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

Serializzazione di oggetti in formato XML nell'ambito del sistema MOMIS

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di
Davide Lenzi

Controrelatore
Chiar.mo Prof. Michele Colajanni

Anno accademico 1999 – 2000

Parole chiave:

DTD
Java
ODL_I^3
object deserialization
serializzazione
XML

Ringraziamenti:

Ringrazio tutti coloro che mi hanno ascoltato, consigliato, aiutato, o semplicemente sopportato. (Così non dimentico nessuno ;)

Indice

Indice delle figure.

Introduzione

Le reti e i sistemi informativi sono diventati parti integranti del sistema produttivo mondiale: ogni giorno nuove informazioni sono immesse in rete e ogni giorno milioni di persone le consultano. Uno dei maggiori ostacoli connessi con l'utilizzo delle informazioni presenti in rete è la difficoltà nel reperirle. Spesso la ricerca di informazioni da parte dell'utente genera un numero molto elevato di possibili contatti, rendendo di fatto inutile la ricerca (*information overload*).

Il valore di un'informazione reperita sulla rete è tanto maggiore quanto più si avvicina alle aspettative dell'utente. Avere a disposizione motori di ricerca altamente selettivi non è una condizione sufficiente per raggiungere l'obiettivo della ricerca: l'utente è comunque costretto a risolvere problemi di svariata natura: la selezione delle sorgenti interessanti, l'analisi e la sintesi dei dati reperiti (spesso duplicati, inconsistenti o di varia natura), ecc.

Sorge dunque il bisogno di disporre di sistemi che siano in grado di integrare in maniera automatica le informazioni, siano in grado di eliminare le ridondanze. Negli ultimi tempi si sono diffusi sistemi come i *Datawarehous*e, i *Datamine*r, i *Sistemi di Work-flo*w, ecc. I domini di impiego di questi sistemi sono innumerevoli: ospedaliero, militare, aziendale, pubblicazioni, ecc.

Questa tesi si colloca nell'ambito di un progetto più ampio denominato **MOMIS (**Mediator Envi**r**Onment for Multiple Information Sources), sviluppato con l'obiettivo di realizzare l'integrazione semi-automatica delle informazioni contenute all'interno di sorgenti eterogenee e distribuite. **MOMIS** adotta un'architettura a tre livelli con un *Mediatore* che si occupa della creazione della vista aggregata degli schemi costitutivi le singole sorgenti, e della gestione delle query poste dall'utente sullo schema globale.

Gli elementi innovativi introdotti in questo progetto sono rappresentati dall'impiego di un approccio semantico e dall'uso di logiche descrittive per la rappresentazione degli schemi locali, elementi che introducono comportamenti intelligenti in grado di rendere semi-automatico il processo di integrazione.

In questa tesi sono stati studiati vari metodi per estendere le possibilità di impiego del sistema MOMIS attraverso l'utilizzo del linguaggio XML, uno standard sempre più diffuso nel web. In particolare si è cercato di risolvere due tipi di problemi:

?? Realizzare un sistema per descrivere in XML, in maniera automatica, le informazioni memorizzate all'interno di classi Java. L'obiettivo è facilitarne lo scambio tra ambienti e sistemi software eterogenei.
?? Studiare la traduzione schemi ODL_I^3 in XML 1.0 e realizzare un software di traduzione automatica. Il traduttore pone le basi per consentire in futuro al sistema MOMIS di apparire agli utenti come una sorgente integrata di dati XML.

La soluzione del primo problema ha portato alla realizzazione di XmlTools, un modulo flessibile in grado di serializzare e ricostruire qualunque grafo di classi Java utilizzando il linguaggio XML. Il modulo è inoltre in grado di fondersi con il meccanismo standard di serializzazione previsto dal linguaggio Java.

La soluzione del secondo problema ha portato alla realizzazione di un software per la generazione automatica di una DTD che mantenga il più possibile le informazioni presenti in uno schema ODL_I^3.

La tesi è organizzata nel seguente modo:

Nel **Capitolo 1** si introduce il concetto di Integrazione Intelligente di Informazioni, descrivendo l'architettura di riferimento I^3 e la struttura di un *Mediator*e. Si elencheranno le scelte implementative adottate per il sistema MOMIS soffermandosi sulla sua architettura.

Nel **Capitolo 2** si è voluto descrivere brevemente il linguaggio XML 1.0 e una sua estensione, i namespace. Vengono fornite le nozioni fondamentali per poter comprendere il lavoro e i risultati di questa tesi. XML 1.0 e i namespace sono analizzati in chiave critica ponendone in evidenza pregi e limiti.

Nel **Capitolo 3** si analizza il problema della serializzazione e ricostruzione di grafi di oggetti in formato binario e XML. Nella seconda parte viene presentato il package XmlTools, un tool per la serializzazione e ricostruzione di grafi di oggetti Java in formato XML. Nel capitolo si affrontano i problemi legati alla progettazione e realizzazione del modulo.

Nel **Capitolo 4** si affronta il problema della traduzione di schemi ODL_I^3 in una DTD di XML 1.0. Viene descritto il linguaggio ODL_I^3 e per ogni sua caratteristica si illustra la proposta di traduzione scelta per implementare il traduttore.

Nel **Capitolo 5** viene illustrato il progetto del software di traduzione, vengono analizzate le varie scelte implementative e si descrive l'output prodotto fornendo anche numerosi esempi.

Sono inoltre presenti cinque appendici: Nell'Appendice A viene riportato un glossario della terminologia utilizzata nella tesi. Nell'Appendice B viene illustrato il protocollo di serializzazione XML implementato in XmlTools. Nell'Appendice C si riporta la DTD utilizzata per descrivere le informazioni semantiche di ODL_I^3 non direttamente traducibili in XML. Nell'Appendice D è presentata la BNF del linguaggio ODL_I^3. Infine nell'Appendice E si riporta una sintetica rappresentazione della grammatica di XML 1.0

Capitolo 1

MOMIS: un sistema di integrazione dei dati

Le reti e i sistemi informativi sono diventati parte integrante del sistema produttivo mondiale, ogni giorno nuove informazioni vengono immesse in rete e ogni giorno milioni di persone le consultano. Uno dei maggiori ostacoli connessi all'utilizzo delle informazioni presenti in rete è la difficoltà nel reperirle. Spesso la ricerca di informazioni da parte dell'utente genera un numero molto elevato di possibili contatti, rendendo di fatto inutile la ricerca (*information overload*).

Contestualmente all'aumento della probabilità di reperire un dato cercato, aumenta anche la difficoltà di recuperare questo dato in tempi e modi accettabili. Questo perché le informazioni ed i dati che le quantificano sono di diversa natura (es. testi, immagini, ecc.) ed appartengono a sorgenti eterogenee (es. pagine HTML, DBMS relazionali o ad oggetti, file system, ecc.).

Gli standard esistenti (TCP/IP, ODBC, OLE, CORBA, SQL, ecc.) risolvono solo parzialmente i problemi relativi alle diversità hardware e software, dei protocolli di rete e di comunicazione tra moduli; rimangono però irrisolti quelli relativi alla modellazione delle informazioni. Infatti, i modelli di dati e gli schemi si differenziano gli uni dagli altri in modo da dare una struttura logica ai numerosi generi di dati da memorizzare, creando così una eterogeneità semantica non risolvibile dagli standard. Un altro problema non trascurabile e l'information overload, ovvero il sovraccarico di informazioni fa sì che l'utente abbia sempre maggiori difficoltà nel discernere ed isolare i dati per lui significativi.

Altre problematiche non trascurabili riguardano: i tempi d'accesso, la salvaguardia della sicurezza ed i costi per il mantenimento della consistenza delle informazioni. Per far fronte alla molteplicità e complessità degli aspetti appena descritti, le architetture dedicate all'integrazione di sorgenti eterogenee devono essere necessariamente flessibili e modulari.

Gli approcci all'integrazione, descritti in letteratura o effettivamente realizzati, presentano diverse metodologie: la reingegnerizzazione delle sorgenti mediante standardizzazione degli schemi e la creazione di un database distribuito; il repository indipendence, un approccio che prevede di isolare al di sotto di una

vista integrata, le applicazioni ed i dati integrati dalle sorgenti, consentendo la massima autonomia e nascondendo al contempo le differenze esistenti; i datawarehouse, che realizzano presso l'utente finale delle viste materializzate, ovvero delle porzioni di sorgenti, replicando fisicamente i dati ed affidandosi ad algoritmi di allineamento per assicurarne la consistenza a fronte di modifiche nelle sorgenti.

Nel seguito verrà descritto l'approccio che in letteratura viene indicato come Intelligent Integration of Information (I^3) [15] ovvero l'approccio seguito da quei sistemi che realizzano l'Integration of Information (I^2), cioè combinano tra di loro informazioni senza replicare fisicamente i dati, utilizzando tecniche di Intelligenza Artificiale (AI).

1.1 Caratteristiche dei sistemi I^3

I sistemi che realizzano l'Integrazione Intelligente delle Informazioni, basandosi sulle descrizioni dei dati, combinano tra loro informazioni provenienti da diverse sorgenti (o parti selezionate di esse) senza dover ricorrere alla duplicazione fisica dei dati.

Questo richiede conoscenza ed intelligenza volte all'individuazione delle sorgenti e dei dati, nonché alla loro fusione e sintesi. Ciò viene raggiunto usando tecniche di Intelligenza Artificiale.

1.1.1 Programma I^3

Dal 1992 è operativo il Programma I^3, un progetto di ricerca fondato e sponsorizzato dall'ARPA (Advanced Research Projects Agency), che si prefigge di individuare un'architettura di riferimento che realizzi in maniera automatica l'integrazione di sorgenti di dati eterogenee [26].

I^3 propone l'introduzione di architetture modulari sviluppabili secondo i principi proposti da uno standard. Lo standard deve porre le basi dei servizi da soddisfare attraverso l'integrazione e deve consentire l'abbattimento dei costi di sviluppo e manutenzione. Questo renderebbe possibile ovviare ai problemi di realizzazione, manutenzione e adattabilità, inoltre, la riutilizzazione della tecnologia già sviluppata, rende la costruzione di nuovi sistemi più veloce e meno difficoltosa, con conseguente abbassamento dei costi. Per poter sfruttare un'elevata riusabilità bisogna disporre di interfacce ed architetture standard. Il paradigma suggerito per la suddivisione dei servizi e delle risorse nei diversi moduli si articola su due dimensioni:

- l'orizzontale, divisa in tre livelli: livello utente, moduli intermedi che fanno uso di tecniche di IA, risorse di dati;
- la verticale: molti domini, con un numero limitato di sorgenti.

I domini nei vari livelli non sono strettamente connessi, ma si scambiano dati ed informazioni la cui combinazione avviene a livello dell'utilizzatore, riducendo la complessità totale del sistema e permettendo lo sviluppo di applicazioni con finalità diverse.

I^3 si concentra sul livello intermedio della partizione, quello che media tra gli utenti e le sorgenti. In questo livello sono presenti vari moduli, quali:

?? **Facilitator e Mediator**: ricercano le fonti interessanti e combinano i dati da esse ricevuti;
?? **Query Processor**: riformula le query aumentando le loro probabilità di successo;
?? **Data Miner**: analizza i dati per estrarre informazioni intensionali implicite.

Nell'Appendice A è presente un glossario di termini comunemente usato in ambito I^3. Questo ha lo scopo di spiegare quei termini che dovessero risultare ambigui o poco chiari, visto il campo recente ed in evoluzione in cui si muove il progetto.

1.1.2 Architettura di riferimento per i sistemi I^3

L'obiettivo del Programma I^3 è di ridurre il tempo necessario per la realizzazione di un integratore di informazioni, fornendo una raccolta e una formalizzazione delle soluzioni prevalenti finora nel campo della ricerca.

Come si è visto, la complessità del processo di integrazione è tale da rendere estremamente utile la proposta di un'architettura di riferimento standard, che rappresenti alcuni dei servizi che un integratore di informazioni deve contenere e le possibili interconnessioni fra di essi.

Il programma individua cinque famiglie di attività omogenee, illustrate in

Figura 1 unitamente ai loro legami. La reciproca interazione tra queste attività consente di eseguire le operazioni di comunicazione, traduzione ed integrazione dei dati nelle sorgenti.

Sono inoltre evidenti due assi, uno orizzontale ed uno verticale, che permettono di intuire i diversi compiti dei vari servizi. Sull'asse orizzontale si hanno i servizi di Coordinamento ed Amministrazione, che hanno il compito di mantenere informazioni sulle capacità delle sorgenti, vale a dire che tipo di dati sono in grado di fornire e come vanno interrogate. Sempre sull'asse orizzontale si hanno poi i servizi Ausiliari, che sono responsabili delle attività di arricchimento semantico e di supporto.

Sull'asse verticale, i servizi di Coordinamento, di Integrazione e Trasformazione semantica e di Wrapping evidenziano come avviene lo scambio di informazioni.

Analizzando i vari servizi nel dettaglio:

Servizi di Coordinamento

Sono servizi ad alto livello che costituiscono l'interfaccia con l'utente, dandogli l'impressione di trattare con un sistema omogeneo.

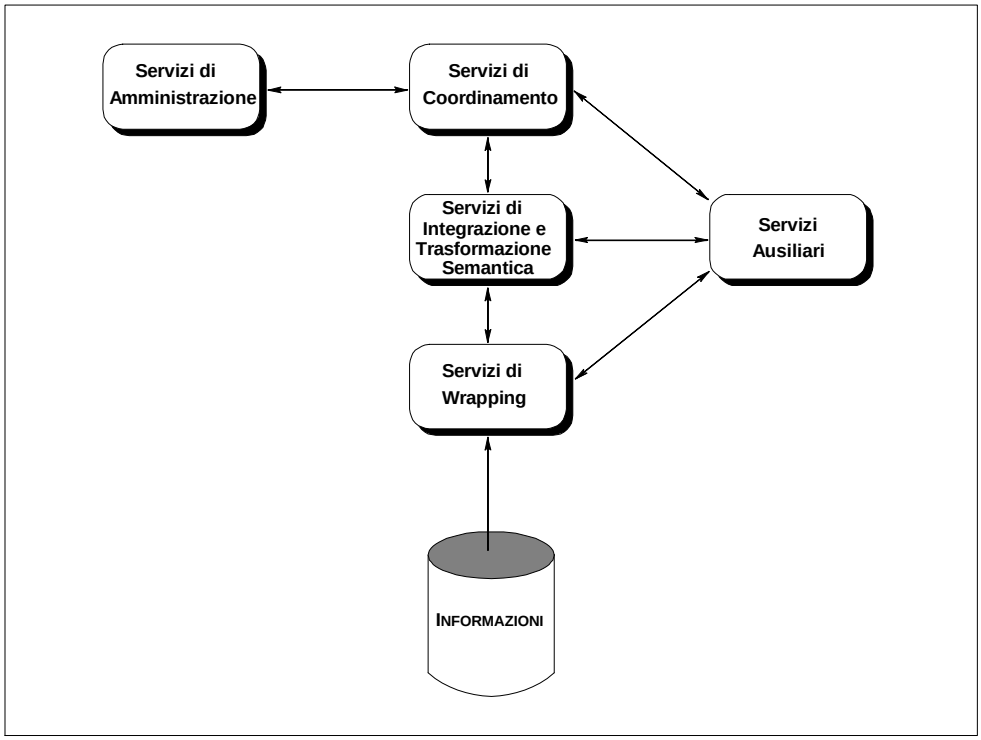

Figura 1: Diagramma dei servizi I^3

Grazie alle funzionalità messe a disposizione dalle altre famiglie di servizi, essi permettono di individuare le sorgenti di dati interessanti, ovvero quelle sorgenti che probabilmente possono dare risposta ad una determinata richiesta dell'utente. Conformemente col tipo di integratore che si è intenzionati a realizzare, i servizi di Coordinamento possono essere:

?? *Facilitation e Brokering Services*: forniscono una selezione dinamica delle sorgenti in grado di soddisfare la richiesta dell'utente. Il sistema usa un deposito di metadadati per individuare il modulo che può trattare direttamente questa richiesta, in particolare si parla di Brokering quando è coinvolto un modulo alla volta, oppure di Facilitatori o Mediatori se vi sono più moduli interessati. In quest'ultimo caso la query iniziale viene decomposta in un insieme di sottoquery da inviare a differenti moduli che gestiscono sorgenti distinte, successivamente vengono integrate le

risposte per fornirne una presentazione globale all'utente. Questo viene realizzato facendo uso di servizi di Query Decomposition e di tecniche di Inferenza (mutuate dall'Intelligenza Artificiale) per una determinazione dinamica delle sorgenti da interrogare.

?? *Matchmaking*: il mapping fra informazioni integrate e locali è effettuato manualmente da un'operatore in fase di inizializzazione. In questo caso tutte le richieste vengono trattate allo stesso modo.

Servizi di Amministrazione

Questi servizi sono utilizzati da quelli di Coordinamento per localizzare le sorgenti utili, determinare la loro capacità, creare ed interpretare Template. I Template sono strutture dati che descrivono i servizi, le fonti ed i moduli da utilizzare per realizzare un determinato task. Queste strutture dati servono quindi per ridurre al minimo le possibilità di decisione del sistema, consentendo di definire a priori le azioni da eseguire a fronte di una determinata richiesta.

Al posto dei Template e possibile usare le Yellow pages, ovvero servizi di directory che mantengono le informazioni sul contenuto delle sorgenti e sul loro stato. In questo modo, le Yellow pages consentono al Mediatore di inviare la richiesta di informazioni alla sorgente giusta o, se non fosse disponibile, ad una equivalente.

Tra questi tipi di servizio vi sono il Browser, che permette di “navigare” tra le descrizioni degli schemi delle sorgenti, recuperando informazioni, e gli Iterative Query Formulation, che aiutano l'utente a rilassare o specificare meglio alcuni vincoli dell'interrogazione al fine di ottenere risposte più precise.

Servizi di Integrazione e Trasformazione Semantica

Questi servizi supportano le manipolazioni necessarie per l'integrazione e la trasformazione delle informazioni. Hanno in input una o più sorgenti di dati, e restituiscono come output la ``vista" integrata o trasformata di queste informazioni. Spesso sono indicati come servizi di Mediazione, essendo tipici dei moduli mediatori. I principali sono:

?? **Servizi di integrazione di schemi**: creano il vocabolario e le ontologie condivise dalle sorgenti, integrano gli schemi in una vista globale, mantengono il mapping tra schemi globali e sorgenti;
?? **Servizi di integrazione di informazioni**: aggregano, riassumono ed astraggono i dati per fornire presentazioni analitiche significative;
?? **Servizi di supporto al processo di integrazione**: sono utilizzati quando una query deve essere scomposta in più sottoquery da inviare a fonti differenti, con la necessità di integrare poi i loro risultati.

Servizi di Wrapping

Il fine di questi servizi e far sì che le fonti di informazione aderiscano ad uno standard. Sono praticamente dei traduttori dai sistemi locali ai servizi di alto livello dell'integratore.

I servizi di Wrapping permettono ai servizi di Coordinamento e di Mediazione di manipolare in modo uniforme le sorgenti locali. Forniscono un'interfaccia che, seguendo gli standard più diffusi (ad esempio: SQL come linguaggio di interrogazione, CORBA come protocollo di scambio), permette alle sorgenti estratte di essere accedute dal maggior numero possibile di sistemi mediatori.

Servizi Ausiliari

Aumentano le funzionalità degli altri servizi. Possono svolgere varie funzioni, tra cui: monitoraggio del sistema, propagazione di aggiornamenti, attività di ottimizzazione, ecc.

1.1.3 Il mediatore

Secondo la definizione proposta da Wiederhold in [16] "un mediatore è un modulo software che sfrutta la conoscenza su un certo insieme di dati per creare informazioni per una applicazione di livello superiore… Dovrebbe essere piccolo e semplice, così da poter essere amministrato da uno, o al più pochi, esperti."

Un mediatore presenta allora i seguenti compiti:

- ?? assicurare un servizio stabile, anche nel caso di cambiamento delle risorse;
- ?? amministrare e risolvere le eterogeneità delle diverse fonti;
- ?? integrare le informazioni ricavate da più risorse;
- ?? presentare all'utente le informazioni attraverso un modello scelto dall'utente stesso.

Il progetto MOMIS, di cui questa tesi fa parte, ha come obiettivo la progettazione e realizzazione di un Mediatore, come descritto in [29][14][3]. L'ipotesi di avere a che fare esclusivamente con sorgenti di dati strutturati e semistrutturati, ha consentito di restringere il campo applicativo del sistema con una conseguente diminuzione delle problematiche riscontrate in fase di progettazione e realizzazione.

L'approccio architetturale scelto e quello classico, che si sviluppa su tre livelli principali:

1. *utente*: attraverso un'interfaccia grafica l'utente pone delle query su uno schema globale e riceve un'unica risposta, come se stesse interrogando un'unica sorgente di informazioni;

2. *mediatore*: il mediatore gestisce l'interrogazione dell'utente, combinando, integrando ed eventualmente arricchendo i dati ricevuti dai wrapper, ma usando un modello (e quindi un linguaggio di interrogazione) comune a tutte le fonti;
3. *wrapper*: ogni wrapper gestisce una singola sorgente, convertendo le richieste del mediatore in una forma comprensibile dalla sorgente, e le informazioni da essa estratte nel modello usato dal mediatore.

Facendo riferimento ai servizi descritti nel paragrafo 1.1.2, l'architettura del mediatore che si è progettato è riportata in Figura 2. In particolare sono stati sviluppati i servizi di Integrazione e Trasformazione Semantica. Inoltre l'impostazione architetturale mostra come il sistema mediatore progettato vuole distaccarsi dall'approccio strutturale e sintattico, tuttora dominante tra i sistemi presenti sul mercato [17][42][43].

Quando si parla di approccio strutturale, si fa riferimento all'uso di un self-describing model per rappresentare gli oggetti da integrare, limitando l'uso delle informazioni semantiche alle regole predefinite dall'operatore. Il sistema non conosce a priori la semantica di un oggetto recuperato da una sorgente, bensì è l'oggetto stesso che, attraverso delle etichette, si autodescrive. I vantaggi di questo approccio sono: la possibilità di integrare in modo completamente trasparente al mediatore basi di dati fortemente eterogenee e magari mutevoli nel tempo; per trattare in modo omogeneo dati che descrivono lo stesso concetto o che hanno concetti in comune, ci si basa sulla definizione manuale di rule, che permettono di identificare i termini, che devono essere condivisi da più oggetti.

Altri progetti, tra cui MOMIS, seguono invece un approccio all'integrazione di tipo semantico, che prevede che siano soddisfatti i seguenti punti:

?? il mediatore deve conoscere, per ogni sorgente, lo schema concettuale (metadati);
?? le informazioni semantiche sono codificate in questi schemi;
?? deve essere disponibile un modello comune per descrivere le informazioni da condividere (e dunque per descrivere anche i metadati);
?? deve essere possibile una integrazione (parziale o totale) delle sorgenti di dati.

In questo modo, sfruttando le informazioni semantiche che necessariamente ogni schema sottintende, il mediatore può individuare concetti comuni a più sorgenti e relazioni che li legano.

1.1.4 Problematiche da affrontare

Pur avendo a disposizione gli schemi concettuali delle varie sorgenti, non è certamente un compito banale individuare i concetti comuni ad esse e le relazioni che possono legarli, ne tantomeno è semplice realizzare una loro coerente integrazione. Trascurando le differenze dei sistemi fisici (alle quali dovrebbero

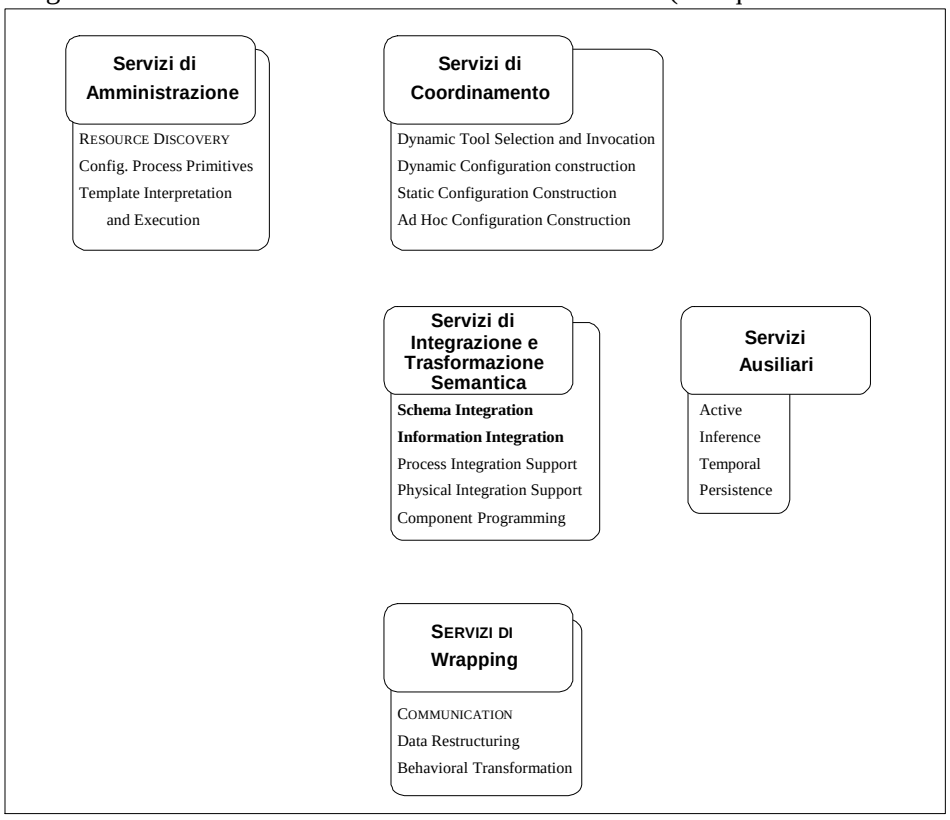

Figura 2: servizi I^3 presenti nel mediatore

provvedere i moduli wrapper) i problemi a livello di mediazione che si e dovuto risolvere (o coi quali si e dovuti scendere a compromessi) sono:

Problemi ontologici

Come riportato in Appendice A, per ontologia si intende, in questo ambito, "l'insieme dei termini e delle relazioni usate in un dominio, per indicare oggetti e concetti". In sostanza con ontologia ci si riferisce a quell'insieme di termini che, in un particolare dominio applicativo, denotano una particolare conoscenza e fra i quali non esiste ambiguità perché sono condivisi dall'intera comunità di utenti del dominio applicativo stesso.

