umana. I termini, infatti, non sono disposti seguendo l'ordine alfabetico (cosa che accosterebbe parole con significati fra loro anche molto diversi), ma per affinità di significato. WordNet tratta quattro categorie sintattiche : nomi, verbi, aggettivi ed avverbi. Ogni categoria sintattica è suddivisa in diversi insiemi di sinonimi; ad ognuno di questi insiemi è associato un unico significato condiviso da tutti i termini ad esso associati. Un termine, ovviamente, può possedere più di un significato ed essere, quindi, presente in molti di questi insiemi ed anche in più di una categoria sintattica. Nel gergo usato all'interno del progetto WordNet un insieme di vocaboli che condivide uno stesso significato viene chiamato *synset*. Altro importante fattore che contraddistingue WordNet da un semplice dizionario di vocaboli, è la presenza di relazioni fra synsets. Vi sono diverse tipologie di relazioni che possono collegare due synsets, come, ad esempio, l'iperonimia e

l'iponimia tramite cui si è in grado di creare, all'interno dell'intera categoria sintattica, gerarchie di significato.

Le versione di WordNet attualmente impiegata a supporto del progetto MOMIS è la 1.6, datata 27 febbraio 1998.

6.2 La terminologia di WordNet

In questo paragrafo si introducono un insieme di temini significativi propri della terminologia di WordNet. Tali termini verranno utilizzati in questo capitolo, che tratta di WordNet, della sua struttura e della sua estensione, ed anche nei capitoli successivi.

- *Categoria sintattica*: sono le grandi categorie in cui sono suddivisi i termini (ed anche i file in cui sono contenuti) di WordNet. Le categorie sintattiche trattate sono quattro: nomi, verbi, aggettivi, avverbi.
- *Lemma*: è la parola, il termine a cui viene associato uno o più significati. A volte un lemma è costituito da due o più parole ed in tal caso i singoli termini sono uniti dal carattere underscore (_)
- *Synset*: un synset rappresenta il significato che viene associato ad un insieme di lemmi appartenenti alla stessa categoria sintattica. In pratica è corretta l'affermazione che ad un synset appartengono un certo numero di lemmi. Un synset, infatti, può essere rappresentato, oltre che tramite la sua Gloss, per mezzo dell'insieme dei sui lemmi (di solito racchiusi all'interno di parentesi graffe eg. $S_j\{l_1,l_2,l_3,l_4,l_5...\}$, dove a l_i corrisponde l'i-simo lemma collegato al synset j-simo)
- *Gloss*: una Gloss (o italianizzandola Glossa) è una descrizione a parole di uno specifico significato, ogni synset oltre a contenere un insieme di sinonimi possiede anche una glossa.
- *Relazione semantica*: si tratta di una relazione presente fra due synsets appartenenti alla stessa categoria sintattica; i diversi tipi di relazioni semantiche saranno trattate in seguito.
- *Relazione lessicale*: è una relazione che collega due lemmi appartenenti a due synsets distinti (ma sempre relativamente alla stessa categoria sintattica); i diversi tipi di relazioni lessicali saranno trattate in seguito.

6.3 La matrice lessicale

Il punto di partenza per la semantica lessicale è la comprensione che esiste una associazione fra la forma di una parola (per forma di una parola si intende il modo in cui viene scritta e letta tale parola) ed il significato ad essa associato. La corrispondenza fra forma della parola e significato non è di tipo univoco, ma rappresenta, invece, una relazione di tipo molti a molti dando luogo ai concetti di:

- *Sinonimia*: proprietà per cui uno stesso significato è esprimibile tramite l'uso due o più parole distinte.

- *Polisemia*: proprietà per cui ad una stessa parola sono associati due o più significati distinti.

	W_1	W_2	W_3	W_4	W_5
M_1	$E_{1,1}$				
M_2		$E_{2,2}$			
M_3		$E_{3,2}$	$E_{3,3}$		
M_4					
M_5				$E_{5,4}$	
M_6					$E_{6,6}$

Tabella 6.1: La matrice lessicale

Le relazioni fra forma della parola e significato possono essere rappresentate in quella che viene chiamata *matrice lessicale*. In tale matrice le righe rappresentano i vari significati che è possibile assumere per una parola e le colonne sono formate dai diversi termini. In pratica, volendo leggere la matrice lessicale tramite la terminologia di WordNet, ad ogni riga è associato un synset e ad ogni colonna un lemma. Ogni elemento non nullo che compare all'interno della matrice implica che il particolare lemma o termine situato in quella riga può essere usato per rappresentare lo specifico significato associato a quella colonna. Se in una colonna sono contenuti due o più elementi si ha un caso di polisemia (il termina associato alla colonna può essere sfruttato per esprimere più di un concetto); se, al contrario, due o più elementi compaiono in una stessa riga si è in presenza di un caso di sinonimia (il significato o synset di tale colonna può essere espresso tramite più parole distinte).

Il concetto di matrice lessicale viene espressa nel database di WordNet tramite, appunto, la separazione fra lemmi e synsets (mantenendo cioè separati termini e significati). Un synset viene espresso nei files usati da WordNet tramite l'insieme dei termini che sono ad esso associati. Tuttavia, nella maggioranza dei casi, solamente un insieme di parole non basta per descrivere un significato (si pensi al caso in cui si stia trattando un significato particolare che può essere descritto da una sola parola), così viene associata a ciascun synset anche una descrizione del significato tramite una frase in Inglese: la Gloss.

6.4 Le relazioni

WordNet presenta al suo interno due grandi gruppi di relazioni che si differenziano a seconda del tipo di operatori cui sono applicate. Si hanno così *relazioni lessicali* quando gli operandi sono entrambi lemmi e *relazioni semantiche* quando, invece, gli operandi sono synsets. Non possono esistere relazioni fra un lemma ed un synset o fra operandi appartenenti a diverse categorie sintattiche (ad esempio fra un nome ed un verbo). All'interno dei files originali di WordNet tutte le relazioni (con l'unica eccezione riguardante la relazione di sinonimia) sono rappresentate tramite puntatori e tramite caratteri speciali che indicano il tipo di relazione specificata. Nei prossimi paragrafi saranno presentate tutte le tipologie di relazioni semantiche e lessicali presenti in WordNet con relativa breve spiegazione.

6.5 Le relazioni semantiche

Le relazioni di tipo semantico coinvolgono sempre due concetti, due significati (due synsets), non semplicemente due termini (o lemmi).

6.5.1 Iponimia (Hyponymy)

Le relazioni di iponimia e iperminia (che ne rappresenta la relazione inversa) possono essere considerate l'equivalente delle gerarchie di specializzazione/generalizzazione per database relazionali o dell'ereditarietà per i modelli ad oggetti. Una relazione di questo tipo è valida solamente per le categorie sintattiche dei nomi e dei verbi (ma per i verbi si parla si troponimia). Una relazione di iponimia lega un concetto (nel nostro caso un synset) ad uno più generale, quello che può essere ritenuto una sua

generalizzazione. Trattandosi di un database di termini in lingua inglese è lecito dire che un synset *X* è un iponimo di un synset *Y* se è corretta l'affermazione *'X is a kind of Y'*. Per quanto riguarda la relazione opposta, quella di *ipernimia*, essa lega un concetto ad uno più particolare, più specializzato. In pratica si può affermare che un synset *X* rappresenta un ipernimo di un synset *Y* , se *Y* presenta tutte le caratteristiche di *X* più, almeno, una sua caratteristica particolare e aggiuntiva. Le relazioni di iponimia e ipernimia sono le relazioni più numerose presenti all'interno del database lessicale di WordNet. Un semplice esempio per meglio comprendere queste importanti relaziono potrebbe essere: ABETE è un iponimo di ALBERO, ALBERO, a sua volta, è un iponimo di VEGETALE. Sono altresì verificate le relazioni opposte: VEGETALE è un ipernimo di ALBERO che è un ipernimo di ABETE. La relazione di iponimia (assieme a quella di iperminia) può essere utilizzata per formare una gerarchia di specializzazione fra i synsets di WordNet.

6.5.2 Meronimia (Meronymy)

Anche la relazione di meronimia lega fra loro due concetti, o synsets, ed anche in questo caso si è in presenza di una relazione inversa, chiamata: *olonimia* (holonymy). Un concetto *X* è detto meronimo di uno *Y* se è lecito, per un madrelingua inglese, pronunciare la frase: *'X is a part of Y'*. Anche la relazione di meronimia, come quella di iponimia, può essere sfruttata per costruire una gerarchia sui synsets di WordNet, in cui uno risulta essere una parte dell'altro. Le relazioni di meronimia e olonimia vengono formulate sulla categoria sintattica dei nomi. Un esempio potrebbe essere rappresentato dai concetti di MURA e FONDAMENTA come meronomi di COSTRUZIONE.

6.5.3 Implicazione (Entailnment)

La relazione di implicazione è posta fra due verbi. La relazione di *entailment* può essere ritenuta simile a quella di meronimia posta sui nomi. Questa relazione è verificata se è verificata la seguente proposizione: un verbo *X* implica un verbo *Y* se *X* non può verificarsi a meno che non si sia verificato (o non si stia verificando) *Y*. L'implicazione non è solamente una relazione semantica, ma è possibile avere anche implicazioni lessicali fra verbi (fra singoli termini). Per comprendere meglio questo concetto si consideri i verdi DORMIRE e SOGNARE: in base alla definizione precedente, SOGNARE risulta implicare DORMIRE (*to dream entails to sleep*), infatti non

è possibile sognare senza dormire. L'implicazione lessicale è una relazione univoca: se un verbo *X* implica un verbo *Y* non può essere vero anche il contrario.

6.5.4 Relazione causale (Cause to)

Le relazione causale è simile alla relazione di implicazione (*entailment*) ma senza inclusione temporale. Un esempio potrebbe essere FORZARE che implica AGIRE.

6.5.5 Raggruppamento di verbi (Verb Group)

Questa relazione viene utilizzata per produrre raggruppamenti nella categoria sintattica dei verbi. In un gruppo formato in tale maniera i synsets hanno tutti un significato semantico molto simile. Un esempio di raggruppamento di verbi è dato da: mistake, confuse, confound, confuse, jumble, confuse, mix_up, confuse, blur, obscure

(Ogni synset dell'esempio precedente è stato riportato tramite l'elenco dei lemmi che lo compongono racchiusi all'interno di parentesi graffe)

6.5.6 Similarità (Similar to)

La relazione di similarità è utilizzata solamente nell'ambito della categoria sintattica riguardante gli aggettivi. Molti synsets di questa categoria sono raggruppati in coppie legate da una relazione di antinomia (si pensi, ad esempio, a synsets trattanti i concetti di PESANTE e LEGGERO, in netta contrapposizione semantica fra loro); tali synsets vengono chiamati synsets principali (o *head synsets*). A questi synsets principali sono collegati per similarità dei synsets *satelliti*, che condividono indirettamente la relazione di antinomia insieme al significato principale cui sono legati. Ricapitolando, un aggettivo descrittivo può avere una relazione di antinomia diretta (si tratta quindi di un synset principale) oppure una indiretta, tramite l'ausilio di una relazione di similarità (synset satellite).

6.5.7 Attributo (Attribute)

La relazione di attributo rappresenta il legame che intercorre fra un aggettivo ed un nome di cui esprime il valore. Gli aggettivi in grado di descrivere il valore di un attributo sono gli aggettivi descrittivi. Per fare un esempio basti pensare ad una frase come: *questa persona è alta*. L'aggettivo descrittivo *alta* indica il valore dell'attributo *altezza* riferito a *persona*. Aggettivi quali *alto* o *basso* sono, quindi, legati al nome *altezza*.

WordNet contiene puntatori fra gli aggettivi descrittivi ed i synsets, appartenenti alla categoria sintattica dai nomi, che rappresentano gli attributi cui conferiscono il valore.

6.5.8 Coordinazione

La coordinazione non è un tipo di relazione base, ma è, si potrebbe dire, derivata. Due synsets sono detti coordinati se possiedono lo stesso ipernimo se, cioè, risultano essere la specializzazione del medesimo concetto.

6.6 Relazioni lessicali

Le relazioni lessicali, diversamente da quelle semantiche, coinvolgono sempre due lemmi, non due synsets.

6.6.1 Sinonimia (Synonymy)

La sinonimia, anche se rappresenta una relazione lessicale, non è espressa formalmente come la altre relazioni di WordNet: non esiste alcun puntatore che collega un termine ad un suo sinonimo. La relazione è espressa, invece, tramite l'appartenenza, da parte dei due vocaboli sinonimi, allo stesso synset. Ricordiamo, infatti, che un synset può essere rappresentato e caratterizzato dall'insieme dei lemmi che lo compongono (è un concetto che può essere espresso tramite la matrice lessicale); il termine synset, (*set of synonyms*) è stato coniato proprio per indicare questa idea. Per ogni coppia di termini appartenenti allo stesso synset esiste dunque, in maniera implicita, una relazione di sinonimia. Due possibili definizioni di sinonimia sono possibili: una attribuita a Leibniz

Due termini sono sinonimi se la sostituzione di uno per l'altro non cambia mai il valore della frase in cui è fatta la sostituzione.

ed una che rende la relazione di sinonimia relativa la contesto della frase:

due termini sono sinonimi, all'interno di un contesto linguistico C, se la sostituzione di un termine con l'altro, all'interno di C, non varia il valore della frase.

La seconda definizione è decisamente più permissiva rispetto alla prima: esistono pochi termini considerati sinonimi nel senso descritto da Leibniz: è estrmemente difficile, infatti, trovare due parole il cui interscambio sia possibile in ogni genere di

contesto. Il database lessicale WordNet, comunque, adotta, per stabilire la relazione di sinonimia, la seconda definizione: due lemmi sono sinonimi solo all'interno di un certo contesto, di un certo synset. Anche tramite la seconda definizione di sinonimia, appare chiaro che due termini appartenenti a categorie sintettiche differenti non potranno in nessun caso essere sinonimi. Proprio per questa ragione WordNet è stato diviso nelle categorie sintattiche di nomi, verbi, aggettivi ed avverbi.

6.6.2 Antinomia (Antynomy)

L'antinomia è una relazione lessicale fra due lemmi. Due termini legati da una relazione di antinomia sono l'uno il contrario dell'altro. Non è sempre corretta, comunque, l'affermazione che *non x* è l'antonimo di *x*. Si pensi, ad esempio, ai termini *ricco* e *povero*: se un individuo non è ricco, non è necessariamente detto che sia povero. Rimane invero evidente la contrapposizione di significato fra i lemmi *ricco* e *povero*. Non è corretto considerare l'antonimia una relazione semantica, quindi fra synsets; i synsets {*rise, ascend*} e {*fall, descend*}, pur essendo concettualmente opposti, non rappresentano degli antinomi. Una relazione di antinomia, invece, è presente fra *rise* e *fall*, ed anche fra i termini *ascend* e *descend*.

6.6.3 Relazione di pertinenza (Pertainym)

La relazione di pertinenza concerne gli aggettivi relazionali. Un aggettivo relazionale svolge un ruolo che può essere riassunto in un espressione come: *associato con*, oppure *pertinente a* o, semplicemente *di* in relazione ad un nome. L'aspetto di un aggettivo relazionale risulta molto simile a quello del nome cui è legato leggermente modificato. Si pensi all'espressione *acutezza mentale*; l'aggettivo relazionale *mentale* è associato al nome *mente* tramite, appunto, una relazione di pertinenza.

6.6.4 Vedi anche (See also)

La relazione detta *vedi anche* è una relazione lessicale e lega singoli lemmi di synsets differenti. I motivi di tale relazione possono essere molto differenti fra loro.

6.6.5 Relazione participiale (Participle)

Questa relazione lega gli aggettivi, detti participiali, ai nomi da cui derivano. Come esempio si può pensare all'aggettivo *bruciato* derivante dal verbo *bruciare* (all'interno

di WordNet esiste un relazione participiale fra i lemmi *burned* e *burn*, appartenenti alle categorie sintattiche, rispettivamente, di aggettivi e verbi).

6.6.6 Derivato da (Derived from)

Alcuni aggettivi relazionali derivano da antichi nomi Greci o Latini. Questa affermazione risulta essere vera sia per la lingua italiana che per quella inglese (idioma su cui è costruito WordNet). L'aggettivo relazionale *verbale*, deriva dal nome neutro latino *verbum*, mentre *lessicale* deriva dal corrispondente nome greco. La relazione *derivato da* lega gli aggettivi ad i nomi stranieri da cui derivano.

6.7 Estendere WordNet

Cosa significa esattamente estendere WordNet?

Tramite l'espressione -estendere WordNet- ci riferiamo alla possibilità di poter aggiungere nuovi termini, nuovi concetti e nuove relazioni a quelli contenuti nelle originali versioni di WordNet. L'interfaccia MOMIS-WordNet progettata ed implementata da Giovanni Malvezzi[30] (si tratta del modulo SLIM) reperisce le informazioni necessarie alla costruzione del *common thesaurus* direttamente dai files propri di WordNet. Ovviamente WordNet non è in grado di fornire la conoscenza richiesta in qualsiasi ambito (basti pensare al desiderio di annotare schemi trattanti campi specialistici come la medicina o l'elettronica) e si potrebbero così verificare perdite di informazioni anche molto importanti. Supponiamo, infatti, di voler annotare un elemento (classe o attributo) il cui nome non è presente all'interno dei files originali di WordNet; si hanno, in questo caso, due possibilità per poter assegnare all'elemento un significato:

- Scegliere una forma base presente all'interno di WordNet ma differente dal nome originale (se i due termini non sono esattamente sinonimi si verifica una perdita parziale di conoscenza)
- Decidere di non annotare l'elemento (si verifica una perdita totale di conoscenza per quanto riguarda l'oggetto considerato)

Proprio per rendere estensibile WordNet ed evitare perdite di conoscenza durante la fase di annotazione degli oggetti presenti nei vari schemi, è stato implementato

da Veronica Guidetti[24] il modulo WordNetEditor. WordNetEditor (o più semplicemente WNEditor) è un browser capace di navigare all'interno di WordNet e di modificarne la struttura creando nuovi lemmi, nuovi synsets e nuove relazioni. WNEditor, comunque, non modifica direttamente i files di WordNet, ma utilizza un database relazionale chiamato MOMISWN (attualmente è implementato tramite MySql) ottenuto a partire dai medesimi files. In pratica, considerando che SLIM crea il theraurus di relazioni a partire dai files originali di WordNet, e non dal database MOMISWN, WNEditor e SLIM sono due moduli non integrati.

Parte del lavoro svolto in questa tesi è stato modificare il modulo SLIM, gli algoritmi utilizzati al suo interno, ed alcune parti di WNEditor al fine di integrare pienamente i due moduli, permettendo così ad un eventuale utente di estendere la conoscenza contenuta in WordNet direttamente durante la fase di annotazione.

Capitolo 7

Modifiche al modulo SLIM

7.1 Introduzione

Il primo passo che porta il sistema MOMIS alla realizzazione di uno schema comune partendo da un insieme di sorgenti eterogenee, è la creazione di un dizionario di relazioni (*common thesaurus*) ricavato dall'analisi di tali fonti. La parte del sistema MOMIS che attua questo processo è il *Global Schema Builder* (GSB). Il GSB, a sua volta, è composto da due moduli in grado di collaborare al fine della creazione del thesaurus:

- *SIM(Source Integrator Module)*: il modulo SIM è in grado di estrarre relazioni intensionali intra-schema (ottenute, quindi, fra operandi appartenenti allo stesso schema), analizzando la struttura delle classi ODL_{I^3} componeneti lo schema. Questo modulo sfrutta l'interazione con il componenete esterno ODB-Tools in grado di validare le relazioni ottenute e di inferirne di nuove.

- *SLIM(Schemata Lessical Integrator Module)*: modulo in grado di estrarre relazioni intensionali inter-schema (quindi è in grado di ricavare relazioni fra componenti appartenenti a schemi differernti) tramite l'interazione con il database lessicale WordNet.

Attualmente, tramite il componente software *SIDesigner* è offerta al progettista una interfaccia grafica per interagire con i moduli SLIM e SIM.

7.1.1 Come funziona attualmente SLIM?

Il modulo SLIM progettato da Giovanni Malvezzi[] svolge il compito di estrarre relazioni intensionali inter-schema dall'analisi delle viste espresse in ODL_{I^3} delle varie sorgenti locali. Tali relazioni possono essere espresse fra due attributi, due interfacce oppure anche tra un attributo ed una interfaccia. Si ricorda che il *common thesaurus* prevede l'utilizzo di quattro tipi di relazioni:

- SYN: synonym-of
- BT: broader-term
- NT: narrow-term
- RT: related-term

L'estrazione di queste relazioni prevede una fase preliminare (affidata al progettista), chiamata *fase di annotazione*, formata da due operazioni distinte:

- *Scelta della forma base*: per ogni nome di classe ed attributo in ogni sorgente selezionata per l'integrazione, SLIM propone una forma base (*word form*). Tale forma base è lo stesso nome dell'oggetto trattato eliminati i suffissi dovuti a declinazioni o coniugazioni. Il progettista è, comunque, sempre libero si cambiare a piacimento la forma base di qualsiasi oggetto per cancellare eventuali ambiguità o selezionare differenti significati.
- *Scelta del significato*: per ogni forma base SLIM propone al progettista una serie di significati fra cui scegliere quello desiderato (o, in alcuni casi, quelli desiderati). Tutti i significati proposti sono ottenuti dal database lessicale WordNet.