Fra i diversi livelli di ontologia esistenti (*top-level ontology, domain and task ontology, application ontology*, ecc…) [22][23], ognuno con le proprie problematiche, si è assunto di muoversi all'interno delle domain ontology, ipotizzando quindi che tutte le fonti informative condividano almeno i concetti fondamentali (ed i termini con cui identificarli).

Problemi semantici

Pur ipotizzando che anche sorgenti diverse condividano una visione similare del problema da modellare, e quindi un insieme di concetti comuni, è improbabile che usino la stessa semantica, cioè gli stessi vocaboli e le stesse strutture dati per rappresentare questi concetti.

Come riportato in [13] la causa principale delle differenze semantiche si può identificare nelle diverse concettualizzazioni del mondo esterno che persone distinte possono avere, ma non è l'unica. Le differenze nei sistemi di DBMS possono portare all'uso di differenti modelli per la rappresentazione della porzione di mondo in questione: partendo cosi dalla stessa concettualizzazione, determinate relazioni tra concetti avranno strutture diverse a seconda che siano realizzate attraverso un modello relazionale o ad oggetti. L'obiettivo dell'integratore, che è fornire un accesso integrato ad un insieme di sorgenti, si traduce allora nel non facile compito di identificare i concetti comuni all'interno di queste sorgenti e risolvere le differenze semantiche che possono essere presenti tra di loro. Possiamo classificare queste contraddizioni semantiche in tre gruppi principali:

1. *eterogeneità tra le classi di oggetti*: benché due classi in due differenti sorgenti rappresentino lo stesso concetto nello stesso contesto, possono usare nomi diversi per gli stessi attributi, per i metodi, oppure avere gli stessi attributi con domini di valori diversi o ancora (dove questo e permesso) avere regole differenti su questi valori;
2. *eterogeneità tra le strutture delle classi*: comprendono le differenze nei criteri di specializzazione, nelle strutture per realizzare una aggregazione, ed anche le discrepanze schematiche, quando cioè valori di attributi sono invece parte dei metadati in un altro schema (come può essere l'attributo SESSO in uno schema, presente invece nell'altro implicitamente attraverso la divisione della classe PERSONE in MASCHI e FEMMINE);
3. *eterogeneità nelle istanze delle classi*: ad esempio, l'uso di diverse unità di misura per i domini di un attributo, o la presenza/assenza di valori nulli.

È però possibile sfruttare adeguatamente queste differenze semantiche per arricchire il nostro sistema: analizzando a fondo queste differenze e le loro motivazioni si può arrivare al cosiddetto arricchimento semantico, ovvero all'aggiungere esplicitamente ai dati tutte quelle informazioni che erano

originariamente presenti solo come metadati negli schemi, dunque in un formato non interrogabile.

1.2 Il sistema MOMIS

Considerando le problematiche descritte nel paragrafo precedente, nonché alcuni sistemi preesistenti [17][42][43][8][25][34][5][19][20][41][40], si è giunti alla progettazione di un sistema intelligente di integrazione di informazioni provenienti da sorgenti di dati strutturati e semistrutturati denominato **MOMIS** (**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources). Il contributo innovativo di questo progetto, rispetto ad altri similari, risiede nell'impiego di un approccio semantico e nell'uso di logiche descrittive per la rappresentazione degli schemi delle sorgenti, elementi che introducono comportamenti intelligenti in grado di rendere semi-automatica la fase di integrazione.

Un lavoro approfondito è stato svolto anche riguardo alla fase di query processing [2][1][30][18], cioè quel processo che dalla query posta sullo schema unificato, provvede a generare automaticamente le sottoquery da inviare alle sorgenti ed ad integrare i risultati. MOMIS nasce all'interno del progetto MURST 40% INTERDATA e si sviluppa in D2I dalla collaborazione tra i gruppi operativi dell'Università di Modena e Reggio Emilia e di quella di Milano.

1.2.1 L'approccio adottato

MOMIS adotta un approccio di integrazione delle sorgenti semantico e virtuale [2]. Il concetto di “semantico” è stato illustrato nella Sezione 1.1.3. Con “virtuale” [27] si intende invece che la vista integrata delle sorgenti, rappresentata dallo schema globale, non viene materializzata. Per ottenere i dati cercati, il sistema accetta query sullo schema globale, le decompone generando delle subquery da inviare alle varie sorgenti locali; lo schema globale dovrà inoltre disporre di tutte le informazioni atte alla fusione dei risultati ottenuti localmente per poter ottenere una risposta significativa.

Le motivazioni che hanno portato all'adozione di un approccio come quello descritto sono varie:

1. la presenza di uno schema globale permette all'utente di formulare qualsiasi interrogazione che sia con esso consistente;
2. le informazioni semantiche che comprende possono contribuire ad una eventuale ottimizzazione delle interrogazioni;
3. l'adozione di una semantica *type as a set* per gli schemi permette di controllarne la consistenza facendo riferimento alle loro descrizioni;
4. la vista virtuale rende il sistema estremamente flessibile, in grado cioè di sopportare frequenti cambiamenti sia nel numero che nel tipo delle

sorgenti, ed anche nei loro contenuti (non occorre prevedere onerose politiche di allineamento);

Si è deciso di adottare, sia per la rappresentazione degli schemi che per la formulazione delle interrogazioni, un unico modello dei dati basato sul paradigma ad oggetti. Il modello comune dei dati utilizzato nel sistema (ODM_{I^3}) è di alto livello e facilita la comunicazione tra il mediatore ed i wrapper. Per definire questo modello si e cercato di seguire le raccomandazioni relative alla proposta di standardizzazione per i linguaggi di mediazione, nata in ambito I^3: un mediatore deve poter essere in grado di gestire sorgenti dotate di formalismi complessi (ad es. quello ad oggetti) ed altre decisamente più semplici (come i file di strutture), è quindi preferibile l'adozione di un formalismo il più completo possibile.

Per la descrizione degli schemi si è arrivati a definire il linguaggio ODL_I^3 [1][14] [2][29] che si presenta come estensione del linguaggio standard ODL proposto dal gruppo di standardizzazione ODMG-93.

Per quanto riguarda il linguaggio di interrogazione si è adottato OQL_I^3 che adotta la sintassi OQL senza discostarsi dallo standard. Questo linguaggio risulta estremamente versatile ed espressivo fornendo la possibilità di sfruttare le informazioni rappresentate nello schema globale.

Inoltre si è cercato di utilizzare uno standard comune di comunicazione tra i vari moduli MOMIS al fine di rendere ancora più agevole l'ampliamento futuro. Si è deciso di adottare lo standard CORBA (Common ORB Architecture) per le comunicazioni tra i moduli corba. CORBA è una tecnologia per l'integrazione, inoltre è ad oggetti ed una modellazione di questo tipo permette di ridurre la complessità di MOMIS: esistono difatti metodologie consolidate per la rappresentazione e progettazione di sistemi ad oggetti (OMT, UML), ma soprattutto per utilizzare un oggetto è sufficiente conoscerne l'interfaccia pubblica e questo favorisce il lavoro degli sviluppatori che verranno.

1.2.2 L'architettura generale di MOMIS

Momis è stato progettato per fornire un accesso integrato ad informazioni eterogenee memorizzate sia in sorgenti strutturate, come database relazionali, database ad oggetti e semplici file, sia in sorgenti semistrutturate, come le sorgenti descritte in XML.

Come si può vedere nella Figura 3, i componenti del sistema MOMIS sono disposti su tre livelli:

Livello Dati

A questo livello si trovano i Wrapper. Posti al di sopra di ciascuna sorgente, sono i moduli che fungono da interfaccia tra il mediatore vero e proprio e le sorgenti locali di dati. Le funzioni da loro svolte sono:

?? in fase di integrazione forniscono una descrizione delle informazioni contenute nelle sorgenti, utilizzando il linguaggio ODL_I^3.

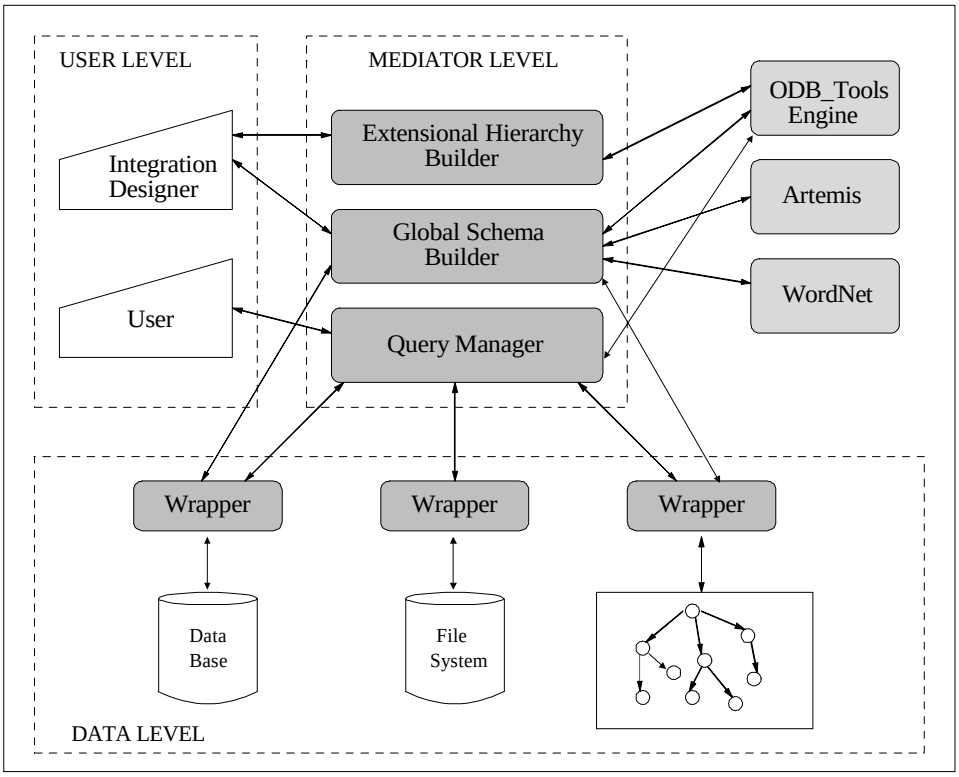

Figura 3: Architettura generale di MOMIS

?? in fase di Query Processing, traducono la query ricevuta dal mediatore (espressa in OQL_I^3) in una interrogazione comprensibile ed eseguibile dalla sorgente stessa. Inoltre i wrapper devono esportare i dati ricevuti in risposta all'interrogazione, presentandoli al mediatore attraverso il modello ODM_I^3.

MOMIS ha un'architettura distribuita su tre livelli, considerando anche le sorgenti i livelli diventano quattro.

Livello Mediatore

Il mediatore è il cuore del sistema ed è composto da tre moduli, ognuno preposto a funzionalità ben precise.

?? *Global Schema Builder*

La sua funzione principale è quella di generare lo Schema Globale. Il modulo riceve in input le descrizioni degli schemi locali delle sorgenti espressi in ODL_I^3 e forniti ognuno dal relativo wrapper. A questo punto (utilizzando strumenti di ausilio quali ODB-Tools Engine, WordNet, ARTEMIS) il Global Schema Builder è in grado di costruire la vista virtuale integrata (Global Schema) utilizzando tecniche di clustering e di Intelligenza Artificiale. In questa fase e prevista anche l'interazione con il progettista il quale, oltre ad inserire le regole di mapping, interviene nei processi che non possono essere svolti automaticamente dal sistema (come ad es. l'assegnamento dei nomi alle classi globali, la modifica di relazioni lessicali, …).

?? *Extensional Hierarchy Builder*
Questo modulo si occupa della generazione della Conoscenza Estensionale (Gerarchie Estensionali e Base Extension) necessaria per ottimizzare le interrogazioni.

?? *Query Manager*
È il modulo di gestione delle interrogazioni. In questa fase la singola query posta in OQL_I^3 dall'utente sullo Schema Globale (che chiameremo Global Query) sarà rielaborata in più Local Query (anch'esse espresse in OQL_I^3) da inviare alle varie sorgenti, o meglio ai wrapper predisposti alla loro traduzione, come abbiamo visto. Questa traduzione avviene in maniera automatica da parte del Query Manager utilizzando la conoscenza intensionale ed estensionale definite nella precedente fase di integrazione.

Livello Utente

L'utente del sistema può interrogare lo schema globale e per lui sarà come interrogare un database tradizionale. La query posta dall'utente sullo schema globale viene passata come input al Query Manager, che interroga le sorgenti e fornisce all'utente la risposta cercata. Tutte queste operazioni, per l'utente, risultano completamente trasparenti.

1.2.3 Tools di supporto

Per realizzare il processo di integrazione degli schemi il sistema mediatore MOMIS sfrutta anche alcuni tool esterni. Questi sono:

ODB-Tools

È uno strumento software sviluppato presso il dipartimento di Ingegneria dell'Università di Modena e Reggio Emilia [9][10][7]. Esso si occupa della

validazione di schemi e dell'ottimizzazione semantica di interrogazioni rivolte a Basi di Dati orientate agli Oggetti (OODB).

L'architettura di ODB-Tools, come si vede in Figura 4: Architettura di ODB-Tools prevede vari componenti, tra cui:

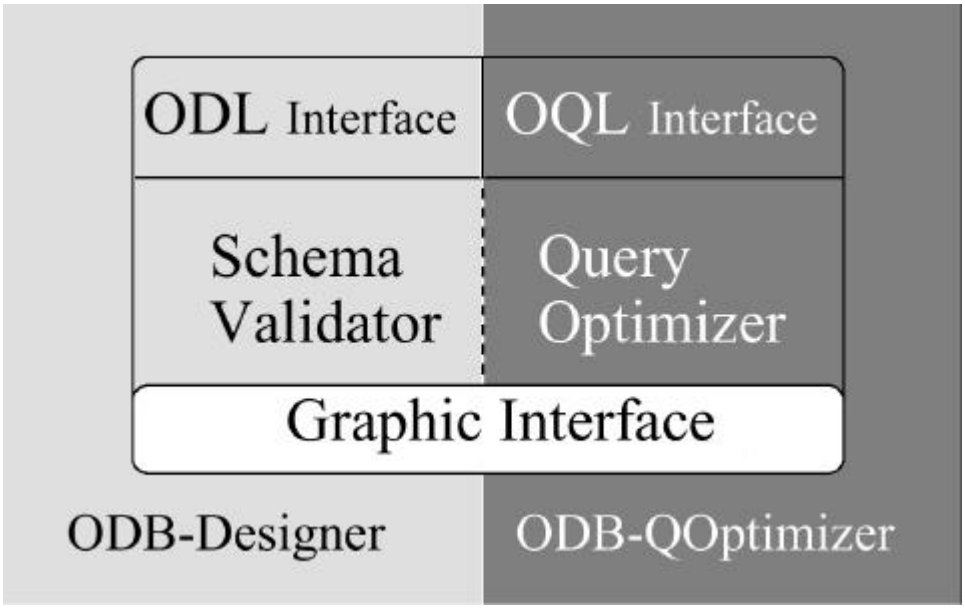

Figura 4: Architettura di ODB-Tools

?? *ODB-Designer* si occupa della validazione di schemi: si può inserire la descrizione di uno schema di database (in ODL) ed il sistema realizzerà automaticamente la sua validazione e la sua riclassificazione (verifica che non vi siano classi incoerenti e calcola relazioni di specializzazione non esplicitate dallo schema).

?? *ODB-Qoptimizer* si occupa dell'ottimizzazione semantica delle interrogazioni: se si inserisce una query (in OQL) posta su di un determinato schema, questa viene automaticamente riformulata in una equivalente, ma più efficiente, sfruttando l'espansione semantica ed i vincoli di integrità.

WordNet

È un database lessicale on-line in lingua inglese. Esso è capace di individuare relazioni semantiche fra termini; cioè dato un insieme di termini, WordNet è in grado di identificare l'insieme di relazioni lessicali che li legano [4].

ARTEMIS

(Analisys and Reconciliation Tool Enviroment for Multiple Information Sources) [28]: è uno strumento software sviluppato presso l'Università di Milano e Brescia. Riceve in ingresso il thesaurus, cioè l'insieme delle relazioni terminologiche (lessicali e strutturali) generate da MOMIS, e sulla base di queste assegna ad ogni classe coinvolta nelle relazioni un coefficiente numerico indicante il suo grado di affinità con le altre classi. Questi coefficienti servono per raggruppare le classi locali in modo tale che ogni gruppo (cluster) comprenda solo classi con coefficienti di affinità simili.

Capitolo 2

Introduzione a XML 1.0

Questo breve capitolo non vuole essere una trattazione esaustiva dello standard XML 1.0, vuole invece descrivere le principali caratteristiche del linguaggio in modo che il lettore possa affrontare i successivi capitoli con un sufficiente background semantico e terminologico.

XML significa `Extensible Markup Language' (estensibile perché non è un formato rigido come HTML). È una "recommendation[1]" emanata dal World Wide Web Consortium (W3C), un importante organismo per lo sviluppo di nuovi standard industriali nel campo informatico. XML è stato progettato in modo da poter utilizzare SGML (un metalinguaggio che è divenuto uno standard internazionale per i sistemi di markup testuali) sul World Wide Web.

XML non è un linguaggio predefinito, non è nemmeno un ***singolo*** linguaggio: è un metalinguaggio – un linguaggio per descrivere altri linguaggi – che permette di definire i propri markup.

Un linguaggio di markup predefinito come HTML definisce un modo per descrivere l'informazione utilizzando solamente una specifica classe di documenti: XML permette invece di definire dei propri linguaggi di markup personalizzati per descrivere innumerevoli classi di documenti. Tutto questo è possibile in quanto XML è scritto in SGML.

XML di per sé è una scatola vuota, le sue specifiche indicano *come* scambiarsi informazioni non dicono *cosa* queste rappresentano. La semantica di un tag è interpretata da chi utilizza il documento XML, se questi non è in grado di attribuire un corretto significato all'informazione che sta esaminando, allora parte delle potenzialità di XML vengono neutralizzate. Le potenzialità sono neutralizzate solo in parte perché il fruitore del documento può semplicemente limitarsi a registrare o ignorare le informazioni contenute nel tag sconosciuto e passare al successivo.

[1] Si veda il glossario per la terminologia interna al W3C.

2.1 XML 1.0

Le specifiche di XML 1.0 forniscono una sintassi universale per la rappresentazione dei dati; con questo linguaggio è possibile definire un qualunque tag e assegnargli il significato che si vuole. L'informazione che si vuole scambiare viene veicolata o come contenuto del tag oppure come valore dei suoi attributi.

```
<my_tag my_attribute0="information"
        my_attribute1="information">
  information
</my_tag>
```

In XML i dati e i metadati, ossia le informazioni che descrivono i dati, sono fusi in un unico formato. Si consideri il seguente esempio di codice XML, relativo a delle pubblicazioni scientifiche:

```
<Publication URL="ftp://db.stanford.edu/pub/papers/xml.ps"
Authors="RG JM JW">
     <Title>From Semistructured Data to XML:
            Migrating the Lore Data Model
            and Query Language
     </Title>
     <Published>Proceedings of the 2nd International
               Workshop on the Web
               and Databases (WebDB '99)
     </Published>
     <Pages>25-30</Pages>
     <Location>
       <City>Philadelphia</City>
       <State>Pennsylvania</State>
     </Location>
     <Date>
       <Month>June</Month>
       <Year>1999</Year>
     </Date>
</Publication>
<Publication RL="ftp://db.stanford.edu/pub/papers/ozone.ps"
             Authors="TL SA JW">
     <Title>Ozone: Integrating Structured and
            Semistructured Data
     </Title>
     <Published>Technical Report
     </Published>
     <Institution>Stanford University Database Group
     </Institution>
```

```
      <Date>
        <Month>October</Month>
        <Year>1998</Year>
      </Date>
</Publication>
<Author ID="SA">S. Abiteboul</Author>
<Author ID="RG">R. Goldman</Author>
<Author ID="TL">T. Lahiri</Author>
<Author ID="JM">J. McHugh</Author>
<Author ID="JW">J. Widom</Author>
```

La struttura del documento è delineata dai tag che descrivono l'informazione contenuta. Un tag `<some_tag some_attribute="some_value">` compreso il suo contenuto, nella terminologia XML è detto *elemento*.

Un elemento comprende dunque il tag iniziale `<some_tag>` , il tag finale `</some_tag>` e tutto il testo racchiuso tra i due. Gli elementi possono contenere altri elementi definendo una struttura con un qualunque livello di profondità.

XML introduce inoltre un meccanismo simile alle macro del C/C++, si possono definire e riferire delle "entity":

```
<!ENTITY product_name "Super boom">

...
<description> &product_name; can blown anything
</description>
...
```

La definizione di entity può avvenire solamente all'interno della "Document Type Declaration" (DTD) che verrà descritta nel seguito. La definizione di una entity non deve necessariamente risiedere all'interno del documento XML ma può essere definita in altri file reperibili tramite la loro URL.

Well formedness.

Un documento XML deve rispettare i vincoli di well-formedness, ossia deve essere correttamente formattato. I vari vincoli che seguono sono espressi in maniera qualitativa in modo che si colga immediatamente il loro significato. Per evitare il sorgere di possibili ambiguità si rimanda alle specifiche [37] per la definizione rigorosa dei vincoli di well-formedness:

?? Ogni start tag `<tag>` deve avere un corrispondente end tag `</tag>` che utilizzi lo stesso identificatore (`tag`). Non devono esistere tag iniziali o finali non accoppiati. Unica eccezione è la seguente: se un elemento è privo di contenuto, può essere riportato in modo equivalente nei due modi seguenti:

```
<tag someattribue=″some_value″></tag>
   oppure
<tag someattribue=″some_value″/>
```

Per contenuto di un elemento si intende ogni informazione contenuta tra i tag iniziale e finale escludendo eventuali attributi.

?? **Gli elementi devono essere correttamente innestati**: possono essere integralmente contenuti in altri elementi, essere vuoti, oppure essere l'elemento radice. Non devono esistere elementi che si sovrappongono solo parzialmente.

?? **L'elemento radice deve essere unico**: ogni documento XML deve avere un unico elemento che contiene tutti gli altri.

?? Un elemento può avere un qualunque numero di attributi. Gli attributi sono contenuti nel tag iniziale, devono essere separati da spazi o simboli di "nuova linea" e sono nella forma:

```
attribute_name = "value"
```

Il valore di un attributo è una stringa di caratteri in cui non è ammesso il carattere "<", che rimane riservato per la definizione di un tag.

?? **Un elemento non può avere attributi duplicati**, ossia più attributi con lo stesso nome.

?? Il contenuto di un elemento può essere formato da testo e/o da altri elementi. Nel testo non deve mai comparire il carattere "<" a meno che non si stia definendo un tag iniziale. Il carattere "<" va sostituito con un riferimento ad una entity predefinita (`<` in modo analogo ad HTML). Anche il carattere "&" può essere utilizzato solamente per riferire una entity, in caso contrario si deve utilizzare l'entity predefinita `&`;

2.1.1 Document Type Declaration (DTD)

Un documento XML che rispetti tutti i vincoli di well-formedness è corretto sintatticamente e verrà accettato da tutti i parser XML ma chi assicura la correttezza degli elementi?

Un documento XML può facoltativamente dichiarare per esplicito la sua struttura. Può indicare quali elementi sono ammessi e dove devono essere collocati. Per ogni elemento è possibile indicare quali attributi possono comparire. Esiste una apposita sezione nel documento XML adibita alle dichiarazioni che ne descrivono la struttura: la DTD.

Validity

L'uso della DTD equivale a definire la grammatica del documento XML e comporta l'introduzione del concetto di *validità*.

Un documento XML si dice valido se rispetta tutti i vincoli di validità imposti dalla DTD. I vincoli imposti restringono gli elementi e gli attributi che possono comparire in un documento XML.

La DTD individua quindi una classe di documenti XML aventi caratteristiche simili. Ogni documento XML valido è spesso indicato come documento istanza di una determinata DTD.

Le dichiarazioni della DTD seguono una sintassi differente rispetto a quella del documento istanza, per dichiarare un elemento si utilizza la clausola

`<!ELEMENT` *element_name content* `>`[2]

element_name è un qualunque identificatore XML, *content* invece può assumere varie forme e descrive il contenuto dell'elemento.

Ad esempio se si vuole dichiarare l'elemento `<case>` in modo che possa contenere solo del testo:

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

#PCDATA (Parsed Character DATA) è l'unico tipo di dato possibile in un documento XML, non vi sono distinzioni tra stringhe o numeri. Il termine "Parsed" indica che il testo è interpretato e può contenere riferimenti ad entity.

Il termine *content* descrive il modello del contenuto dell'elemento (content model) e può assumere le seguenti forme:

- `EMPTY` : l'elemento che si sta dichiarando non può contenere nulla, né testo , né altri elementi.
- `ANY` : l'elemento può contenere uno qualunque degli elementi dichiarati nella DTD.
- Elementi figli.
- Mixed content.

Per gli ultimi due casi occorre spendere qualche parola in più:

Gli elementi figli indicati con i nomi di tag, possono comparire in sequenza (operatore ",") o in alternativa (operatore "|"). Nel documento istanza i vari elementi devono occupare precisamente la posizione assegnata loro dalla DTD in caso contrario il documento non è valido.