In pratica, tramite la fase di annotazione, si puo associare un significato (quindi un synset) ad ogni oggetto (classe o attributo) di ogni sorgente da integrare. Il *common thesaurus* di relazioni inter-schema viene generato proprio a partire da questi synsets (e, ovviamente, dalle relazioni che, all'interno di WordNet, li legano). Le relazioni di WordNet utilizzate al fine di raggiungere tale scopo sono:

- *Sinonimia* mappata in una relazione di tipo SYN
- *Ipernimia* mappata in una relazione di tipo BT

- *Iponimia* mappata in una relazione di tipo NT
- *Meronimia* mappata in una relazione di tipo RT
- *Olonimia* mappata in una relazione di tipo RT
- *Correlazione* mappata in una relazione di tipo RT

Si supponga, ad esempio, di avere in una sorgente *A* una classe di nome *student* (studente) ; si supponga anche che esista, in una sorgente *B*, una classe chiamata *law_student* (studente in legge). Durante la scelta della forma base i nomi delle due classi non vengono variati ed i significati scelti nel passo successivo sono:

per *student*: pupil,student,educatee — 'a learner who is enrolled in a educational institution'

e per *law_student*: {law_student} — 'a student in a law school'

(il formato in cui vengono espressi i synsets contiene fra parentesi graffe i lemmi collegati al synset, seguiti, dopo il simbolo —, dalla Gloss)

All'interno del database lessicale di WordNet fra questi due synset è definita una relazione di iponimia (*law_student* è un iponimo di *student*); seguendo il mapping utilizzato fra relazioni semantiche e relazioni prodotte da SLIM otteniamo:

B.law_student NT A.student

7.2 L'integrazione di SLIM e WordNetEditor

Abbiamo visto all'interno del capitolo 6 cosa significa esattamente estendere WordNet: poter aggiungere al database lessicale nuovi termini (lemmi), nuovi significati (synset) e nuove relazioni per poter evitare perdite di conoscenza durante la fase di annotazione degli schemi ODL_{I^3}. La realizzazione di questo intento avviene per mezzo del database MOMISWN, contenente tutti i synset, tutti i lemmi e tutte le relazioni della versione 1.6 di WordNet e facilmente estensibile per mezzo del browser WordNetEditor (WNEditor) progettato e realizzato da Veronica Guidetti[24].

Integrare il modulo SLIM con WNEditor significa effettuare due azioni principali:

1. Riscrivere gli algoritmi e le funzioni di SLIM atte a reperire i dati(synset, lemmi, relazioni sematiche) in modo che utilizzino il database relazionale MOMISWN.

2. Creare un'interfaccia *user friendly*, quindi facilmente utilizzabile per un eventuale utente, che colleghi al modulo SLIM il browser WNEditor.

Consideriamo il primo punto. Il modulo SLIM originariamente progettato e realizzato da Giovanni Malvezzi[30] raccoglie le informazioni di interesse (i significati connessi ad un termine durante la fase di annotazione e le relazione semantiche fra synset durante la fase di creazione del thesaurus di relazioni inter-schema) direttamente dai file originali di WordNet. Per poter integrare SLIM con WNEditor è dunque necessario riscrivere le funzioni usate da SLIM in modo che reperiscano i dati dal database relazionale impiegato da WNEditor: MOMISWN.

Veniamo ora al secondo punto. Il modulo WNEditor è stato realizzato originariamente cone una *fase* di MOMIS, al pari dei moduli SAM, SIM e SLIM. Questo significa che, supponendo di aver già realizzato il primo punto, ogni volta che deve essere annotato un termine che non esiste all'interno del database MOMISWN bisogna uscire dall'interfaccia grafica del modulo SLIM ed entrare in quella di WNEditor. Si può ben comprendere come questo procedimento, se ripetuto molte volte all'interno di una stessa fase di annotazione, possa risultare noioso e poco efficiente. Bisogna, quindi, realizzare un'interfaccia che permetta l'apertura di WNEditor direttamente all'interno della GUI di SLIM.

Entrambi i punti precedentemente elencati e spiegati sono stati realizzati nell'ambito di questa tesi. Una nuova versione del modulo SLIM è stata, dunque, prodotta in modo da utilizzare il database relazionale MOMISWN. Per ottenenre questo sono state riscritte le funzioni che ricercano i synset durante la fase di annotazione ed i metodi per la realizzazione del thesaurus di relazioni inter-schema. Il modulo WNEditor si può, inoltre, usare ora facilmente all'interno dell'interfaccia grafica di SLIM, alleggerendo notevolmente il compito dell'utente.

Nei prossimi paragrafi saranno spiegate in maniera approfondita le modifiche effettuate al software onde ottenere i risultati voluti.

7.3 Realizzazione dell'integrazione

Il processo di integrazione fra i moduli SLIM e WNEditor ha implicato la riscrittura delle classi Java e degli algoritmi necessari al fine della creazione di un thesaurus di relazioni intensionali ed inter-schema.

Si ricorda che la versione originale del software SLIM crea il thesaurus di relazioni intensionali a partire dalle relazioni semantiche reperite dai file originali di WordNet.

Le informazioni contenute nei files originali di WordNet sono state trasferite nel database relazionale MOMISWN; tale database è estensibile e modificabile tramite il tool grafico WNEditor (in particolare, tramite WNEditor si possono modificare solamente le nuove relazioni prodotte ed i nuovi lemmi e synsets creati. In questo modo si ha sempre la parte del database costruita a partire dai files di WordNet consistente con tali files).

Le modifiche apportate a SLIM ed ai moduli del software di SIDesigner collegati con esso sono orientate alla interazione con il database MOMISWN e con il programma WNEditor.

Nei prossimi paragrafi saranno spiegate le principali differenze nel funzionamento e nella organizzazione del software fra la versione originale e quella modificata di SLIM.

7.3.1 Organizzazione del software di SLIM originale

Il programma SIDesigner, che fornisce un supporto interattivo e semiautomatico, oltre che grafico, per l'integrazione di sorgenti eterogenee, è stato progettato e scritto in Java. Per spiegare l'organizzazione del software della versione originale di SLIM e le differenze presenti con la nuova versione, verranno usati alcuni termini specifici inerenti questo linguaggio di programmazione (come ad esempio Package: insieme di classi situate nella stessa directory e concorrenti ad un fine comune).

L'organizzazione originale del modulo SLIM, e dei moduli da esso utilizzati, è molto importante al fine di comprendere le scelte che sono state fatte per produrre la nuova versione del software.

Le principali classi ed i package più importanti impiegati nel processo di creazione del *common thesaurus* di relazioni inter-schema sono (per quanto riguarda la versione originale di SLIM):

- *Package SLIM*: package contenente le classi (compresa la classe SLIM) atte a permettere l'annotazione dei vari schemi e a visualizzare il thesaurus di relazioni inter-schema prodotto tramite l'interazione con WordNet
- *Classe CORBA_WordNet*: classe i cui metodi vengono chiamati da SLIM e che rappresenta un'interfaccia con il package mWordNet
- *Package mWordNet*: package contenente la classi e le funzioni per l'accesso ai files di WordNet.
- *Package wn2Slim*: package contenente la classi tramite cui mWordNet deve presentare le informazioni ottenute dai files di WordNet alla classe CORBA_WordNet (e quindi a SLIM)

L'interazione fra i precedenti componenti Java è espressa dalla Figura(7.1)

Come si può notare il package SLIM (e tutte la classi contenute in esso) comunica solamente con la classe CORBA_WordNet. Il processo di interazione tra i precedenti oggetti (e gruppi di oggetti) Java può essere spiegato, seguendo la Figura(7.1), tramite i seguenti cinque passi:

1. Le classi contenute all'interno del package SLIM sfruttano, per svolgere operazioni quali ritrovamento dei significati di un termine e creazione del thesaurus, i metodi della classe CORBA_WordNet
2. La classe CORBA_WordNet impiega i metodi delle classi contenute all'interno del package mWordNet per rispondere alle richieste di SLIM
3. Le classi del package mWordNet interagiscono con i files originali del database lessicale WordNet. Vengono così soddisfatte le richieste, da parte di CORBA_WordNet, riguardanti il ritrovamento di lemmi, synsets o relazioni semantiche atte a formare il thesaurus

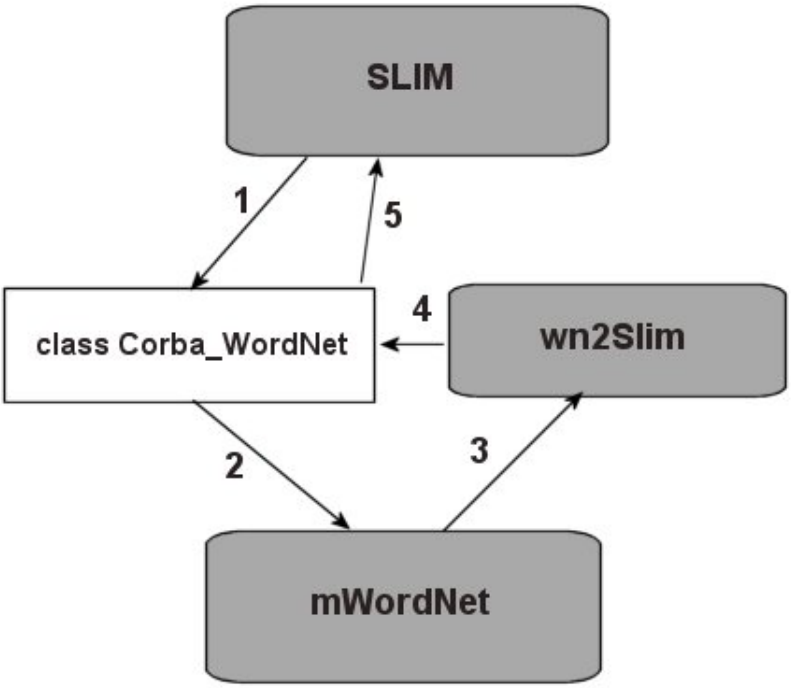

Figura 7.1: Interazione di SLIM con i Package di SIDesigner

4. Le informazioni raccolte da mWordNet vengono presentate alla classe CORBA_WordNet nei formati espressi dalle classi del package wn2Slim (i synsets vengono presentati tramite la classe Synset del package wn2Slim; le relazioni ritrovate tramite la classe ThesaurusEntry ...)

5. La classe CORBA_WordNet risponde alle richieste originarie di SLIM in un formato ad esso comprensibile (il formato dato dalle classi del package wn2Slim). SLIM è ora in grado di presentare graficamente le informazioni ritrovate all'utente che può così proseguire nell'opera di integrazione delle fonti.

Lo schema rappresentato in Figura(7.2) fornisce una descrizione, seguendo il modello UML, delle principali classi coinvolte nel processo di interazione descritto in precedenza. Si ricorda che in un modello UML di questo tipo:

- Le classi sono rappresentate tramite rettangoli divisi in tre parti. Nella parte superiore è presente il nome della classe, nella parte centrale sono situati gli attributi di tale oggetto mentre, in quella inferiore, sono collocati i metodi.

- I simboli (+, -, #) presenti prima di ogni nome di attributo o di metodo significano: + attributo o metodo pubblico; - attributo o metodo privato; # attributo o metodo protetto.

- Una freccia che collega due classi rappresenta una gerarchia di specializzazione: la classe cui punta la freccia à la classe padre, mentre quella da cui parte la freccia rappresenta la classe figlio (che eredita attributi e metodi da quella padre).

- Una linea che inizia con un rombo rappresenta una relazione di appartenenza che lega due classi (il rombo è in prossimità della classe che possiede l'altra).

- Una semplice linea che lega due classi rappresenta una relazione. I numeri fra parentesi tonde indicano le cardinalità (minime e massime) delle due classi nella stessa relazione.

Per carenza di spazio ed una maggiore comprensibilità, all'interno delle classi rappresentate in Figura(7.2) non sono stati riportati tutti i metodi e gli attributi che tali oggetti possiedono, ma solamente quelli più significativi.

Possiamo osservare che la classe SLIM (appartenente al package omonimo) deriva dalla classe SIDPhase. SIDPhase contiene i metodi che devono essere implementati da tutte le cosiddette *fasi* di SIDesigner. Queste fasi sono costituite dai moduli (come il modulo SAM per gestire le fonti, il modulo SIM per l'utilizzo di ODB-Tools e la validazione di schemi, il modulo SLIM per la fase di annotazione e la creazione del thesaurus di relazioni inter-schema ...) che implementano in pratica le effettive fasi del processo di integrazione svolto dal sistema MOMIS. SIDPhase è, inoltre, in relazione con la classe GlobalSchemaProxy che mantiene un collegamento con l'oggetto GlobalSchema (si tratta dello schema mediato prodotto nel processo di integrazione) ed altre informazioni che devono essere comuni a tutte la fasi del procedimento. Si consideri ora la classe SlimTree (appartenente anch'essa, come la classe SLIM, al package SLIM); tale oggetto *appartiene* alla classe SLIM nel senso che viene generato ed utilizzato all'interno di essa (anche visivamente, tramite l'interfaccia GUI, l'albero binario di SlimTree appare contenuto nel modulo SLIM). I metodi di SlimTree trovaSensi, trovaParenti, creaThes e trovaHashObject chiamano

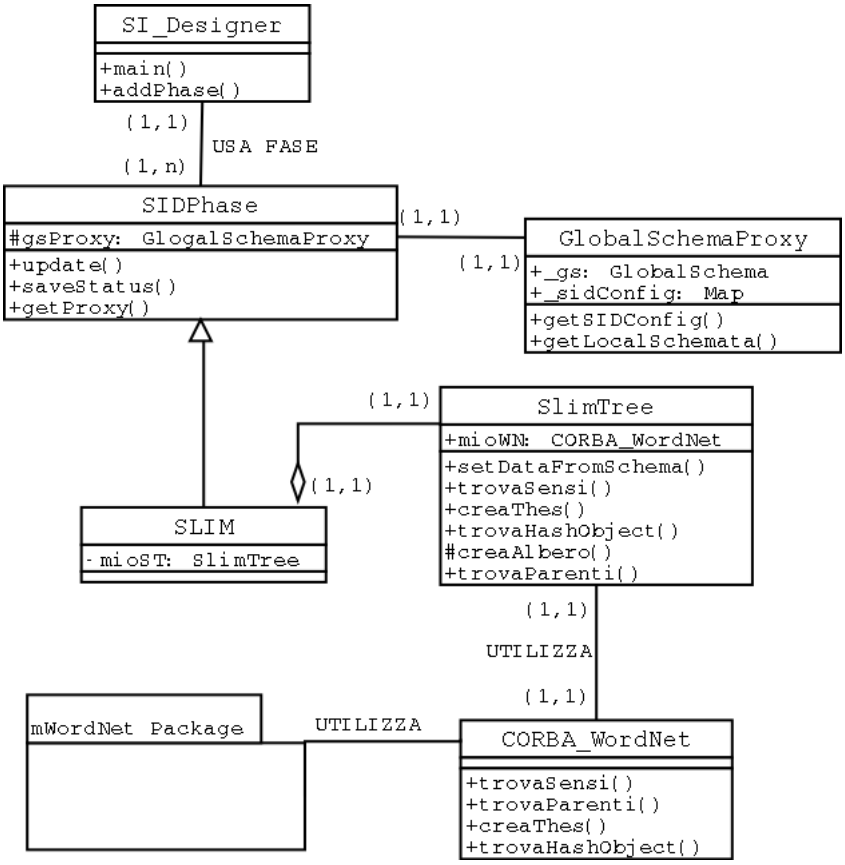

Figura 7.2: Schema UML delle classi principali di SLIM e di quelle da esso utilizzate

le funzioni omonime apprtenenti alla classe CORBA_WordNet, la quale si interfaccia direttamente con gli oggetti del package mWordNet in grado di reperire le informazioni desiderate dai files di WordNet. Una breve spiegazione della funzione svolta dai metodi citati in precedenza è la seguente:

- *TrovaSensi*: funzione capace di associare ad un lemma tutti i synsets in cui, all'interno di WordNet, compare esplicitemente.
- *CreaThes*: funzione che genera il thesaurus di relazioni inter-schema sfruttando le relazioni semantiche fra i synsets di WordNet.
- *TrovaParenti*: funzione in grado i synsets considerati parenti di un synset dato, fra cui trovare i sinonimi di un lemma dato.
- *TrovaHashObject*: funzione usata per la creazione del grafico rappresentante tutti gli iperonimi di un synset.

Nel prossimo paragrafo verrà spiegata la nuova struttura del software in grado di relazionarsi anche con il database MOMISWN

7.3.2 Organizzazione del software nella nuova versione di SLIM

Il software di SIDesigner è stato modificato al fine di permettere al modulo SLIM di creare il thesaurus di relazioni inter-schema non più a partire dai files originali di WordNet, ma ricercando le informazioni necessarie ispezionando il database MOMISWN. Il punto di partenza alle modifiche del software è stato osservare che tutte le richieste fatte dalle classi del package SLIM (e dalla classe SlimTree in particolare) per interagire con WordNet, attraversano la classe CORBA_WordNet. CORBA_WordNet risulta essere, quindi, una vera e propria interfaccia fra SLIM ed il database lessicale WordNet. Appare dunque chiaro come il modo più semplice per ottenere il risultato voluto sia stato riscrivere la classe CORBA_WordNet in modo che interagisca con una nuova libreria di funzioni per l'accesso al database MOMISWN.

In realtà ciò che è stato fatto (come mostra la Figura(7.3)) è leggermente diverso: per mantenere la compatibilità con la versione precedente di SIDesigner è stata creata una nuova classe, chiamata MDB_WordNet, in grado di soddisfare le richieste del modulo SLIM interagendo con la nuova libreria di funzioni situata nel package mDBWordNet; tale classe non sostituisce definitivamente CORBA_WordNet ma, all'avvio del programma SIDesigner, è possibile scegliere tramite un parametro (-dbw) in quale modalità far funzionare SLIM: tramite i files originali di WordNet (e quindi attraverso CORBA_WordNet) oppure tramite il database estensibile MOMISWN (usando MDB_WordNet). I due percorsi possibili sono visibili in Figura(7.3). Il percorso che si ottiene seguendo le frecce tratteggiate è sempre quello rappresentato in Figura(7.1), i cui passi sono stati spiegati in precedenza. Il percorso ottenuto seguendo le frecce continue, invece, rappresenta la seconda nuova modalità di funzionamento di SLIM (attraverso il database MOMISWN), i cui passi fondamentali sono:

1. (punto 6 in figura) Le classi contenute all'interno del package SLIM sfruttano, per svolgere operazioni quali ritrovamento dei significati di un termine e creazione del thesaurus, i metodi della classe MDB_WordNet

2. (punto 7 in figura) La classe CORBA_WordNet impiega i metodi delle classi contenute all'interno del package mDBWordNet per rispondere alle richieste di SLIM

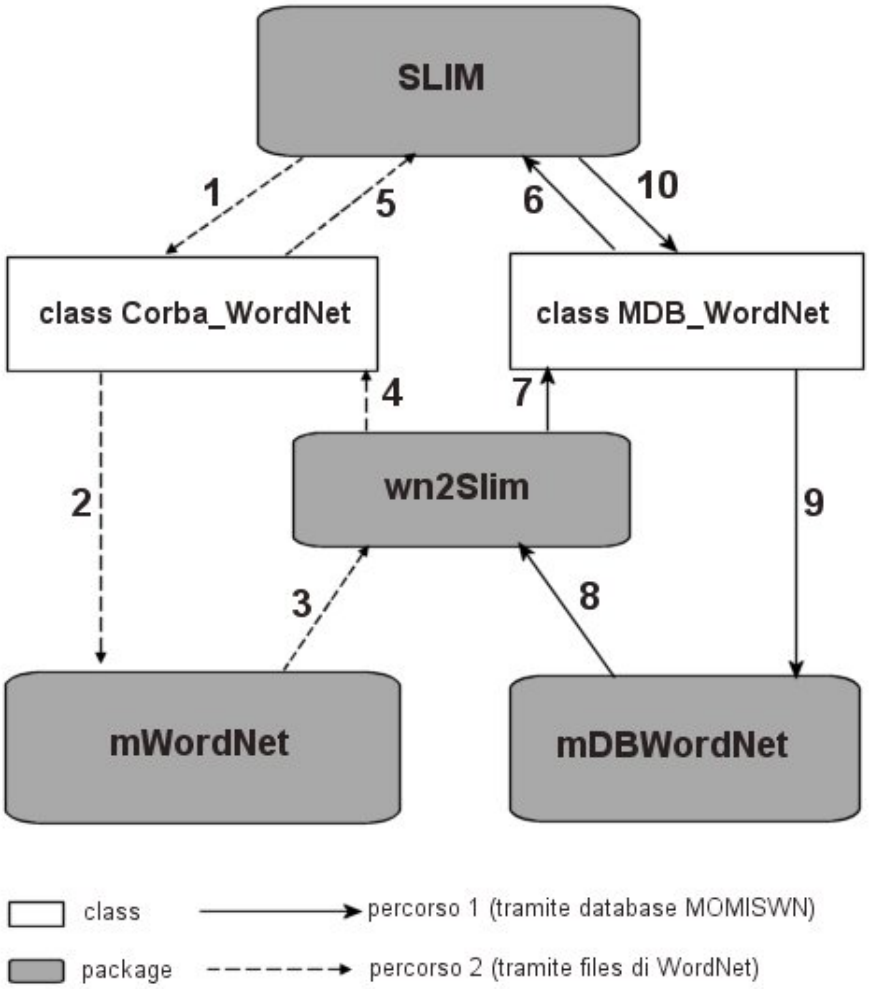

Figura 7.3: Schema dei package usati nella nuova versione di SLIM

3. (punto 8 in figura) Le classi del package mDBWordNet interagiscono con i dati contenuti nel database MOMISWN. Vengono così soddisfatte le richieste, da parte di MDB_WordNet, riguardanti il ritrovamento di lemmi, synsets o relazioni semantiche atte a formare il thesaurus

4. (punto 9 in figura) Le informazioni raccolte da mDBWordNet vengono presentate alla classe MDB_WordNet nei formati espressi dalle classi del package wn2Slim (i synsets vengono presentati tramite la classe Synset del package wn2Slim; le relazioni ritrovate tramite la classe ThesaurusEntry ...)

5. (punto 10 in figura) La classe MDB_WordNet risponde alle richieste originarie

di SLIM in un formato ad esso comprensibile (il formato dato dalle classi del package wn2Slim). SLIM è ora in grado di presentare graficamente le informazioni ritrovate all'utente che può così proseguire nell'opera di integrazione delle fonti.

Si noti come sia la classi contenute nella libreria mWordNet, sia quelle appartenenti a mDBWordNet presentino i dati raccolti nello stesso formato: tramite le classi del package wn2Slim. Proprio per questo motivo i due metodi di funzionamento possibili di SLIM risultano essere pienamente compatibili e trasparenti all'utente.

Per maggiori informazioni sull'organizzazione vera e propria del software si osservi la Figura(7.4). La classe MWordNetManager è una classe astratta che possiede i metodi invocati da SlimTree (in precedenza SlimTree invocava direttamente i metodi di CORBA_WordNet). Le due classi, CORBA_WordNet e MDB_WordNet, che fungono da interfaccia, rispettivamente, con WordNet originale e con il database MOMISWN, derivano da MWordNetManager e ne riscrivono i metodi in dipendenza del loro effettivo utilizzo. In questo modo il package SLIM, sfruttando i metodi di un oggetto MWordNetManager, è in grado di collegarsi indifferentemente sia con la classe CORBA_WordNet che con MDB_WordNet.

Un'altra classe ad essere stata modificata è GlobalSchemaProxy. Questa classe è condivisa da tutte la fasi (intendendo con il termine fasi anche gli oggetti che ereditano da SIDPhase) del sistema SIDesigner. Si è, quindi, scelto di inserire un oggetto MWordNetManager direttamente al suo interno, in modo da poter essere sfruttato da tutti i moduli del sistema (SLIM compreso).

7.3.3 Collegamento di SLIM con il browser WordNetEditor

Il modulo WordNetEditor (WNEditor) sviluppato da Veronica Guidetti è un browser in grado di navigare all'interno del database MOMISWN ed aggiungere nuovi termini (lemmi), nuovi significati (synsets) e nuove relazioni, sia semantiche che lessicali, a quelle già esistenti. WNEditor crea, inoltre, automaticamente per ogni relazione aggiunta al database anche la relazione inversa (nel caso sia possibile). Il modulo WordNetEditor, la cui classe principale è WordNetEditorPanel, era originariamente implementato e visualizzato come una *fase* di SIDesigner (WordNetEditorPanel deriva direttamente dalla classe SIDPhase, già citata in precedenza). Per facilitarne l'u-

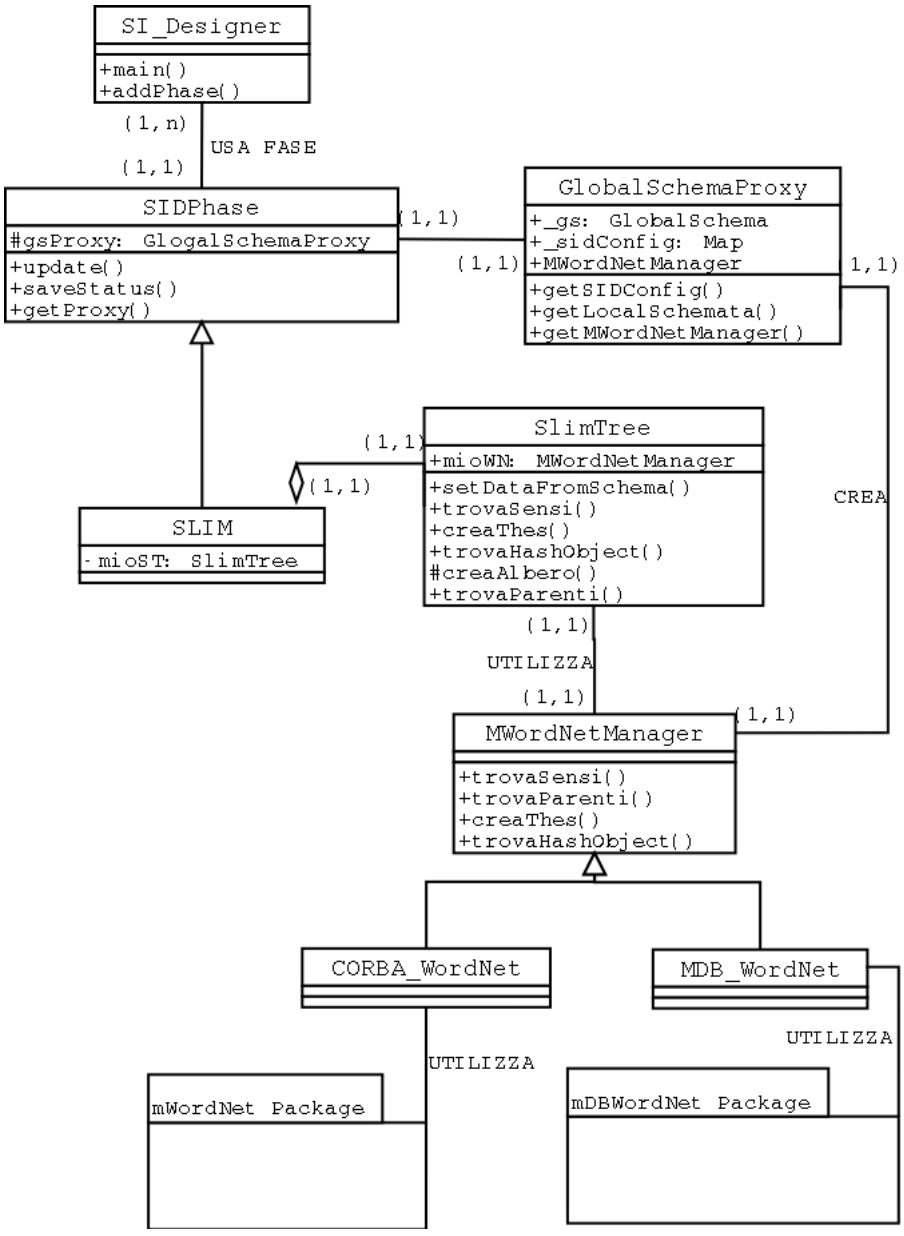

Figura 7.4: Schema UML delle classi nella nuova versione di SLIM

tilizzo ed offrirne un impiego immediato durante la fase di annotazione degli schemi, WNEditor è ora visualizzato tramite una finestra che può essere aperta direttamente dall'interno del modulo SLIM (e più precisamente dal menu collegato ad ogni SlimNode).

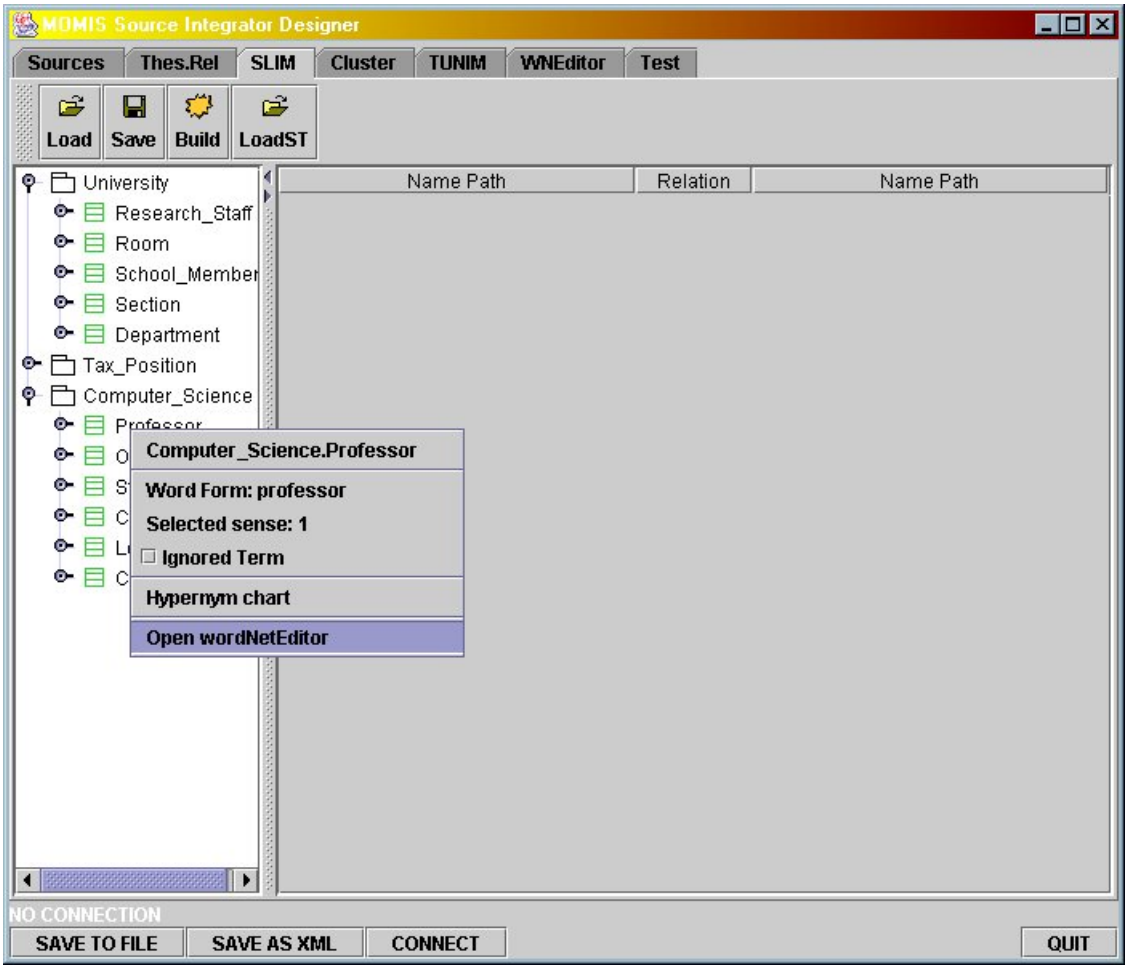

Figura 7.5: Apertura di WNEditor dal modulo SLIM

In Figura(7.5) è possibile notare il menu che viene visualizzato a partire dallo SlimNode concernente l'oggetto (interfaccia) *professor*. Scegliendo, a questo punto, l'opzione *Open wordNetEditor* comparirà il browser visualizzato in Figura(7.6).

E' ora possibile collegare, ad esempio, nuovi significati al termine *professor*. Nell'esempio rappresentato dalla Figura(7.6), è stato aggiunto il significato (synset non

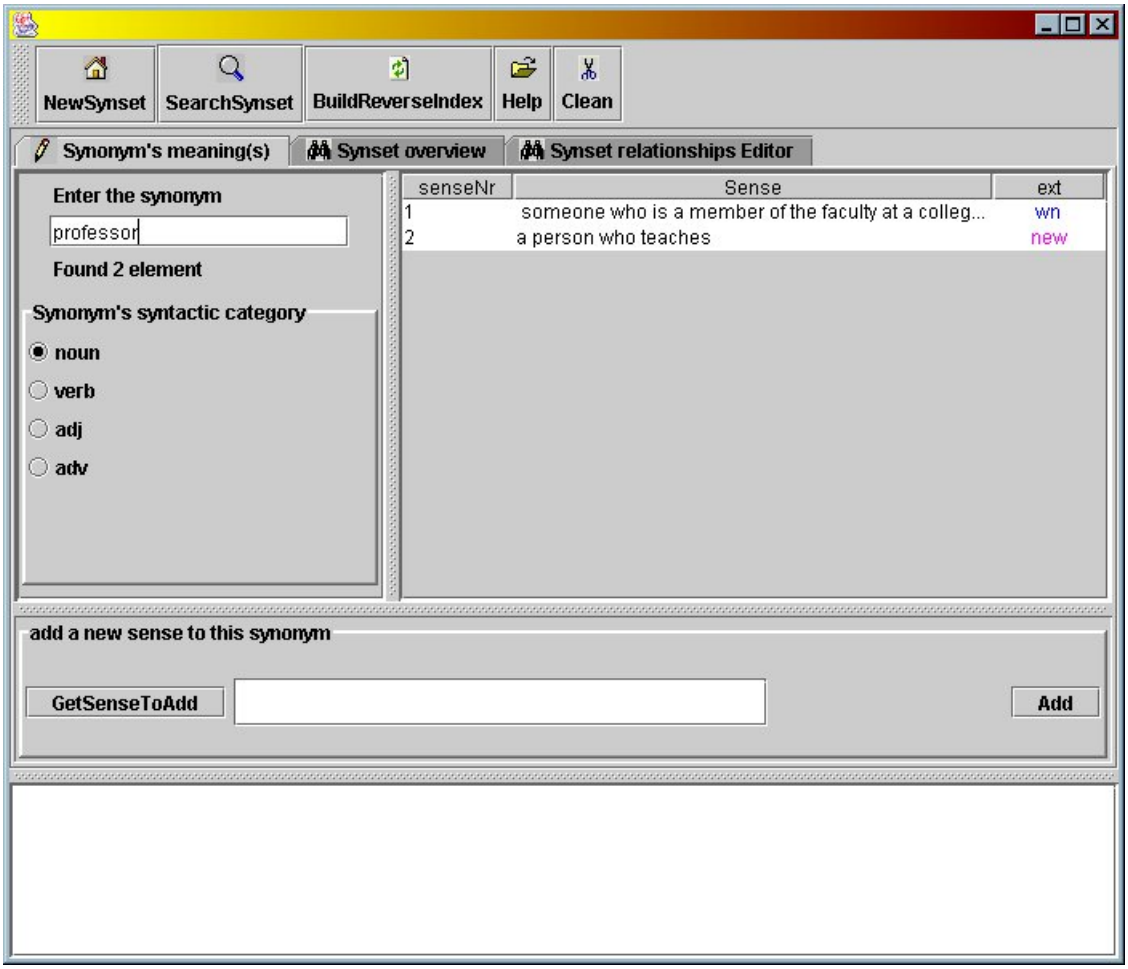

Figura 7.6: Vista del browser WNEditor

presente fra quelli originari di WordNet) *'a person who teaches'*. Grazie alla nuova versione di SLIM il significato aggiunto può essere scelto per annotare l'oggetto *professor*, come visualizzato in Figura(7.7). Anche il thesaurus di relazioni inter-schema, prodotto premendo il pulsante *Build* del modulo SLIM, risulta essere consistente con il database modificato.

Se si è interessati alla modalità in cui SLIM e WNEditor sono stati integrati, si osservi la Figura(7.8).

La classe WNSlim è stata aggiunta al package SLIM al fine di creare e visualizzare una finestra in cui mostrare WNEditor. Un oggetto di tipo WNSlim viene

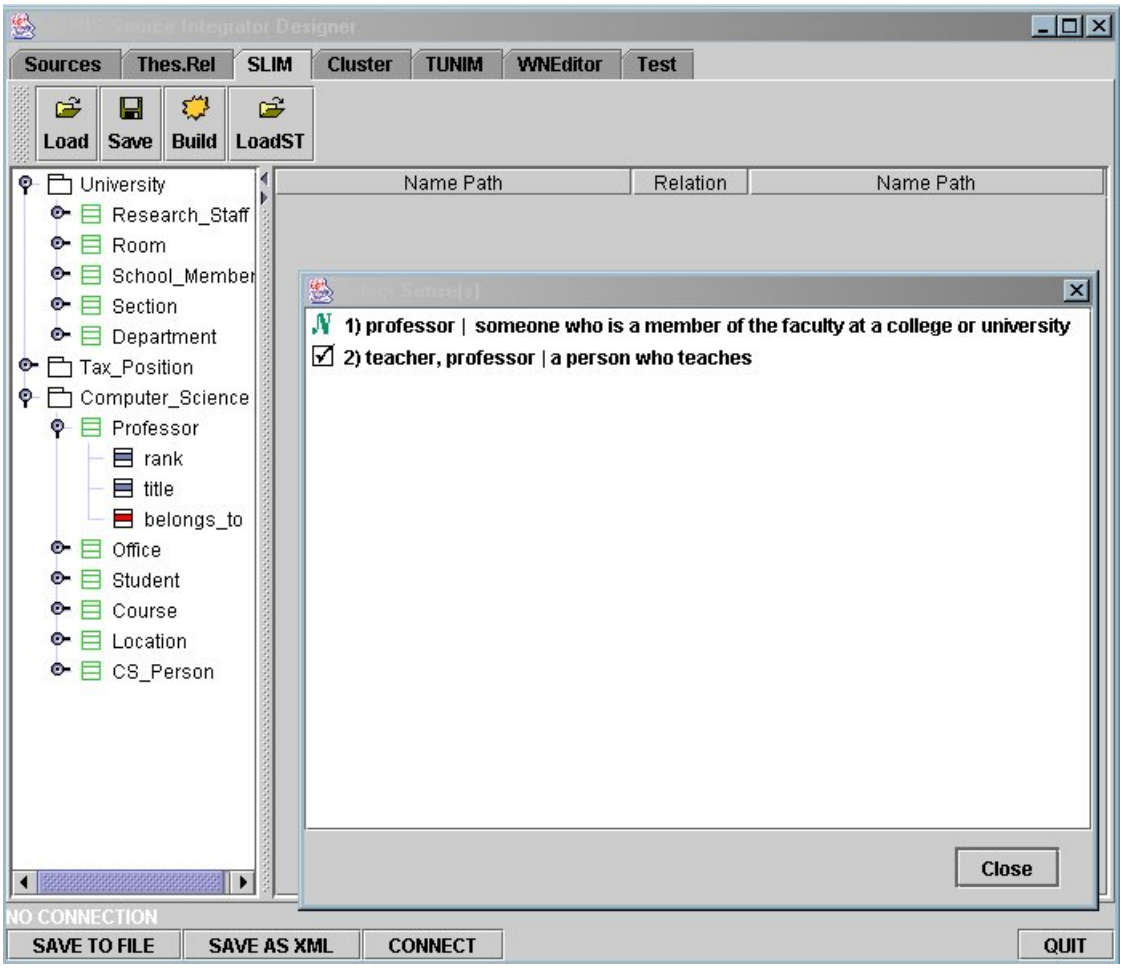

Figura 7.7: Scelta del muovo significato con SLIM

creato dalla classe SlimTree ed inizializzato (sfruttando il metodo inizializza() della classe WNSlim). Inizializzare WNSlim vuol dire caricare in memoria il modulo WNEditor. Avendo progettato in questo modo il software, WNEditor viene caricato in memoria una volta soltanto, rendendo efficiente la sua visualizzazione grafica all'utente (si noti, infatti, che tutti gli oggetti SlimNode, dal cui menu viene visualizzato WNEditor, sono in relazione con lo stesso oggetto WNSlim, e quindi con lo stesso WordNetEditorPanel).

7.4 Principali funzioni ed algoritmi usati

Verranno di seguito spiegate le principali funzioni ed i principali algoritmi che sono stati implementati (nelle classi del package mDBWordNet) al fine di reperire i synsets,

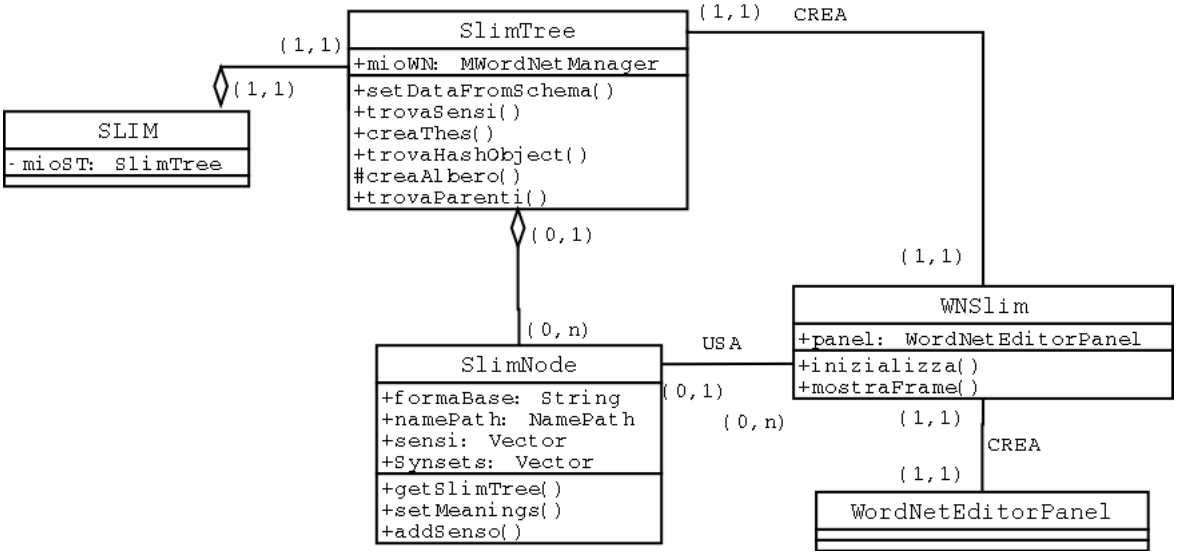

Figura 7.8: Schema UML delle classi per l'integrazione di SLIM e WNEditor

le relazioni e costruire il thesaurus a partire dalle informazioni contenute nel database relazionale MOMISWN. Per poter comprendere alcune scelte compiute, comunque, non è possibile prescindere dalla struttura del database MOMISWN e dalla tecnologia utilizzata per costruire ed interrogare tale database: Torque.