Esempio:

[2] Per una BNF formale si rimanda all'Appendice E o alle specifiche [37].

Se si vuole dichiarare che l'elemento <computer> può contenere solo gli elementi <case> , <CPU> e <keyboard> si indica:

```
<!ELEMENT computer (case, CPU, keyboard)>
```

mentre

```
<!ELEMENT CPU (Intel | AMD)>
```

Impone che l'elemento CPU possa contenere solo un elemento da scegliersi tra l'elemento Intel o AMD.

Tra gli operatori "," e "|" non è definita alcuna priorità, le parentesi devono essere utilizzate obbligatoriamente per evitare situazioni ambigue, **è infatti un errore dichiarare**:

```
<!ELEMENT computer (case, keyboard, mouse | trackball)>
```

La DTD supporta inoltre gli operatori *, + e ? per indicare la cardinalità di un elemento:

?? * zero o più occorrenze dell'elemento
?? + una o più occorrenze
?? ? l'elemento è facoltativo

la dichiarazione

```
<!ELEMENT computer (case, PCI_cards*, keyboard, mouse?)>
```

indica che l'elemento computer può contenere un numero qualunque di schede ed eventualmente un mouse.

Altra tipologia di contenuto per un elemento è il "mixed content" con questa modalità è possibile alternare testo ed elementi in maniera analoga all'HTML.

Ad esempio:

```
<!ELEMENT htmltext (#PCDATA | em| b| i| h1 |h2)* >
```

Esistono delle limitazioni all'uso del tipo #PCDATA nei content model: bisogna seguire rigorosamente le specifiche sul mixed content: #PCDATA deve essere sempre la prima delle alternative, deve comparire una volta sola e alla fine deve essere presente l'operatore *.

Attributi

Per ogni elemento è possibile dichiarare una lista di attributi ammissibili ed è possibile specificarne varie caratteristiche. La dichiarazione di un attributo avviene utilizzando la clausola `ATTLIST`:

```
<!ATTLIST element_name
  attribute_i      type          default_spec
...
>
```

element_name	nome dell'elemento a cui si riferiscono gli attributi.
type	determina quali valori può assumere un attributo
default_spec	indica il valore di default dell'attributo, se è obbligatorio ecc.

i … indicano che in una clausola `ATTLIST` possono essere dichiarati più attributi.
type può assumere le seguenti forme:

type	**descrizione**
`(value1 | value2 | ...)`	Una lista di valori in alternativa tra loro
`CDATA`	una semplice stringa di testo
`ID`	un valore univoco non condiviso da altri attributi ID.
`IDREF`	un riferimento ad un valore di un attributo ID definito altrove nel documento.
`IDREFS`	una lista di riferimenti ad altri attributi ID separata da spazi.
`ENTITY`	Il nome di una entity definita nella DTD.
`ENTITIES`	una lista di entity separata da spazi.
`NMTOKEN`	un nome XML valido composto di lettere, numeri, trattini, underscore e ":"
`NMTOKENS`	una lista separata da spazi di nomi XML.
`NOTATION`	Il nome di una notation definita nella DTD; una notation specifica dei dati non in formato XML; questa specifica sta divenendo obsoleta, quindi non verrà trattata.

Lista di valori

L'attributo può assumere solamente uno dei valori indicati nella DTD, ogni altro valore è proibito.

ID, IDREF

Questi sono attributi di tipo speciale, non possono esistere due attributi ID in tutto il documento XML che abbiano lo stesso valore. Questo è il meccanismo che di solito viene utilizzato per accertarsi dell'identità di un elemento XML, in questo caso il valore dell'attributo ID deve rappresentare una chiave unaria per l'entità che rappresenta l'elemento XML.

Gli attributi IDREF possono assumere solo dei valori assegnati ad attributi ID, ogni altro valore è proibito. La presenza di un attributo di questo tipo assicura l'esistenza dell'attributo riferito.

default_spec	**Significato**
`#REQUIRED`	L'attributo è obbligatorio e il suo valore deve essere specificato dalle istanze.
`#IMPLIED`	L'attributo è facoltativo e può anche non essere presente nelle istanze.
`"defaultValue"`	Se il documento non specifica un valore per l'attributo allora viene utilizzato questo valore di default.
`#FIXED "fixedValue"`	Se il documento specifica questo attributo, il suo valore deve coincidere con quello specificato altrimenti si intende che l'attributo abbia comunque questo valore.

Ad esempio:

```
<!ELEMENT CPU (Intel, AMD)>
<!ATTLIST CPU
  speed     (1.5GHz | 1.0GHz)      "1.0GHz"
  model      CDATA                  #REQUIRED
>
```

dichiara che l'attributo speed deve valere o 1.5GHz o 1.0GHz e ha come valore di default il secondo; inoltre model è un attributo contenente una stringa che deve sempre essere specificato negli elementi CPU.

In un documento XML può essere dichiarata una sola DTD anche se può essere formata dall'unione da tanti moduli sparsi in file differenti. Per includere una DTD esterna tra le definizioni della DTD che si sta dichiarando è sufficiente dichiarare una entity e referenziarla, ad esempio:

```
<!ENTITY % external_dtd SYSTEM "/dtd/my_external.dt d">
%external_dtd;
```

Per la definizione e l'espansione di entity all'interno della DTD si usa il simbolo % per differenziarle da quelle ordinarie utilizzate nel documento istanza.

Per dichiarare l'utilizzo di una DTD all'interno di un documento istanza si utilizza la clausola `DOCTYPE`. Tramite questo costrutto si dichiara in oltre quale sarà l'elemento radice del documento e la DTD può essere specificata come una DTD locale oppure pubblica accessibile attraverso la rete.

`<!DOCTYPE` *root_name location* `[` *local definitions* `]>`

In un documento è possibile utilizzare un unico `DOCTYPE`.

root_name	è il nome dell'elemento che contiene tutti gli altri elementi presenti nel documento.
location	identifica l'ubicazione della DTD, è facoltativo.
local definitions	le dichiarazioni di elementi, entity e attributi incluse in questa posizione sono considerate precedenti a tutte quelle contenute nella DTD riferita.

Esempio:

```
<!DOCTYPE laboratory PUBLIC "www.pc-lab.org/computers.dtd"[

  <!ELEMENT laboratory (computer*)>
]>
```

Una DTD permette inoltre la definizione di sezioni condizionali, ossia insiemi di dichiarazioni che possono essere utilizzate o meno dai parser ai fini della validazione. Il criterio con cui una sezione entra o no a far parte della DTD effettiva si basa sulla definizione di entity nel documento istanza. Una trattazione dettagliata di questi aspetti esula dagli scopi di questo breve capitolo introduttivo e può essere trovata in [11].

2.1.2 Limiti della DTD

La DTD consente di dichiarare la struttura di un documento XML e di validare il suo contenuto, tuttavia gli strumenti che fornisce sono molto limitati.

?? Gli elementi dichiarati nella DTD devono avere un nome univoco: non sono ammesse multiple definizioni di un elemento, questo causa molti problemi quando si devono integrare DTD provenienti da sorgenti diverse.

?? Scarso contenuto semantico delle dichiarazioni: tramite dichiarazioni DTD si definiscono relazioni *part-of* , relazioni di tipo *kind-of* vengono espresse solo implicitamente, si veda il Capitolo 4 per maggiori dettagli. Relazioni come "*is a*" non sono direttamente supportate dal linguaggio.

?? Non si può limitare il contenuto di un elemento testuale: non è possibile definire dei pattern di caratteri ammissibili per il contenuto di un elemento, ad esempio per far sì che un elemento contenente un numero telefonico contenga solo cifre.

?? Supporto all'identificazione degli elementi insufficiente: gli attributi ID consentono di specificare solamente chiavi surrogate, si veda il paragrafo 4.3.3 per maggiori dettagli.

?? Impossibilità di esprimere veri vincoli di integrità referenziale: gli attributi IDREF assicurano solamente l'esistenza di un attributo ID con il valore indicato, non si può specificare alcuna informazione sul tipo di elemento in cui compare l'attributo riferito.

Per superare queto tipo di problemi, sono state avanzate molte proposte di estensione dello standard: XML Schema, XML Namespaces sono solo alcuni esempi.

2.1.3 W3C DOM recommendation

Il W3C Document Object Model definisce un'interfaccia, indipendente dalla piattaforma e dal linguaggio utilizzati, per permettere a programmi e script di accedere e aggiornare dinamicamente il contenuto, la struttura e lo stile dei documenti. Il modello fornisce un insieme standard di oggetti per rappresentare documenti HTML e XML, un modello standard per combinarli tra loro e un'interfaccia per poterli accedere e manipolare.

Non è necessario che l'informazione rappresentata dal DOM sia effettivamente un documento HTML o XML, un database relazionale per esempio potrebbe rappresentare le sue informazioni come un DOM.

Il supporto per il DOM viene fornito come interfaccia standard che gli sviluppatori possono utilizzare senza doversi legare a qualche specifica API, in modo da aumentare l'interoperabilità sul web.

Un programma che supporti la DOM API, non solo consente ai suoi dati di essere manipolati da altre routine, ma lo fa in maniera tale da permettere una forte riusabilità del codice, consentendo l'impiego di differenti implementazioni del DOM e di poter sfruttare soluzioni già scritte per altri progetti che utilizzano il DOM. Questa interoperabilità consente ai programmatori di investire nell'apprendimento del modello e delle sue routine, certi di poter applicare le conoscenze acquisite in contesti molto diversificati.

Lo scopo è avere un'API standard per poter intercambiare liberamente le implementazioni del DOM con le applicazioni basate su di esso. Un esempio di

questa interoperabilità sono gli script dell'HTML dinamico: accettando il DOM come rappresentazione standard dei loro documenti, gli script possono essere scritti in maniera tale da funzionare correttamente su tutti i browser.

Anche se le varie implementazioni del DOM possono interoperare, le loro dimensioni, la richiesta di memoria e le prestazioni delle singole operazioni possono variare enormemente.

Come in qualunque altro insieme di interfacce generalizzate, utilizzare il DOM può avere anche aspetti negativi, le chiamate fornite dal DOM possono risolvere un gran numero di problemi, ma potrebbero non rappresentare la soluzione ottimale per alcuni specifici casi. Tuttavia i vantaggi connessi con l'interoperabilità e la famigliarità agli utenti compensano pienamente gli svantaggi in molte applicazioni.

2.1.4 DOM tree

Il DOM rappresenta i dati di un documento in forma di un albero, i cui nodi possono essere di molti tipi differenti: elementi, nodi testuali, attributi, commenti ecc. Un documento XML viene convertito in un albero in cui il nodo radice rappresenta l'elemento radice del documento e i nodi figli rappresentano il suo contenuto.

Ad esempio il frammento XML seguente:

```
<car>
   <engine power=”5500W”>
     <supplier>Bingo Motors</supplier>
   </engine>
   <note>Cheap model</note>
</car>
```

viene rappresentato con lo schema di Figura 5. Nel DOM tree sono riportati con dei nodi testuali vuoti anche gli spazi che separano i vari tag.

Sul DOM tree sono disponibili varie operazioni, è possibile aggiungere o togliere nodi, creare, modificare o rimuovere attributi, vi sono routine che ritornano tutti gli elementi aventi un determinato nome ecc.

Allo stato attuale delle specifiche DOM livello 1 e 2 [35][36] il modello non prevede la creazione e la modifica di DTD, costringendo gli sviluppatori a ricorrere ad API non standard. Nel traduttore presentato al Capitolo 5 si è reso necessario implementare un modulo specifico per il supporto alle DTD.

Nelle specifiche DOM livello 3, attualmente in fase di progettazione questo problema dovrebbe venire risolto fornendo inoltre un supporto a XML Schema.

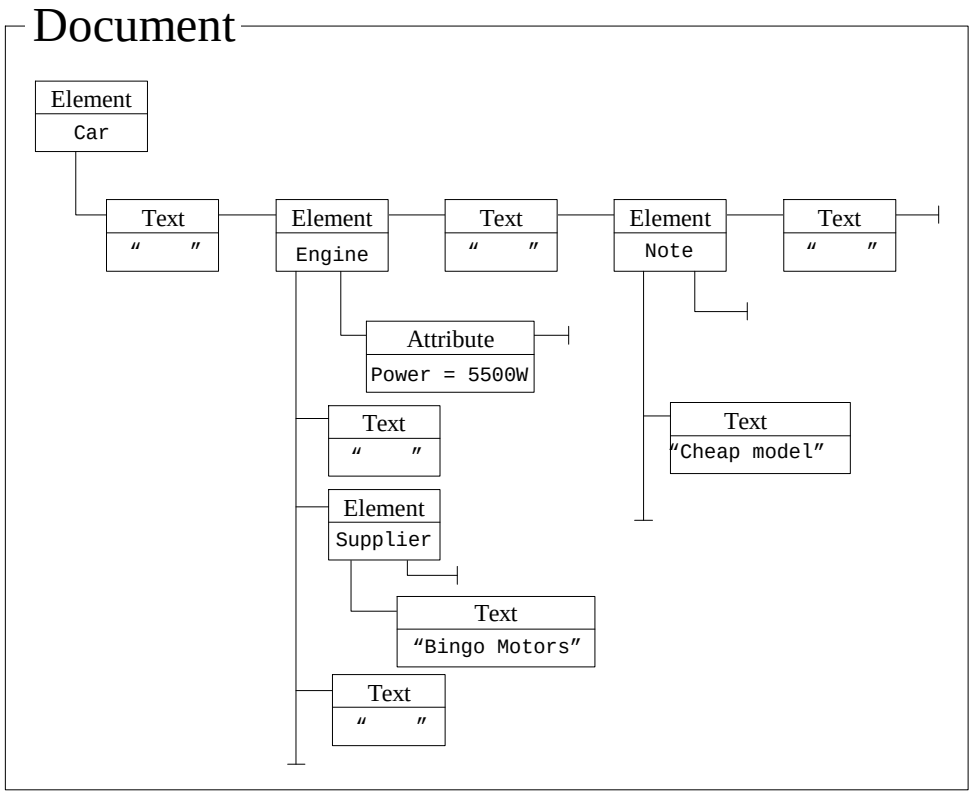

Figura 5: DOM tree di un frammento XML

2.2 XML namespaces

Il concetto di "spazio di nomi" è largamente utilizzato in svariati campi dell'informatica per risolvere i problemi legati alle omonimie. Si pensi ai nomi degli attributi delle classi C++ ai nomi dei package Java ecc.

Questi spazi di nomi sono comunemente denominati spazi di nomi "tradizionali", secondo la terminologia XML, e indicano degli *insiemi* di nomi, nel senso matematico del termine.

Nelle specifiche di XML 1.0 il concetto di namespace non esiste, o meglio è presente un unico spazio di nomi al quale devono appartenere tutti gli identificatori utilizzati. Lavorando in XML spesso ci si imbatte nel problema di dover rinominare alcuni elementi o attributi poiché i loro nomi collidono con quelli utilizzati da altri.

Si consideri un'applicazione che debba processare codice XML come quello riportano nei seguenti frammenti:

```
<?xml version="1.0" ?>
  <Address>
     <Street>Wilhelminenstr. 7</Street>
     <City>Darmstadt</City>
     <State>Hessen</State>
     <Country>Germany</Country>
     <PostalCode>D-64285</PostalCode>
  </Address>
```

e

```
  <?xml version="1.0" ?>
  <Server>
     <Name>OurWebServer</Name>
     <Address>123.45.67.8</Address>
  </Server>
```

Il termine `Address` nel contesto in cui l'applicazione lavora, è chiaramente ambiguo in quanto ha un significato profondamente differente nelle due definizioni. Il primo frammento address è un indirizzo del mondo reale, mentre il secondo è l'IP di un web server.

Una soluzione banale al problema è quella di rinominare uno dei due elementi, questo è sempre possibile, ma non è una soluzione utile sul lungo termine. Una delle speranze legate a XML è che con il tempo si sviluppino linguaggi XML standardizzati che risolvano problematiche relative ad un certo settore: la descrizione di formule matematiche o chimiche ecc. Se ogni volta che c'è un conflitto si procede a rinominare una delle parti si dovrebbe modificare anche il codice che lavora nel suo contesto e si aspetta il vecchio nome.

Per risolvere problematiche di questo tipo sono state progettate le XML namespace recommendation emanate dal W3C come estensione alla semantica di XML 1.0. Secondo queste specifiche uno spazio di nomi XML è dichiarato utilizzando uno speciale attributo `xmlns`. Il valore di questo attributo determina lo spazio di nomi cui appartiene l'elemento che lo contiene. Tutti gli elementi XML che non dichiarano esplicitamente un namespace si intendono appartenere al namespace del padre. Ad esempio

```
<Server xmlns=”some_unique_identifier/IPAddress”>
  <Name>OurWebServer</Name>
  <Address>123.45.67.8</Address>
</Server>
```

dichiara che `Server` e `Name` e `Address` appartengono allo spazio di nomi. Si possono anche dichiarare dei prefissi per riferire un dato namespace anziché un altro. Un prefisso è dichiarato concatenandolo all'attributo `xmlns`:

```
<ip:address  xmlns:ip=”unique_identifier/IPAddress”>
   192.168.1.1
</ip:address>
```

Un prefisso dichiarato si utilizza componendolo con il nome dell’elemento o dell’attributo che si vuole caratterizzare, utilizzando i “:” come separatore.

Un possibile modo per utilizzare all’interno di un elemento comune elementi aventi lo stesso nome locale è il seguente:

```
<real:Department xmlns:real=”unique_identifier/real”
                 xmlns:ip=”unique_identifier/IPAddress”
 >
  <real:Address>
    <real:Street>Wilhelminenstr. 7</real:Street>
    <real:City>Darmstadt</real:City>
    <real:State>Hessen</real:State>
    <real:Country>Germany</real:Country>
    <real:PostalCode>D-64285</real:PostalCode>
  </real:Address>
  <ip:address  xmlns:ip=”unique_identifier/IPA ddress”>
   192.168.1.1
  </ip:address>
</real:Department>
```

Si noti inoltre che gli identificatori che compaiono negli attributi `xmlns` sono spesso delle URI o delle URL, ma non per questo referenziano qualche tipo di risorsa, sono utilizzate all’unico scopo di definire un identificatore univoco.

2.2.1 Limiti degli spazi di nomi

Gli XML namespace non sono spazi di nomi nel senso tradizionale del termine: *non sono insiemi* matematici ai quali i vari identificatori appartengono. Gli XML namespace forniscono solamente un sistema di nominazione diviso in due parti: prefisso, nome locale e assomigliano più a delle collezioni di elementi ed attributi.

Namespace e DTD

Gli XML namespace non si applicano alla DTD e non modificano il modo con cui i parser XML 1.0 la processano. Il concetto di validità introdotto dalla DTD in XML 1.0 non viene minimamente modificato. Si consideri ad esempio

```
<!DOCTYPE doc [
  <!ELEMENT doc (x)>
  <!ATTLIST doc
    xmlns CDATA #FIXED "http://www.unimo.it/"
    xmlns:foo CDATA #FIXED "http://www.unimo.it/">
```

```
  <!ELEMENT x EMPTY>
]>
<doc>
  <foo:x/>
</doc>
```

Seguendo la semantica dei namespace ci si aspetterebbe che il documento sia valido in quanto l'elemento foo:x e l'elemento x rappresentano la stessa entità ma in realtà le stringhe dei nomi sono interpretate alla stessa maniera di XML 1.0 e quindi l'elemento foo:x risulta non definito nella DTD.

Gli attributi `xmlns` sono trattati come normali attributi e non rappresentano delle dichiarazioni di namespace. I nomi interamente qualificati (prefisso:nome_locale) sono visti come qualunque altro nome senza nessun particolare significato.

Ai fini di questo lavoro di tesi sarebbe stato molto utile disporre di un processo di validazione namespace-aware (si veda il Capitolo 4 e il paragrafo 4.3.2) ma questa funzionalità come si è visto non è prevista dalle [38]. Il motivo di ciò è dovuto al fatto che le DTD hanno molte altre mancanze e attualmente si sta tentando di realizzare un meccanismo completamente nuovo per la validazione degli schemi: XML Schema.

Capitolo 3

I/O di oggetti in formato XML: il package XmlTools

3.1 Descrizione di oggetti

Il package XMLTools è nato dall'esigenza di poter descrivere il contenuto di oggetti residenti in memoria in una lingua franca che facilitasse lo scambio di informazioni tra piattaforme diverse. Il software prodotto dal progetto MOMIS ha raggiunto una complessità notevole, si è così pensato di realizzare uno strumento che consenta in maniera automatica di ottenere una rappresentazione di un dato grafo di oggetti residenti in memoria.

Uno degli obiettivi del linguaggio XML è di rendere agevole lo scambio di informazioni tra applicativi. È risultato dunque naturale sceglierlo come output della descrizione.

La procedura di descrizione deve essere trasparente al progettista di una classe ma se questi ha necessità particolari, il tool deve poter fornire gli strumenti necessari per personalizzare l'output XML.

Un secondo requisito era quello di poter ricostruire all'occorrenza le classi originali a partire dalla descrizione XML, in modo che qualunque applicativo che incorpori il tool, possa agevolmente importare i dati in formato XML. Questo requisito ha contribuito a spingere alla realizzazione di uno strumento per la serializzazione di oggetti che nel contempo avesse la possibilità di personalizzare enormemente l'output XML per creare descrizioni di grafi di oggetti.

Il progetto MOMIS è sviluppato in Java, questo linguaggio è dunque un vincolo di progetto per la realizzazione del tool.

3.2 Serializzazione

3.2.1 Serializzazione di un oggetto

Tradizionalmente nel modo di concepire il software, le entità da analizzare venivano modellate in termini di dati e codice in maniera separata; con l'avvento e la diffusione dei primi linguaggi ad oggetti, gli sviluppatori hanno a disposizione un nuovo modello che permette di superare la dualità dati/codice attraverso il concetto di incapsulazione.

I dati di un oggetto sono ora "nascosti" da una barriera di metodi rappresentanti l'interfaccia con cui le altre entità interagiscono, i metodi sono i soli responsabili dell'utilizzo delle variabili interne, in questo modo i dettagli implementativi delle funzionalità di un oggetto vengono nascoste guadagnando in semplicità delle interfacce, manutenibilità e affidabilità del codice.

Quando un'istanza risiede in memoria tutte le sue proprietà sono accessibili in maniera casuale, le proprietà pubbliche possono essere consultate ed utilizzate da altre classi, quelle private sono ad uso esclusivo della classe che le ha definite: anche in questo caso i metodi della classe possono accedere casualmente a qualsiasi variabile membro privata.

Le proprietà di una classe sono implementate attraverso la definizione, nel caso più semplice, di variabili membro di tipo primitivo oppure, nei casi più generali e complessi, possono arrivare a definire strutture dati con liste, alberi, mappe di oggetti che riferiscono a loro volta altri oggetti e così via. L'insieme dei valori delle variabili membro di un oggetto rappresenta il suo **stato**.

Se da un lato la metodologia ad oggetti per lo sviluppo del software semplifica la vita dello sviluppatore, dall'altro ha portato alla luce un nuovo problema: l'I/O di oggetti. I primi linguaggi ad oggetti non supportavano l'I/O diretto di oggetti e costringevano i programmatori a costose operazioni di codifica e decodifica dei dati incapsulati per poterli scrivere o leggere su qualche dispositivo.

Oggi molti applicativi complessi necessitano di scambiare informazioni direttamente nel formato con cui sono state modellate, senza costringere lo sviluppatore ad implementare operazioni aggiuntive per l'I/O. Per raggiungere questo obiettivo è stato necessario sviluppare degli strumenti che consentono di automatizzare le operazioni di codifica o decodifica.

Il problema può essere inquadrato facendo riferimento al seguente schema: una sorgente invia uno o più oggetti ad una destinazione attraverso un canale per sua natura sequenziale: un file su disco, un collegamento punto-punto di una WAN ecc. Perché la comunicazione possa avvenire lo stato interno di ogni oggetto deve essere "serializzato" ossia i valori delle sue variabili membro

devono essere scritti in sequenza sul canale seguendo un protocollo prefissato, si devono scrivere inoltre sufficienti informazioni in modo che la destinazione possa ricostruire l'oggetto originale.

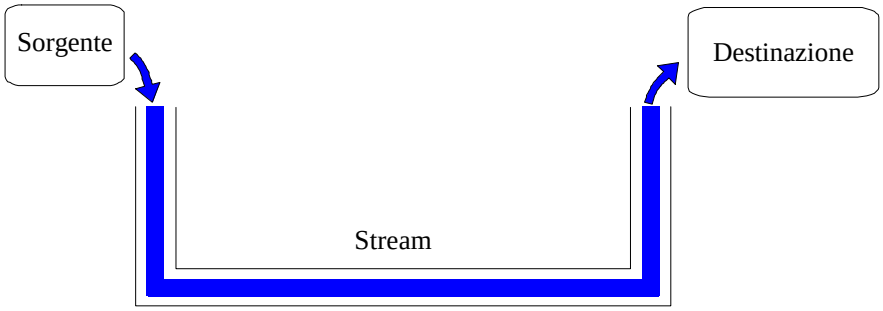

Figura 6 Schema di riferimento per la serializzazione

La destinazione degli oggetti deve a sua volta leggere la sequenza di dati e conoscendo il protocollo utilizzato per la scrittura deve essere in grado di allocare un nuovo oggetto e ricostruire ("deseserializzare[3]") lo stato dell'oggetto originale. In questo modo si ottiene una nuova istanza del tutto equivalente a quella trasmessa o scritta.

Il punto chiave del problema è che tutte queste operazioni devono avvenire in maniera trasparente per chi progetta la classe. Lo sviluppatore deve essere lasciato libero di mettere poche, o preferibilmente, nessuna informazione riguardanti la serializzazione all'interno della classe che sta progettando, in modo che non debba preoccuparsi dei dettagli di come lo stato venga scritto o letto dal canale.

L'interfaccia verso l'utilizzatore di un tool di serializzazione deve quindi mettere a disposizione semplici comandi come "scrivi questo oggetto" oppure "leggi un oggetto". Un buon tool per l'I/O di oggetti dovrebbe inoltre introdurre una certa insensibilità a variazioni della struttura interna di un oggetto: la versione della classe presso la sorgente che scrive un oggetto potrebbe non essere la stessa di quella in possesso della destinazione.

Si pensi come esempio ad una nuova versione di un applicativo che deve poter continuare a importare dati dalla vecchia. Gli sviluppatori devono essere messi nella condizione di poter implementare questa funzionalità riducendo al minimo la scrittura di metodi di conversione più o meno complicati.

[3] Neologismo derivante dal termine inglese "deserialize", utilizzato nelle specifiche [32].

Quando si serializza un oggetto bisogna fare attenzione al fatto che molto spesso è una aggregazione di più oggetti in quanto i suoi campi[4] possono contenere riferimenti a intricati grafi, anche ciclici, di oggetti (innestamento ricorsivo).

Ogni oggetto raggiungibile a partire dall'oggetto che si sta serializzando deve essere serializzato a sua volta facendo attenzione a non serializzare più volte lo stesso oggetto. Se non si fa attenzione si rischia di sprecare banda trasmissiva sul lato sorgente mentre la destinazione si troverebbe ad allocare più memoria per fare posto ad oggetti ridondanti e il grafo ricostruito non sarebbe più equivalente a quello originario.

Il fatto che un tool per la serializzazione debba operare in maniera trasparente comporta automaticamente un problema notevole: gli oggetti letti o scritti sono delle "black box": si deve trattare con oggetti di cui a tempo di compilazione non si conosce né il tipo della classe né la struttura interna.

Per poter implementare il tool sarà dunque necessario un linguaggio di programmazione e un ambiente di sviluppo che metta a disposizione degli strumenti di basso livello per accedere alle *run-time type information*. Queste informazioni sono dei metadati che descrivono il tipo e la struttura interna di un oggetto a tempo di esecuzione.

I più diffusi linguaggi di programmazione Java, Delphi e molte implementazioni di C++, mettono a disposizione potenti gerarchie di metaclassi; in Java è possibile, ad esempio, anche invocare metodi i cui nomi e parametri sono ignoti a *compile time.*

La serializzazione di oggetti viene spesso utilizzata in tre campi applicativi:

?? ***Remote method invocation (RMI)*** - per realizzare una comunicazione tra oggetti in ambienti distribuiti
?? ***Lightweight persistence*** - Grafi di oggetti vengono archiviati dagli applicativi per un utilizzo successivo.
?? ***Socket communication*** – Scambio di oggetti tramite una rete locale o geografica.

Si pensi per esempio ad una applicazione client-server per un grande distributore con vari punti vendita sparsi sul territorio che inviano in tempo reale dati al server sull'andamento delle vendite: il lato server dell'ipotetica applicazione potrebbe ricevere gli "oggetti" inviati dai client rappresentanti i prodotti venduti.

[4] Così sono chiamate le variabili membro di una classe che deve essere serializzata.

Gli oggetti verrebbero serializzati dai client e deserializzati dal server, un ulteriore vantaggio fornito da un approccio "trasparente" alla serializzazione consente di aggiungere, senza modificare una sola riga di codice del modulo di comunicazione, nuovi tipi di prodotto (nuove classi) e grazie all'ereditarietà è molto probabile che il lato server non debba subire grosse modifiche.

Moltissime applicazioni oggi usano la serializzazione per "congelare" il loro stato permettendo all'utente, dopo un riavvio, di riprendere il lavoro esattamente dove lo si era interrotto.

3.2.2 Serializzazione in formato binario

La serializzazione di oggetti avviene quasi sempre in formato binario: l'oggetto viene convertito in una uniforme sequenza di byte, un flusso (stream), e scritto su file o trasmesso via socket.

In uno stream binario i dati che descrivono il contenuto di un oggetto e i metadati sono codificati secondo uno speciale protocollo dipendente dall'implementazione del meccanismo di serializzazione.

Questo comporta almeno tre conseguenze:

?? **Altissima efficienza**: il numero di byte utilizzati per descrivere gli oggetti è molto vicino al minimo indispensabile, lo stream risulta molto compatto con minor ingombro sia di banda trasmissiva che di spazio occupato nel dispositivo di storage utilizzato.
?? **Incompatibilità tra implementazioni diverse:** differenti produttori devono convenire un formato comune se vogliono che le loro implementazioni siano compatibili.
?? **Difficoltà di accesso all'informazione:** Lo stream binario risulta difficilmente comprensibile a qualunque strumento che non abbia specifiche conoscenze del protocollo utilizzato per la codifica dei campi.

L'alto livello di efficienza dello stream binario ottenuto dalla serializzazione è sicuramente il fattore determinante nella scelta di questo formato. La serializzazione binaria inoltre garantisce prestazioni migliori, come tempo di CPU impiegato per il processo.

Scegliere di serializzare in binario significa isolare le informazioni contenute negli oggetti *ostacolandone l'utilizzo al di fuori degli scopi per cui il software è stato progettato*.

Sotto il profilo della sicurezza questo può anche essere un fatto positivo ma appare ovvio che esistono molti ambiti applicativi in cui non lo è:

?? Condivisione di dati e documenti tra applicativi: occorrono specifici filtri di import – export per usufruire delle informazioni.

?? Software realizzato in ambienti di ricerca: bisogna evitare la serializzazione binaria se si vogliono scambiare dati tra software di diversi gruppi di ricerca. Il progetto MOMIS ne è appunto l'esempio lampante.

Dal punto di vista dello sviluppatore, avendo a disposizione comandi ad alto livello come quelli citati prima, la serializzazione è uno strumento potente e molto comodo: con essa è infatti possibile memorizzare o comunicare le informazioni elaborate dal software in modo diretto senza la necessità di progettare esplicitamente dei metodi di conversione.

Finché la sorgente e la destinazione delle informazioni sono entità interne allo stesso progetto ha senso utilizzare la serializzazione, non appena però la destinazione o la sorgente diventano entità esterne, ecco che si rende necessario implementare dei metodi che trasformino il formato interno delle informazioni in un qualche formato standard o viceversa.

Dal discorso appena fatto si evince che, disporre di un tool che serializza/deserializza in un formato universalmente riconosciuto, quale appunto è XML 1.0, significa risolvere alcuni dei problemi sopra citati.

Il linguaggio Java nell'implementazione fornita da Sun Microsystems mette a disposizione degli sviluppatori una potente gerarchia di classi per l'I/O, oltre a consentire la scrittura di tipi primitivi in un formato indipendente dalla piattaforma alcune classi sono state progettate con lo scopo di permettere l'I/O di tipi complessi come le istanze di classi.

Le linee guida che animano le "Java object serialization specification" sono:

?? Definire un meccanismo semplice ed espandibile
?? Supportare il marshaling e unmarshaling[5], necessario per gli oggetti remoti
?? Richiedere l'implementazione di metodi solo per la personalizzazione
?? Permettere ad ogni oggetto di poter definire il proprio formato esterno.

Le specifiche definiscono due speciali stream binari: `ObjectOutputStream` per la serializzazione e `ObjectInputStream` per la deserializzazione.

Una qualunque classe per poter essere serializzata o deserializzata deve implementare l'interfaccia `Serializable` del package `java.io`:

```
public interface Serializable {
```

[5] Differenti piattaforme utilizzano formati numerici differenti: fare il marshaling, significa porre un dato binario in un formato convenuto (ad es. little endian). L'unmarshaling è l'operazione inversa: si passa da un formato convenuto a quello realmente utilizzato dalla macchina. In questo modo i file serializzati su una macchina sono portabili al pari del codice java.

```
}
```

Questa interfaccia in realtà non richiede l'implementazione di alcun metodo aggiuntivo, serve solo per discriminare le classi che sono state pensate per l'I/O dalle altre. Poiché l'interfaccia da implementare è vuota risulta immediato convertire una classe esistente in un'entità serializzabile.

3.2.3 Serializzazione in formato XML

Il formato XML fornisce un valido punto di partenza per la progettazione di un tool di serializzazione: in XML si può definire una struttura con elementi annidati, rappresentabile da un albero i cui rami possono raggiungere una profondità qualunque.

Ad esempio, prendiamo in considerazione un'istanza della classe `rectangle` che rappresenta l'omonima figura geometrica, una possibile dichiarazione Java potrebbe essere la seguente:

```
public class Rectangle {
  public Rectangle (Point p1, Point p2) {
    top_left=p1; bottom_right=p2;
  }

  private Point top_left;
  private Point bottom_right;
}

public class Point {
  public Point (int x, int y) {
    coord_x=x;
    coord_y=y;
  }

  private int coord_x;
  private int coord_y;
}
```

Si può progettare un tool che trasformi lo stato di una sua istanza nel seguente frammento XML:

```
<Rectangle objectID=”000”>
  <Point field=”top_left” objectID=”001” >
    <int field=”coord_x” value=”32”/>
    <int field=”coord_y” value=”10”/>
  </Point>
  <Point field=”bottom_right” objectID=”002” >
    <int field=”coord_x” value=”50”/>
    <int field=”coord_y” value=”64”/>
```

```
  </Point>
</Rectangle>
```

Se una classe definisce dei campi complessi, questi si possono rappresentare in XML annidando un elemento che rappresenti il tipo complesso in esame, la classe `Point` vista prima ne è un esempio. L'attributo objectID serve per mantenere l'identità delle istanze serializzate e verrà descritto in dettaglio nel seguito.

La struttura prodotta contiene tutte le informazioni per poter ricostruire l'istanza originale: i tag `Rectangle` e `Point` indicano i tipi di classi coinvolte, i tag `int` rappresentano i valori interi assunti dai vari campi.

Quando si serializza lo stato di un oggetto, anche il contenuto dei suoi campi privati viene esposto agli utenti del documento XML generato, questo genera potenzialmente un problema sulla sicurezza delle informazioni: il tool di serializzazione dovrà dunque fornire strumenti per evitare l'esposizione di informazioni sensibili quali ad esempio il numero di una carta di credito o un file descriptor ritornato dal sistema operativo.

Si può notare che XML fornisce una rappresentazione degli oggetti serializzati molto intuitiva, questo risultato è ottenuto mescolando le informazioni da rappresentare (le coordinate dei punti nell'esempio precedente) con le metainformazioni necessarie per rappresentare la struttura (i tag).

Con un file serializzato di questo tipo è persino possibile editarne il contenuto per modificare il comportamento di un software durante una sessione di debug.

Adottando XML come formato di output si facilita lo scambio di informazioni tra ambienti eterogenei: un file di oggetti Java serializzati in XML, può essere analizzato con facilità da un filtro scritto in C++ per scopi differenti da quelli previsti dal progettista Java. I due applicativi scritti in Java e C++ possono essere progettati senza conoscere nulla l'uno dell'altro, ovviamente la deserializzazione sarà possibile solo a tools che condividono lo stesso protocollo di serializzazione ma le informazioni risultano comunque condivisibili.

La flessibilità nell'utilizzo dei tag ha comunque un costo in termini di efficienza: la **ridondanza**.

Uno dei difetti di XML 1.0 è l'alto livello di ridondanza insito nella sua sintassi, la dimensione di un documento XML infatti, risulta uno o due ordini di grandezza maggiore di una rappresentazione alternativa in formato binario, anche al di fuori del contesto di serializzazione.

Serializzare in formato XML dunque, comporta anche lo studio di particolari accorgimenti che limitino in maniera apprezzabile la ridondanza.

3.3 Il package XMLTools: aspetti generali

Il tool è stato realizzato in Java per operare su classi Java. Il linguaggio è un vincolo del progetto in quanto gran parte del codice che compone il sistema MOMIS è scritto in Java.

3.3.1 Rappresentazione XML di grafi di oggetti.

Il primo problema da affrontare è la progettazione della struttura che assumerà il documento XML contenente gli oggetti serializzati. Consideriamo un insieme di classi residenti in memoria, tra loro in generale sussistono relazioni di associazione, aggregazione (*part-of*) e generalizzazione/specializzazione (*isa*).

Le relative istanze formano un grafo in cui le associazioni e aggregazioni si manifestano sotto la forma di campi contenenti dei riferimenti ad altre istanze. Lo stato di una istanza specializzata sarà formato dall'unione dello stato relativo alle sue superclassi e dallo stato relativo ai campi specifici della sua classe di appartenenza.

Consideriamo come esempio il seguente diagramma OMT di un parcheggio a pagamento:

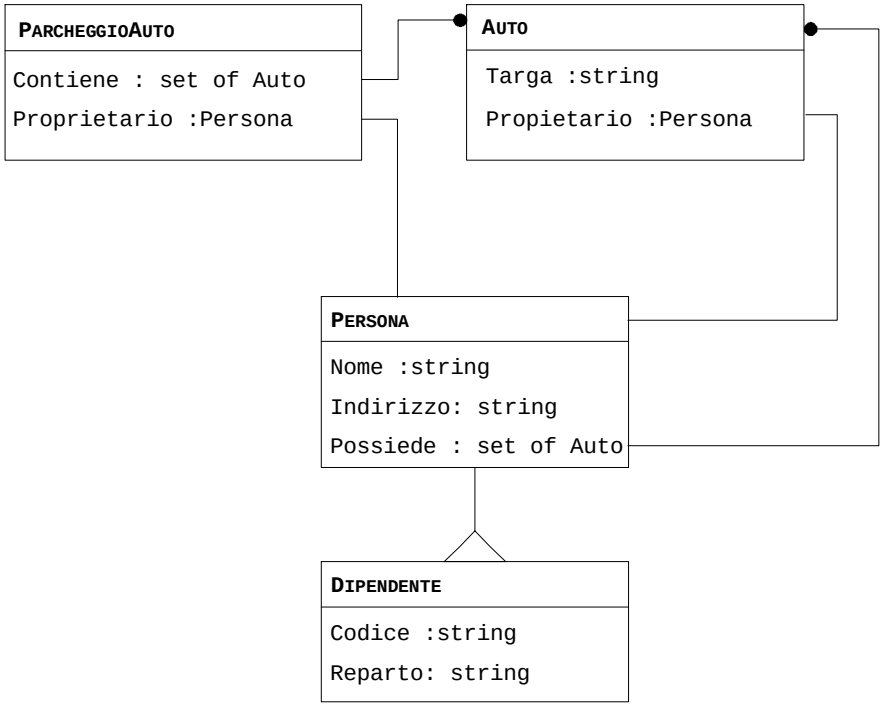

Figura 7 Esempio di uno schema da serializzare

In un suo grafo di istanze, molte auto potrebbero avere lo stesso proprietario e il parcheggio potrebbe contenere anche le auto del suo proprietario. Qualunque sia l'algoritmo utilizzato per serializzare, occorre evitare di scrivere più di una volta la stessa istanza in modo da tagliare, in qualche punto del grafo, i riferimenti ciclici (si pensi all'auto che è posseduta da una persona che a sua volta può possedere più auto.)

Per spezzare un riferimento ciclico o per non serializzare due volte la stessa istanza occorre definire un riferimento nel documento XML, una sua possibile implementazione potrebbe essere:

```
<NomeTag ObjectREF=”[object_ID]” >
```

l'attributo ObjectREF ha come valore, l'Object_ID dell'istanza riferita.

Un altro aspetto da tenere in considerazione per la progettazione è come gestire l'ereditarietà tra classi. XML 1.0 non fornisce un supporto diretto alla descrizione dell'ereditarietà ma permette l'annidamento di elementi complessi: Una classe specializzata può quindi contenere ricorsivamente al suo interno la serializzazione dell'istanza relativa alla sua immediata superclasse.

Un possibile formato di serializzazione per un'istanza della classe `Dipendente` potrebbe essere ad esempio:

```
<Dipendente ObjectID=”2001”>
  <string field=”codice” ObjectID=”2002” >
      <![CDATA[AB-123-CD]]>
  </string>
  <string field=”reparto” ObjectID=”2003” >
      <![CDATA[Piano 2B]]>
  </string>
  <superclass class=”Persona”>
      <string field=”nome” ObjectID=”2004”>
            <![CDATA[Mario Rossi]]>
      </string>
      <string field=”indirizzo” ObjectID=”2005”>
            <![CDATA[via qualunque, 22]]>
      </string>
      <set field=”Possiede” ObjectID=”2006”>
        <Auto ObjectREF=”3231” />
        <Auto ObjectREF=”1234” />
      </set>
  </superclass>
</Dipendente>
```

Le stringhe andranno memorizzate come sezioni CDATA dato che essendo stringhe generiche non è detto che contengano solo testo. Il tipo `#PCDATA` utilizzato normalmente come contenuto degli elementi XML infatti, richiede che alcuni caratteri come "<" "& " siano sostituiti da appositi riferimenti a entity per non confonderli con i simboli di inizio tag e inizio entity reference; questo fatto lo rende particolarmente scomodo nel descrivere stringhe generiche. Inoltre in una stringa gli spazi sono significativi, mentre non lo sono in `#PCDATA`.

Maggiori dettagli sul protocollo di rappresentazione dei tipi Java verranno forniti nell'Appendice B.

3.3.2 Architettura

Nella progettazione di `XmlTools` sono state tenute in forte considerazione le specifiche [32] in modo da realizzare un tool che fornisse all'utente un'interfaccia molto simile a quella già nota per la serializzazione binaria. Sono state seguite le medesime linee guida giungendo alla realizzazione di un tool che opera in maniera completamente trasparente e richiede la scrittura di metodi su misura solamente per personalizzazioni che esulano da quelle già previste dal tool.

Cosa è possibile serializzare.

Le specifiche sulla serializzazione di Java richiedono ad ogni classe che debba essere serializzata di implementare l'interfaccia `Serializable`. Anche `XmlTools` segue questo principio e per default se si tenta di serializzare una classe non serializzabile viene lanciata una eccezione.

Non c'è alcun motivo tecnico per cui una classe che non implementa `Serializable` non possa essere serializzata, l'interfaccia serve solamente a identificare la semantica di "essere serializzabile". `XmlTools` consente di disabilitare, se richiesto, questo controllo quindi è certamente possibile descrivere in XML *qualunque* grafo di oggetti Java.

Tuttavia in questo caso non sarà sempre possibile deserializzare i documenti prodotti: si pensi a classi con campi che memorizzano informazioni volatili quali handle di sistema o file descriptor.

Classi

Trattandosi di un tool per la serializzazione `XmlTools` è composto da due classi principali:

?? `XmlWriter` per la serializzazione e descrizione di classi
?? `XmlReader` per la deserializzazione

Queste classi non estendono gli speciali stream forniti da Java (`ObjectOutputStream` e `ObjectInputStream`), per fornire ugualmente un supporto al meccanismo standard di serializzazione sono stati quindi implementati due specifici wrapper che estendono le funzionalità di `ObjectOutputStream` e `ObjectInputStream` ai documenti XML, consentendo ad XmlTools una perfetta integrazione con il meccanismo standard descritto nelle [32], si veda il paragrafo 3.6 per maggiori dettagli.

`XmlWriter` e `XmlReader` mettono a disposizione dell'utente metodi per la gestione di tipi primitivi, oggetti e array, l'output di `XmlWriter` è un albero DOM standard, mentre per `XmlReader` l'input può essere sia uno stream che un albero DOM.

Le DOM specification (Document Object Model) sono delle direttive emanate dal W3C per la definizione di un'API standard che riesca a processare, creare e modificare documenti XML o HTML in maniera indipendente dall'implementazione utilizzata.

La scelta di non utilizzare direttamente degli stream XML è stata dettata dal voler conferire una maggiore flessibilità all'utilizzo dell'output di `XmlWriter`: un DOM tree permette infatti di poter modificare e personalizzare facilmente il documento XML prodotto, consentendo l'integrazione della serializzazione nei documenti XML più disparati.

La trasformazione di un DOM tree in uno stream XML è un'operazione meccanica e banale spesso riconducibile ad una chiamata ad un unico metodo.

Un diagramma semplificato delle classi di `XmlTools` secondo il modello OMT è il seguente:

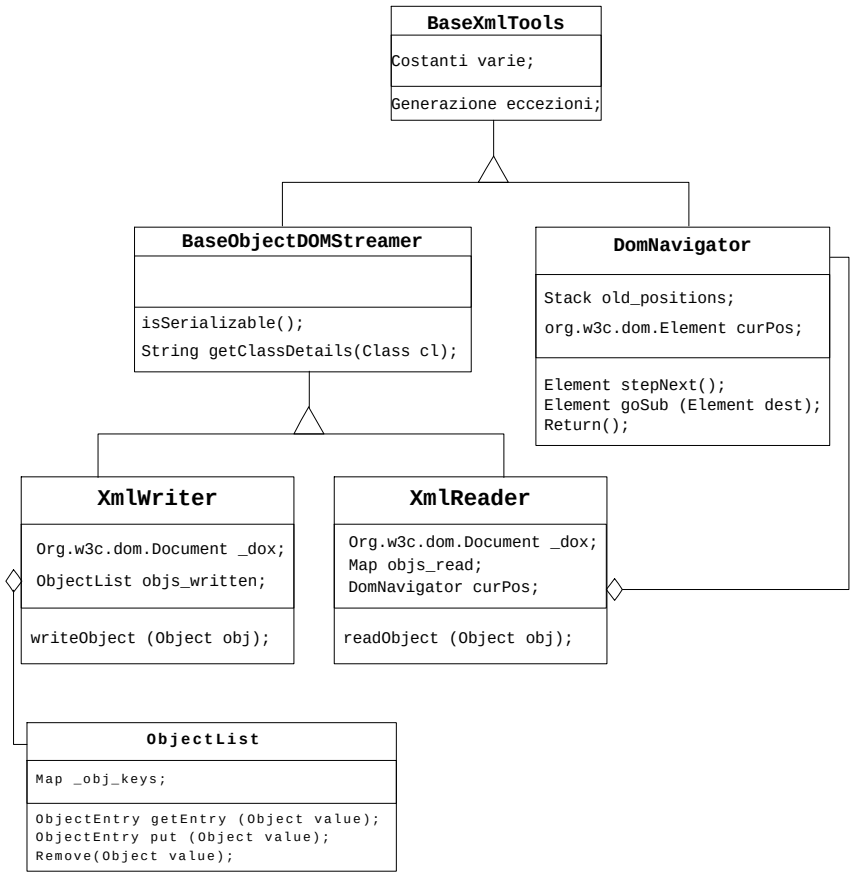

Figura 8 Schema semplificato delle classi di XmlTools

Da questo schema restano escluse le eccezioni, le interfacce e i wrapper per l'integrazione con il meccanismo standard di Java per la serializzazione nonché alcune classi minori. I metodi riportati sono solamente quelli più utili e significativi per questo punto della trattazione.

BaseXmlTools

Questa classe è la radice da cui ereditano le altre classi, in essa sono definite varie costanti pubbliche, tra cui gli identificatori utilizzati come nomi per attributi o per alcuni tag con significato speciale.

Ai discendenti vengono inoltre forniti dei metodi per la gestione centralizzata dei messaggi di debug e per una generazione uniforme delle eccezioni. Le eccezioni generate infatti oltre al consueto messaggio di errore riportano anche l'elemento XML che lo ha provocato o trasportano altre eccezioni di più basso livello come un errore di lettura o scrittura su un dispositivo.

BaseObjectDomStreamer

Scopo di questa classe è fornire dei metodi per l'estrazione a run-time di informazioni utili sia per la serializzazione sia per la deserializzazione.

XmlWriter

`XmlWriter` è un vero e proprio componente software:

?? Fornisce una semplice interfaccia pubblica mettendo a disposizione dell'utente metodi con un alto grado di astrazione come `writeObject` per serializzare o descrivere oggetti in XML. Esiste inoltre un metodo per scrivere ogni tipo primitivo.
?? Consente un notevole grado di personalizzazione dell'output impostando semplicemente il valore di alcune proprietà.
?? Mette a disposizione degli sviluppatori una folta schiera di metodi protetti per poterne facilmente modificare il comportamento, anche solo in piccole parti. Maggiori informazioni al riguardo possono essere trovate nella documentazione relativa alle API di XmlTools.

Tutti i metodi di scrittura hanno come risultato la creazione di elementi XML che vengono inseriti nella posizione corrente dell'albero che rappresenta il documento XML che si sta creando. A questo scopo `XmlWriter` mantiene al proprio interno un riferimento all'elemento corrente del DOM tree che si sta producendo.

Si può creare un'istanza di `XmlWriter` in modo che produca un documento XML ex novo oppure gli si può passare un DOM tree esistente e specificare un determinato elemento di partenza che conterrà l'intero output della serializzazione/descrizione.

XmlWriter ha necessità di tenere traccia di quali oggetti sono già stati serializzati e quali no, objs_written è l'istanza della classe che gestisce tutte le informazioni relative agli oggetti scritti: objectID utilizzati, riferimenti agli elementi XML già prodotti ecc..

XmlReader

Questa è la classe duale a `XmlWriter`, ogni metodo di scrittura in `XmlWriter` ha un metodo corrispondente per la lettura in `XmlReader`. `XmlReader` viene istanziato passandogli o uno stream contenente dati XML o un DOM tree esistente con relativo elemento di partenza nel caso questi non coincida con l'elemento radice. Se viene fornito uno stream XML, `XmlReader` provvede ad effettuare il parsing e a costruire un DOM tree rappresentativo per il documento.

Gli oggetti vengono ricostruiti a partire dalla posizione corrente nel DOM tree, la classe utilizza uno speciale cursore, un'istanza di DomNavigator, l'uso di

questa classe si è reso necessario per semplificare il codice di esplorazione dei dati XML e per supportare la deserializzazione di DOM tree bilanciati.