7.4.1 Il database relazionale MOMISWN

La struttura del database MOMISWN è visibile in Figura(7.9), che ne rappresenta lo schema E/R (Entity/Relationship). Ad ogni rettangolo corrisponde un'entità, quindi una tabella nel database MOMISWN, mentre i rombi rappresentano le relazioni che intercorrono fra le entità. Le entità presenti nel database sono le seguenti:

- WN_LEMMA (wn_lemma_id, lemma, syntactic_category, sense_cnt, wn_extender_id): in questa tabella sono contenuti tutti i lemmi, o termini, presenti in WordNet. Il campo sense_cnt rappresenta il numero di synsets collegati al lemma tramite WN_LEMMA_SYNSET.

- WN_SYNSET (wn_synset_id, byte_offset, syntactic_category, word_cnt, gloss, wn_extender_id): tabella contenente tutti i synsets di WordNet. Il campo word_cnt rappresenta il numero di ternimi che formano il synset.

- WN_LEMMA_SYNSET (wn_lemma_synset id, wn_lemma_id, wn_sense_id, lemma_number, sense_number, wn_extender_id): tabella che mette in relazione i

lemmi (contenuti in WN_LEMMA) con i synset (contenuti in WN_SYNSET). In pratica, se un lemma ed un synset compaiono in questa relazione il lemma appartiene al synset specificato.

- WN_RELATIONSHIP_TYPE (wn_relationship_type_id, symbol, description, reflex): tabella contenente I simboli (campo symbol) delle varie relazioni presenti in WordNet (eg. @ rappresenta l'iperminia).

- WN_RELATIONSHIP (wn_relationship_id, wn_source_synset_id, wn_target_synset_id, wn_source_lemma_id, wn_ target_lemma_id, wn_extender_id, wn_relationship_type_id): tabella che contiene tutte le relazioni, sia semantiche che lessicali, contenute in WordNet. Per le relazioni semantiche i campi wn_source_lemma_id e wn_target_lemma_id vengono posti a zero.

- WN_EXTENDER (wn_extender_id, name, description): tabella che contiene tutti gli extender che è possibile usare. Il tipo di extender serve per contraddistinguere gli oggetti caricati nel database dai file originali di WordNet da quelli immessi in un tempo successivo. Non è possibile cancellare o modificare tramite WNEditor gli oggetti marcati come originali (extender WN).

7.4.2 Un accenno alla tecnologia utilizzata: Torque

Torque è un livello di persistenza che genera tutte le risorse di un database richieste da una possibile applicazione Java. Torque, inoltre, fornisce anche il supporto necessario per sfruttare la classi generate a tempo di esecuzione.

Torque è stato sviluppato come parte del progetto Jakarta ed è disponibile gratuitamente in rete presso il sito **http://jakarta.apache.org/turbine/torque/**.

Il software Torque utilizza un singolo documento XML per generare il codice SQL in grado di costruire il database voluto ed alcuni oggetti Java in grado di rappresentare il database relazionale. Tramite questo procedimento è possibile generare anche la documentazione HTML per visionare la schema XML del database. Il prodotto offerto da Torque è un file per Ant (*Apache Ant* è un compilatore Java, simile al compilatore *Make* per codice scritto in C) che può essere aggiunto al progetto. Per quanto riguarda la parte a tempo di esecuzione, Torque offre due tipologie di classi:

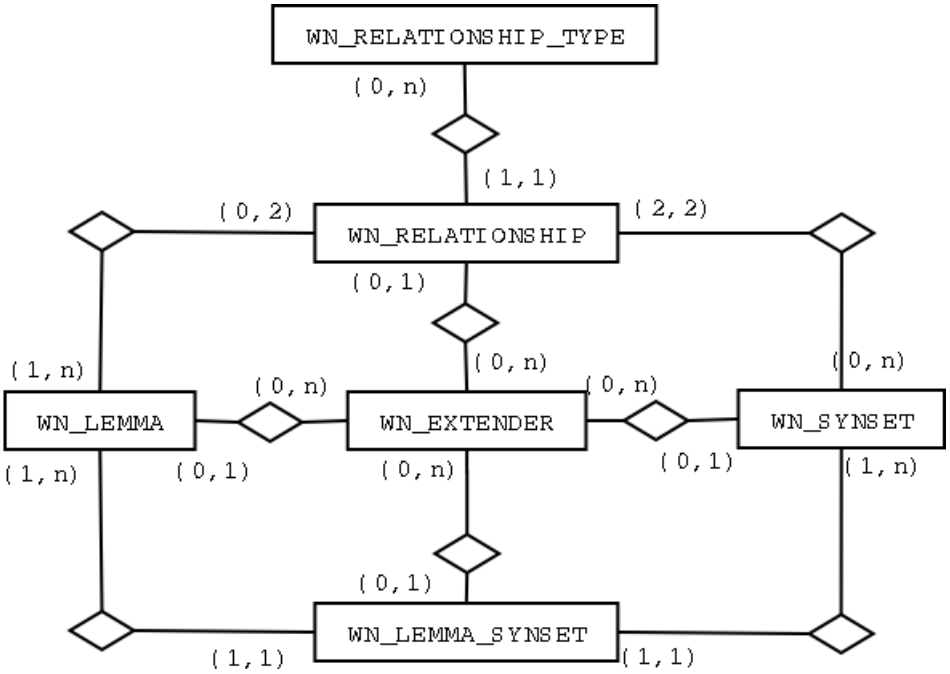

Figura 7.9: Schema E/R del database relazionale MOMISWN

classi di tipo *base* e classi di tipo *peer-base* (costruite automaticamente a partire dal file XML) in grado di fornire operazioni di selezione, cancellazione, inserimento e modifica sulle tebelle del database.

In realtà, per ogni tabella presente nel database, Torque genera quattro classi distinte; supponendo che una tabella si chiami WnSynset la classi generate sono: BaseWnSynset, BaseWnSynsetPeer, WnSynset e WnSynsetPeer. Non è possibile aggiungere codice proprio alle prime due classi, poiche esse vengono sovrascritte ad ogni nuova compilazione tramite *Ant*, ma è possibile farlo nelle classi WnSynset e WnSynsetPeer che ereditano, rispettivamente, da BaseWnSynset e BaseWnSynsetPeer. La principale differenza fra il funzionamento delle classi di tipo *peer* e quelle di tipo *data* (non *peer*) è che le prime lavorano sulle intere tabelle del database fornendo operazioni di inserimento, selezione e cancellazione, mentre le seconde lavorano sulle singole righe delle tabelle (in pratica lavorano sulle singole tuple ritrovate tramite gli oggetti *peer*).

Il file XML contenente la descrizione del database MOMISWN si trova nel file *momiswn-schema.xml*. Un estratto da tale file è il seguente, rappresentante la tabella WN_SYNSET:

```
<table name="WN_SYNSET" description="WN_synset" idMethod="null"
    skipSql="false" abstract ="false">
  <column name="WN_SYNSET_ID" required="true" primaryKey="true"
    autoIncrement="true" type="INTEGER" description="Synset_Id"
    inheritance ="false" />
  <column name="BYTE_OFFSET" required="false" type="INTEGER"
    description ="Synset_offset" primaryKey="false"
    autoIncrement="false" inheritance ="false" />
  <column name="SYNTACTIC_CATEGORY" required="true" type="INTEGER"
    description ="Syntactic_category" primaryKey="false"
    autoIncrement="false" inheritance ="false" />
  <column name="WORD_CNT" required="false" type="INTEGER"
    description ="Total_number_of_synonyms_in_this_synset"
    primaryKey="false" autoIncrement="false" inheritance ="false" />
  <column name="GLOSS" required="true" type="LONGVARCHAR"
    description ="Synset_gloss" primaryKey="false"
    autoIncrement="false" inheritance ="false" />
  <column name="WN_EXTENDER_ID" required="false" type="INTEGER"
    description ="Origin_of_this_record_(who_is_the_extender)"
    primaryKey="false" autoIncrement="false" inheritance ="false" />
  <foreign-key foreignTable="WN_EXTENDER" name="WN_SYNSET_EXTENDER_FK"
    onUpdate="none" onDelete="none">
    <reference local ="WN_EXTENDER_ID" foreign="WN_EXTENDER_ID/>
  </foreign-key>
</table>
```

Gli oggetti Java (*peer* e non) generati da Torque si trovano all'interno del package *om*, utilizzato dalla nuova libraria di funzioni del package mDBWordNet al fine di interfacciare SLIM con il database MOMISWN, si veda Figura(7.7). Tramite tale libraria di funzioni si è utilizzato, al fine di porre query sul database, l'oggetto Criteria, fornito anch'esso da Torque.

Si supponga, ad esempio, di voler selezionare tutti gli oggetti, appartenenti alla tabella WN_LEMMA, il cui campo lemma è pari a 'professor'. La relativa query SQL risulta essere:

```
select * from WN_LEMMA where lemma = 'professor'
```

All'interno di un programma Java, sfruttando il costrutto Criteria, è possibile esprimere la medesima query tramita il codice:

```
Criteria  c = new Criteria ();
c.add(WnLemmaPeer.LEMMA,"professor");
List  l = WnLemmaPeer.doSelect(c );
```

Ogni tupla che rappresenta il risultato della query è inserita nell'oggetto *List l.*

(tutte le query citate nei prossimi paragrafi saranno, anche se espresse in SQL, da intendersi come create tramite tale metodo)

7.4.3 Funzioni ed algoritmi

Si è mostrato, nei paragrafi precedenti, come SLIM richieda le informazioni volute al database MOMISWN tramite la classe MDB_WordNet. Questa classe, per svolgere i compiti richiesti, utilizza le classi della libreria di funzioni, prodotta anch'essa nell'ambito di questa tesi, del package mDBWordNet. In Figura(7.10) è rappresentata l'interazione fra MDB_WordNet e le principali classi del modulo mDBWordNet tramite un diagramma delle classi UML.

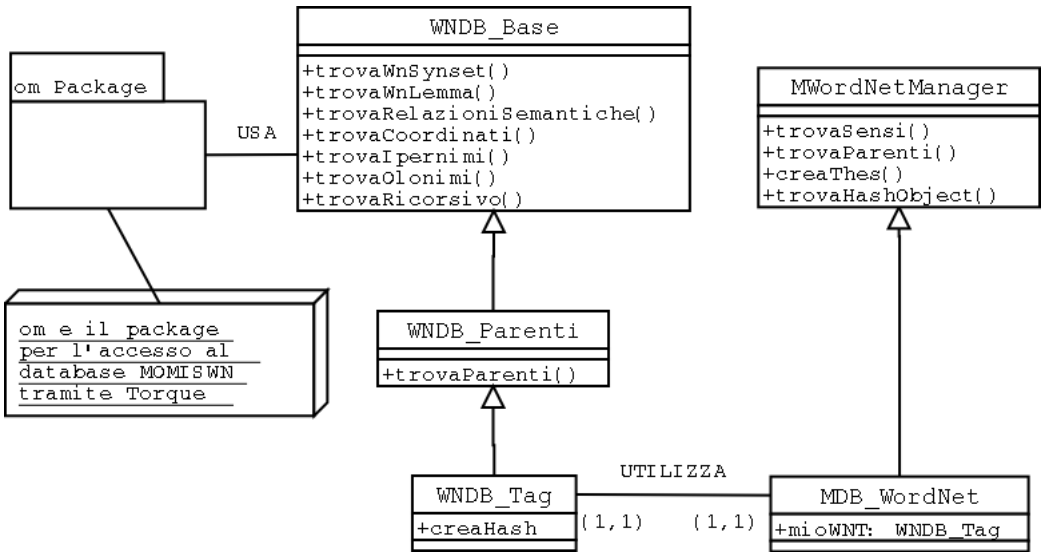

Figura 7.10: Schema E/R del database relazionale MOMISWN

Si può notare come la classe MDB_WordNet utilizzi una istanza dell'oggetto WNDB_Tag. La classe WNDB_Tag ha il compito di creare la tabella Hash (funzione *creaHash()*) che fornisce la base per poter creare il thesaurus di relazioni inter-schema, prodotto finale del modulo SLIM. WNDB_Tag eredita i metodi sia dalla classe WNBD_Parenti che da WNDB_Base. Per poter analizzare l'algoritmo implementato dalla funzione *creaThes()* di MDB_WordNet è necessario comprendere, in via preliminare, le funzioni principali situate nelle classi del package mDBWordNet.

Funzioni della classe WNDB_Base La classe WNDB_Base implementa i metodi

di base per l'interazione con il package om, contenente le classi create da Torque (come le classi WnSynset, WnLemma, WnSynsetPeer ... che utilizzano le relative tabelle del database MOMISWN). Le sue principali funzioni sono:

public WnSynset synset2WnSynsetNew(Synset s): trasforma un oggetto Synset (classe appartenente al package wn2Slim) nel corrispondente oggetto WnSynset (package om)

public WnSynset2SynsetNew(WnSynset WnS): trasforma un oggetto WnSynset in uno Synset.

public Vector trovaWnLemma(WnSynset WnS): ritorna un vettore contenente tutti i lemmi che formano il synset *WnS* (utilizza la tabella WN_LEMMA_SYNSET).

public Vector trovaWnSynset(String word): ritorna un vettore contenente ogni synset contenente il lemma *word* (utilizza la tabella WN_LEMMA_SYNSET).

public Vector trovaRelazioniSemantiche(Synset s, String tipo): ritorna un vettore contenente tutti i synsets collegati tramite il synset *s* dalla relazione semantica rappresentata dal simbolo tipo (utilizza la tabella WN_RELATIONSHIP).

public Vector trovaCoordinati(Synset inpSynset): ritorna un vettore contenente i synsets ipomini dello stesso ipernimo di *inpSynset*.

public Vector trovaIpernimi(Synset inpSynset): trova tutti gli ipernimi del synset *inpSynset* (sfrutta la funzione trovaRicorsivo()).

public Vector trovaOlonimi(Synset inpSynset): trova tutti gli olomini del synset *inpSynset* (sfrutta la funzione trovaRicorsivo()).

public Vector trovaRicorsivo(Synset inpSynset, String tipo): trova ricorsivamente tutti i synsets collegati a *inpSynset* tramite la relazione avente come simbolo *tipo*

Funzioni della classe WNDB_Parenti Classe per cercare i synsets semanticamente equivalenti ad un synset dato.

public Vector trovaParenti(Synset mpS, String word): chiama la funzione *trovaWnSynset(word)* per trovare tutti i synsets contenenti il lemma *word* (anche il synset mpS deve contenere word) il risultato di trovaWnSynset viene posizionato in un vettore chiamato *vTemp*. Da *vTemp* viene eliminato *mpS*. Viene creato un vettore *vParenti* che conterrà il risultato della funzione *trovaParenti()*. A *vParenti* si aggiungono i synset appartenenti a *vTemp* che soddisfano una di queste due condizioni:

1. hanno almeno tre lemmi uguali a quelli di *mpS*
2. hanno lo stesso ipernimo di *mpS*

Si aggiunge a *vParenti* il Synset *mpS*. Si ritorna il vettore *vParenti* contenente i synsets considerati semanticamente equivalenti a *mpS*.

Funzioni della classe WNDB_Tag La classe WNDB_Tag serve per creare la struttura necessaria, contenuta in una tabella Hash, per l'algoritmo di creazione del thesaurus.

public Hashtable creaHash(Vector vNamePath): questa funzione viene invocata usando come paramentro un vettore di NamePath (classe del package wn2Slim). Ogni NamePath contiene un oggetto Synset *s* ed una stringa (chiamata *namePath*) rappresentante il path dell'oggetto SlimNode dalla cui annotazione è stato selezionato *s*. Alla funzione creaHash() viene passato un vettore contenente tutti i NamePath ottenuti dalla fase di annotazione (uno per ogni oggetto annotato). La tabella Hash viene costruita nella seguente maniera: Per ogni synset ottenuto dalla fase di annotazione (contenuto in un NamePath)

1. si chiama la funzione *trovaParenti()* e si cercano i suoi synset semanticamente equivalenti.
2. per ogni parente trovato, se non è già presente nella tabella Hash si aggiunge una entry formata da ((chiave)synset parente, (valore)*namePath* corrispondente); se è già presente si aggiunge un elemento al vettore dei namePaths corrispondenti.

In questo modo si ottiene una tabella in cui ogni entry ha come chiave un synset ottenuto dalla fase di annotazione, e come valore un vettore contenente tutti i path degli oggetti i cui nomi possiamo considerare *sinonimi (relazione SYN nel thesaurus)*.

Algoritmo per la creazione del thesaurus (funzione creaThes())

Questo algoritmo serve per trovare un insieme di relazioni fra i path dei vari nodi degli schemi rappresentanti le sorgenti locali di dati. Ogni nodo è un oggetto SlimNode, può essere annotato, possiede una forma base (termine scelto per rappresentarlo) ed un attributo di tipo NamePath contenente il suo path. L'algoritmo si comporta nella seguente maniera:

1. si chiama la funzione *creaHash()* con parametro un vettore contenete tutti i *NamePath* degli SlimNode annotati
2. per ogni entry nella tabella così creata
   - si crea una relazione di sinonimia fra gli elementi (namePaths) associati alla chiave attuale
   - si cercano gli iperonimi del synset rappresentante la chiave attuale e, per ogni iperonimo, si cerca la sua esistenza nella tabella. Se esiste si scrive una relazione di tipo NT per ogni elemento ad esso associato.
   - si cercano gli olomini del synset rappresentante la chiave attuale e, per ogni olonimo, si cerca la sua esistenza nella tabella. Se esiste si scrive una relazione di tipo RT per ogni elemento ad esso associato.
   - si cercano i coordinati del synset rappresentante la chiave attuale e, per ogni coordinato, si cerca la sua esistenza nella tabella. Se esiste si scrive una relazione di tipo RT per ogni elemento ad esso associato.

7.5 Il processo di ottimizzazione

La nuova versione del modulo SLIM, utilizzante il database relazionale MOMISWN, risulta essere più lenta della versione di SLIM implementata originariamente da Giovanni Malvezzi. La causa di questa perdita di performance è dovuta alla struttura stessa del database MOMISWN: tale database, infatti, è stato progettato e costruito con lo scopo di essere facilmente estensibile e comprensibile, non per fornire migliori

prestazioni rispetto ai files originali di WordNet (il database MOMISWN, infatti è stato progettato in forma minima). Basti pensare che, per ogni categoria sintattica (nomi, verbi, aggettivi, avverbi) WordNet prevede solamente due files:

- un file che rappresenta un indice inverso costruito sui lemmi (per ogni lemma vengono elencati tutte le relative posizioni (byte-offset) dei synsets in cui compare).
- un file che contiene tutti i synsets. Per ogni synset vengono specificati: i lemmi che lo compongono; tutte le relazioni (sia semantiche che lessicali) con gli altri synsets; la Gloss (definizione a parole del concetto rappresentato).

Si può ben comprendere come, tramite solamente due accessi a tali files, la versione precedente di SLIM fosse in grado di reprire tutte le informazioni di interesse riguardo ad uno specifico synset.

Supponiamo ora di voler, utilizzando il database MOMISWN, trovare tutti i synsets conteneti il lemma *student*. La query da compiere è la seguente:

```
select * from WN_LEMMA l, WN_SYNSET s, WN_LEMMA_SYNSET ls
  where
      l.LEMMA = 'student'
      AND l.WN_LEMMA_ID = ls.WN_LEMMA_ID
      AND s.WN_SYNSET_ID = ls.WN_SYNSET_ID
```

Supponendo che student sia contenuto in due synsets differenti, tramite i files originali di WordNet è possibile avere le informazioni volute in tre accessi solamente: uno per l'indice e due nel file contenente i synsets. Sfruttando il database MOMISWN, e quindi la query citata in precedenza, gli accessi a disco (ritenendo le informazioni non già immagazzinate in memoria RAM) che occorrono sono più di tre: occorrono un accesso alla tabella WN_LEMMA, due alla tabella WN_LEMMA_SYNSET e due alla tabella WN_SYNSET. Si hanno così due accessi in più, si deve, inoltre, tenere presente che tramite la query non si hanno informazioni sulle relazioni o sugli altri lemmi componenti i synsets reperiti; tali informazioni devono essere trovate al costo di altri, numerosi, accessi al disco.

Per porre rimedio a questo problema e stato studiato un procedimento di ottimizzazione delle operazione di estrazione delle informazioni comprendente due punti fondamantali:

- Creazione di opportuni indici sul database MOMISWN
- Creazioni di caches per non estrarre mai due volte la stessa informazione dal disco fisso

Verranno analizzati di seguito i due punti elencati in precedenza.

7.5.1 Creazione di indici

La creazione di indici sul database MOMISWN ha incrementato notevolmente le prestazioni del software che, pur non essendo pari a quelle della prima versione di SLIM, riducono lo scarto da un fattore di circa dieci (senza indici, lo SLIM utilizzante il database è circa dieci volte più lento rispetto a quello utilizzante i files di WordNet) ad un fattore di due o tre (in dipendenza delle diverse procedure da effettuare).

Si tenga presente che il database è stato implementato sfruttando MySql, DBMS che non crea automaticamente nessun indice, neppure sulle chiavi primarie delle varie tabelle.

Gli indici da creare nel database sono stati scelti in base alle query sfruttate nei metodi delle classi del package mDBWordNet, creati proprio allo scopo di reperire le informazioni necessarie a SLIM nella creazione del thesaurus. Inoltre, al fine di considerare opportuna la realizzazione di un indice, i costi di modifica (update, insert e delete) del database MOMISWN usando il browser WNEditor sono stati considerati trascurabili, ritenendo le operazioni di selezione di gran lunga più frequenti.

Di seguito verranno elencate le funzioni (appartenenti al package mDBWordNet, si veda Figura()) in base alle quali sono stati decisi gli indici; assieme ad esse verranno elencate la query SQL che implementano (si ricorda che in realtà il codice SQL non è presente, ma è stato sfruttato il software Torque) ed gli indici creati. (NOTA: Anche per gli indici, come per le tabelle del database MOMISWN, è possibile la generazione automatica del codice SQL a partire dalla loro definizione nello schema XML usato da Torque; il formato con cui è possibile produrre un indice in questa maniera è:

```
<index name = ”nome”>
    <index−column name=”colonna_di_indice”/>
</index>
```

il precedente codice è da inserire all'interno della descrizione XML della tabella desiderata; è possibile specificare più colonne per l'indice all'interno del tag index.)

public WnSynset synset2WnSynsetNew(Synset s)

query: select * from WN_SYNSET
where WN_SYNSET_ID = s.byte−offset

tabella :
WN_SYNSET

index:
<index name = " synset_id_index ">
<index−column name="WN_SYNSET_ID"/>
</index>

public Vector trovaWnLemma(WnSynset WnS)

query: select l.* from WN_LEMMA_SYNSET ls, WN_LEMMA l where
ls.WN_SYNSET_ID = WnS.getwnSynsetIn()
AND l.WN_LEMMA_ID = ls.WN_LEMMA_ID

tabella :
WN_LEMMA

index:
<index name = "lemma_Id_index">
<index−column name="WN_LEMMA_ID"/>
</index>

tabella :
WN_LEMMA_SYNSET

index:
<index name = "lemma_synset_synsetId_index">
<index−column name="WN_SYNSET_ID"/>
</index>

public Vector trovaWnSynsetUno(String word, int pos)

query : select l.* from WN_LEMMA_SYNSET ls, WN_LEMMA l, WN_SYNSET s
where l.LEMMA = 'word'
AND l.WN_LEMMA_ID = ls.WN_LEMMA_ID
AND ls.WN_SYNSET_ID = s.WN_SYNSET_ID

tabella :
WN_LEMMA

index:
<index name = "lemma_lemma_index">
<index−column name="LEMMA" size="10"/>
</index>

public Vector trovaRelazioniSematiche(Synset s ,Stringtipo)

query : select r.WN_TARGET_SYNSET_ID from WN_RELATIONSHIP r where
r.WN_SOURCE_SYNSET_ID = s.byte−offset
AND r.WN_SOURCE_LEMMA_ID = 0
AND r.WN_TARGET_LEMMA_ID = 0

tabella :
WN_RELATIONSHIP

index:
<index name = " relationship_sourceSynset_index ">
<index−column name="WN_SOURCE_SYNSET_ID"/>
</index>

7.5.2 Creazione e gestione delle caches

Allo scopo di ottimizzare in maniera consistente la velocità di risposta di SLIM alle richieste dell'utente, sono state introdotte alcune caches che immagazzinano i dati recuperati dal database MOMISWN. Le caches fondamentali a questo scopo sono due:

- *la cache dei lemmi* (oggetto Java HashMap di nome LemmaMap)
- *la cache delle relazioni* (oggetto Java HashMap di nome RelMap)

La cache dei lemmi è una tabella Hash strutturata in questo modo: ad ogni entry della tabella corrisponde un valore di chiave che rappresenta l'identificatore univoco di un synset s, ed un valore collegato a tale chiave, rappresentato da un oggetto formato da un vettore stringhe (sono tutti i lemmi che compongono il synset s) e dalla gloss di s. Le cache dei lemmi viene sfruttata dalla funzione *trovaWnLemma(WnSynset WnS)* che, prima di interrogare il database, controlla l'esistenza delle informazioni cercate nella tabella Hash. Se le informazioni cercate non sono reperibili all'interno della cache vengono cercate nel database e successivamente inserite nella tabella Hash.

L'utilizzo della cache delle relazioni è molto simile a quello della cache dei lemmi. All'interno di questa tabella Hash ad ogni entry corrisponde un valore di chiave, rappresentante sempre l'identificatore univoco di un synset *s*, ed un valore collegato alla chiave formato da un vettore contenente tutti gli identificatori dei synsets collegati (tramite relazioni semantiche) al synset *s*. La cache delle relazioni viene utilizzata dalla funzione *trovaRelazioniSemantiche(Synset s, String tipo)* che, prima di cercare nel database MOMISWN le relazioni semantiche di *s*, controlla la loro esistenza nella tabella Hash.

Entrambe queste tabelle Hash vengono salvate in files (rel.map e lem.map) situati nella directory *momis/var* che possono essere eliminati senza problemi. Questi files, se vengono ritrovati, vengono caricati in memoria ad ogni utilizzo del software SIDesigner.

Un problema che si è dovuto risolvere per conciliare buone prestazioni ed affidabilità del software è stato implementare un metodo affinche le caches rimanessere sempre consistenti con il database MOMISWN. Ricordiamo, infatti, che il database può essere modificato tramite l'uso del browser WNEditor; possono essere creati nuovi lemmi e nuovi synsets (bisogna modificare la cache dei lemmi) e possono essere create anche nuove relazioni (bisogna modificare la cache della relazioni). Tutte le operazioni per il mentenimento della consistenza nelle due caches vengono effettuate dalla funzione *mantieniCoerenza()* situata all'interno della classe WNSlim che funge, come si puo notare in Figura(), da interfaccia fra SLIM e WNEditor. Il metodo *mantieniCoerenza()* viene eseguito ogni volta che si chiude la finestra contenente WNEditor.

Capitolo 8

Un modulo per la riscrittura dei path

8.1 Introduzione

In questo capitolo viene presentato un modulo software per risolvere il problema del confronto fra i path specificati in una query (eg. XQuery) e quelli contenuti negli schemi dei documenti digitali di un archivio.

Riprendiamo brevemente il discorso riguardante il problema della riscrittura delle query, descritto all'interno del primo capitolo: un utente vuole porre una query su un grande deposito di documenti digitali XML, ma non è a conoscenza della struttura di tutti i documenti dell'archivio. Chiaramente, la probabilità di porre una query in grado di soddisfare le reali aspettative dell'utente è molto bassa. Tale richiesta deve, quindi, essere automaticamente riscritta per poter essere formulata su ogni documento ritenuto di interesse.

Come scegliere i documenti di interesse per cui riformulare una query?

Un grande aiuto, in questo senso, può essere dato dai metadati, ovvero, nel caso di un archivio di documenti in formato XML, da schemi sullo stampo di Xml-Schema. Questi schemi possono essere tradotti in ODL_{I^3}, da appositi wrapper, per interagire col sistema MOMIS. MOMIS, ed in particolare il modulo detto SLIM, è in grado, interagendo col database lessicale WordNet, di arricchire il contenuto semantico di ogni schema. Tale 'incremento semantico' viene svolto attraverso la *fase di annotazione*, in cui è possibile associare un significato, estratto da WordNet, ad ogni oggetto contenuto in ogni schema ODL_{I^3}. Il modulo SLIM, inoltre, è in grado di creare, sempre

per mezzo dell'interazione con WordNet, un thesaurus di relazioni inter-schema; proprio questo thesaurus può essere ritenuto il punto di partenza per una buona fase di riscrittura delle query.

Il modulo software descritto nei prossimi paragrafi è in grado, dato un *path* in ingresso, di selezionare i percorsi più simili riscontrati all'interno di tutti path contenuti in ogni schema ODL_{I^3} annotato (e catalogarli in base al loro grado di similarità). Questa operazione è effettuata tramite l'applicazione di un algoritmo di Edit Distance modificato in modo da considerare le relazioni (appartenenti al thesaurus prodotto da SLIM) fra gli elementi di due path.

Vediamo ora, riprendendo per l'occasione l'esempio citato nel primo capitolo di questa tesi (e modificandolo leggermente), come un modulo proposto per la selezione dei path semanticamente simili può essere sfruttato al fine della riscrittura delle query. In un archivio di documenti XML, due tipologie di questi sono rappresentate visivamente dagli schemi di Figura (8.1) (la figura esprime tramite un grafico la struttura ad albero naturalmente espressa tramite schemi come Xml-Schema).

La query posta dall'utente (interessato ai titoli dei capitoli dei libri il cui argomento è 'XML') è la seguente:

```
for $a in /Trattato/Capitolo
   where $a/Argomento = 'XML'
   return $a/Titolo
```

(si rammenta che l'utente non è a conoscenza della struttura dei documenti in archivio).

La query espressa in precedenza può essere ritenuta composta da due percorsi (ottenuti sostituendo il valore della variabile $a dove richiesto):

- /Trattato/Capitolo/Argomento
- /Trattato/Capitolo/Titolo

Sono proprio questi due path a selezionare i documenti di interesse per la richiesta dell'utente. Osservando la Figura(8.1), però, possiamo facilmente notare che (supponendo non vi siano altri schemi concernenti libri o trattati che descrivono i documenti presenti in archivio) nessun documento potrà essere selezionato tramite i path

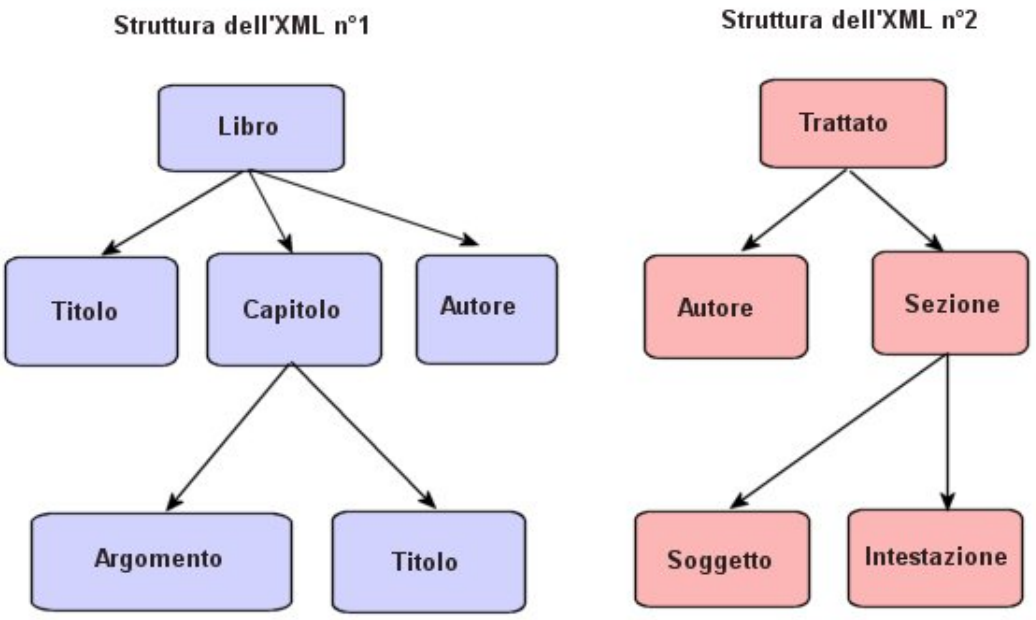

Figura 8.1: Schemi di due documenti XML in archivio

componenti questa query. La richiesta espressa dall'utente, comunque, risulta essere piuttosto chiara e tramite l'utilizzo degli schemi semanticamente arricchiti (per mezzo della fase di annotazione) vedremo come sarà possibile riformularla correttamente per i documenti esistenti.

Abbiamo osservato che una query può essere vista come un insieme di percorsi, o path. Se riuscissimo a trovare, per ogni percorso espresso nella query, i percorsi semanticamente simili degli schemi, potremmo facilmente utilizzarli per riscrivere la query in base alla reale struttura dei documenti stessi.

Il modulo software proposto in questo capitolo svolge esattamente la funzione richiesta basandosi, oltre che su un algoritmo di Edit Distance, sulle relazioni inter-schema ottenute dal modulo SLIM di MOMIS(ed eventualmente modificabili dal progettista). Se, infatti, consideriamo che gli schemi (eg. Xml-Schema) dei due documenti siano stati tradotti in schemi ODL_{I^3} per mezzo di wrappers appositi e, successi-

vamente, annotati in maniera appropriata, è possibile ottenere all'interno del thesaurus di relazioni inter-schema (prodotto da SLIM) ad esempio:

/ Trattato /Sezione SYN /Libro/Capitolo

/Libro SYN /Trattato

/ Trattato / Capitolo /Argomento SYN /Libro/Capitolo/Soggetto

/ Trattato / Capitolo / Intestazione SYN /Libro/Capitolo/ Titolo

Tali relazioni legano strettamente i path citati, poichè, essendo classificati come sinonimi, i significati di *Libro* e *Trattato*, *Argomento* e *Soggetto*, *Intestazione* e *Titolo* , *Sezione* e *Capitolo* risultano identici (hanno, cioè, lo stesso synset assegnato nella fase di annotazione). Basandosi su un thesaurus di relazioni di questo genere risulta abbastanza evidente come da un path quale */Trattato/Capitolo/Argomento* possano essere reperiti i percorsi (esistenti negli schemi XML) */Libro/Capitolo/Argomento* e */Trattato/Capitolo/Soggetto*. Infatti, tramite l'utilizzo degli schemi annotati e del thesaurus possiamo risalire, senza problemi, dal termine *Trattato* a *Libro*, e da *Argomento* a *Soggetto*.

Appare ora chiaro come, tramite l'estensione semantica del contenuto dei metadati impiegati, risulti agevole il confronto semantico dei path espressi in una query con quelli contenuti negli schemi, e, quindi, la riscrittura approssimata delle query.

Nei prossimi paragrafi sarà illustrato più dettagliatamente il procedimento per il confronto fra i path, compresi gli algoritmi e gli accorgimenti adottati per implementarlo.

(NOTA: L'esempio precedentemente illustrato in Figura(8.1) non può essere implementato direttamente tramite MOMIS: WordNet è, infatti, solamente in lingua Inglese ed i termini usati dovrebbero, prima, essere tradotti in tale idioma.)

8.2 Un metodo sintetico per la rappresentazione di path

Il metodo per la rappresentazione sintetica di path mostrato in questo paragrafo è stato implementato dal professor P.Tiberio dell'Università di Modena e dai professori

P.Ciaccia e D.Maio dell'Università di Bologna, in [39].
Tale metodo è stato pensato e progettato non per l'ambito di cui tratta questa tesi, ma nell'ambito dei DBMS relazionali, al fine di migliorare la possibilità, da parte di tali sistemi, di gestire gerarchie di oggetti. L'idea adottata è quella di non immagazzinare la chiave della tupla padre, ma creare una codice univoco da cui sia possibile risalire a tutto il percorso dei predecessori della tupla. L'utilità di questo metodo in un modulo di confronto fra path è data dalla possibilità di poter immagazzinare, tramite pochissimi bytes, un percorso che può essere anche molto lungo, migliorandone l'efficienza (basta sapere il codice di un oggetto per sapere anche tutti quelli dei suoi predecessori nello schema di un documento).

Il metodo con cui il codice associato al un percorso viene creato è strettamente collegato all'utilizzo delle *simple continued fractions (SICF)*[34, 40] Una SICF è un'espressione matematica che assume la seguente forma:

$$\cfrac{p_1}{q_1+\cfrac{p_2}{q_2+\cfrac{p_3}{q_3+\ldots}}}$$

Le frazioni continue di interesse, comunque, assumono una forma particolare per soddisfare alle seguenti proprietà:

- il numero di termini p_i e q_i è finito
- $p_i = 1$ (i = 1,2,3...m)
- tutti i q_i sono interi maggiori di zero

La forma assunta dalle SICF che rispettano le proprietà elencate è la seguente:

$$\cfrac{1}{q_1+\cfrac{1}{q_2+\cfrac{1}{q_3+\cfrac{1}{q_a+\ldots\cfrac{1}{q_{m-1}+\cfrac{1}{q_m}}}}}}$$

I termini q_i vengono chiamati quozienti parziali della SICF. Per questione di comodità una SICF può essere rappresentata per mezzo della forma:

$[q_1, q_2, q_3, \ldots, q_{m-1}, q_m]$

Ogni SICF gode della seguenti proprietà:

1. *Proprietà 1*: Ogni SICF è l'espansione di un numero razionale N/D (N/D = $[q_1, q_2, q_3, ..., q_{m-1}, q_m]$) assumendo N e D numeri interi primi fra loro. Vice versa, ogni numero razionale N/D ($0 < N/D < 1$) può essere univocamente rappresentato da una SICF solamente se l'ultimo termine q_m è maggiore di 1.

2. *Proprietà 2*: A causa dell'unicità dell'espansione di una SICF, è possibile associare un numero razionale N_i/D_i (i = 1,2,.., m) alla SICF parziale di ordine i, in modo che:

   $N_i/D_i = [q_i..., q_m]$

   in cui $N_m/D_m = 1/q_m$ e $N_1/D_1 = N/D$.//

   Il numero razionale associabile alla SICF di ordine i è il seguente:

   $N_i/D_i = \frac{1}{N_{i+1}/D_{i+1}}$

   in cui $D_{i+1} = N_i$, per cui è possibile ottenere che:

   $q_i = D_i \; div \; N_i$

   $N_{i+1} = D_i \; mod \; N_i$

Vedremo ora come le SICF possono essere utilizzate per rappresentare i percorsi (*path*) di strutture ad albero:

I passi fondamentali da seguire sono i seguenti:

- Alla radice dell'albero è assegnato un numero 1/S ($S > 1$)

- Ad ogni nodo figlio si assegna un numero intero positivo; tale numero, che chiameremo n, può essere ottenuto, ad esempio, numerando i figli, da sinistra a destra, tramite la sequenza dei numeri naturali (1,2,3...). I numeri associati ai figli di un nodi possono essere ottenuti anche secondo differenti criteri, ma due figli dello stesso nodo non possono avere due numeri n uguali.

- ad ogni nodo dell'albero è associato, in dipendenza da *n*, il numero razionale definito *codice*:

  $codice(NODO) = \frac{1}{n+codice(NODO-PADRE)}$

  Se, ad esempio, ad un nodo, padre di tre figli, è associato il codice 11/30, i codici dei suoi figli vengono ottenuti (supponendo che i numeri *n* siano, rispettivamente 1, 2 e 3) da:

  $\frac{1}{1+\frac{11}{30}} = 30/71$; $\frac{1}{2+\frac{11}{30}} = 30/41$; $\frac{1}{3+\frac{11}{30}} = 30/101$

Allo scopo di comprendere come questi codici possano essere usati per ritrovare i codici appartenenti a tutti i predecessori di un nodo (recuperando cioè l'intero path a partire dal codice di un nodo), si consideri la Figura(8.2).

In questa figura è rappresentato una struttura ad albero (lo schema di un documento XML) ed i codici che vengono generati sfruttando le SICF. Si supponga di voler risalire partendo dal codice associato all'elemento *Argomento* a quelli associati ai suoi predecessori. Usando la proprietà 2 delle SICF otteniamo che:

$N_1/D_1 = 5/7$ (codice del nodo *Argomento*) $q_1 = 1(7\ div\ 5)$

$N_2/D_2 = 2/5$ (codice del nodo *Capitolo*) $q_2 = 2(5\ div\ 2)$ il denominatore è ottenuto dal numeratore del nodo padre, mentre il denominatore è il resto della divisione fra il denominatore ed il numeratore del nodo padre

$N_3/D_3 = 1/2$ (codice del nodo *Libro*) $q_3 = 2(2\ div\ 1)$ il denominatore è ottenuto dal numeratore del nodo padre, mentre il denominatore è il resto della divisione fra il denominatore ed il numeratore del nodo padre

L'espansione della SICF è rappresentata da N/D = [1,2,2], mentre la sequenza dei codici (ovvero il path di *Argomento*) è: 5/7; 2/5; 1/2.