Anche `XmlReader` fornisce sia una semplice interfaccia pubblica sia una serie di metodi protetti per modificarne il comportamento standard.

DomNavigator

Questa classe funziona come un cursore che indica la posizione corrente all'interno del documento XML che si sta deserializzando. All'interno di un albero DOM esistono vari tipi di nodi: testo, elementi, commenti ecc.

Per filtrare i nodi non interessanti dell'albero e semplificare il codice di navigazione di `XmlReader` è stata progettata questa classe: i suoi metodi ritornano solo i tag e consentono in qualunque momento di interrompere la scansione sequenziale del documento XML per saltare in altri punti memorizzando la posizione corrente, un po' come avviene durante la chiamata a una sub-routine.

3.3.3 The reflection API

Per poter effettivamente implementare un tool di serializzazione usando un linguaggio compilato tradizionale si sarebbe dovuto ricorrere a una programmazione di basso livello: le istanze andrebbero viste come delle qualunque sequenze di byte residenti in memoria, questo fatto renderebbe molto problematica la distinzione dei riferimenti ad altre istanze dai campi ordinari senza rendere obbligatoria la scrittura di qualche metodo speciale in ogni classe da serializzare.

Inoltre il supporto per le RTTI (Run-Time Type Information) non è sempre uno standard del linguaggio: ad esempio in C++ le RTTI sono una estensione del linguaggio e potrebbero non essere supportate dalla totalità dei compilatori con ovvi problemi di portabilità.

Il caso di Java invece è differente: Java definisce sin dalla prima versione una potente API per determinare dinamicamente le classi di appartenenza di oggetti e non solo. Utilizzando la reflection API si ha accesso a un'intera gerachia di meta-classi che riflettono classi, campi e metodi presenti in un programma.

Tramite la reflection API è possibile:

?? Determinare dinamicamente la classe di un oggetto
?? Ottenere informazioni su classi, campi, metodi, costruttori e superclassi, determinandone il nome, il tipo di accessibilità (pubblica, protetta, privata ecc.)
?? Determinare quali costanti e dichiarazioni di metodi appartengono a una interfaccia.

?? Creare istanze di una classe il cui nome rimane sconosciuto fino a tempo di esecuzione.
?? Leggere e scrivere il valore di campi (variabili membro) anche se il loro nome è noto solo a tempo di esecuzione.
?? Invocare un metodo su una particolare istanza anche se il suo nome e parametri sono ignoti a compile-time.
?? Creare e modificare array, anche a più dimensioni, di il cui tipo dei componenti è noto solo a run-time

Si potrebbe ironicamente obiettare che anche il linguaggio macchina consente una simile elasticità a scapito dei principi di isolabilità e information hiding enunciati da Pressman e Mayers.

Utilizzare la reflection API va sicuramente contro questi principi ma è importante notare che durante la serializzazione i dati sono solamente letti e durante la deserializzazione i valori scritti nei vari campi sono i *medesimi* che questi avevano al momento della serializzazione, quindi se un'istanza era valida durante serializzazione, risulterà valida anche l'istanza ricostruita[6].

I vantaggi offerti dall'I/O di oggetti in maniera trasparente al progettista, sono notevoli e giustificano il ricorso a questo strumento.

Utilizzare un API standard fornita come parte integrante del linguaggio di programmazione riduce comunque marcatamente il rischio di errori di programmazione e consente di mantenere la portabilità del codice prodotto; inoltre l'accesso a campi e metodi non pubblici di una classe è regolato da una classe per la gestione della sicurezza progettata per bloccare operazioni non consentite.

3.4 Il processo di serializzazione

Sino a questo punto si è cercato di dare un'idea del tipo di problematiche da che si sono affrontate nella progettazione di XmlTools. Ora viene presentato il meccanismo di serializzazione vero e proprio utilizzato da `XmlWriter`.

3.4.1 Funzionamento

In fase di inizializzazione viene costruita la struttura dati che conterrà tutte le informazioni relative ad ogni oggetto che verrà serializzato. La struttura in

[6] Questo non è sempre vero: XmlTools consente di definire dei callback per effettuare la validazione di un'istanza deserializzata.

questione è una speciale mappa[7] le cui chiavi sono gli oggetti già serializzati mentre i valori sono delle classi `ObjectEntry`.

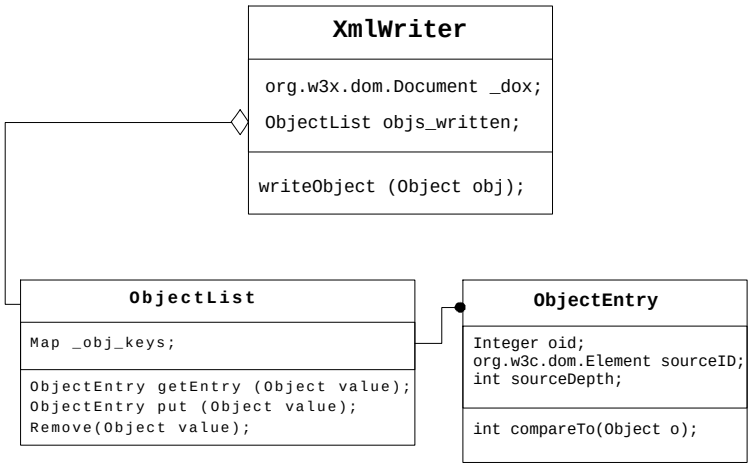

Figura 9 La struttura dati di XmlWriter

`ObjectList` differisce dalle altre mappe in quanto gli oggetti utilizzati come chiavi vengono comparati utilizzando l'operatore `==` anziché il metodo `equals`() in questo modo le istanze si diversificano in base al loro indirizzo di memoria e non su altri criteri.

Le classi `ObjectEntry` contengono l'objectID assegnato ad un determinato oggetto, la sua serializzazione in XML (tramite un riferimento ad un nodo del DOM tree) e altre informazioni utili se si desidera bilanciare l'albero.

Quando viene chiamato `writeObject()` per un oggetto, viene prima consultata la mappa degli oggetti serializzati, se l'oggetto è presente, significa che è già stato serializzato in precedenza, quindi anziché ripetere la sua traduzione in XML, viene inserito un riferimento nella posizione corrente del documento XML. Il riferimento indica l'elemento che contiene effettivamente i dati.

L'algoritmo di serializzazione opera nel seguente modo: sia `obj` il riferimento all'oggetto da serializzare:

1. Se `obj` è `null` viene scritto uno speciale elemento `<null/>` nella posizione corrente del documento XML

[7] Una mappa è una struttura dati in cui gli oggetti contenuti sono identificati da una chiave. Normalmente ad una chiave cosrrisponde un unico oggetto.

2. Se obj è già stato serializzato allora si reperisce l'objectID assegnato in precedenza e si inserisce nel documento un riferimento all'elemento contenente i dati.
3. Se un diescendente di `XmlWriter` sa come serializzare una particolare classe di cui `obj` è un'instanza, viene invocato un metodo specializzato.
4. Se la *classe* di `obj` richiede un trattamento speciale, viene eseguito il metodo più opportuno per ottenere la serializzazione corretta dell'istanza. Ricadono in questa categoria gli array, i wrapper per i tipi primitivi di Java, i meta oggetti `Class` e le stringhe.
5. Se nessuna delle precedenti condizioni è vera allora `obj` è un'istanza di una classe Java "ordinaria": se la classe di `obj` implementa l'interfaccia `Serializable` viene processata, altrimenti viene lanciata una eccezione.
6. Viene generato un nuovo `objectID` e si crea un elemento XML vuoto che dovrà contenere la serializzazione di `obj`.
7. L'oggetto `obj` viene inserito nella speciale mappa degli oggetti serializzati.
8. Se obj implementa una delle interfacce per la personalizzazione dell'output allora il controllo della serializzazione passa a questi metodi poi si esce.
9. I campi specifici di `obj` vengono serializzati tramite apposite chiamate, anche ricorsive, ai metodi di `XmlWriter` o facoltativamente, al metodo `writeObject()` implementato nella classe di `obj`, si veda il paragrafo 3.4.2 in proposito. I campi destinati ad essere serializzati sono tutti quelli non contrassegnati come statici, finali o "`transient`".
10. I campi di ogni superclasse serializzabile di obj vengono a loro volta scritti nel documento XML.
11. Giunti alla prima superclasse non serializzabile il processo si considera concluso.

Alcune classi richiedono un trattamento particolare:

Array

Gli array in Java sono implementati come oggetti, ma non sono oggetti comuni (vengono supportati a livello di linguaggio, non sono definiti in nessun file .java) e inoltre non implementano l'interfaccia `Serializable`. Per poter serializzare correttamente gli array occorre esaminarne il tipo dei componenti e la dimensione utilizzando la classe `Array` nella reflection API.

Queste informazioni vengono riportate nell'elemento XML che definisce l'array quindi per ogni componente viene chiamato ricorsivamente il metodo `writeObject()`[8] di `XmlWriter` che si incarica di completare la serializzazione.

[8] In realtà viene chiamato ricorsivamente il metodo protetto che implementa le funzionalità di `writeObject()`.

Class

Per le istanze di `Class` viene riportato solo il nome ritornato dal metodo `getName()`.

Per le stringhe e i wrapper dei tipi primitivi viene fornita una rappresentazione XML molto compatta.

Eventuali discendenti di `XmlWriter` possono definire altri tipi di classe per cui si desideri dare un supporto diretto, questa è comunque una procedura insolita e andrebbe evitata in favore di altri metodi come quelli descritti nel seguente paragrafo.

3.4.2 Personalizzazione dell'output

Oltre a supportare il meccanismo di personalizzazione descritto nelle specifiche [32], XmlTools introduce un modello molto simile per la personalizzazione dell'output articolato in quattro livelli, il primo dei quali non richiede la scrittura di codice.

Uno sviluppatore può scegliere a quale livello di dettaglio vuole scendere nella rappresentazione di una classe:

1. Utilizzare le proprietà di personalizzazione `XmlWriter`.
2. Personalizzare solamente alcuni campi specifici della classe da serializzare tramite l'implementazione di un opportuno metodo nella classe in questione.
3. Personalizzare il contenuto di un'intera classe (superclassi comprese) implementando l'interfaccia `XmlExternalizable`
4. Personalizzare tutti gli aspetti di una classe implementando l'interfaccia `FullXmlExternalizable`.

La scelta di implementare un meccanismo duale a quello standard è motivata dal fatto di voler supportare operazioni specifiche per la rappresentazione in XML.

Mentre il codice scritto nei vari metodi descritti nelle specifiche è comune a tutti gli stream utilizzati, il codice scritto nei metodi che vengono descritti ora può contenere anche istruzioni specifiche per la rappresentazione XML come l'inserimento di commenti o di elementi XML che non hanno alcun legame con i campi della classe.

Le proprietà di XmlWriter per la personalizzazione dell'output

XmlWriter mette a disposizione tre proprietà booleane per modificare l'aspetto dell'output XML, i loro nomi sono un po' lunghi ma ben rappresentano la loro funzione:

?? `ClassNameAsTagName`

?? `FullJavaClassName`
?? `FullClassNameAsAttribute`.

`ClassNameAsTagName`: Se vera, i tag che descrivono le istanze usano come nomi gli identificatori di classe. Se falsa, l'inizio di un'istanza è marcata dal tag `<object >`. Usando questo secondo modo è possibile scrivere una DTD per il documento XML in maniera molto agevole.

`FullJavaClassName`: Se vera, gli identificatori di classe, utilizzati come nomi di tag, utilizzano il nome corto: la classe `java.lang.String` è indicata semplicemente come `String`.

`FullClassNameAsAttribute`: Se vera, negli elementi che definiscono un oggetto è presente un attributo `class` che ha come valore l'identificatore di classe dell'oggetto in forma estesa.

Non tutte le combinazioni producono documenti deserializzabili: perché `XmlReader` possa ricostruire un'istanza con successo deve poter accedere al suo identificatore di classe.

Esempi di output prodotti di cui è possibile effettuare la deserializzazione sono i seguenti:

classNameAsTagName = **false**
fullJavaClassName = **true** | **false**
fullClassNameAsAttribute = **true**

```
<object field="value" objectID="1"
        class="java.lang.String" >
```

classNameAsTagName = **true**
fullJavaClassName = **false**
fullClassNameAsAttribute = **true**

```
<String field="value" objectID="1"
        class="java.lang.String" >
```

classNameAsTagName = **true**
fullJavaClassName = **true**
fullClassNameAsAttribute = **flase**

```
<java.lang.String field="value" objectID="1" >
```

Implementare writeObject()

Una classe può dichiarare uno specifico metodo per personalizzare la serializzazione o la descrizione solo dei suoi campi specifici. Le signature dei metodi sono:

```
private void writeObject(XmlWriter wrt);
   //per la descrizione di oggetti
private void writeObject(ObjectOutputStream out);
   //per la serializzazione
```

Questi metodi sono facoltativi e servono se si vuole fornire una rappresentazione alternativa da quella di default.

Lo sviluppatore non si deve preoccupare della coordinazione con le superclassi, queste verranno gestite da `XmlWriter`.

Il secondo metodo è quello standard introdotto dalle specifiche Java e produce in maniera polimorfica o un frammento XML o uno stream binario in base al tipo di stream utilizzato, per i dettagli sul tipo di supporto offerto da XmlTools al meccanismo standard di serializzazione si veda il paragrafo **Errore. L'origine riferimento non è stata trovata.**.

All'interno di questi metodi è possibile richiamare uno qualunque dei metodi di scrittura, oppure nel caso si usi il primo è anche possibile creare propri elementi o commenti XML e aggiungerli nella posizione corrente.

In qualunque momento all'interno di questi metodi è possibile invocare la scrittura di default per i campi specifici della classe. Un possibile utilizzo di questo sistema è il seguente:

```
private void writeObject(XmlWriter wrt) {
  wrt.defaultWrite(); //Invoke the default mechanism
  // write your code here.
}
```

Il meccanismo descritto è molto simile a quello presentato nelle specifiche Java: il metodo `defaultWrite()` è attivo solo durante la chiamata a `writeObject()`, se viene chiamato al di fuori di questo contesto si verifica una eccezione.

È importante notare che la dichiarazione di un metodo `writeObject` **deve** essere esattamente quella sopra riportata: se il metodo viene dichiarato pubblico, protetto o con dei parametri differenti XmlTools lo ignora.

Il metodo va dichiarato privato perché in questo modo se più discendenti di una classe dichiarano un proprio metodo per la personalizzazione, questi risulta indipendente dagli altri. Se il metodo fosse pubblico o protetto sarebbe invocata l'implementazione relativa alla classe più specializzata e risulterebbe impossibile una personalizzazione senza doversi coordinare esplicitamente con le superclassi.

Implementare l'interfaccia XmlExternalizable o FullXmlExternalizzable

Questa interfaccia è l'analogo di Externalizable introdotta dalle specifiche Java. Tramite l'implementazione di questa interfaccia è possibile avere un controllo totale su come vengono serializzati i campi sia della classe che delle sue superclassi.

A tal proposito i metodi di questa interfaccia si devono coordinare esplicitamente con le superclassi per ottenere una corretta serializzazione o deserializzazione.

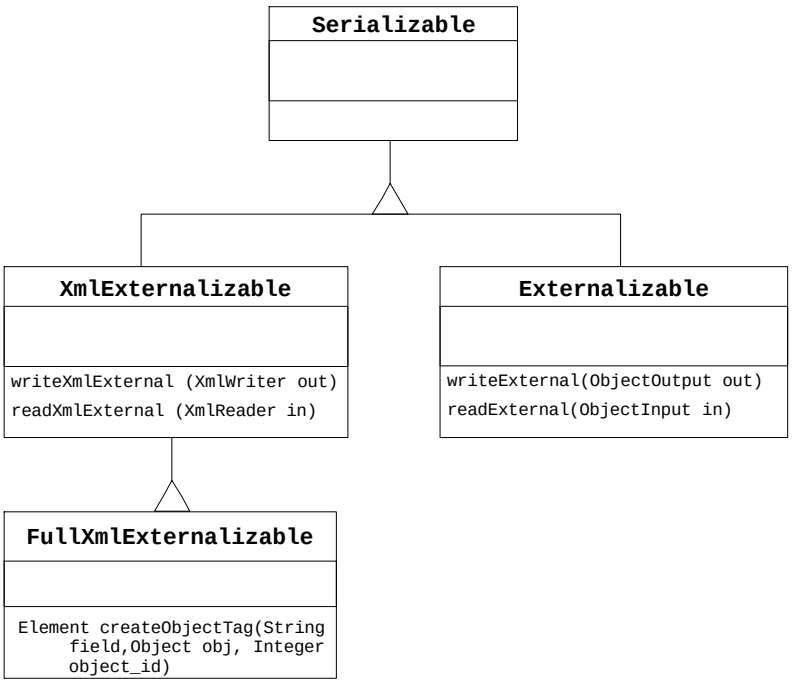

Figura 10 Interfacce per la personalizzazione del formato esterno di un oggetto

Nel momento in cui `XmlWriter` invoca il metodo `writeXmlExternal()` di una classe, l'elemento XML che descrive l'oggetto comprendente il suo `objectID`, il nome del campo in cui l'oggetto è riferito e il nome della classe è già stato creato, in questo modo lo sviluppatore dovrà solamente concentrarsi sulla rappresentazione interna della classe.

Se si desidera invece personalizzare anche il tag iniziale di un oggetto allora bisognerà implementare `FullXmlExternalizable`. Quest'ultima interfaccia richiede di implementare una funzione che ritorni l'elemento DOM che rappresenta l'intestazione di un oggetto e fornisce tutte le informazioni sotto

forma di parametri: il nome del campo di appartenenza dell'oggetto, l'istanza da serializzare e il suo `objectID`.

3.4.3 Normalizzazione degli identificatori di classe

Il set di caratteri validi per un identificatore di classe Java differisce da quello consentito da XML 1.0 come nome per un tag. Se si vuole produrre un output XML con i nomi di classi usati come nomi di tag (`XmlWriter` fornisce una apposita proprietà al riguardo), bisogna procedere a una conversione dagli identificatori Java ai nomi XML.

Ad esempio, uno dei simboli proibiti nei nomi XML è il dollaro. L'identificatore di classe, utilizzato da Java, per una classe interna è nella forma "`NomeClasse$NomeClasseInterna`", il simbolo dollaro verrà convertito da `XmlWriter` in '`_S`'.

`XmlReader` è in grado di ricostruire il nome originale grazie all'adozione di un particolare formalismo, si vedano le API di `XmlTools` per maggiori dettagli.

3.4.4 Protezione delle informazioni sensibili

Alcuni tipi di classe necessitano una maggiore cura da parte dello sviluppatore. Le classi che forniscono accessi a delle risorse di sistema fanno parte di questa categoria.

Ad esempio un file descriptor fornisce un handle che permette di accedere ad una risorsa del sistema operativo. Durante la deserializzazione anche lo stato di una classe viene ricostruito, essere in grado di forgiare un file descriptor potrebbe consentire forme di accesso illegali.

Per questa ragione un tool di serializzazione deve seguire un approccio conservativo e non deve fare affidamento sul fatto che lo stream o il documento XML contengano sempre e solo rappresentazioni valide di oggetti.

Per evitare di compromettere una classe, i campi sensibili di una classe non devono essere ripristinati dallo stream, oppure dovrebbero essere nuovamente validati dalla classe.

Le specifiche Java sulla serializzazione presentano alcune tecniche per proteggere lo stato sensibile di un oggetto e XmlTools è in grado di sfruttarle: per evitare che un campo sia serializzato basta contraddistinguerlo con la parola riservata *transient* del linguaggio Java. Quando `XmlWriter` individua uno di questi campi lo ignora e passa al successivo.

Quando si ha a che fare con classi particolarmente sensibili, si dovrebbe sempre prendere in considerazione l'idea di non renderla serializzabile.

`XmlReader` consente alle classi che vengono deserializzate di registrare un proprio metodo di *callback* per effettuare tutte le operazioni di validazione. Il metodo in questione viene chiamato una volta che l'intero grafo di oggetti è stato deserializzato ma prima che questi venga restituito all'applicativo.

3.4.5 Riduzione della ridondanza

Come si accennava all'inizio del capitolo, XML è un formato per la descrizione di informazioni con un alto livello di ridondanza intrinseca nella sintassi.

Risulta quindi importante adottare delle misure che limitino in maniera considerevole le dimensioni dei documenti XML prodotti dal processo di serializzazione, anche se la ridondanza intrinseca non sarà mai eliminabile.

In XmlTools sono stati implementati alcuni accorgimenti in tal senso:

1. Durante la serializzazione di array di oggetti, tutti gli elementi contenti valori `null` non vengono serializzati, ogni componente non nullo dell'array viene serializzato e riporta l'informazione della posizione occupata originariamente.
   Questo semplice accorgimento si ripercuote positivamente anche su strutture dati più complesse come i `Vector`, le Collezioni e le Mappe che implementano algoritmi di hash, dato che internamente fanno largo uso di array con una certa ricorrenza di valori `null`.
2. Gli elementi XML con un significato speciale per il tool di deserializzazione, sono stati ridotti al minimo indispensabile: il tag `<superclass>` ad esempio, viene usato per identificare le superclassi appartenenti ad un oggetto. Questo tag viene scritto solamente se la superclasse cui fa riferimento, contiene campi da serializzare. Per maggiori dettagli su questo tag si veda l'Appendice B. per ora basti sapere che è necessario per gestire in modo trasparente classi con più versioni.

3.4.6 Bilanciamento del DOM tree

Seguendo il meccanismo sin ora descritto, ogni nuovo oggetto incontrato da `XmlWriter` viene immediatamente serializzato in un elemento XML e inserito nella posizione corrente del DOM tree. I documenti prodotti risultano avere elementi molto annidati, spesso il primo elemento contiene quasi tutto il grafo di oggetti mentre poi seguono soprattutto dei riferimenti.

Si consideri per esempio la serializzazione di una possibile istanza di ParcheggioAuto introdotto in Figura 7:

```
<ParcheggioAuto objectID=”10”>
  <java.util.HashSet field="Contiene" objectID="11">
     . . . .
     <Auto objectID="12">
       <java.lang.String field=”Targa” objectID="13 ">
         <![CDATA[AB-123-CD]]>
       </java.lang.String>
       <Persona field=”Proprietario” objectID=”14”>
         <java.lang.String field=”Nome” objectID="15">
           <![CDATA[Mario Rossi]]>
```

```
          </java.lang.String>
          <java.lang.String field=”Indirizzo” objectID="16">
            <![CDATA[via del Monte, 22]]>
          </java.lang.String>
          <java.util.HashSet field=”Possiede” objectID=”17”>
             . . . .
             <Auto objectID=”18”>
               <java.lang.String field=”Tar ga” objectID="19"
                ><![CDATA[AA-000-BB]]>
               </java.lang.String>
               <Persona field=”Proprietario” objectREF=”14”
                />
             </Auto>
             <Auto objectREF=”12” />
          </java.util.HashSet>
        </Persona>
      </Auto>
      <Auto ObjectREF=”18” />
      <Auto . . .
    </java.util.HashSet>
    <Persona field=”Proprietario” objectREF=”14”/>
</ParcheggioAuto>
```

Dentro alla prima istanza di una persona sono contenute tutte le sue auto che quindi verranno riportate con un riferimento nell'insieme delle auto contenute nel parcheggio.

Questa struttura gode delle seguenti proprietà:

?? Favorisce la deserializzazione in quanto scandendo sequenzialmente il documento si è in grado di ricostruire il grafo di oggetti.

?? Se il documento fosse trasmesso i dati potrebbero essere deserializzati on-the-fly senza attendere il completamento del trasferimento.

Tuttavia se l'utilizzatore del documento XML fosse interessato solamente all'elenco delle auto contenute nel parcheggio, sarebbe costretto molto spesso a scendere in profondità nell'albero alla ricerca delle istanze di `Auto`.

Il documento inoltre risulta di scarsa leggibilità.

Per questi motivi si è deciso di implementare un algoritmo di bilanciamento del DOM tree. L'algoritmo viene utilizzato solamente su esplicita richiesta dell'utilizzatore in quanto comporta un certo overhead sia durante la serializzazione sia durante la deserializzazione.

Ai fini della sola serializzazione, l'algoritmo di bilanciamento non dovrebbe essere utilizzato, il discorso cambia se il documento XML deve essere condiviso tra ambienti eterogenei. Per aumentare la flessibilità del tool, in `XmlReader` è stato implementato il codice per supportare DOM bilanciati.

L'algoritmo di bilanciamento

Definiamo profondità di un elemento, il numero di elementi "padre" che contengono l'elemento in esame.

Durante la serializzazione di un oggetto vengono memorizzati il riferimento all'elemento XML che contiene la sua definizione e la sua profondità. L'idea di fondo è la seguente:

Si analizza l'oggetto da serializzare:
1. Se non è ancora stato serializzato, si serializza e ci si ricorda dove è stato scritto e la sua profondità.
2. Se è già stato serializzato, si guarda la profondità del riferimento che si sta per scrivere: se questa è minore della profondità dell'elemento contenente i dati dell'oggetto allora si **scambia** il riferimento con la definizione.

L'obiettivo è quello di ottenere un albero in cui ogni riferimento ad un oggetto abbia una profondità maggiore dell'elemento che contiene la sua definizione.

Tutte le informazioni necessarie all'algoritmo sono contenute nelle istanze `ObjectEntry`, lo schema visto nel paragrafo 3.4.1 ne riportava solamente alcune.

Questo tipo di algoritmo agirebbe durante la serializzazione e risulterebbe molto leggero ed efficiente… *Peccato che non funzioni*!
Un elemento contenente la definizione di un oggetto può a sua volta contenere dei riferimenti ad altri elementi. Quando viene effettuato uno scambio tra un elemento referente e la sua definizione, i riferimenti in essa contenuti potrebbero venirsi a trovare ad una nuova profondità minore delle rispettive definizioni. Tuttavia quei riferimenti, essendo già stati scritti non avrebbero più occasione per provocare nuovi scambi.