8.3 La metrica di similarità Edit Distance

Il metodo impiegato, nell'ambito di questa tesi, per identificare il grado di similarità che intercorre fra due path è basato su un algoritmo costruito sul concetto di *edit*

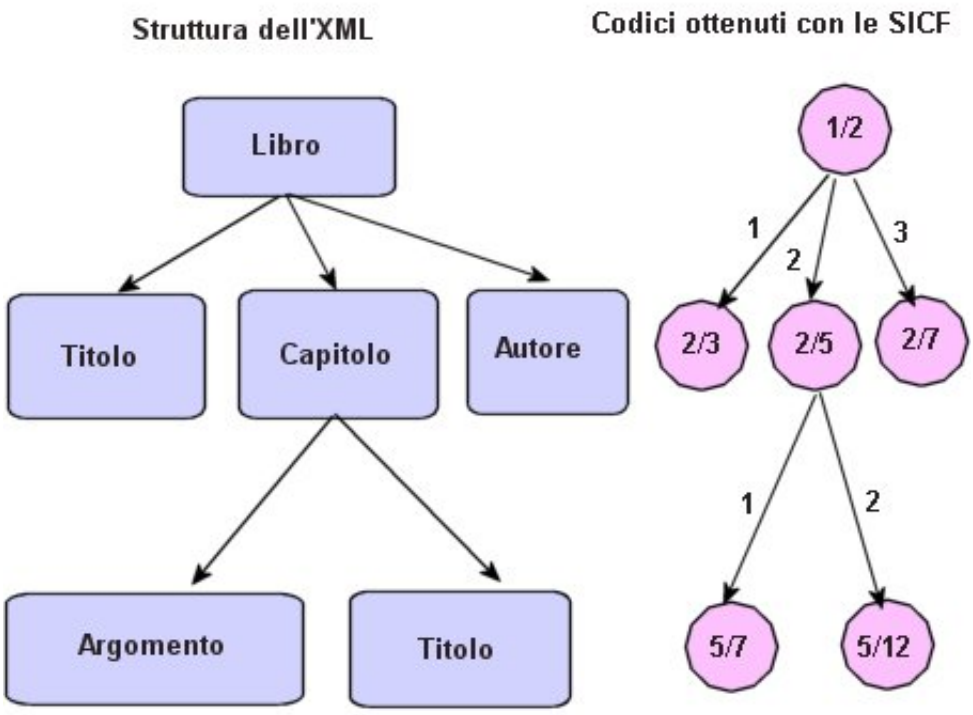

Figura 8.2: Codici collegati ad un albero

distance. La prima cosa da fare è, dunque, comprendere cosa funziona la metrica di similarità basata sull'*edit distance* e per cosa, di solito, viene utilizzata.

L'*edit distance* nasce come metodo per il confronto fra due stringhe di caratteri. Normalmente, in ogni linguaggio di programmazione, vengono proposti algoritmi per il confronto totale di due stringhe, in grado, quindi, di verificarne la perfetta uguaglianza. L'edit distance, invece, è in grado di dare una definizione del concetto di similarità fra due stringhe di caratteri, dunque, basandosi su di esso, si è in grado di identificare parole o frasi 'approssimativamente uguali' fra loro.

Una definizione formale della nozione classica di edit distance (quindi applicato alle stringhe di caratteri) è la seguente:

Definizione: *L'edit distance ed(s_1, s_2) fra due stringhe s_1 ed s_2 è il minimo costo della sequenza di operazioni che trasforma s_1 in s_2. Il costo della sequenza di operazioni è la somma dei costi delle operazioni individuali. Le operazioni sono un insieme finito di regole nella forma $\delta(s_3, s_4) = c$, dove s_3 ed s_4 sono due stringhe diverse e c è un numero reale non negativo. Una volta che una operazione ha convertito una sottostringa s_3 in una s_4, nessuna ulteriore operazione può essere svolta su s_4.*

Per il confronto fra due stringhe di caratteri le operazioni possibili sono le seguenti tre, ognuna delle quali ha un costo di applicazione unitario:

- *Inserimento*: inserire una lettera in una stringa
- *Cancellazione*: eliminare una lettera da una stringa
- *Sostituzione*: sostituire una lettera della stringa s_2 con una appartenente alla stringa s_1

8.3.1 L'algoritmo per il calcolo dell'Edit Distance

Verrà ora presentato un semplice algoritmo in grado di calcolare la distanza fra due stringhe (l'algoritmo utilizzato fa riferimento alla versione mostrata in[21]).
Il metodo utilizzato prevede il riempimento, in maniera dinamica, di una matrice C_{m*n}, dove *m* ed *n* rappresentano la lunghezza totale (misurata contando il numero di caratteri che le compongono) delle due stringhe da confrontare. Il calcolo viene effettuato affrontando i seguenti passaggi algebrici:

- $C_{i,0} = i$
- $C_{0,j} = j$
- $C_{i,j} = C_{i-1,j-1}$ se $s_1(i) = s_2(j)$, altrimenti si sceglie il minino fra
  1. $C_{i-1,j-1} + 1$
  2. $C_{i,j-1} + 1$
  3. $C_{i-1,j} + 1$

Il risultato finale, che rappresenta la distanza (*edit distance*) fra le due stringhe è leggibile nell'ultima cella in basso a destra della matrice (la cella $C(m,n)$). Vediamo

		C	A	N	E
	0	1	2	3	4
C	1	0	1	2	3
A	2	1	0	1	2
S	3	2	1	1	2
A	4	3	2	2	**2**

Tabella 8.1: Matrice per il calcolo dell'edit distance fra casa e cane

ora una semplice applicazione dell'algoritmo per il confronto di due brevi stringhe quali *casa* e *cane*.

Il risultato dell'operazione per il calcolo dell'edit distance è rappresentato nella matrice dal numero **2** (cella in basso a destra) , evidenziato in neretto. L'edit distance, quindi, fra le stringhe di caratteri *casa* e *cane*, è pari a 2. La motivazione del risultato è abbastanza chiara: se un risultato pari a zero indica che le due stringhe sono perfettamente identiche ed una pari a quattro indica che sono completamente diverse, il 2 significa che le stringhe sono uguali solo a metà (infatti le prime due lettere di casa e cane sono uguali).

8.3.2 Edit Distance fra path

Numerose sono stati negli anni gli algoritmi proposti, basati sul concetto di edit distance, atti a confrontare elementi nei campi più disparati, dalla struttura molecolare[28] ad intere frasi[31]. La versione dell'algoritmo proposta in questa sede confronta fra loro due path. Particolarmente utile è il caso in cui uno di questi due path appartenga ad una query ed il secondo appartenga, invece, ad uno schema rappresentante la struttura di un documento digitale: tramite il risultato dell'algoritmo, infatti, siamo in grado di comprendere quanto il percorso espresso all'interno della query, possa adattarsi al documento considerato. Ovviamente, nel caso in cui vogliamo riscrivere la query per ogni documento di interesse, l'algoritmo per il confronto dovrà essere usato per confrontare il path della query con ogni percorso, presente all'interno di ogni schema, che possa fornire un elevato grado di similarità. Una definizione per l'edit distance fra due percorsi può essere la seguente:

Definizione: *L'edit distance ed(p_1, p_2) fra due path p_1 ed p_2 è il minimo costo della sequenza di operazioni che trasforma p_1 in p_2. Il costo della sequenza di operazioni è la somma dei costi delle operazioni individuali. Le operazioni sono un insieme finito di regole nella forma $\delta(p_3, p_4) = c$, dove p_3 ed p_4 sono due path diversi*

e c è un numero reale non negativo. Una volta che una operazione ha convertito una sottopath p_3 in uno p_4, nessuna ulteriore operazione può essere svolta su p_4.

Le operazioni definite per questa metrica di confronto sono tre:

- *Inserimento*: inserire un elemento in un path
- *Cancellazione*: eliminare un elemento da un path
- *Sostituzione*: sostituire un elemento del path p_2 con uno appartenente al path p_1

Fino a questo punto l'edit distance applicato ai path risulta essere molto simile a quello classico che lavora sulle stringhe. La differenza fondamentale fra i due si trova nel costo dell'operazione di sostituzione: mentre le operazioni di inserimento e sostituzione hanno un costo fisso unitario, l'operazione di sostituzione ha un costo variabile; tale costo dipende, infatti, dalla relazione (ritrovata nel thesaurus di relazioni inter-schema prodotto dal modulo SLIM di MOMIS) che intercorre fra gli elementi dei due path tramite i quali viene fatta la sostituzione. Intuitivamente, cambiare un elemento in un percorso con un suo sinonimo (relazione SYN nel thesaurus fra i due elementi) sarà equivalente a mantenere il path invariato, ed il costo dell'operazione dovrà essere nullo. Sostituire, altresì, un elemento con un'altro col quale è presente una relazione di tipo NT, BT oppure RT dovrà costare meno che sostituirlo con un elemento col quale non sussiste alcuna relazione (il costo dovrà , quindi essere compreso fra zero e uno). I costi scelti per l'implementazione dell'algoritmo di confronto sono:

(p_1 e p_2 rappresentano i due path, mentre $p_1[i]$ e $p_2[j]$ si riferiscono ai singoli elementi dei path)

- *Costo per un inserimento*: sempre 1
- *Costo per una cancellazione*: sempre 1
- *Costo per una sostituzione*:
  1. se $p_1[i] = p_2[j]$ il costo sarà 0
  2. se $p_1[i]$ *SYN* $p_2[j]$ il costo sarà 0
  3. se $p_1[i]$ *BT* $p_2[j]$ il costo sarà 0.2

4. se $p_1[i]$ NT $p_2[j]$ il costo sarà 0.2
5. se $p_1[i]$ RT $p_2[j]$ il costo sarà 0.5

I costi per le sostituzioni sono stati scelti basandosi sui *pesi*, o coefficienti, che vengono forniti alle relazioni (SYN, BT, RT, NT) all'interno del sistema MOMIS, al fine di creare i clusters per le classi globali. Tali pesi sono i seguenti:

- SYN peso = 1
- BT peso = 0.8
- NT peso = 0.8
- RT peso = 0.5

Tali coefficienti rappresentano un metro per la similarità fra due oggetti appartenenti a due schemi differenti mentre, per il nostro scopo, sono richiesti coefficienti che pesino la differenza che intercorre fra due elementi; ciò che si è deciso di fare, quindi, è semplicemente porre i costi di sostituzione pari a: $1 - c$; dove c rappresenta il peso collegato alla relazione considerata (vedere l'elenco precedente).

Vedremo ora come è stato implementato l'algoritmo di confronto fra path basato sull'edit distance, ed in seguito parleremo di come scegliere i path fra i quali cercare la similarità nel caso si voglia riscrivere una query basandosi su un insieme di schemi di documenti.

Perchè l'algoritmo di confronto fra i path abbia successo, ad ogni elemento dei due percorsi (radice, nodo o foglia che sia) deve essere associato un codice in grado di fornire il significato associato ad esso. Per ottenere questo risultato, a tutti gli oggetti di tutti gli schemi tradotti in ODL_{I^3}, memorizzati all'interno del sistema MOMIS ed *annotati* tramite il modulo SLIM, viene associato un codice univoco ottenuto tramite l'applicazione delle SICF (metodo visto all'interno di questo stesso capitolo). Assumendo la fase di annotazione terminata è, dunque, ben comprensibile come ad ogni codice sia associato un significato, o synset, usando la terminologia propria di WordNet. Dato un codice rappresentante un oggetto di uno schema è inoltre possibile ottenere, oltre al significato ad esso collegato, anche tutti i codici, e quindi tutti i significati, dei suoi predecessori. Ricordiamo, infatti, che un codice ottenuto con le SICF è in grado ri rappresentare un intero path per una struttura ad albero. L'algoritmo per

il confronto fra path lavora, ancora una volta, per mezzo della costruzione di una matrice C_{m*n} (m lunghezza del primo path, n lunghezza del secondo) il cui termine finale (in basso a destra) rappresenta la distanza fra i due path. Tale matrice viene costruita in maniera molto simile a quella sfruttata dal classico algoritmo per il confronto fra stringhe, in più si tiene conto dei costi di sostituzione modificati controllando le relazioni sul thesaurus prodotto da SLIM; seguiamo i passaggi algebrici:

- $C_{i,0} = i$
- $C_{0,j} = j$
- $C_{i,j} = C_{i-1,j-1}$ se $p_1(i) = p_2(j)$, altrimenti si sceglie il minino fra
  1. $C_{i-1,j-1} + Costo\ Sostituzione$
  2. $C_{i,j-1} + 1$
  3. $C_{i-1,j} + 1$

Si consideri il seguente esempio:

- Path 1: A B C
- Path 2: D B E

I termini A, B, C, D, E rappresentano i codici collegati agli elementi dei due path. Dopo aver annotato gli schemi, il modulo SLIM fornisce le seguenti relazioni (ottenute tramite l'ausilio di WordNet):

- D SYN A (costo di sostituzione pari a 0)
- E NT C (costo di sostituzione pari a 0.2)

Usando l'algoritmo basato sull'edit distance fra path definita in precedenza la tabella risultante è la seguente:

L'ultima cella in basso a destra della matrice rappresenta il risultato (edit distance) che classifica il grado di differenza che intercorre fra i due percorsi considerati. Nonostante i due path presentino due elementi di differenza (l'edit distance classico avrebbe dato, infatti, risultato 2) tramite le relazioni trovate da SLIM (ed al contenuto semantico degli schemi annotati) la distanza fra ia due percorsi risulta praticamente nulla (0.2) rendendo possibile un *mapping* fra le due strutture ad albero (eg. query e schema XML) come mostra la Figura 8.3.

		D	B	E
	0	1	2	3
A	1	0	1	2
B	2	1	0	1
C	3	2	1	**0.2**

Tabella 8.2: Matrice per il calcolo dell'edit distance fra due path

8.4 Applicazione dell'algoritmo di confronto fra path

Nel paragrafo precedente è stato mostrato come funziona l'algoritmo di confronto fra due path appartenenti a schemi differenti, basato sul concetto di edit distance. Cerchiamo ora di capire come questo algoritmo può essere applicato per la riscrittura di una query su un insieme di schemi XML (che possono essere,ad esempio, in formato Xml-Schema). Consideriamo una query espressa usando il linguaggio XQuery:

```
for $a in /Trattato/Capitolo
   where $a/Argomento = 'XML'
   return $a/Titolo
```

Si può facilmente notare che la query è formata da due path distinti:

- /Trattato/Capitolo/Argomento
- /Trattato/Capitolo/Titolo

Il path espresso a seguito della clausola *for*, infatti, viene memorizzato all'interno della variabile *$a* per essere sfruttato più agevolmente nella rimanente parte della query. Tramite l'algoritmo di confronto fra path precedentemente descritto è possibile cercare e trovare, all'interno della totalità degli schemi posseduti, quali sono i percorsi maggiormente simili a quelli espressi dalla query. Una volta effettuata questa operazione sarà possibile, per ogni documento ritenuto interessante (cioè per il quale vengono ritrovati percorsi sufficientemente simili a quelli espressi nella query) riformulare la query per mezzo dei path semanticamente più simili.

Ricordiamo che l'algoritmo per il calcolo dell'edit distance fra path agisce costruendo una matrice a partire dai codici univoci affidati (per mezzo delle frazioni continue, ovvero delle SICF) ad ogni oggetto di ogni schema. La prima operazione da fare, avuta una query in ingresso, è selezionare ogni suo path e, per ogni path, affidare ad

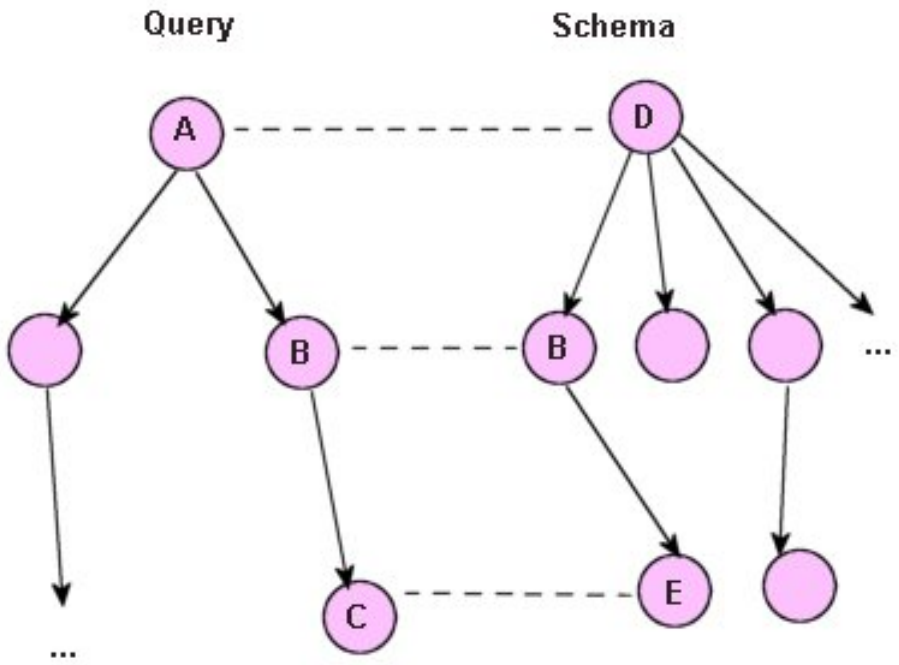

Figura 8.3: Mapping fra la query e lo schema per i due percorsi considerati

ogni elemento un codice tramite cui applicare l'algoritmo. Per ogni elemento $p_i[j]$ situato all'interno del path p_i della query, si possono verificare due casi:

- L'elemento $p_i[j]$ esiste all'interno dell'insieme degli schemi annotati, ed è unico
- L'elemento $p_i[j]$ *non* esiste all'interno dell'insieme degli schemi annotati
- L'elemento $p_i[j]$ esiste più volte all'interno dell'insieme degli schemi annotati

Nel primo caso sarà affidato il codice, e quindi anche il significato, dell'elemento omonimo appartenente all'insieme degli schemi; nel secondo caso, invece, all'elemento $p_i[j]$ sarà affidato un codice non usato da nessun elemento contenuto negli schemi (ad esempio un numero negativo, i codice generati dalle SICF, infatti, sono tutti positivi). Il terzo caso è leggermente più complicato rispetto ai precedenti; questa

situazione si può verificare piuttosto spesso, consideriamo ancora la Figura 8.1 situata nell'introduzione a questo capitolo. In tale raffigurazione, infatti, ed in particolare nello schema sulla sinistra, l'elemento *titolo* compare due volte, per indicare sia il titolo di un libro che quello di un suo capitolo. Se una query, quindi, possiede un path contenente l'elemento titolo, ad esso potranno essere associati più codici, ed anche più significati. In questo caso l'algoritmo di confronto fra path dovrà essere provato una volta per ogni differente significato (e quindi codice) associabile all'elemento della query.

A questo punto (dopo aver assegnato codici e significati agli elementi del path della query) si deve affrontare un problema piuttosto importante nell'ottica dello sviluppo di un modulo per la riscrittura di query: con quali percorsi, appartenenti agli schemi, si deve confrontare il path della query?

Confrontare il percorso ottenuto dalla query con tutti i possibili percorsi ritrovabili all'interno dell'insieme di schemi che descrivono l'archivio è, comprensibilmente, un po' eccessivo. Se l'archivio di cui ci si sta occupando contiene molti documenti XML, è probabile che sia molto elevato anche il numero degli schemi che li descrivono. Eseguire l'algoritmo di confronto fra path per ogni percorso diviene, quindi, computazionalmente troppo pesante. Oltre a questa considerazione si consideri una richiesta che contenga, ad esempio, il path */Libro/Autore* , tale path non deve essere necessariamente confrontato con un percorso quale /Libro/Capitolo, oppure /Libro/Titolo. I significati espressi dagli *elementi finali* dei percorsi (oppure, parlando in termini di strutture ad albero, dalle *foglie* dei path) sono molto differenti fra loro, ed è inutile confrontarli. Fortunatamente, causa l'azione preliminare di *annotazione* degli schemi, una selezione preliminare dei path degli schemi su cui effettuare il confronto è possibile. Vediamo come:

1. Dato un path p_Q, appartenente alla query Q, di lunghezza l, sia $p_Q[l]$ il suo elemento finale.

2. Si cercano, all'interno degli schemi annotati, tutti gli oggetti omonimi di $p_Q[l]$ (che possono avere differenti significati associati)

3. Per ognuno di questi oggetti si selezionano tutti gli elementi collegati ad essi tramite qualche relazione del thesaurus creato da SLIM (SYN, BT, RT, NT)

4. Per ognuno di questi elementi (compresi quelli omonimi di $p_Q[l]$) si ritrova, a partire dal codice SICF ad esso associato, il path da cui deriva all'interno dell'insieme degli schemi

5. Per ogni path così selezionato si applica l'algoritmo di confronto basato sull'edit distance.

Una volta definiti a quali path si deve applicare l'algoritmo basato sull'edit distance modificato, bisogna considerare un ultimo problema: l'edit distance fornisce una misura della differenza che intercorre fra due path; per la maniera in cui viene stabilita questa distanza risulta non essere uniforme se applicata a path di lunghezza differente. Si consideri, infatti, una coppia di percorsi la cui p_1 e p_2 la cui lunghezza sia di tre elementi ciascuno; la massima distanza possibile rilevata dall'algoritmo è tre. Ripetendo lo stesso discorso per una coppia di path p_3 e p_4 di lunghezza, ad esempio, dieci, si nota che la distanza fra i due può arrivare sino a dieci (nel caso in cui tutti gli elementi siano diversi e non vi siano relazioni del thesaurus che li legano). Sorge dunque il problema di rendere le misure di distanza (e di similarità) uniformi per ogni coppia di path considerata. Ciò che è stato pensato è di adottare una definizione di edit distance fra path normalizzata, ottenuta dividendo l'edit distance fornita dall'algoritmo per la massima lunghezza dei percorsi considerati (nella coppia fra cui si ricerca la distanza)e chiamata *edn* (edit distance normalizzata):

$$edn(p_1, p_2) = \frac{ed(p_1,p_2)}{max(|p_1|,|p_2|)}$$

Il risultato fornito dalla formula precedente sarà sempre compreso fra zero ed uno, consentendo di uniformare la metrica definita per la distanza fra due path alla lunghezza degli stessi. Definiamo, inoltre, una misura normalizzata della similarità fra due path, chiamata *simPath* ed ottenuta, semplicemente, sottraendo ad uno il valore di *edn*:

$$simPath(p_1, p_2) = 1 - edn(p_1, p_2)$$

Possiamo ora definire una variabile, detta *sogliaSim*, compresa fra zero ed uno, il cui valore rappresenta la minima *simPath* che vogliamo prendere in considerazione per la riscrittura di un path di una query sui path degli schemi dei documenti.

Per ogni path p_Q appartenente ad una query, l'insieme dei path, appartenenti agli schemi descriventi la struttura dei documenti digitali e scelti come descritto in precedenza, che risultano possedere una $simPath(p_Q, p_i)$ maggiore della soglia *sogliaSim*

indicata, risulta essere il prodotto finale del modulo software di confronto fra path implementato.

Rivediamo dunque, in maniera sintetica, quali sono i passi fondamentali descritti per il ritrovamento dei path più semanticamente simili a quelli espressi in una query Q:

- Assegnazione dei codici, ottenuti dalle SICF, agli elementi della query Q
- Selezione dei percorsi negli schemi per i quali effettuare il confronto (tutti i path ottenuti a partire dagli elementi degli schemi omonimi, o collegati tramite qualche relazione del thesaurus, all'ultimo elemento del path p_Q)
- Applicazione dell'algoritmo basato sull'edit distance modificato per path
- Selezione solamente di quei percorsi la cui similarita $simPath(p_Q, p_i)$ è maggiore di una soglia *sogliaSim* (compresa fra zero e uno) specificata.

8.4.1 Un esempio

Per comprendere meglio come il metodo di confronto fra path possa rappresentare la base per un modulo di risoluzione delle richieste, poste dagli utenti finali, su un archivio di documenti XML, si ripropone l'esempio su Libri e Trattati esposto nell'introduzione di questo capitolo. Si ricorda che la query, non risolvibile in maniera esatta, proposta è:

```
for $a in /Trattato/Capitolo
   where $a/Argomento = 'XML'
   return $a/Titolo
```

I due path che compongono la query sono:

- /Trattato/Capitolo/Argomento
- /Trattato/Capitolo/Titolo

Per questi percorsi mostriamo ora l'applicazione dell'algoritmo di confronto fra path. In Figura 8.4 sono mostrati gli schemi XML a cui si conformano i documenti di un ipotetici archivio. I numeri presenti all'interno dei rettangoli indicanti i nodi degli

alberi, sono i codici univoci assegnati sfruttando le frazioni semplici continue (SICF) (per creare un albero unico ed assegnare codici univoci a tutti i nodi degli schemi è stata inserita una radice fittizia).

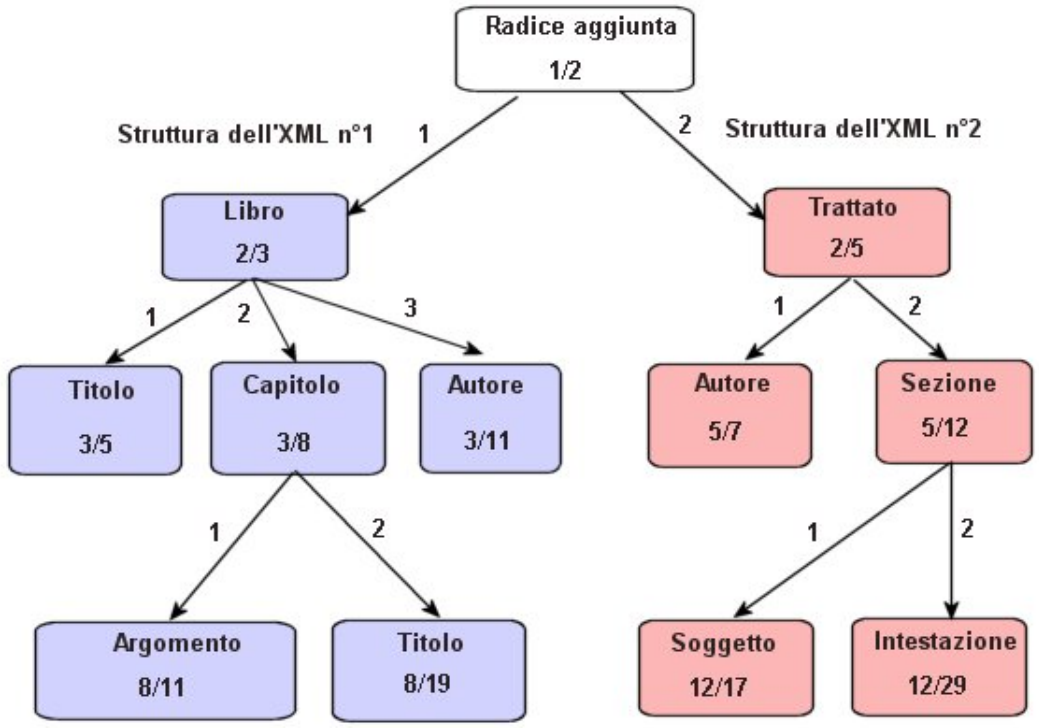

Figura 8.4: Schemi XML per l'esempio del metodo di confronto fra path

Supponiamo, ora, che gli schemi XML siano stati annotati, per mezzo del modulo SLIM. Si supponga, anche, che il thesaurus di relazioni inter-schema risultante dal processo di annotazione sia:

Trattato	NT	Libro
Trattato/Sezione/Soggetto	SYN	Libro/Capitolo/Argomento
Libro/Titolo	SYN	Trattato/Sezione/Intestazione
Libro/Capitolo/Titolo	SYN	Trattato/Sezione/Intestazione
Trattato/Sezione	NT	Libro/Capitolo

Nel processo di annotazione per mezzo del modulo SLIM, quindi, sono stati assegnati a *Titolo* ed *Intestazione* synset considerati sinonimi, così come a *Soggetto* ed

Argomento. A *Libro*, invece, è stato assegnato un synset ipernimo di *Trattato*, così come a *Capitolo* è stato assegnato un significato ipernimo di *Sezione*.

Le stesse relazioni, sfruttando i codici creati per mezzo delle frazioni continue, risultano apparire come segue (si ricorda che un codice di questo tipo ha il potere espressivo di un intero percorso):

2/5	NT	2/3
12/17	SYN	8/11
3/5	SYN	12/29
8/19	SYN	12/29
5/12	NT	3/8

Stabiliamo ora un valore massimo di *simPath*, cioè il massivo grado di similarità (in una scala da zero a uno) per cui un percorso scelto fra quelli appartenenti agli schemi sarà considerato semanticamente simile ad uno della query.

Supponiamo tale valore pari a **0.75**

Iniziamo con il ricercare i percorsi simili al path */Trattato/Capitolo/Argomento*, che compare nella clausola *where* della query. Seguendo il procedimento descritto nel paragrafo precedente, si attribuiscono i codici, presi dall'albero degli schemi, corrispondenti agli elementi che compaiono nel percorso della richiesta. Fortunatamente, per questo primo caso, tutti i termini rappresentanti i nomi degli elementi del percorso sono presenti, una sola volta, nei vari schemi ed, assumiamo, sono stati annotati. Le corrispondenze che sorgono sono le seguenti:

Trattato	associato a	2/5
Capitolo	associato a	3/8
Argomento	associato a	8/11

Il passo successivo è la scelta dei percorsi degli schemi per i quali rilevare le affinità semantiche con il path */Trattato/Capitolo/Argomento*. A tal scopo si considera l'ultimo elemento del percorso della query (la foglia del ramo dell'albero rappresentato dal percorso); il codice ad esso associato è 8/11. L'algoritmo di confronto si applica a tutti quei percorsi (negli schemi) che hanno come codice associato all'ultimo elemento 8/11 o un codice ad esso legato tramite una relazione semantica nel thesaurus; in questo caso, seguendo la relazione *8/11 SYN 12/17* l'algoritmo si applica sia al

percorso selezionato da 8/11 che da 12/17 (ricordiamo, infatti, che da questi codici è possibile, per mezzo di un algoritmo iterativo, ricavare i codici di tutti gli elementi del percorso precedente, fino alla radice dell'albero).

8/11 fornisce il percorso *Libro/Capitolo/Argomento*
12/17 fornisce il percorso *Trattato/Sezione/Soggetto*

Vediamo ora di rappresentare le matrici per il calcolo dell'edit distance fra il percorso */Trattato/Capitolo/Argomento*, originario della query, ed i possibili path simili individuati.

		2/5	3/8	8/11
	0	1	2	3
2/3	1	0.2	1.2	2.2
3/8	2	1.2	0.2	1.2
8/11	3	2.2	1.2	**0.2**

Tabella 8.3: Matrice per il calcolo dell'edit distance fra /Trattato/Capitolo/Argomento e Libro/Capitolo/Argomento

		2/5	3/8	8/11
	0	1	2	3
2/5	1	0	1	2
5/12	2	1	0.2	1.2
12/17	3	2	1.2	**0.2**

Tabella 8.4: Matrice per il calcolo dell'edit distance fra /Trattato/Capitolo/Argomento e Trattato/Sezione/Soggetto

Possiamo comprendere facilmente che entrambi i percorsi considerati sono, grazie alle relazioni presenti nel thesaurus, molto simili a quello della query; l'edit distance è, infatti, per entrambi pari a 0.2. Il valore dell'edit distance normalizzato sulla lunghezza del massimo percorso è 0.067 (0.2 / 3) Troviamo comunque il valore di similarità $simpath$ per un edit distance pari a 0.2:

$simpath$ = 1 - edn = **0.933**

Il valore di similarità dei due percorsi trovati (uno per ogni schema) è maggiore rispetto alla soglia di 0.75 posta. Possiamo, quindi, considerare i due path semanticamente simili a quello della query (indicativamente, se per ogni schema viene trovato un

percorso simile anche al secondo path della query, */Trattato/Capitolo/Titolo*, la richiesta potrà essere riscritta per entrambi gli schemi, sostituendo al posto dei percorsi originali quelli reperiti).

Consideriamo ora il secondo percorso che compare nella richiesta: */Trattato/Capitolo/Titolo*. Assegnamo anche ad esso i codici da cui possiamo risalire ai significati semantici. Notiamo subito che *Titolo* compare due volte all'interno dell'insieme degli schemi, per cui ad esso dovranno essere associati due codici differenti: 3/5 e 8/19.

Trattato	associato a	2/5
Capitolo	associato a	3/8
Titolo	associato a	3/5 e 8/19

Dato che ad un elemento del percorso della query sono associabili due codici (e due significati potenzialmente differenti) l'intero procedimento di confronto (dall'identificazione dei percorsi da valutare al calcolo della similarità) dovrà essere ripetuto due volte.

1. *Applicazione del metodo di confronto avendo il codice di Titolo = 3/5*

   I percorsi (scelti fra quelli che formano gli schemi) da confrontare in questo caso sono identificati dal codice 3/5 (elemento finale del path della query) e dal codice 12/29, per via della relazione *3/5 SYN 12/29*.

3/5	fornisce il percorso	*Libro/Titolo*
12/29	fornisce il percorso	*Trattato/Sezione/Intestazione*

   Vediamo come appaiono le matrici per il calcolo dell'edit distance fra il path della query (con il codice di *Titolo* pari a 3/5) ed i due percorsi selezionati.

		2/5	3/8	3/5
	0	1	2	3
2/3	1	0.2	1.2	2.2
3/5	2	1.2	1.2	**1.2**

Tabella 8.5: Matrice per il calcolo dell'edit distance fra /Trattato/Capitolo/Titolo(3/5) e Libro/Titolo

   Possiamo notare che l'edit distance fra */Trattato/Capitolo/Titolo* e */Libro/Capitolo* è pari a 1.2. L'edit distance normalizzata è, invece, 0.4 (1.2 / 3). Per cui:

		2/5	3/8	3/5
	0	1	2	3
2/5	1	0	1	2
5/12	2	1	0.2	1.2
12/17	3	2	1.2	**0.2**

Tabella 8.6: Matrice per il calcolo dell'edit distance fra /Trattato/Capitolo/Titolo(3/5) e Trattato/Sezione/Intestazione

simPath = 1 - 0.4 = **0.6**

Il valore di similarità non è superiore al valore di soglia scelto, pari a 0.75. I due percorsi, in questo caso non possono essere considerati semanticamente simili.

L'edit distance, invece, fra */Trattato/Capitolo/Titolo* e */Trattato/Sezione/Intestazione* è pari a 0.2. L'edit distance normalizzata è, invece, 0.067 (0.2 / 3). Per cui:

simPath = 1 - 0.067 = **0.933**

Il valore di similarità è inferiore al valore di soglia scelto, pari a 0.75. I due percorsi, in questo caso, sono da ritenersi simili.

2. *Applicazione del metodo di confronto avendo il codice di Titolo = 8/19*

   I percorsi (scelti fra quelli che formano gli schemi) da confrontare in questo caso sono identificati dal codice 8/19 (elemento finale del path della query) e dal codice 12/29, per via della relazione *8/19 SYN 12/29*.

   8/19 fornisce il percorso *Libro/Capitolo/Titolo*
   12/29 fornisce il percorso *Trattato/Sezione/Intestazione*

   Vediamo come appaiono le matrici per il calcolo dell'edit distance fra il path della query (con il codice di *Titolo* pari a 8/19) ed il primo percorso selezionato (il secondo percorso, infatti, è già stato analizzato in precedenza ed è risultato simile al path della query).

   Possiamo notare che l'edit distance fra */Trattato/Capitolo/Titolo* e */Libro/Capitolo/Titolo* è pari a 0.2. L'edit distance normalizzata è, invece, 0.067 (0.2 / 3). Per cui:

		2/5	3/8	8/19
	0	1	2	3
2/3	1	0.2	1.2	2.2
3/8	2	1.2	0.2	1.2
8/19	3	2.2	1.2	**0.2**

Tabella 8.7: Matrice per il calcolo dell'edit distance fra /Trattato/Capitolo/Titolo(8/19) e Libro/Capitolo/Titolo

simPath = 1 - 0.067 = **0.933**

Il valore di similarità è superiore al valore di soglia scelto, pari a 0.75. I due percorsi, in questo caso possono essere considerati semanticamente simili.

Conclusioni tratte dall'esempio: analizzando i risultati ottenuti, possiamo osservare che per la query:

```
for $a in /Trattato/Capitolo
   where $a/Argomento = 'XML'
   return $a/Titolo
```

i cui path componenti sono:

- /Trattato/Capitolo/Argomento
- /Trattato/Capitolo/Titolo

sono stati individuati, come semanticamente simili, i percorsi:

- /Libro/Capitolo/Argomento
- /Libro/Capitolo/Titolo

apparteneti al primo schema XML, ed individuati i percorsi:

- /Trattato/Sezione/Soggetto
- /Trattato/Sezione/Intestazione

appartenenti al secondo schema. Risulta piuttosto evidente come, utilizzando i percorsi reperiti in questa maniera, sia semplice riscrivere correttamente una query per i documenti realmente esistenti in archivio (e conformi agli schemi).

8.4.2 Lavori futuri

Il modulo software progettato e costruito nell'ambito di questa tesi al fine di selezionare i path più simili (all'interno di un insieme di schemi annotati) ad uno dato, potrà essere sfruttato in un modulo, ancora da implementare, di gestione delle query. All'interno del software di MOMIS, infatti, risulta tuttora assente un package in grado di utilizzare query in formato XQuery. Una volta che questo sarà stato implementato risulterà possibile la riscrittura vera e propria, non solo dei path che formano una richiesta, ma della richiesta nel suo complesso.

Vediamo come, una volta ottenuti per ogni path p_Q della query i percorsi più simili, potrebbe essere possibile riscrivere una query.

Si supponga, ad esempio, che una query sia formata da tre differenti path; per ognuno di essi siamo ora in grado di identificare i percorsi identici, o maggiormente simili semanticamente, sfruttando il modulo di confronto. Ogni percorso ritrovato appartiene ad uno schema S_i che descrive un insieme di documenti XML. Consideriamo l'unione di tutti gli schemi S_i selezionati in tale maniera. Per ognuno di essi è possibile ottenere un valore di similarità complessivo fra lo schema e la query, basandosi sui valori di similarita *simPath* fra i percorsi della query e quelli (trovati a per mezzo del modulo di confronto fra path) appartenenti allo schema. Per ogni schema la cui similarità complessiva, definita per esempio *simPathTot*, risulti superiore ad una determinata soglia, sarà possibile riscrivere la query sostituendo i path originari con quelli, ritenuti simili, appartenenti allo schema.

Una proposta per assegnare un punteggio di similarità complessivo fra uno schema (eg. Xml-Schema tradotto in ODL_{I^3} ed annotato) ed una query è la seguente:

Chiamiamo *n* il numero di path presenti all'interno della query che si vuole riscrivere. Chiameremo ogni percorso della query p_{Qi} dove [i = 1..*n*]. Per ogni p_{Qi} sia p_{Si} il path, appartenente allo schema *S*, che meglio lo approssima (i valori di similarità sono dati dall'applicazione dell'algoritmo di confronto fra path, spiegato in precedenza). Fra il path e la query, il grado di similarita *simPathTot* complessivo può essere definito:

$$simPathTot(Q,S) = \sum_{i=1..n} simPath(p_{Qi}, p_{Si})$$

Oppure, se vogliamo una versione normalizzata compresa fra zero ed uno della similarità complessiva:

$$simPathTot(Q,S) = \frac{\sum_{i=1..n} simPath(p_{Qi}, p_{Si})}{n}$$

In pratica si sommano tutti i contributi *simPath* (compresi fra zero ed uno), per ogni schema, ottenuti dall'algoritmo di confronto fra path applicato ad ogni percorso della query. Tale valore, eventualmente diviso per il numero di percorsi che formano la query, rappresenta il grado di affinità fra la richiesta e lo schema. Se tale affinità supera una data soglia, da definire sperimentalmente , si può ritenere che lo schema soddisfi (tutti o in buona parte) i requisiti espressi dalla query. La query può, quindi, essere riscritta per quello schema sostituendo i suoi percorsi originali p_{Qi} con quelli,semanticamente simili, contenuti nello schema (se il grado totale di affinità fra la query e lo schema supera la soglia, ma un percorso della query non trova riscontro nello schema, cioè il suo *simPath* è pari a zero, durante la riscrittura la parte della query concernente tale path deve essere omessa).

8.5 La struttura del software

In questo paragrafo sarà presentata e commentata la struttura del modulo software prodotto, che permette, dato un path in ingresso (ottenuto, ad esempio dall'espressione di una query in XQuery) di trovare i percorsi sugli schemi che meglio lo approssimano.

Prima di iniziare la spiegazione delle classi e delle funzioni del modulo si deve comprendere che tale software, per funzionare, ha bisogno, come supporto, del sistema MOMIS. In particolare vengono sfruttati i moduli SAM (per l'acquisizione degli schemi dai wrapper) e SLIM (per la annotazione degli schemi e la creazione del thesaurus di relazione inter-schema). Gli altri moduli del sistema MOMIS non servono al fine del funzionamento del software prodotto nell'ambito del confronto fra path in questa tesi (sarebbe possibile anche utilizzare il thesaurus di relazioni prodotto dal modulo SIM, che comprende anche le relazioni ritrovate per mezzo dei riferimenti a chiave e per mezzo delle gerarchie di specializzazione, ma per il momento si è deciso di utilizzare solamente il thesaurus prodotto dal modulo SLIM). In effetti il software è stato prodotto partendo da una versione di MOMIS comprendente solamente i moduli citati (con l'aggiunta del browser WNEditor, ora integrato all'interno dell'ambiente di SLIM).

Entriamo ora nel vivo della descrizione del software. Il modulo per il confronto

fra path è composto da due package Java che interagiscono fra loro.Il package SICF ed il package CPath. Nei prossimi paragrafi potrà essere vista la struttura di questi package ed il meccanismo per al loro interazione.

8.5.1 Il package SICF

Il package chiamato SICF, serve per la definizione e l'implementazione di alcuni algoritmi per la gestione delle frazioni continue semplici (le SICF, appunto). La struttura di questo package è rappresentata in Figura 8.5

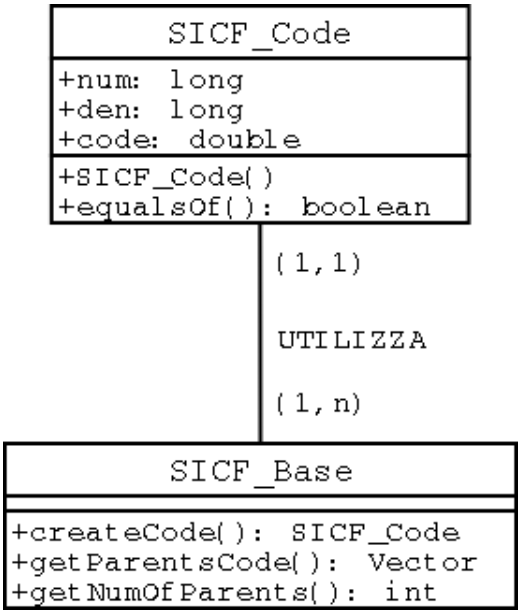

Figura 8.5: Schema UML delle classi del package SICF

Il package SICF è attualmente formato da due classi, chiamate rispettivamente SICF_Code e SICF_Base. La prima delle due classi (SICF_Code) rappresenta il metodo che è stato implementato per l'utilizzo dei codici generati per mezzo delle SICF. In pratica ogni istanza di questa classe rappresenta l'espansione di una frazione continua semplice, che può essere generata a partire da una struttura ad albero come quella gestita nella classe SlimTree del package SLIM.