La strada che si è seguita è invece la seguente:
1. Produrre un albero non bilanciato e memorizzare le informazioni relative alla profondità delle definizioni e dei riferimenti durante la sua costruzione.
2. Per ogni elemento di definizione (quelli che contengono i dati) se esiste almeno un riferimento con profondità minore allora scambia i due elementi e aggiorna la struttura dati.
3. Se si sono verificati scambi nell'ultima iterazione allora ripeti 2

L'algoritmo implementato effettivamente è una versione ottimizzata dell'algoritmo appena descritto, tuttavia il consumo di memoria e tempo di elaborazione è comunque rilevante.

Lo stesso esempio di prima dopo il bilanciamento diverrebbe:

```
<ParcheggioAuto objectID=”10”>
  <java.util.HashSet field="Contiene" objectID="11">
     . . . .
     <Auto objectID="12">
       <java.lang.String field=”Targa” objectID="13">
         <![CDATA[AB-123-CD]]>
       </java.lang.String>
       <Persona field=”Proprietario” objectREF=”14”/>
     </Auto>
     <Auto objectID=”18”>
       <java.lang.String field=”Targa” objectID="19">
         <![CDATA[AA-000-BB]]>
       </java.lang.String>
       <Persona field=”Proprietario” objectREF=”14” />
     </Auto>
     <Auto . . .
   </java.util.HashSet>
   <Persona field=”Proprietario” objectID=”14”>
     <java.lang.String field=”Nome” objectID="15">
        <![CDATA[Mario Rossi]]>
     </java.lang.String>
     <java.lang.String field=”Indirizzo” objectID="16">
       <![CDATA[via del Monte, 22]]>
     </java.lang.String>
     <java.util.HashSet field=”Possiede” objectID=”17”>
        . . . .
        <Auto ObjectREF=”18” />
        <Auto objectREF=”12” />
     </java.util.HashSet>
   </Persona>
</ParcheggioAuto>
```

Come si può notare le definizioni delle istanze di auto sono riportate tutte in sequenza migliorando l’accessibilità delle informazioni.

Questo formato comunque comporta una certa complicazione al momento della deserializzazione: scandendo sequenzialmente il documento XML si incontrano dei riferimenti a ad istanze non ancora definite, bisogna dunque supportare una navigazione non lineare all’interno del documento, si veda il paragrafo 3.5.4 in proposito.

3.5 Il processo di ricostruzione (object deserialization)

`XmlReader` è la classe incaricata ricostruire le classi ristabilendo il vecchio stato memorizzato nello stream XML, questa classe si incarica inoltre di fornire una certa insensibilità ad eventuali modifiche apportate ad una classe, in modo che sia possibile deserializzarne ugualmente il vecchio stato.

`XmlReader` dispone inoltre di una dettagliata interfaccia protetta, in modo che in futuro sia possibile estendere agevolmente le sue funzionalità.

La versione attualmente implementata di `XmlReader` lavora su un albero DOM, un eventuale stream XML viene prima convertito in un albero DOM poi ha inizio il processo di ricostruzione.

3.5.1 Funzionamento

Dall'elemento radice per la serializzazione, che non deve necessariamente coincidere con la radice del documento XML, vengono estratte le informazioni riguardanti il formato utilizzato dal documento e la possibilità che esso rappresenti un archivio di oggetti valido.

Viene quindi inizializzato un cursore posizionandolo sul primo elemento figlio del nodo radice. Il cursore (un'istanza della classe `DomNavigator`) scandirà l'albero DOM filtrando gli elementi di interesse e scartando eventuali commenti o nodi di testo.

In condizioni normali l'albero è visitato analizzando uno dopo l'altro i figli del nodo radice, quando uno di questi elementi rappresenta un oggetto si scende di un livello e ricorsivamente si analizzano i suoi figli. In questo modo, l'ordine degli elementi incontrati nell'albero coincide con l'ordine degli elementi del documento XML originario.

Durante la deserializzazione viene costruita una mappa che associa ad ogni objectID il suo corrispondente oggetto deserializzato, la mappa funziona come una cache.

Quali classi sono deserializzabili?

Le specifiche Java sulla serializzazione richiedono esplicitamente che una classe implementi l'interfaccia `Serializable` e che la prima superclasse non serializzabile fornisca almeno un costruttore di default, ossia un costruttore che non richieda passaggio di parametri.

`XmlReader` implementa questo tipo di controllo per rispettare la semantica espressa dall'interfaccia `Serializable` e per mantenere la compatibilità con le specifiche Java. Se una classe non ha i requisiti richiesti viene lanciata un'eccezione. Se l'utente lo desidera il controllo può essere disabilitato.

L'algoritmo di deserializzazione.

I passi necessari per giungere alla ricostruzione di un grafo di oggetti equivalente a quello originale sono riassumibili in due problemi principali:

?? Allocare una nuova istanza appartenente alla classe specificata nel documento XML, sfruttando eventualmente la cache.
?? Ripristinare lo stato mantenendo se possibile la compatibilità con le serializzazioni di versioni precedenti della stessa classe.

I passi che portano alla soluzione del primo problema sono qui riportati:

1. Viene analizzato l'elemento corrente che chiameremo `tag`.
2. Se `tag` è il tag nullo si ritorna il valore `null`.
3. Se `tag` è un riferimento ad un altro elemento, controlla se in cache è presente un oggetto che corrisponde all'objectID riferito, quindi ritorna immediatamente l'oggetto presente in cache. In caso contrario viene effettuato un tentativo di lettura fuori ordine, si veda il paragrafo 3.5.4 per maggiori dettagli, se il tentativo fallisce viene lanciata un'eccezione in quanto si è incontrato un riferimento ad un oggetto non definito nel documento XML.
4. Se `tag` è un elemento che definisce un'istanza allora si guarda a che classe appartiene e viene controllato che la versione della classe presente in memoria sia compatibile con l'immagine serializzata in XML.
5. Se la classe in questione è tra quelle che XmlWriter (o i suoi eventuali discendenti) supporta direttamente allora viene chiamato un metodo protetto che si occuperà di deserializzare l'istanza. Ricadono in questa categoria gli oggetti `Class`.
6. Se `tag` rappresenta un oggetto stringa o un wrapper per tipi primitivi viene creata l'istanza corrispondente con il valore letto dal documento XML. Il nuovo oggetto è inserito nella cache in corrispondenza del proprio objectID e viene restituito come valore di ritorno.
7. Se nessuna delle condizioni 5 e 6 è verificata allora si è in presenza di un'instanza di una classe ordinaria e una nuova istanza viene allocata. Il suo stato attualmente non è valido e verrà ripristinato nei passi successivi.
8. Se la classe implementa una delle interfacce per la personalizzazione del processo di input (`XmlExternalizable` o `Externalizable`) la deserializzazione viene portata a termine dai metodi presenti in tali interfacce.
9. Il cursore viene posizionato sul primo elemento figlio dell'elemento corrente e viene invocato l'algoritmo di ripristino dello stato dell'oggetto (implementato dalla funzione `readFields()`)
10. L'istanza ricostruita è aggiunta alla cache e viene restituita come valore di ritorno.

Quando XmlReader deve ripristinare il valore di un campo di un'istanza dispone di tre informazioni:

?? La classe effettiva di appartenenza del campo
?? Il nome del campo
?? Il tipo del campo

In base al protocollo utilizzato per la serializzazione, la posizione occupata da un elemento XML che descrive il campo di una classe, determina univocamente il tipo effettivo della classe. Quindi in una codifica del tipo:

```
<Dipendente objectID=”123”>
   . . . .
   <superclass class=”Persona”>
      . . . .
      <int field=”anno_nascita” value=”1943” />
      . . . .
   </superclass>
</Dipendente>
```

il campo `anno_nascita` ha come classe effettiva `Persona` anche se l'istanza dell'oggetto da deserializzare è di tipo `Dipendente`.

Sia ora `obj` l'istanza allocata al punto 7 appena visto. L'algoritmo per il ripristino dello stato di un oggetto, una volta istanziato, opera nel seguente modo:

Per ogni classe cui appartiene `obj`, partendo dalla classe più specializzata e terminando alla prima superclasse non serializzabile vengono eseguiti i passi:

1. Se la classe corrente implementa uno dei metodi `readObject()` per la personalizzazione dell'input, gli si cede il controllo in modo che solamente i valori dei campi specifici per questa classe vengano ripristinati. Quindi si passa alla superclasse successiva.
2. Se la condizione 1 è falsa o se uno dei metodi di personalizzazione invoca il meccanismo standard per il ripristino dello stato, vengono esaminati tutti e soli gli elementi XML relativi alla classe corrente. Per ogni elemento, si cerca tra i campi della classe corrente quello che ha esattamente lo stesso nome e lo stesso tipo specificati nell'elemento XML. Se non viene trovato alcun campo con queste caratteristiche l'elemento corrente viene ignorato e si passa al successivo. Se viene trovato il campo cercato si legge il suo valore e lo si assegna all'istanza `obj`.

Questo sistema per deserializzare lo stato di un oggetto consente una notevole insensibilità alle eventuali modifiche che possono occorrere alle versioni di una classe, si veda il paragrafo 3.5.5 per maggiori dettagli.

3.5.2 Personalizzazione dell'input

Nella classe `XmlReader` sono state implementati tutti gli strumenti per supportare i vari livelli di personalizzazione introdotti da `XmlWriter`. I vari formati ottenuti dalla combinazione delle proprietà booleane di `XmlWriter` sono gestiti in modo trasparente: nell'elemento radice per la serializzazione sono riportati degli attributi con il valore delle vari proprietà.

Per supportare le personalizzazioni introdotte dai metodi writeObject() di XmlWriter, una classe può dichiarare i seguenti metodi:

```
private Object readObject(XmlReader rdr);
private Object readObject(ObjectInputStream in);
```

Per questi metodi valgono considerazioni analoghe a quelle fatte per i metodi `writeObject()`, unico vincolo da tenere bene presente è che i dati scritti in XML devono essere letti esattamente nello stesso ordine. Se uno di questi metodi non legge tutti i dati, `XmlReader` è in grado di ignorare i dati opzionali in eccesso, ma lancerà un'eccezione se si richiederanno più dati di quelli effettivamente presenti nel documento.

All'interno di questi metodi è possibile registrare un metodo di *callback* per la validazione dell'istanza una volta che questa sia stata interamente deserializzata. È inoltre possibile invocare il meccanismo standard per deserializzare i campi specifici della classe corrente in qualunque momento.

Le interfacce `XmlExternalizable` e `Externalizable` definiscono al loro interno un metodo per la lettura di informazioni, anche in questo caso la classe che li implementa deve coordinarsi esplicitamente con le sue superclassi.

3.5.3 Il problema del costruttore fantasma

Per riuscire a deserializzare lo stato di un'istanza bisogna prima essere in grado di allocarla. In circostanze comuni questo non è affatto un problema ma nella realizzazione di tool di questo genere il tipo di istanza da allocare risulta noto solo a tempo di esecuzione.

Il meta oggetto `Class` mette a disposizione un metodo per istanziare una qualunque classe ma pone un vincolo molto pesante: richiede la presenza di un costruttore di default.

Inoltre in base alle specifiche Java, per allocare correttamente un'istanza bisogna prima invocare, ai fini della deserializzazione, il costruttore di default della prima superclasse che non implementa l'interfaccia `Serializable`. In

questo modo, si riescono a inizializzare con un valore di default tutti i campi che non sono stati scritti nello stream per le ragioni viste in precedenza.

In pratica, si deve essere in grado di allocare un'istanza di una classe specializzata invocando però soltanto il costruttore di default di una sua superclasse. Da esperienze pratiche svolte sul campo è risultato chiaro che questa è proprio l'operazione svolta dal meccanismo standard per la deserializzazione di Java.

Tuttavia, utilizzando gli strumenti convenzionali forniti dal linguaggio Java questa operazione risulta impossibile: se si invoca il costruttore di default per la prima superclasse non serializzabile, il tipo effettivo dell'istanza sarà quello della superclasse, il che rende impossibile il ripristino dello stato delle classi specializzate.

Se invece si sceglie di invocare il costruttore della classe specializzata ci si trova davanti a due scelte:

?? Richiedere la presenza di un costruttore di default ed invocarlo.
?? Invocare uno dei costruttori presenti nella classe, qualunque siano i loro parametri.

Usando la prima soluzione si limita la flessibilità del tool: per le nuove classi si dovrebbe sempre prevedere questo costruttore, mentre si dovrebbe rinunciare a deserializzare le classi già esistenti che non lo implementano.

La seconda soluzione appare impraticabile: se da un lato è possibile invocare un costruttore, imparando solo a tempo di esecuzione di quali parametri necessita, dall'altro non si può prevedere quali valori passare a questi parametri.

I valori passati dovrebbero funzionare per qualunque tipo di classe, ma alcuni di questi potrebbero anche innescare, nel costruttore, operazioni non desiderate come la visualizzazione di finestre o, peggio ancora, il lancio di eccezioni.

La soluzione adottata da XmlTools è stata quella di implementare un *costruttore fantasma*, ossia un metodo nativo privato di `XmlReader` in grado di istanziare nel modo richiesto dalle specifiche Java una qualunque classe.

Java mette a disposizione una libreria di funzioni native per l'interazione con gli altri linguaggi di programmazione.

La Java Native Interface (JNI) [33] fornisce delle funzioni che sono in larga misura uno standard e almeno per quanto riguarda le primitive utilizzate, sono supportate da tutte le Java Virtual Machine in circolazione.

Il metodo nativo è scritto in ANSI C e deve essere compilato come una libreria dinamica, le operazioni di caricamento e rilascio della libreria sono gestite in automatico dalla Virtual Machine.

Utilizzando le primitive della JNI per l'allocazione dell'istanza, si è sicuri che il metodo ritorni un oggetto Java la cui memoria è correttamente allocata.

Per non limitare la portabilità di una applicazione Java che faccia uso di XmlTools, la presenza della libreria dinamica *non è obbligatoria*: quando viene inizializzata un'istanza di `XmlReader` viene controllata la presenza del codice nativo, se questo non è disponibile `XmlReader` funzionerà ugualmente, ma richiederà la presenza di costruttori di default per ogni classe che deve essere deserializzata.

3.5.4 Deserializzazione di DOM tree bilanciati

Se il documento XML da deserializzare è stato bilanciato con l'algoritmo introdotto da `XmlWriter`, può capitare che alcuni riferimenti precedano nel documento XML l'elemento che definisce l'oggetto.

I riferimenti di questo tipo vanno risolti subito, non si può uscire dal metodo `readObject` di `XmlReader` senza aver ricostruito un'istanza valida.

Quando viene incontrato un riferimento fuori ordine si salta alla destinazione indicata, si deserializza l'oggetto con il solito metodo e si ritorna alla posizione precedente; questa procedura è gestita in maniera trasparente senza che l'entità chiamante se ne accorga.

La classe `DomNavigator` che implementa il cursore utilizzato da `XmlReader` per la navigazione del DOM tree supporta due speciali operazioni: `goSub(Element dest)` e `Return()`.

In caso il documento XML risulti bilanciato, in fase di inizializzazione viene costruita una struttura che associa ad ogni objectID l'oggetto Element che contiene la serializzazione dell'oggetto corrispondente. In questo modo quando si incontra un riferimento ad un certo objectID si conosce la destinazione del salto.

3.5.5 Evoluzione delle classi.

Le classi non sono entità immutabili, con il passare del tempo possono subire modifiche e non è raro che la versione della classe che ha scritto il documento XML sia differente da quella che lo legge.

Gli obiettivi che si pongono le specifiche Java e XmlTools sono:

?? fornire un supporto trasparente e il più possibile automatizzato alla serializzazione di classi con più versioni, senza che sia necessario scrivere del codice in ogni classe per mantenere la compatibilità.

?? Consentire una comunicazione bidirezionale tra nuove e vecchie versioni delle classi.

Gestire la serializzazione di queste classi significa porsi degli interrogativi di fondo riguardo a cosa sia l'identità di una classe e quali cambiamenti siano compatibili con la versione precedente.

Le specifiche Java dicono che un cambiamento è definito compatibile se non modifica il contratto presente tra la classe ed un suo utilizzatore. Questo significa considerare due aspetti:

?? La classe evoluta non deve infrangere le assunzioni presenti nell'interfaccia della classe non evoluta, in questo modo la nuova classe può sostituire in pieno quella vecchia.
?? Quando si comunica con la versione precedente della classe devono essere fornite informazioni equivalenti e in quantità sufficiente affinché la vecchia classe possa continuare a soddisfare il contratto non evoluto.

Si consideri il seguente schema, le classi evolute (sulla destra) devono mantenere gli stessi impegni presi dalle loro dirette superclassi.

Il protocollo di serializzazione utilizzato per comunicare tra versioni differenti non fa parte di questo contratto: è specifico per ogni classe.

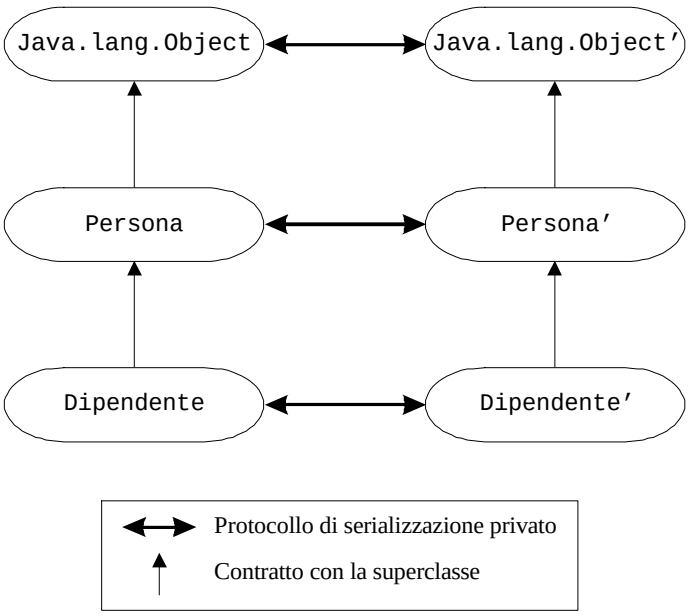

Figura 11 Classi evolute e contratti

Nel caso in cui una classe dichiari dei metodi per la personalizzazione dell'I/O, sarà suo compito mantenere la compatibilità tra versioni differenti; in caso contrario, sarà il meccanismo di serializzazione a doversene preoccupare.

Se la classe evoluta è ottenuta dalla versione precedente attraverso la somma di cambiamenti compatibili, la compatibilità è assicurata dal meccanismo di serializzazione; in caso contrario, le due classi non saranno compatibili.

Esempi di cambiamenti compatibili sono: aggiungere campi, aggiungere classi nella gerarchia, cambiare il modificatore di accesso ad un campo (da public a private ecc.).

Esempi di cambiamenti incompatibili sono: cancellare dei campi, scambiare delle classi all'interno della gerarchia, trasformare campi dinamici in campi statici o "`transient`".

L'elenco completo delle assunzioni fatte dal meccanismo di gestione delle versioni e gli elenchi dei cambiamenti considerati compatibili o incompatibili possono essere trovati nelle [32].

Per mantenere la compatibilità con il meccanismo di gestione delle versioni introdotto dalle specifiche Java, XmlTools utilizza lo stesso sistema per identificare la versione di una classe.

Per determinare con quali versioni è compatibile una classe, ogni classe presente sullo stream o documento XML, ha associato un identificatore numerico chiamato SUID (Stream Univoque IDentifier). Lo SUID è un intero a 64 bit calcolato da un algoritmo di hashing[9] applicato allo stream di byte ottenuto concatenando varie informazioni inerenti la classe quali: il nome della classe, delle interfacce implementate, dei methods, e dei campi.

`XmlWriter` scrive in testa al documento una tabella in cui ad ogni identificatore di classe che compare nel documento è associato lo SUID calcolato al momento della serializzazione.

Un esempio di questa tabella è il seguente frammento XML:

```
<meta xmlns="momis.project.dsi.unimo.it">
  <suid.entry class="Persona"
              suid="-2047135734441853794" />
  <suid.entry class="java.lang.String"
              suid="-6849794470754667710" />
  <suid.entry class="java.util.HashMap"
              suid="362498820763181265" />
  <suid.entry class="java.util.HashSet"
              suid="-5024744406713321676" />
  <suid.entry class="java.lang.Class"
```

[9] L'algoritmo è il Secure Hash Algorithm (SHA-1) del National Institute of Standards and Technology (NIST)

```
              suid="3206093459760846163" />
  <suid.entry class="[Ljava.lang.String;"
              suid="-5921575005990323385" />
</meta>
```

Per default una nuova versione di una classe è considerata non compatibile con la precedente, infatti è sufficiente cambiare anche solo in nome di un metodo perché lo SUID cambi radicalmente.

In fase di inizializzazione, `XmlReader` carica la tabella degli SUID e quando si deve istanziare una classe presente nel documento XML, il suo vecchio SUID viene confrontato con quello attuale; se sono diversi viene lanciata un'eccezione.

Se si vuole che una classe sia compatibile con una versione precedente, si deve dichiarare esplicitamente lo SUID della precedente versione. Un possibile esempio di dichiarazione è il seguente:

```
static final long serialVersionUID = 5611216200593098020L;
```

Per conoscere lo SUID di una classe si può utilizzare un tool fornito con l'ambiente di sviluppo, per JDK si utilizza il tool "`serialver`".

3.6 Supporto alle specifiche Java per la serializzazione di oggetti

XmlTools è in grado di fondersi con il meccanismo standard di serializzazione di Java. `XmlWriter` e `XmlReader` non sono discendenti di `ObjectOutputStream` o `ObjectInputStream`. Le specifiche Java impongono che una classe che voglia implementare un metodo personalizzato `writeObject()` o `readObject()`, ricevano come parametro un discendente di questi due stream.

Per supportare dunque questa funzionalità sono stati implementati due wrapper, uno per `XmlWriter` e uno per `XmlReader`:

?? `XmlObjectOutput`
?? `XmlObjectInput`

`XmlWriter` e `XmlReader` mettono a disposizione degli sviluppatori anche un metodo per sostituire questi wrapper con altri discendenti di `ObjectOutputStream` o `ObjectInputStream`.

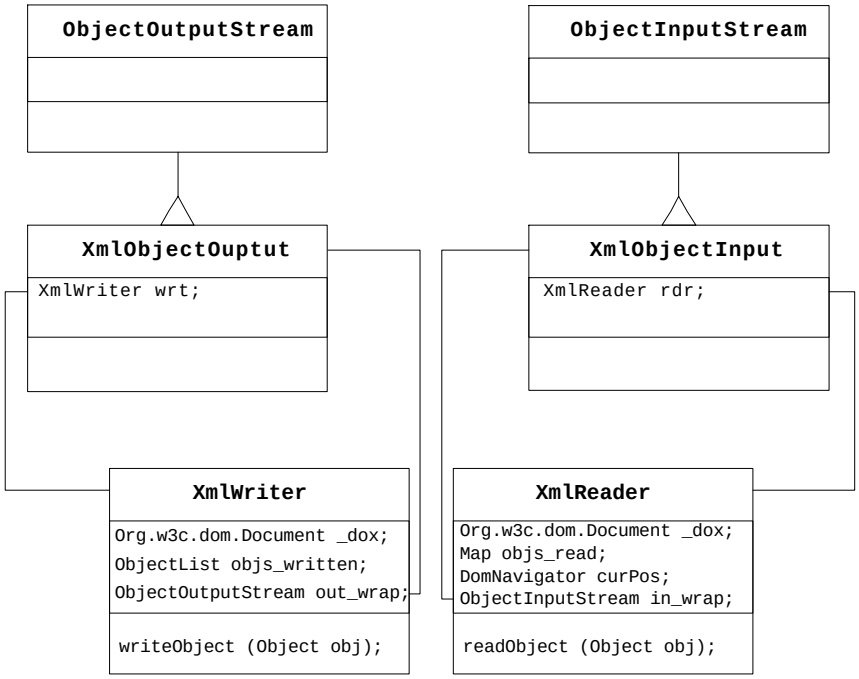

Figura 12 Diagramma delle classi per i wrapper di XmlReader e XmlWriter

I due wrapper estendono gli stream standard di Java e reimplementano da zero gran parte dei loro metodi.

Ogni volta che `XmlWriter` o `XmlReader` individuano un metodo di personalizzazione dell'I/O conforme alle specifiche Java, utilizzano uno di questi wrapper per poterlo invocare correttamente.

L'utente può anche utilizzare direttamente i wrapper, se lo desidera, in questo caso la serializzazione in XML risulta assolutamente trasparente all'applicativo.

3.7 Utilizzo di XmlTools

Nella progettazione di XmlTools si è voluto rendere il suo utilizzo identico a quello del meccanismo standard di serializzazione, in modo che anche applicativi non pensati per la serializzazione XML possano ugualmente trarne un vantaggio.

Serializzazione

Si inizia con l'istanziare o `XmlWriter` o `XmlObjectOutput` utilizzando uno dei loro molteplici costruttori, ad esempio:

```
XmlObjectOutput out = new XmlObjectOutput();
```

Quindi si cominciano a serializzare gli oggetti che interessano, sia `obj` un riferimento a una HashMap ad esempio, per serializzarla in XML sarà sufficiente invocare:

```
out.writeObject(obj);
```

Deserializzazione

In maniera del tutto analoga per deserializzare un oggetto occorrerà prima creare un'istanza di XmlReader o XmlObjectInput, magari passandogli uno stream XML:

```
FileInputStream fin = new FileInputStream ("My_file.xml");
XmlObjectOutput in = new XmlObjectOutput(fin);
```

quindi sarà sufficiente invocare:

```
HashMap map = (HashMap)in.readObject();
```

si noti che `readObject()` ritorna un generico `Object`; per utilizzare tutte le proprietà e i metodi dell'oggetto appena ricostruito è necessario effettuare un cast dinamico sul tipo dell'oggetto che ci si aspetta (in questo caso una `HashMap`).