La seconda classe, SICF_Base, implementa i principali algoritmi che possono essere utilizzati sfruttando le frazioni continue (ovvero le istanza della classe SICF_Code). Le principali funzioni appartenenti a questa classe sono:

public SICF_Code createCode(long labeled_edge, long f_NUM, long f_DEN): questa funzione crea il codice di un nodo figlio a partire dal codice di un padre. I valori *f_NUM* ed *f_DEN* rappresentano il numeratore ed il denominatore della frazione equivalente al numero razionale codice del nodo padre. Il valore *labeled_edge*, invece, è il valore *n* univoco che deve essere assegnato ad ogni figlio di un nodo per poterne determinare il codice con il metodo delle frazioni continue.

public static int getNumOfParents(SICF_Code Code): a partire dal codice *Code*, questa funzione restituisce, il numero di nodi parenti, o antenati, del nodo associato a tale codice.

public static Vector getParentsCode(SICF_Code Code): a partire dal codice *Code*, questa funzione restituisce, memorizzati in un vettore, tutti i codici dei parenti, o antenati, del nodo associato a tale codice.

8.5.2 Il package CPath

Il package CPath, strettamente connesso al package SICF, è il package principale del modulo di confronto fra path. Il suo scopo è, appunto, quello di prelevare le informazioni interessanti dal modulo SLIM (la struttura dell'albero che rappresenta gli schemi in ODL_{I^3} ed il thesaurus di relazioni prodotto) e, dato un path in ingresso, selezionare i percorsi semanticamente più simili fra quelli presenti all'interno degli schemi dei dati. La struttura delle classi di questo package è rappresentata (usando come al solito uno schema UML per le classi) in Figura 8.6.

La classe principale è quella chiamata *PathComparator*. Le funzioni di questa classe sono:

public Vector trovaPathsSimili(): è al funzione che cerca i path semanticamente simili a quello espresso come riferimento (eg. un path estratto da una query XQuery). Sfrutta la classe EditDist per effetuare il suo compito.

public public Vector trovaPathsDaProvare(SICF_Code queryCode): a partire dal codice *queryCode*, rappresentante l'ultimo elemento del path della query (o, comunque, l'ultimo elemento del percorso per cui trovare i path simili), questa funzione individua i percorsi appartenenti agli schemi su cui effettuare il confronto.

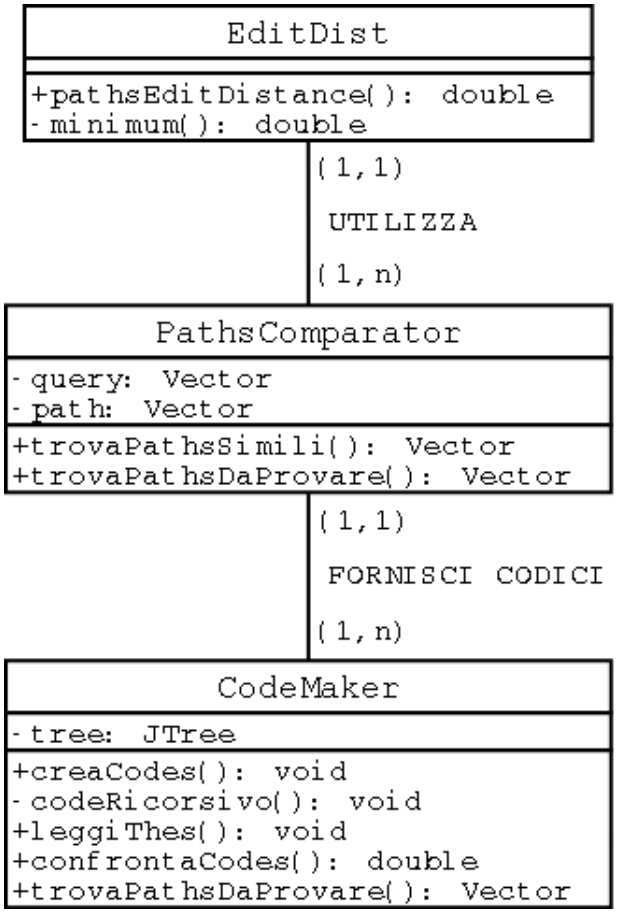

Figura 8.6: Schema UML delle classi del package CPath

La classe EditDist è quella che si occupa dell'implementazione dell'algoritmo di confronto fra path. Il confronto viene effettuato per mezzo della funzione *pathEdit-Distance()* che, ricevendo in ingresso due vettori contenenti i codici degli elementi dei due path, produce la matrice il cui ultimo termine (in basso a destra) viene restituito come risultato. Tale valore rappresenta, appunto, l'edit distance che intercorre fra i due percorsi. Questa classe, per svolgere il suo compito, si interfaccia con la classe CodeMaker, che possiede il thesaus di relazioni di SLIM, riscritto per essere usato velocemente per mezzo dei codici ottenuti con le SICF.

L'ultima classe ad essere contenuta nel package CPath è la classe CodeMaker. Tale classe si occupa di reperire le informazioni necessarie dal modulo SLIM (struttura dell'albero di SlimTree e thesaurus di relazioni fornito dalla classe ThesaurusTable) e gestire l'assegnamento dei codici ottenuti per mezzo delle frazioni continue. Le principali funzioni di questa classe sono:

public void creaCodes(): è la funzione che si occupa di assegnare i codici SICF ai nodi di una struttura ad albero gastita dalla classe java JTree. In particolare, dato che SlimTree possiede un albero JTree rappresentante la struttura degli schemi ottenuti dai wrapper, è in gradi di assegnare i codici agli elementi degli schemi.

public void leggiThes(Vector entryThes): riscrive il thesaurus di relazioni inter-schema prodotto da SLIM (e situato in una istanza della classe ThesaurusTable del package omonimo) usando i codici SICF_Code collegati agli elementi degli schemi.

8.5.3 Relazioni fra i package

Le relazioni che intercorrono fra le classi appartenenti ai package SICF e CPath, oltre che quelle contenute nel package SLIM, sono raffigurate in un diagramma delle classi UML in Figura 8.7.

Dalla figura è possibile notare come il package CPath si appoggi, per le interazioni sia con il package SICF, che con SLIM, alla classe CodeMaker. Compito di questa classe, infatti, è gestire le informazioni ottenute dal modulo SLIM (il thesaurus di relazioni e la struttura ad albero contenente gli schemi ODL_{I^3} di SlimTree) e produrre, in collaborazione con il package SICF, i codici univoci necessari all'algoritmo di confronto.

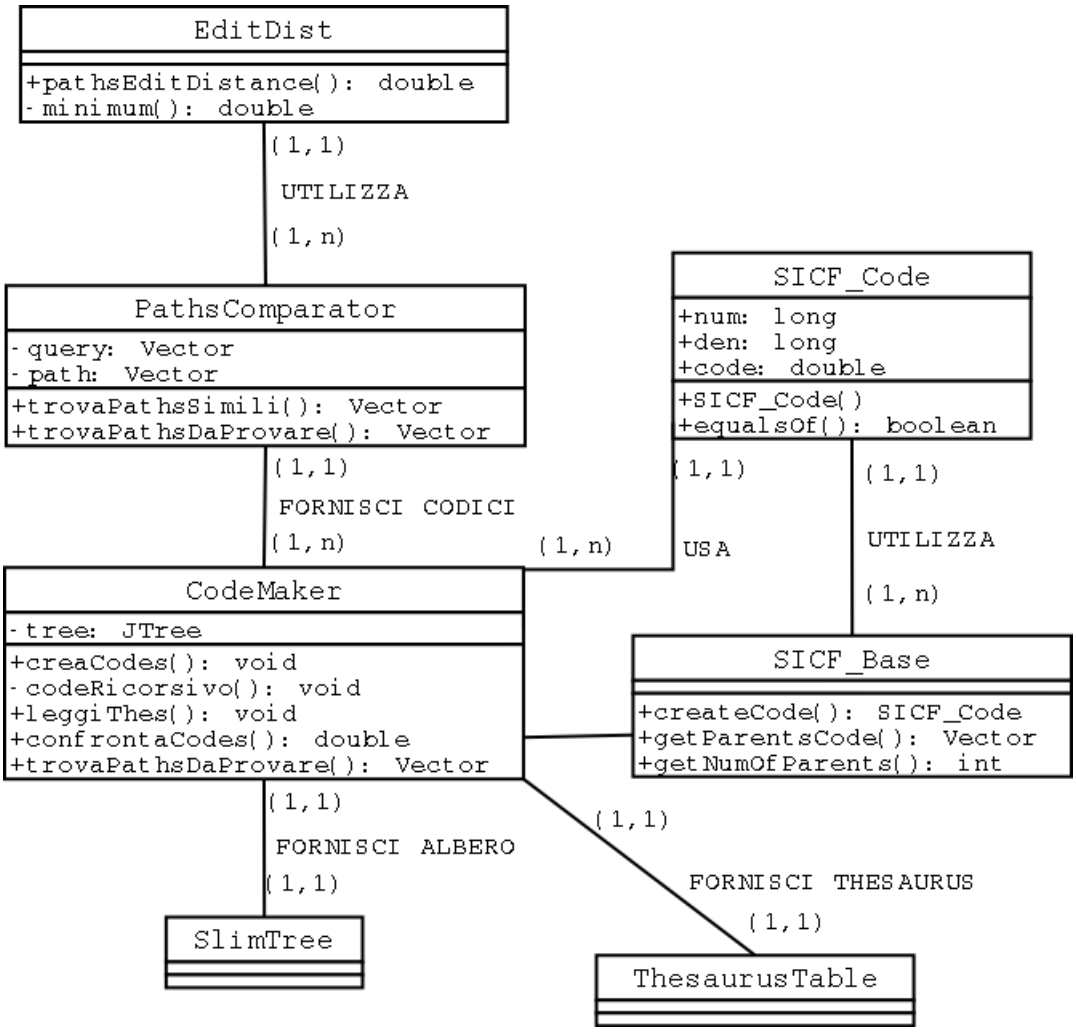

Figura 8.7: Schema UML complessivo del modulo di confronto fra path

Conclusione

In questa tesi si è studiato un metodo innovativo in grado di rispondere efficientemente ed efficacemente alle interrogazioni di un utente poste su un archivio di documenti XML.

Il metodo proposto si basa sull'utilizzo del modulo mediatore MOMIS unitamente all'impiego di ontologie in grado di arricchire semanticamente i metadati utilizzati per descrivere i documenti. MOMIS è in grado di operare su schemi, descriventi i documenti presenti in una fonte locale di informazione, solamente dopo che essi sono stati tradotti, da appositi wrapper, in schemi ODL_{I^3}. Il primo passo compiuto in questa tesi è stato, quindi, la specifica di un wrapper per Xml-Schema (Xml-Schema è un nuovo e flessibile linguaggio per la descrizione e la validazione si documenti XML, proposto e consigliato dal World Wide Web Consortium). Dopo aver analizzato in modo approfondito la sintassi di Xml-Schema e quella del linguaggio per la mediazione ODL_{I^3}, utilizzato da MOMIS, è stata proposta una traduzione da Xml-Schema a ODL_{I^3}, in modo da permettere la futura implementazione del wrapper.

Assumendo poi che il sistema mediatore MOMIS sia in grado di interagire, per mezzo di un futuro wrapper Xml-Schema, con un archivio di dati semistrutturati XML, si è progettata e realizzata una nuova versione del modulo SLIM (utilizzato da MOMIS), in grado di interagire con una versione estensibile di WordNet. La conoscenza del database lessicale WordNet, infatti, è stata riportata in un database di nome MOMISWN da [24]. Gli algoritmi per la creazione del thesaurus inter-schema (prodotto finale del modulo SLIM) e le funzioni per il reperimento delle informazioni al fine di annotare (arricchire semanticamente per mezzo dei significati forniti da WordNet) gli schemi ODL_{I^3}, sono stati riscritti ed ottimizzati per sfruttare questo database facilmente estensibile. L'utilizzo di una versione estensibile di WordNet permette, tramite l'aggiunta di nuovi termini, significati o relazioni fra essi, di eliminare totalmente le possibili perdite di conoscenza nella fase di annotazione degli schemi

(nel caso in cui il nome di un elemento di uno schema non sia presente all'interno di WordNet esso può essere aggiunto).

Oltre all'implementazione della nuova versione del modulo SLIM è stato progettato e realizzato il software per una interfaccia *user friendly* fra WordNetEditor (un browser per l'utilizzo del database relazionale MOMISWN) e lo stesso SLIM. Il browser può ora essere usato direttamente all'interno dell'interfaccia grafica di SLIM in modo da ottimizzare il lavoro compiuto dal progettista in fase di annotazione degli schemi.

L'ultimo passo effettuato in questa tesi è stata l'implementazione di un *modulo per l'individuazione della similarità fra path*. Tale modulo utilizza il thesaurus di relazioni prodotto dalla nuova versione di SLIM, ed un algoritmo basato del concetto di edit distance, per reperire i percorsi (presenti all'interno dell'insieme degli schemi annotati per mezzo di SLIM) semanticamente simili ad uno dato. Questo software può essere considerato il punto di partenza di un modulo (per il momento ancora da implementare) per la gestione, tramite il sistema MOMIS, delle interrogazioni poste da un utente su un archivio di documenti XML. Tramite il modulo di confronto fra path, infatti, a partire dai percorsi che costituiscono una richiesta (sullo stampo di quelle formulate con XQuery, linguaggio per l'interrogazione di documenti XML), possono essere individuati i path maggiormente simili per ogni schema di interesse. Tramite i percorsi recuperati in questo modo sarà molto semplice riscrivere la richiesta dell'utente, correttamente, per ogni schema sufficientemente vicino ai requisiti richiesti dalla query.

Si ricorda che questa tesi si è svolta nell'ambito del progetto ECD (Enhanced Content Delivery) atto a fornire, usando mezzi come biblioteche digitali e sistemi mediatori, contenuti arrichiti agli utenti finali, in modo da rispondere più efficacemente alle loro aspettative.

Bibliografia

[1] Chaitanya Baru, Vicent Chu, Amarnath Gupta, Bertram Ludascher, Richard Marciano, Yannis Papakonstantinou, and Pavel Velikhov. Xml-based information mediation for digital libraries. *ACM Digital Libraries*, pages 214–215. Reperibile presso http://citeseer.nj.nec.com/article/baru99xmlbaseb.html.

[2] I. Benetti, D. Beneventano, S. Bergamaschi, A. Corni, F. Guerra, and G. Malvezzi. Sidesigner: a tool for intelligent integration of information. *International Conference on Sytstem Siences (HICSS2001)*, Gennaio 2001. Reperibile presso http://www.dbgroup.unimo.it/prototipo/paper/hicss2001.ps.gz.

[3] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Inteligence*, (4):185–203, 1994.

[4] S. Bergamaschi, M. Vincini, and S. Castano. Semantic integration of semistructured and structured data sources. *SIGMOS records*, 28(1), Marzo 1999.

[5] S. Bergamaschi, M. Vincini, S. Castano, and D. Beneventano. Semantic integration of heterogeneous information sources. *Journal of Data and Knowledge Engineering*, 36(3):215–249, 2001.

[6] S. Bergamaschi, M. Vincini, S. Castano, D. Beneventano, A. Corni, G. Malvezzi, R. Guidetti, and M. Melchiori. Information integration: the momis project demonstration. in amr el abbadi and michael l.brodie and sharma chakravarthy

and umeshvar dayal and nabil kamel and kyuyoung wang editors. *VLDB2000, Proceedings of 26th Conference on Very Large Data Bases, September 10-14, 2000, Cairo*, pages 611–614, 2000. Morgan Kaufmann.

[7] S. Castano and V. De Antonellis. Deriving global conceptual views from multople infotmation sources. *preProc. of ER'97 Preconference Symposium on Conceptual Modeling. Historical Perspectives and Future works*, 1997.

[8] S. Chawathe, Garcia Molina, H. J. Hammer, K. Ireland, J. Ulmann Y. Papakonstantinou, and J. Widom. The tsimmis project: Integration of etherogeneous information sources. *IPSJ conference, Tokyo, Japan*, 1994. ftp://db.stanford.edu/pub/chawathe/tsimmis-overvew.ps.

[9] Paolo Ciaccia and Wilma Penzo. Approximate tree pattern matching. *EEXTT: Efficiency and Effectiveness of XML Tolls and Tecniques*, pages 22–34, 2002.

[10] World Wide Web Consortium. Xml path language (xpath). version 1.0, Novembre 1999. http://W3C/TR/xpath/.

[11] World Wide Web Consortium. Xml schema part 0: Primer, Maggio 2001. http://W3C/TR/xmlschema-0/.

[12] World Wide Web Consortium. Xml schema part 1: Datatypes, Maggio 2001. http://W3C/TR/xmlschema-2/.

[13] World Wide Web Consortium. Xml schema part 1: Structure, Maggio 2001. http://W3C/TR/xmlschema-1/.

[14] D.Beneventano, J.P.Ballerini, S.Bergamaschi, C.Sartori, and M.Vincini. A semantic-driven query optimizer for oodb. *Int. Workshop on Descriptiond Ligics*, Giugno 1995.

[15] D.Beneventano, S.Bergamaschi, C.Sartori, and M.Vincini. Odb-tools: a tool for semantic query optimization in oodb. *Prec. of Int. on Data Engeenering, ICDE'97*, Aprile 1997.

[16] Istituto CNR di Pisa. Open dlib official site. informazioni presso http://opendlib.iei.pi.cnr.it/.

[17] World Wide Web Consortium Working Draft. Xml syntax for xquery 1.0 (xqueryx), Giugno 2001. http://W3C/TR/xqueryx.

[18] World Wide Web Consortium Working Draft. Xquery 1.0: An xml query language, Agosto 2002. http://W3C/TR/xquery.

[19] Gio Wiederhold et al. Integrating arificial intelligence and database technology. *Journal of Intelligent Integration Systems*, 2/3, Giugno 1996.

[20] H. Garcia Molina et al. The tsimmis approach to mediation: Data models and languages. *NGITS workshop*, 1995. ftp://db.stanford.edu/pub/garcia/1995/tsimmis-models-languages.ps.

[21] G.Navarro. A guided tour to approximate string matching. Technical report, Dept. of Computer Science. University of Chile, 2001.

[22] N. Guarino. Understanding, building and using ontologies: A commentary to 'using explicit ontologies in kbs development', by van heijst, schreiber, and wielinga. *International Journal of Human and Computer Studies*, (46):293–310, 1997.

[23] N. Guarino. *Formal ontologies and information systems*. IOS Press, Amsterdam, 1998.

[24] Veronica Guidetti. Intelligent information integration systems: extending lexicon ontology. Master's thesis, Università di Modena e Reggio Emilia, 2001. reperibile presso http://sparc20.ing.unimo.it.

[25] R. Hull and R. king et al. Arpa i3 reference architecture. 1995. reperibile presso: http://www.isse.gmu.edu/I3_Arch/index.html/.

[26] Morgan Kaufmann. *The object database standard: ODMG-93*. R.G.G. Cattel, 1994.

[27] Morgan Kaufmann. *The object database standard: ODMG2.0*. R.G.G. Cattel, 1997.

[28] K.Zhang, J.T.L.Wang, and D.Shasha. On editing distance between undirected acyclic graphs. *International Journal of Foundation of Computer Science*, 1995.

[29] Dongwon Lee and Wesley W.Chu. Comaparative analysis of six xml schema languages. *ACM SIGMOD Record*, Settembre 2000.

[30] Giovanni Malvezzi. Estrazione di relazioni lessicali con wordnet nel sistema momis. Master's thesis, Università di Modena e Reggio Emilia, 2000. reperibile presso http://sparc20.ing.unimo.it.

[31] Riccardo Martoglia. Extra: Progetto e sviluppo di un ambiente per traduzioni multilingua assistite. Master's thesis, Università di Modena e Reggio Emilia, 2001. reperibile presso http://sparc20.ing.unimo.it.

[32] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph mathing algorithm and its application schema matching. *ICDE: International Conference on Data Engineering*, 2002.

[33] George A. Miller. Wordnet. a lexical database for english. *Communications for the ACM*, 38(11):39–41, 1995. il sito di WordNet si trova in: http://www.cogsci.princeton.edu/ wn/.

[34] C. D. Olds. *Confinued Fractions. New Methematical Library*. Random House, 1963.

[35] S.Abiteboul, D.Quass, J.McHoug, and J.Widom. The lorel query language for semistructured data. *International Journal on Digital Libraries*, pages 68–88, April 1997.

[36] S.Bergamaschi, D.Beneventano, M.Vincini, and S.Castano. Integrazione di informazione: il linguaggio odli3 e la logica descrittiva olcd. Technical Report TR-R03, Università di Modena e Reggio Emilia, 1998.

[37] Torsten Schlieder. Apporxql: design and implementation of an approximate pattern matching language for xml. Technical Report Report B01-01, Freie Universitat, Berlin, 2001.

[38] Torsten Schlieder and Felix Naumann. Approximate tree embedding for querying xml data. *Proceeding of the workshop on the Web Information and Data*.

[39] Paolo Tiberio, Dario Maio, and Paolo Ciaccia. A method for hierarchy processing in relational systems. *Information Systems*, 14(2):93–105, 1989.

[40] H. S. Wall. *Analytic Theory of Continued Fractions*. Van Nostrand, 1948.

[41] Jason T.L. Wang, Huijuan Shan, Kaizhang Zhang, and Dannis Shasha. Atreegrep. approximate searching in unordered trees. 2002.

[42] Alberto Zanoli. Si-designer, un tool di ausilio all'integrazione di sorgenti di dati eterogenee distribuite: progetto e realizzazione. Master's thesis, Università di Modena e Reggio Emilia, 1998. reperibile presso http://sparc20.ing.unimo.it.

[43] Kaizhang Zhang, Rich Statman, and Dannis Shasha. On the editing distance between unordered trees. *Information processing letters 42*, pages 133–139, 1992.