Capitolo 4

Esportazione di schemi ODL_I^3 in XML 1.0

Durante lo svolgimento della tesi si è cercato di porre le basi affinché il sistema MOMIS in futuro possa supportare una sorgente integrata di dati in formato XML. Attualmente il sistema accetta il linguaggio ODL_I^3 (evoluzione del linguaggio ODL di ODMG) per la descrizione degli schemi sorgenti e il linguaggio OQL_I^3 per le interrogazioni; entrambi i linguaggi non sono degli standard universalmente riconosciuti e quindi si cerca con questo lavoro di aprire una finestra sul mondo di XML.

Nel capitolo verrà analizzato come sia possibile tradurre uno schema, espresso mediante il linguaggio ODL_I^3, in una struttura XML che possa essere utilizzata per produrre i risultati di una interrogazione.

Verranno analizzate le principali caratteristiche del linguaggio ODL_I^3 e per ognuna si proporrà una possibile traduzione che risulti facilmente implementabile per il modulo incaricato di esportare i dati.

L'obiettivo dello studio era giungere alla realizzazione di un software che realizzi la traduzione in maniera automatica. Nel prossimo capitolo verrà illustrato il progetto del traduttore.

4.1 Il linguaggio ODL_I^3

Per una ricca rappresentazione semantica degli schemi sorgenti e degli object pattern[10] associati con le sorgenti di informazione da integrare, si è introdotto un linguaggio object-oriented, chiamato ODL_I^3.

Seguendo le raccomandazioni ODMG e I^3, il modello ad oggetti ODM_I^3 si avvicina molto al modello ODM. ODL_I^3 è un linguaggio indipendente dalla sorgente per l'estrazione di informazioni utilizzato per descrivere schemi eterogenei e dati semistrutturati in un formato comune.

[10] Si veda il glossario per una definizione di object pattern.

Rispetto ad ODL, ODL_I^3 introduce le seguenti estensioni:

Union: il costrutto di unione denotato con `union`, è stato introdotto per esprimere strutture dati alternative nella definizione di una classe ODL_I^3, in modo da poter catturare le caratteristiche dei dati semistrutturati.

Attributi opzionali: sono denotati da (?) e sono stati introdotti per indicare che un determinato attributo è opzionale per un'istanza (cioè potrebbe non venire specificato nell'istanza). Anche questo costrutto è stato introdotto per catturare altri requisiti dei dati semistrutturati.

Regole per vincoli di integrità: Questo tipo di regola è stato introdotto in ODL_I^3 perché si potesse esprimere, in maniera dichiarativa, vincoli di integrità di tipo if-then sia all'interno della stessa sorgente, sia tra sorgenti differenti.

Relazioni intensionali: sono relazioni terminologiche che esprimono la conoscenza per lo schema sorgente. Le relazioni intensionali sono definite tra classi e attributi e sono specificate utilizzando i nomi classe/attributo, denominati termini. In ODL_I^3 possono essere specificate le seguenti relazioni:

?? Sinonimia (SYN), definita tra due termini t_i e t_j, con $t_i ? t_j$, che siano considerati sinonimi in tutte le sorgenti considerate (cioè, t_i e t_j possono essere utilizzati indifferentemente in ogni sorgente per indicare un certo concetto).

?? Ipernimia (Boarder terms) (BT), definita tra due termini t_i e t_j tali che t_i ha un significato più generale di t_j. La relazione BT non è simmetrica. La relazione contraria a BT è NT (Narrower Terms) o iponimia.

?? Termini correlati (RT) o associazione positiva, definita tra due termini t_i e t_j che siano generalmente utilizzati insieme nello stesso contesto nella sorgente considerata.

Relazioni estensionali. Le relazioni intensionali SYN, BT e NT definite tra due classi C_1 e C_2 possono essere "rafforzate" dichiarando che esse sono anche relazioni estensionali. Di conseguenza in ODL_I^3 possono essere definite le seguenti relazioni estensionali:

?? C_1 SYN_{ext} C_2: questo significa che le istanze di C_1 sono le stesse di C_2.

?? C_1 BT_{ext} C_2: questo significa che le istanze di C_1 sono un sovrainsieme delle istanze di C_2.

?? C_1 NT_{ext} C_2: questo significa che le istanze di C_1 sono un sottoinsieme delle istanze di C_2.

Inoltre, le relazioni estensionali vincolano la struttura delle due classi C_1 e C_2, ossia C_1 NT_{ext} C_2 è semanticamente equivalente a una relazione "*isa*". Per riassumere:

?? una relazione estensionale C_1 NT_{ext} C_2 equivale a una relazione "*isa*" C_1 ISA C_2 più una relazione intensionale C_1 NT C_2;
?? una relazione estensionale C_1 BT_{ext} C_2 equivale a una relazione "*isa*" C_2 ISA C_1 più una relazione intensionale C_1 BT C_2;
?? una relazione estensionale C_1 SYN_{ext} C_2 equivale a due relazioni "*isa*" " C_1 ISA C_2 e C_2 ISA C_1 più una relazione intensionale C_1 SYN C_2.

Una relazione "*isa*" " C_1 ISA C_2 è esprimibile in ODL_I^3 dalla seguente regola di integrità:

```
rule Rule forall X in C1 then X in C2
```

Regole di mapping (mapping rules): questo tipo di regola è stato introdotto in ODL_I^3 per esprimere le relazioni che esistono tra la descrizione dello schema integrato e la descrizione ODL_I^3 degli schemi delle sorgenti originali.

I wrapper di MOMIS traducono gli schemi delle varie sorgenti, in schemi ODL_I^3. Inoltre un wrapper è responsabile anche dell'aggiunta del nome della sorgente e del tipo (relazionale, ad oggetti, semistrutturato). La traduzione in ODL_I^3 seguendo la corretta sintassi (si veda l'Appendice D) è eseguita come segue. Data una relazione (un pattern `<l,A>`), la traduzione comprende i seguenti passi:

1. Una classe ODL_I^3 è definita con un nome corrispondente al nome della relazione (di `l`), rispettivamente.
2. Per ogni attributo della relazione (label `l' ? A`), viene definito un attributo nella classe ODL_I^3 corrispondente. Quindi vengono estratti i domini per gli attributi.

Come esempio, viene riportata la rappresentazione ODL_I^3 alcune classi locali: (un esempio completo di uno schema ODL_I^3 può essere trovato in [31])

```
interface Fast_Food (
        source semistructured ED)
{
   attribute Owner owner ?;
   attribute Fast_Food nearby ?;
   attribute long phone ?;
   attribute Address address;
   attribute string name;
   attribute string category;
   attribute set<string> specialty;
   attribute long midprice ?;
};

interface Restaurant (
        source relational FD
        key (r_code)
```

```
        foreign_key (pers_id) references Person (pers_id)
) {
     attribute long tourist_menu_price;
     attribute string zip_code;
     attribute string r_code;
     attribute string name;
     attribute long category;
     attribute string special_dish;
     attribute long pers_id;
     attribute string street;
};
```

Partendo dalle descrizioni ODL_I^3 delle varie sorgenti locali, il sistema MOMIS è in grado di produrre uno schema integrato formato da varie classi globali.

4.1.1 La mapping table

Le classi globali sono il risultato di una operazione di clustering sulle classi ODL_I^3. Per ogni cluster individuato viene definita una classe globale che rappresenta una vista integrata di tutte le classi appartenenti al cluster.

Per ogni classe globale è definito un insieme di attributi globali e per ognuno di essi è fornito il mapping intensionale con gli attributi locali (cioè quegli attributi che appartengono alle classi del cluster).

Gli attributi globali sono ottenuti in due passi:

1. L'unione degli attributi di tutte le classi appartenenti al cluster;
2. Fusione degli attributi simili;

nel secondo passo le ridondanze sono eliminate in maniera semi-automatica prendendo in considerazione le relazioni presenti nel common thesaurus. Per ogni classe globale viene generata una tabella persistente (la mapping table) che memorizza tutte le infomazioni sul mapping intensionale; la mapping-table è una tabella le cui colonne rappresentano l'insieme delle classi locali appartenenti al cluster e le righe rappresentano gli attributi globali.

	Computer_Science	Computer_Science	Computer_Science	University	University	tax_position_xml
Person	CS_Person	Professor	Student	Research_Staff	School_Member	Student
belongs_to	NULL	belongs_to	NULL	NULL	NULL	NULL
dept_code	NULL	NULL	NULL	dept_code	NULL	NULL
e_mail	NULL	NULL	NULL	e_mail	NULL	NULL
globalAttribute0	NULL	rank	rank	NULL	NULL	NULL
globalAttribute1	NULL	NULL	year	NULL	year	NULL
globalAttribute2	NULL	NULL	NULL	NULL	faculty	faculty_name
globalAttribute3	last_name	last_name	last_name	name	name	name
globalAttributeX0	first_name	first_name	first_name	NULL	NULL	NULL
section_code	NULL	NULL	NULL	section_code	NULL	NULL
student_code	NULL	NULL	NULL	NULL	NULL	student_code
takes	NULL	NULL	takes	NULL	NULL	NULL
tax_fee	NULL	NULL	NULL	NULL	NULL	tax_fee

Figura 13: Esempio di Mapping table

Un elemento *MT*[*ag*][*L*] rappresenta come l'attributo globale *ag* viene mappato nella classe locale *L*. Ogni elemento *MT*[*ag*][*L*] può assumere uno dei seguenti valori:

- *MT*[*ag*][*L*] = *al* : l'attributo globale *ag* mappa sul solo attributo locale *al*.
- *MT*[*ag*][*L*] = al_1 `and` al_2 `and` al_3 `and` … `and` al_n : viene usato questo valore quando l'attributo globale *ag* è la concatenazione dei valori assunti da un insieme di attributi al_i appartenenti alla stessa classe locale *L*.
- *MT*[*ag*][*L*] = `case of` *al* $const_1$: al_1, … $const_n$: al_n : questa situazione si verifica quando l'attributo globale *ag* può assumere uno dei valori di un insieme di attributi locali al_i appartenenti alla stessa classe locale *L*. La scelta del giusto attributo si basa sul valore di un terzo attributo *al*, appartenente alla stessa classe che agisce come selettore.
- *MT*[*ag*][*L*] = `const` : in questo caso l'attributo globale non fa riferimento a nessun attributo locale e un valore costante è assegnato dal progettista.
- *MT*[*ag*][*L*] = `null` : In questo caso nessun attributo della classe L corrisponde all'attributo globale *ag*.

4.2 MOMIS come sorgente integrata di dati in formato XML

Il sistema MOMIS può essere concettualmente diviso in due macro blocchi: uno primo blocco per la creazione del global virtual schema (GVS), un secondo per la gestione delle query sullo schema integrato: il query manager.

Il sistema MOMIS è in grado di acquisire schemi provenienti da sorgenti eterogenee e di produrre uno schema globale integrato. Per descrivere lo schema globale viene utilizzato il linguaggio ODL_I^3; mentre il query manager, attualmente in fase di sviluppo, accetta come linguaggio OQL_I^3.

Entrambi i linguaggi sono proprietari e quindi non rappresentano uno standard riconosciuto. Per estendere il campo di applicazione del sistema MOMIS, si sta studiando la possibilità di fornire un supporto al linguaggio XML a due livelli:

- A livello di descrizione dello schema integrato.
- A livello di interrogazioni.

A livello di integrazione si vuole fornire la possibilità di tradurre uno schema espresso in ODL_I^3 in una struttura XML utilizzando gli strumenti propri di questo linguaggio.

A livello di interrogazioni si sta valutando la possibilità di supportare linguaggi di interrogazione specifici per XML.

L'idea è trasformare MOMIS in una sorgente integrata di dati XML: l'utente riceve una vista XML dello schema integrato ed invia al mediatore delle query in XML-QL o altro linguaggio di interrogazione per XML; il sistema fornisce i risultati delle interrogazioni creando un documento XML conforme alla vista fornita in precedenza.

Questo meccanismo non esclude quello tradizionale, ma lo estende per poter servire anche richieste di applicativi non espressamente progettati per ODL_I^3 e OQL_I^3.

Attualmente esistono due metodi per descrivere una struttura in XML:

?? Utilizzare una Document Type Declaration
?? Definire uno schema secondo le specifiche XML-Schema

La DTD è uno strumento previsto da XML 1.0 per descrivere e vincolare il contenuto di elementi e attributi XML, XML-Schema sono delle specifiche che estendono le capacità espressive di XML 1.0 ma non sono ancora state standardizzate.

I documenti XML sono in realtà delle istanze delle dichiarazioni di strutture presenti nella DTD o dalla dichiarazione dello schema contenuta negli elementi specifici di XML-Schema.

Il query manager può utilizzare queste informazioni per generare dei documenti istanza validi secondo la DTD o lo schema prodotti.

Ai fini del presente lavoro di tesi si è studiata la realizzazione di un traduttore che partendo da uno schema descritto in ODL_I^3, sia in grado di produrre una struttura XML.

La struttura prodotta è espressa mediante una DTD, durante lo studio si sono inoltre poste le basi per la realizzazione di un traduttore in XML Schema.

4.3 Semantica ODL_I^3 e DTD a confronto.

Come si è visto, il linguaggio ODL_I^3 è costituito dalle seguenti relazioni fondamentali:

?? *Isa* – le interfacce possono specializzare altre interfacce ereditandone tutte le proprietà.
?? *Part-of* – le interfacce possono essere aggregate tra loro per costruire classi molto complesse.
?? *Kind-of* – è possibile definire nuovi tipi di dato dichiarandone l'utilizzo quando necessario.

È inoltre possibile definire vincoli di integrità, chiavi primarie e candidate, vincoli di integrità referenziale, relazioni terminologiche e regole di mapping tra attributi globali e attributi locali.

La semantica del linguaggio ODL_I^3 è molto ricca, per contro una DTD di XML 1.0 ha le seguenti caratteristiche:

?? Consente di definire relazioni di *part-of*.
?? Esprime solo in maniera implicita relazioni *kind-of*.
?? Fornisce un supporto primitivo all'identità di un elemento.
?? Consente di imporre l'esistenza di un attributo riferito, ma non consente di vincolare quale entità riferire.

In una DTD XML non si possono definire tipi di dato, non è possibile indicare che un determinato elemento debba contenere un numero intero piuttosto che un reale o una stringa, gli unici tipi di dato definito è il PCDATA (Parsed Character DATA) e CDATA.

Entrambi i tipi descrivono una sequenza di caratteri, ma in PCDATA gli spazi non sono significativi e alcuni caratteri devono essere interpretati e non possono comparire direttamente nella sequenza se non attraverso dei richiami ad ENTITY[11].

Questi sono gli unici strumenti che le specifiche di XML 1.0 mettono a disposizione: ogni costrutto, ogni relazione esprimibile in ODL_I^3 deve essere ricondotta ad un utilizzo combinato di questi strumenti.

In letteratura esistono varie proposte per estendere i vincoli e la semantica di XML 1.0 (ad es. Xml-Schema, RDF Schema) ma nessuna di queste è ancora uno standard.

Appare chiaro che non tutte le caratteristiche di ODL_I^3 potranno essere mantenute in una traduzione verso XML 1.0, in questo studio si cercherà di capire quali di queste possono essere tradotte direttamente, quali possono essere descritte e quali invece non hanno un corrispettivo nell'output del traduttore.

4.3.1 Linee generali di traduzione

In generale le classi globali, le classi locali e gli attributi di ODL_I^3 vengono tradotte come elementi XML. ODL_I^3 consente di definire attributi di tipo complesso: è quindi opportuno utilizzare elementi anche nel caso degli attributi.

[11] Le ENTITY in XML funzionano analogamente alle macro in C.

Si ricorrerà agli attributi XML solamente per descrivere le proprietà di una determinata entità, ad esempio per indicare un riferimento, il valore di un tipo booleano, la definizione un valore univoco, ecc.

Perché la DTD prodotta abbia una qualche utilità, bisogna definire tutti gli elementi XML che rappresentano lo schema ODL_I^3. L'obiettivo della traduzione è una DTD che possa essere utilizzata da un componente quale il Query Manager per produrre documenti istanza contenenti i risultati di una interrogazione.

Il processo di traduzione comincia dalle singole sorgenti locali, quindi si passa al global virtual schema, dopo aver descritto il contenuto di ogni classe globale si esaminano le rispettive mapping table e si produce una valida traduzione per tutti gli attributi globali.

Il traduttore produce gli elementi corrispondenti alle classi globali, alle classi locali, agli attributi, ai tipi, ecc. fornendo una sorta di libreria dove il Query Manager potrà scegliere quali elementi utilizzare.

La struttura del documento istanza non è nota sino al momento in cui è nota la query, ogni query richiede in generale una struttura del documento istanza differente. Si considerino ad esempio le seguenti interrogazioni espresse in XML-QL:

```
WHERE <ComputerScience.SchoolMember>
          <stud_id> $s </stud_id>
          <first_name> $n </first_name>
               <last_name> $l </last_name>
            </ComputerScience.SchoolMember>
IN ‘‘dbgroup.dsi.unimo.it/data.xml’’
CONSTRUCT <Student>
            <stud_id> $s </stud_id>
            <first_name> $n </first_name>
            <last_name> $l </last_name>
          </Student>

WHERE <ComputerScience.SchoolMember>
         <last_name> $l </last_name>
         <address>$a</address>
      </ComputerScience.SchoolMember>
IN ‘‘dbgroup.dsi.unimo.it/data.xml’’
CONSTRUCT <Student>
            <last_name> $l </last_name>
            <address> $a </address>
          </Student>
```

producono due strutture XML differenti:

```
<Student>
```

```
    <stud_id>2054-941</stud_id>
    <first_name>Giacomo</first_name>
    <last_name>Alberghini</last_name>
  </Student>
  <Student>
    <stud_id>2054-942</stud_id>
    <first_name>Michele</first_name>
    <last_name>Merlo</last_name>
  </Student>
  …
```

per quanto riguarda la prima query, mentre per la seconda:

```
  <Student>
    <last_name>Alberghini</last_name>
    <address>
      <street>24, Parco della Vittoria</street>
      <city>Bologna</city>
    </address>
  </Student>
  <Student>
    <last_name>Merlo</last_name>
    <address>
      <street>45, v.le Giardini</street>
      <city>Bologna</city>
    </address>
  </Student>
  …
```

la prima avrebbe una dichiarazione DTD:

```
<!ELEMENT Student (stud_id, first_name, last_name)>
```

la seconda invece:

```
<!ELEMENT Student (last_name, address)>
```

La definizione degli elementi componenti l'elemento `Student`: `address`, `last_name` ecc. è contenuta tra quelle prodotte dal traduttore, l'elemento `Student` invece definisce specificamente la struttura del risultato della query.

Volendo ad esempio dare la possibilità all'utente di validare il risultato di una sua interrogazione, il modulo che crea il documento istanza dovrebbe includere tra le definizioni locali della DTD la dichiarazione della struttura del risultato della query.

Nell'esempio precedente il documento istanza dovrebbe dichiarare la propria definizione di `Student` nel seguente modo:

```
<!DOCTYPE query_res SYSTEM "ComputerScience.dtd"[
   <!-- Defines the instance document structure -->
   <!ELEMENT query_res (Student)*>

   <!ELEMENT Student (last_name, address)>
]>
```

supponendo che gli elementi `address`, `last_name` ecc. siano definiti nel file ComputerScience.dtd generato dal traduttore.

4.3.2 Il problema delle omonimie

Quando si dichiara una DTD bisogna prestare attenzione ad un vincolo molto stringente: *non si possono dichiarare due elementi che abbiano lo stesso nome*. Inoltre anche all'interno di uno stesso elemento non possono comparire attributi omonimi.

Il vincolo è stringente in quanto in ODL_I^3 le omonimie sono ampiamente tollerate. I casi in cui si possono verificare sono i seguenti:

- ?? classi appartenenti a sorgenti differenti possono avere lo stesso nome
- ?? classi differenti possono dichiarare attributi omonimi
- ?? una stessa classe può avere attributi omonimi in union tra loro

Come rimuovere le omonimie?

Si potrebbe essere tentati di utilizzare i namespace in quanto sono stati approvati come standard da parte del W3C.

Le "XML names specification" sono delle "recommendation" progettate per risolvere problemi di questo genere. Con queste specifiche è possibile

definire degli spazi di nomi nei quali includere gli elementi. Se gli elementi omonimi risiedono in spazi di nomi differenti non sorgono problemi di ambiguità.

ODL_I^3 richiede che all'interno di ciascun corpo di interfaccia gli attributi abbiano nomi univoci. Questo significa che si dovrebbe definire uno spazio di nomi per ogni body tradotto.

C'è però un grave ostacolo che impedisce l'utilizzo dei namespace: **queste specifiche estendono la semantica di XML 1.0 ma non ridefiniscono il concetto di validità di un documento rispetto a una DTD**.

In pratica i namespace associano al nome di un elemento o di un attributo uno spazio di nomi. Questa associazione viene esplicitata nelle istanze dei documenti

XML e non nella DTD la quale continua ad avere il vincolo dell'univocità dei nomi degli elementi definiti.

I namespace non si applicano dunque alle DTD ma solo alle loro istanze, questo fatto rende impossibile la soluzione del problema delle omonimie senza rilassare qualche vincolo.

I nomi delle classi globali non presentano problemi in quanto è garantita la loro univocità, per le classi locali invece sarà sufficiente utilizzare il loro nome in dotted notation.

Per risolvere il problema delle omonimie per gli attributi ODL_I^3 si potrebbe pensare di tradurli tutti in attributi XML, in questo modo si risolverebbe il problema per gli attributi omonimi in classi differenti: questa sarebbe solamente una soluzione parziale ed inoltre non è neppure praticabile in quanto un attributo ODL_I^3 potrebbe essere di tipo complesso (es. una struct), mentre gli attributi XML sono definiti solamente su domini semplici formati da sequenze di caratteri.

Appare quindi evidente che l'unica soluzione possibile consiste nel rinominare gli attributi ODL_I^3 omonimi. Il criterio utilizzato non potrà essere arbitrario, bisognerà sceglierne uno che fornisca le seguenti caratteristiche:

- ?? rimuova tutte le omonimie tipiche di ODL_I^3
- ?? consenta di ottenere il nome dell'attributo ODL_I^3 conoscendo solo il nome dell'elemento XML che lo rappresenta. Questa informazione può essere necessaria al query manager per effettuare il mapping tra nomi ODL_I^3 e nomi XML.

Un criterio che soddisfa questi requisiti è quello di anteporre al nome di un attributo ODL_I^3 omonimo, il nome della classe globale a cui appartiene e un l'identificatore associato ad ogni body della classe.

L'identificatore serve nel caso l'interface dichiari più attributi in union tra loro e rappresenta il discriminante delle varie definizioni.

Esempio:

```
interface UN_Person (
   source relational University
   extent UN_Persons
   key (first_name, last_name)
){
   attribute string first_name;
   attribute string last_name;
   attribute string address;
} union second_definition {
```

```
  typedef struct location {
                            int number;
                            string street;
                            string city;
                           } location;

   attribute string first_name;
   attribute string last_name;
   attribute location address;
}
```

L'attributo address è omonimo nei due body dell'interfaccia, in uno è una semplice stringa, nell'altro è una struttura. Le due definizioni di elementi che descrivono address diventerebbero:

```
<!ELEMENT University.UN_Person.address (…)>
<!ELEMENT University.UN_Person.second_definition.address
(…)>
```

Si sono mantenuti i punti '.' come separatori in quanto non sono caratteri validi per l'identificatore di un tag.

Questa procedura di cambiamento dei nomi ODL_I^3 in nomi XML, sebbene corretta, tende a generare delle DTD, e quindi dei documenti XML, molto prolissi: la dimensione di un documento che utilizzi questi tag raggiungerebbe dimensioni ragguardevoli anche con pochi dati.

Nel traduttore si è quindi implementato un meccanismo differente: gli attributi vengono tradotti tentando di utilizzare nell'ordine:

- il nome semplice
- il nome esteso (sorgente.classe.attributo)
- un nome univoco ottenuto aggiungendo al nome semplice un identificatore numerico univoco (ad esempio per l'attributo address: address1)

L'informazione sul mapping tra nomi ODL_I^3 e nomi XML viene inserita in uno speciale elemento XML contenente delle meta-informazioni. Il meta elemento non ha solamente questa funzione, per una trattazione completa si veda il paragrafo 4.6.

Nella sezione relativa ad ogni classe viene inserito il tag `odli3.attribute`:

```
<odli3_meta>
 . . . .
 <odli3.source name="University" type="relational">
   . . . .
   <odli3.interface name="UN_Person" extents="UN_Persons">
     . . . .
     <odli3.attribute odli3Name="address"
```

```
                    type_id="location"
                    DTDName="address1"/>
   </odli3.interface>
 </odli3.source>
</odli3_meta>
```

In questo esempio si nota che all'elemento `address1` è associata la definizione della struct location dell'attributo ODL_I^3:
`University.UN_Person.address`
L'informazione sul tipo ODL_I^3 originale serve per identificare univocamente l'attributo tra tutti gli omonimi, il traduttore provvederà a fornire una valida rappresentazione del tipo dell'attributo nella definizione vera e propria della DTD.

Le informazioni contenute nel meta elemento sono facoltative ai fini della creazione di un documento XML valido secondo la DTD prodotta dal traduttore, se l'utente le richiede possono essere inserite nel risultato di una query o fornite separatamente.

4.3.3 Il problema delle chiavi

ODL_I^3 fornisce un pieno supporto al concetto di chiave. Essa si definisce come minimo insieme di attributi che con il loro valore identificano univocamente un'istanza.

È possibile definire tre tipi di chiavi:

- Chiave Primaria
- Chiavi Candidate
- Foreign key

Tuttavia nella DTD di XML 1.0 non viene introdotto alcun meccanismo per la definizione di chiavi, esiste solo il concetto di attributo XML con valore univoco.

Un attributo di tipo ID ha il vincolo di assumere un valore unico in ogni elemento in cui compare, questo deve valere per l'intero documento. È chiaro che con uno strumento di questo tipo si possono definire solamente delle chiavi unarie, ossia formate da un solo attributo. Ad esempio:

```
<!ELEMENT UN_Person (name, address)>
<!ATTLIST UN_Person
  fiscal_code     ID    #REQUIRED
>
```

L'attributo `fiscal_code` è a tutti gli effetti una chiave per `UN_person`. Non è importante cosa effettivamente sia contenuto come valore, l'importante è che nel documento non esistano altri elementi che dichiarino un attributo ID identico.

Si noti che l'attributo ID deve valere non solo per gli elementi con un determinato nome ma anche per tutti gli altri tipi di elementi che dichiarano un attributo di questo tipo.

Salvo il caso di chiavi unarie come quello appena visto, un attributo ID è inadeguato per esprimere chiavi con molteplici attributi ma può essere efficacemente utilizzato per definire chiavi surrogate.

In effetti un attributo ID è di per se una chiave surrogata per gli elementi XML. È bene ricordare che mentre una chiave identifica universalmente l'istanza di una classe, una chiave surrogata ha un limitato campo di validità che nel caso degli attributi ID coincide con il singolo documento XML.

Come tradurre dunque una chiave ODL_I^3?

Nella progettazione del traduttore si sono proposte numerose soluzioni ma tutte richiedevano la definizione di speciali elementi, il cui significato doveva essere noto all'utilizzatore del documento per interpretare correttamente la chiave.

Essendo quindi impossibile una traduzione diretta delle chiavi ODL_I^3, si è deciso di introdurre in ogni classe una chiave surrogata: `id_xml` e riportare una descrizione degli attributi componenti la chiave nell'elemento contenente le meta informazioni.

4.3.4 Vincoli di integrità referenziale e attributi "IDREF"

Anche per i vincoli di integrità referenziale la DTD di XML 1.0 non fornisce un particolare supporto, unico strumento utilizzabile sono gli attributi di tipo IDREF.

Questi attributi sono costretti ad assumere il valore di un attributo dichiarato come ID, il valore deve esistere in uno qualunque degli elementi del documento. Quando si utilizza questo tipo di attributi si è certi dell'esistenza dell'entità riferita ma non si sa nulla di che tipo essa sia.

La traduzione di una foreign key risulta impossibile, non è possibile tradurre nemmeno una chiave formata da un unico attributo in quanto non si può specificare l'entità riferita.

Anche in questo caso per non perdere completamente l'informazione sulle foreign key si è ricorsi a una descrizione mediante speciali elementi XML inseriti nell'elemento contenente le meta informazioni.

4.4 Traduzione di una sorgente locale

Una sorgente locale è caratterizzata da un nome univoco e da una descrizione sul tipo di sorgente: relazionale, ad oggetti, semistrutturata ecc. Il nome di una

sorgente viene riportato nei tag di tutte le sue classi locali ed il meta elemento contiene la descrizione relativa ad ogni sorgente:

```
. . .
<odli3.source name="ComputerScience" type="relational" >
  <. . .
```

Come si è visto, la DTD di XML 1.0 esprime solamente relazioni di tipo *part-of*, dichiarare un tipo ODL_I^3 invece significa utilizzare una relazione *kind-of* esplicita. Inoltre si sa che una DTD esprime solo implicitamente le relazioni di questo tipo per cui ci si chiede: è possibile trasformare una relazione *kind-of* esplicita in una implicita utilizzando solo relazioni *part-of* ?

Consideriamo la seguente dichiarazione ODL_I^3:

```
interface A_interface (...) {
attribute type attr_name;
};
```

supposto di creare un elemento `type` che fornisca gli attributi XML e gli elementi necessari per descrivere il tipo in questione, una traduzione per l'attributo `attr_name` può essere:

```
<!ELEMENT attr_name (type)>
<!ELEMENT type (a_content)>
<!ATTLIST type
   . . . .
>
```

Seguendo questa metodologia si mantiene l'informazione sul tipo dell'attributo ma si paga un alto prezzo in termini di ridondanza: ogni istanza di `A_interface` avrebbe una struttura del tipo:

```
<A_interface …>
  <attr_name>
     <type …>
         <a_content …
     </type>
  </attr_name>
</A_interface>
```

Se si pensa di utilizzare questa metodologia per tutto lo schema ODL_I^3 con decine e decine di attributi si capisce che un documento istanza potrebbe raggiungere dimensioni ragguardevoli anche per query che ritornano pochi dati.

Per migliorare la situazione si deve rendere ulteriormente implicita l'informazione sul tipo: nella DTD precedente `attr_name` porta l'informazione

sull'identità dell'attributo, mentre `type` ne definisce il contenuto, copiando le caratteristiche di `type` direttamente nell'elemento `attr_name`, si ottiene:

```
<!ELEMENT attr_name (a_content)>
<!ATTLIST attr_name
  . . . .
>
```

La nuova struttura è più compatta e l'informazione sul tipo dell'attributo è implicita nel suo contenuto. Nel caso si presenti la necessità di conoscere il tipo ODL_I^3 originale di un attributo, si può comunque consultare la corrispondente descrizione nel meta elemento.

Il discorso è diverso se si considerano tipi lista, array o bag. In questo caso l'unica maniera per mantenere i confini e gli eventuali attributi XML insiti nel tipo dei componenti (si veda ad esempio un array di booleani) è quella di definire un elemento "`type`" e replicarlo più volte a seconda della struttura da tradurre. Per motivazioni differenti, come verrà spiegato nel prossimo paragrafo, anche nel caso di attributi di tipo classe si è preferito mantenere la forma con l'elemento del tipo indicato.

I tipi ODL_I^3 sono divisi in due categorie: tipi valore e tipi classe. Segue ora la descrizione delle rispettive traduzioni.

4.4.1 Traduzione dei tipi classe

La traduzione delle classi locali presenta alcune problematiche che vanno risolte per poter giungere ad una corretta traduzione. La traduzione delle classi globali avverrà in maniera analoga, ma con l'aggiunta di alcune particolarità specifiche.

L'ereditarietà e la traduzione della relazione "*isa*" nei riferimenti ad altre classi sono i maggiori problemi da affrontare nella traduzione di una classe. Per meglio comprendere le soluzioni apportate a questi problemi viene ora esaminata la traduzione degli aspetti ordinari che caratterizzano una classe: attributi, corpi in union ecc.

Le classi ODL_I^3 possono dichiarare chiavi primarie, chiavi candidate e foreign key, non essendo possibile dare una traduzione diretta di questi costrutti, il traduttore genera una chiave surrogata e fornisce una descrizione delle chiavi contenuta nell'elemento con le meta informazioni.

Nell'intestazione di una classe ODL_I^3 è inoltre specificato se si tratta di una classe persistente e/o di una vista. Queste informazioni vengono riportate come attributi XML dell'elemento che traduce la classe. Ad esempio:

```
interface Person (
   source relational University
```

```
   extent CS_Persons
   . . . .
){
   attribute string first_name;
   attribute string last_name;
} union second_def {
   attribute string name;
   attribute location address;
};
```

Viene tradotto con il seguente elemento:

```
<!ELEMENT University.Person (. . .)>
<!ATTLIST University.Person
  view         (true|false)         #FIXED "false"
  persistent   (true|false)         #FIXED "true"
  id_xml           ID               #REQUIRED
>
```

Il valore degli attributi `view` e `persistent` è imposto in base alle informazioni fornite dalla dichiarazione dell'interfaccia, `id_xml` è la chiave surrogata la cui validità è l'intero documento XML.

La conoscenza relativa all'extent di una classe è riportata nel meta elemento.

Attributi

Gli attributi vengono tradotti con la definizione di elementi. Il nome di questi elementi rispecchia il criterio descritto alla fine del paragrafo 4.3.2. Un elemento che traduce un'interfaccia avrà come contenuto la sequenza degli elementi corrispondenti ai suoi attributi ODL_I^3.

```
<!ELEMENT University.Person ((first_name, last_name) |
                              (name, address) >
```

Le definizioni in union dei vari corpi di interfaccia sono perfettamente tradotte nella DTD.

Gli attributi opzionali di ODL_I^3 vengono tradotti con elementi opzionali all'interno del content model di un elemento classe, ad esempio:

```
attribute integer age*;
```

diventa

```
<!ELEMENT … (…, age?, …)>
```

Ereditarietà

Quando si parla di ereditarietà in ODL_I^3 si intende ereditarietà multipla: ogni classe può ereditare le caratteristiche di più superclassi contemporaneamente. ODL ed ODL_I^3 non stabiliscono alcuna regola per dirimere le ambiguità questo concetto si porta appresso come succede ad esempio se una classe eredita da due superclassi con uno o più parenti in comune.

La discussione dettagliata di queste problematiche esula dallo scopo di questa tesi: l'ereditarietà viene ricondotta dal traduttore ad una aggregazione tra classi.

Una classe specializzata può essere vista come l'unione dei suoi parenti diretti e delle definizioni di attributi introdotti nella classe. Adottando un ottica di questo tipo la traduzione DTD per una classe specializzata assume la forma:

```
<!ELEMENT source.specialized_class (parent_0, parent_1, … ,
                                    (specific_declarations))>
```

i vari `parent_0`, `parent_1` ecc. rappresentano gli elementi con cui sono state tradotte le varie superclassi da cui `specialized_class` eredita.

Ad esempio:

```
interface Research_Staff : Person
    ( source relational University
      extent Research_Staff
      ...)
{
      attribute string name;
      attribute string relation;
      attribute string e_mail;
      attribute integer dept_code;
     attribute integer section_code;
};
```

Viene tradotta con

```
<!ELEMENT University.Research_Staff (University.Person,
  name, relation, e_mail, dept_code, section_code)>
```

Riferimenti a classi

Spesso nelle classi sono presenti attributi di tipo classe per realizzare delle aggregazioni. Quando si traduce uno di questi attributi bisogna considerare che tra la classe specificata e i suoi discendenti esiste una relazione di tipo "*isa*". Questo comporta che ad un attributo di tipo classe può essere assegnata sia un'istanza della classe specificata, sia uno qualsiasi dei suoi discendenti.
Si consideri la seguente gerarchia come riferimento:

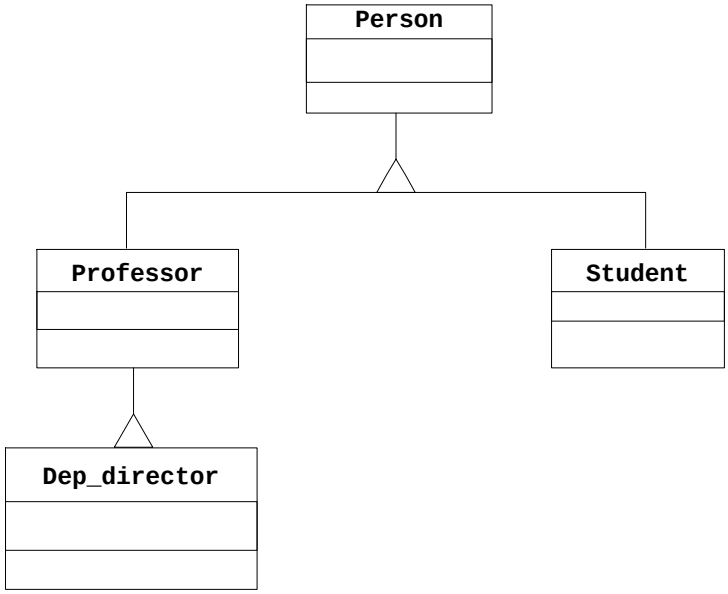

Figura 14: esempio di una possibile gerarchia di classi

La DTD non supporta le relazioni di tipo *isa*, se ad esempio un attributo

```
attribute Person a_person;
```

venisse tradotto

```
<!ELEMENT a_person (Person)>
```

si impedirebbe che ad `a_person` possa essere assegnato un'istanza di `Professor` o `Student`.

Occorre dunque che il traduttore esplori tutta la gerarchia dei discendenti della classe specificata nell'attributo e generi un content model che contempli tutte le possibili alternative. Nel caso dell'attributo `a_person`:

```
<!ELEMENT a_person (Person |Professor |dep_director
                   |Student )>
```

Questa definizione di `a_person` è formalmente corretta, tuttavia possono esistere casi in cui risulti impossibile generare un documento XML valido per questa DTD. Se infatti una delle classi coinvolte nella relazione *isa* possiede un attributo che fa riferimento a se stessa o a un suo parente, si genera una

definizione ricorsiva. Questa situazione è tollerata dalle specifiche sulla DTD ma non esiste in pratica la possibilità di creare un documento XML valido.

Si consideri ad esempio il caso che Person dichiari un attributo:

```
interface Person ( . . .)
{
  . . .
  attribute Person father; };
```

Nella creazione di un documento istanza bisognerebbe innestare dentro all'attributo `father` l'istanza di `Person` relativa al padre ma questa a sua volta dovrebbe contenere il proprio padre e così via…

Bisogna dunque introdurre un meccanismo che permetta di rompere la ricorsione. Se si introduce tra le varie alternative un elemento che rappresenti un riferimento ad un'istanza generica, il Query Manager si troverebbe nella condizione di poter scegliere se riportare per esteso la definizione di una istanza o mettere solamente un riferimento.

Il traduttore riporta in una sezione condizionale della DTD la definizione di un riferimento ad una classe generica:

```
<!ELEMENT odli3_reference EMPTY >
<!ATTLIST odli3_reference
  reference IDREF #IMPLIED
>
```

La definizione è resa condizionata per permettere al creatore del documento XML di dare una propria definizione del riferimento; in questo modo si potranno specificare altre informazioni dipendenti dall'implementazione che si sta realizzando.

L'attributo `reference` è di tipo IDREF e quindi è costretto a referenziare un attributo ID quale ad esempio la chiave surrogata `id_xml` di una istanza.

La traduzione corretta dell'attributo `a_person` è quindi:

```
<!ELEMENT a_person (Person |Professor |dep_director
                   |Student | odli3_reference)>
```

Il traduttore riporta sempre l'elemento odli3_reference tra le alternative possibili per un attributo di tipo classe. Grazie a questo elemento, chi crea il documento XML con i risultati di una query può scegliere, sulla base delle politiche che riterrà opportune, se riportare per esteso un'istanza riferita oppure indicare il riferimento.

Relationship

ODL_I^3 eredita da ODL degli speciali attributi: le relationship. Questi attributi rappresentano una relazione binaria tra due classi. Una relationship coinvolge sempre due classi ed è possibile specificare vari tipi di cardinalità: uno a uno, uno a molti, ecc. Nella classe riferita è garantita la presenza della definizione di una relazione inversa.

Le relationship utilizzano spesso tipi set o liste per ottenere relazioni con cardinalità maggiore di uno. La traduzione avviene in maniera analoga agli attributi di tipo classe ed utilizza eventualmente la forma introdotta per i set o le liste, si veda in proposito la trattazione sulla traduzione dei template type.

Nella traduzione in XML 1.0 l'indicazione della relazione inversa che viene riportata in un attributo #FIXED oltre che nel meta elemento. Ad esempio:

```
interface Course (. . .) {
relationship Professor taught_by
             inverse Professor::teaches;
};
```

viene tradotta in:

```
<!ELEMENT taught_by(University.Professor|odli3_reference)>
<!ATTLIST taught_by
  inverse   CDATA   #FIXED "University.Professor::teaches"
>
```

4.4.2 La traduzione dei tipi valore

I tipi valore si suddividono in tipi semplici e tipi "complessi": constructed type; vediamo i tipi semplici.

In questo paragrafo la traduzione dei vari tipi fa uso degli elementi con il nome del tipo tradotto; occorre tenere presente che questi elementi sono utilizzati solamente nei template type, mentre per gli attributi ODL_I^3 il contenuto dei tipi, compresi gli attributi XML, viene attribuito all'attributo stesso.

Tipi semplici

I tipi semplici sono tutti i tipi atomici di base (integer, float, char, boolean, etc…), i vari tipi collezione "template type" (set, list, bag, array), nonché i tipi definiti dall'utente, i "defined type"

tipi base:

I tipi base con la sola eccezione dei booleani vengono tradotti con elementi contenenti testo:

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

```
<!ELEMENT integer (#PCDATA)>
<!ELEMENT any (#PCDATA)>
<!ELEMENT char (#PCDATA)>
<!ELEMENT floating_point (#PCDATA)>
<!ELEMENT octet (#PCDATA)>
<!ELEMENT string (#PCDATA)>
```

Si può immediatamente notare che la semantica di questi tipi viene persa, l'utilizzatore della DTD non può avere la certezza che il tipo char contenga un solo carattere o che un elemento integer contenga solo cifre numeriche. Un elemento char poteva anche essere tradotto come:

```
<!ELEMENT char EMPTY>
<!ATTLIST char
  character  (&#0;|&#1;|&#2;|.....|&#65;|......)
              #REQUIRED
>
```

in questo modo il carattere sarebbe contenuto in un attributo XML ma si sarebbe uno soltanto ma a costo di dover produrre un pattern di alternative contenente tutti i caratteri definiti dallo standard UNICODE. Per ragioni di semplicità si è scelto il metodo precedente.

tipo boolean:

```
<!ELEMENT boolean EMPTY >
<!ATTLIST boolean
  value   (true|false)   #REQUIRED
>
```

Un tipo booleano può assumere solo due possibili valori: true o false, il valore deve essere specificato nell'attributo obbligatorio `value`.

Tipi definiti: template type

Volendo privilegiare la chiarezza espressiva della struttura, si è scelto di esportare i tipi definiti semplicemente definendo un elemento che racchiuda il tipo effettivo.

Ad esempio:

ODL_I^3 ;

```
typedef integer Vector[30];
```

DTD:

```
<!ELEMENT Vector    (integer*)>
<!ATTLIST Vector
```

```
  arraysize        CDATA            #FIXED "30"
>
```

Per ridurre la ridondanza di un documento istanza un array di tipi primitivi (int, float, boolean ...) si potrebbe utilizzare invece il tipo #PCDATA intendendo che il contenuto sia una lista di valori separati da spazi. Tuttavia questa soluzione non traduce in modo corretto le stringhe in quanto non si sarebbe più in grado di distinguere i confini dei singoli componenti.

Per tradurre un array di stringhe, di booleani o di strutture si deve ricorrere agli elemento string, boolean ecc. Per fornire una descrizione omogenea del contenuto di un template type si è preferito utilizzare la traduzione con gli elementi in tutti i casi possibili senza trattamenti speciali.

Set:

Per definire attributi set o list la soluzione proposta è:

ODL_I^3:

```
attribute set<something> a_set;
```

DTD:

```
<!ELEMENT a_set (something*)>
```

Qui viene perso il concetto di insieme: non si può garantire che il contenuto di `a_set` non abbia elementi duplicati.

List:

Le liste sono tradotte in maniera analoga, nella traduzione di una lista il concetto di ordine è mantenuto in quanto in XML, è importante la posizione che un elemento occupa rispetto agli altri.

Per aumentare la chiarezza espressiva si potrebbe essere tentati di definire i tipi set o list nel seguente modo:

```
<!ELEMENT set (some_element*)>
<!ELEMENT list (some_other_element*)>
```

ma ***questa dichiarazione è errata*** in quanto se nello schema ODL_I^3 sono dichiarati più insiemi contenenti elementi di tipo diverso, seguendo questa strada si viola il vincolo sull'unicità della definizione di un elemento. Ad esempio:

```
attribute set<evaluation> evaluation_set;
attribute set<course> study_plan;
```

non può essere tradotto in

```
<!ELEMENT set (voti*)>
```

```
<!ELEMENT set (corsi*)>
```

e nemmeno in

```
<!ELEMENT set (voti* | corsi*)>
```

in quanto nel primo caso si ha un errore di "definizione duplicata", nel secondo invece non si impedisce che un piano di studio possa contenere dei "voti" o `evaluation_set` possa contenere dei "`course`".

Array:

Il traduttore descrive gli array allo stesso modo delle list, tuttavia vale la pena spendere due parole su traduzioni alternative: gli array hanno associata una dimensione. Non è strettamente necessario ma se si vuole mantenere una distinzione dai set o dalle list, si può aggiungere un attributo XML che descriva la dimensione dell'array.

Ad esempio:

```
attribute float numbers[15];

<!ELEMENT numbers (float*)>
<!ATTLIST numbers
  arraysize        CDATA        #FIXED "15"
>
```

In XML 1.0 è difficile (ma non impossibile) imporre un numero di occorrenze minimo e massimo degli elementi contenuti in un elemento padre, quindi l'attributo arraysize sarebbe solamente informativo.

Usando un piccolo trucco, un modo per limitare il numero delle occorrenze di un elemento esiste: bisogna elencare esplicitamente nel content model del padre quante volte deve comparire un elemento, ad esempio:

```
<!ELEMENT father (child, child, child?, child?)>
```

Impone che `father` possa avere da 2 a 4 elementi `child`.

Non è conveniente tuttavia utilizzare questo sistema per limitare la dimensione di un array: nei casi con cardinalità molto elevate si otterrebbero delle DTD illeggibili.

Tipi definiti dall’utente: Constructed Type

I tipi definiti dall’utente si dividono in tipi enumerativi, tipi union e strutture.

Tipi enumerativi:

Fortunatamente almeno per i tipi enumerativi, XML fornisce un valido supporto a livello di DTD. L'enumerativo viene tradotto con un attributo XML i cui possibili valori possono essere scelti solo tra quelli dichiarati nella DTD.

Vediamo un esempio:

```
enum book_copy { one, two, three };
```

viene tradotto in:

```
<!ELEMENT book_copy EMPTY>
<!ATTLIST book_copy
  enum    (one | two | three)        #REQUIRED
>
```

Tipi Union

I tipi union in ODL_I^3 non sono ancora pienamente supportati, il componente software che effettua il parsing dei sorgenti ODL_I^3 infatti ritorna un'unica stringa contenente le varie definizioni non interpretate.

Per questo motivo il traduttore si limita a riportare tale stringa come attributo fisso.

Si consideri la seguente definizione di union

```
union unionName switch(integer) {
      case 10: int an_int;
      case 20:
      case 30:
               string a_String;
      default case:
               float  a_Float;
    }
```

attualmente viene riportata in:

```
<!ELEMENT unionName EMPTY>
<!ATTLIST unionName
  caseList        CDATA        #FIXED "case 10: int an_int"
case 20: case 30: string a_String; default case: float
a_Float;"
>
```

Se si potesse disporre dell'interpretazione della union una possibile traduzione sarebbe

```
<!ELEMENT unionName (int | string | float)>
```

Si perde il concetto di scelta in base al valore dello switch ma nel complesso la traduzione sarebbe corretta.

Strutture

Una struttura può essere efficacemente espressa in XML 1.0 ad esempio

```
typedef struct tagname {
    int a;
    string b;
    float c;
} struct_type
```

viene tradotta in:

```
<!ELEMENT struct_type (a,b,c)>
<!ELEMENT a (int)>
<!ELEMENT b (string)>
<!ELEMENT c (float)>
```

4.5 Traduzione del global virtual schema

La traduzione del global virtual schema si effettua traducendo ogni classe globale in esso definita. La traduzione di una classe globale è molto simile alla traduzione delle classi locali, l'unica grossa differenza è la traduzione degli attributi globali.

Un attributo globale GA può essere tradotto definendo un elemento il cui contenuto è ottenuto esaminando la mapping rule che definisce GA.

Poiché ogni possibile mapping rule è definita a partire dalla mapping table associata alla classe globale, il traduttore si concentra sulla traduzione in DTD della mapping table.

Il contenuto di un elemento XML relativo ad un attributo globale è dato dalle strutture degli attributi locali su cui mappa unite in maniera opportuna. La traduzione è quindi ottenuta esaminando l'intera linea della mapping table relativa all'attributo in questione.

4.5.1 Traduzione della mapping table

Ad ogni colonna della mapping table corrisponde una classe locale. Gli insiemi di attributi relativi ad ogni classe locale devono essere posti in union tra loro, il valore dell'elemento della mapping table determina il tipo di struttura da generare nella DTD. I possibili valori degli elementi della mapping table, come si è visto nel paragrafo 4.1.1 sono i seguenti:

?? Singolo attributo locale: viene tradotto con la struttura relativa all'attributo locale specificato:

```
<!ELEMENT global_attribute ((local_attribute)
                        |(other_element_definition) |…>
```

?? Concatenazione di attributi locali: viene generata una sequenza rappresentante l'and tra i vari attributi locali:

```
<!ELEMENT global_attribute ((local_attribute1,
          local_attribute2, local_attribute3,…) |
                    (other_element_definition) |… )>
```

?? **Una lista di attributi locali in union tra loro**: il giusto attributo è selezionato in base al valore di un altro attributo appartenente alla stessa classe globale. Nella traduzione il concetto di selettore viene perso, le strutture degli attributi elencati vengono semplicemente poste in alternativa tra loro.

```
<!ELEMENT global_attribute ((local_attribute1 |
          local_attribute2 |local_attribute3 |…) |
                    (other_element_definition) |… )>
```

?? **Valore costante**: in questo caso l'attributo globale è tradotto come un elemento vuoto contenente un attributo XML il cui valore è costretto a coincidere con la costante specificata:

```
<!ELEMENT global_attribute EMPTY>
<!ATTLIST global_attribute
  default_value    CDATA #FIXED "constant value"
>
```

In questo caso tutti gli altri elementi della riga della mapping table possiedono un valore NULL.

?? **Valore NULL**: In questo caso nessuna traduzione viene effettuata, si passa direttamente alla prossima colonna. In una riga di una mapping table esiste sempre almeno un valore diverso da NULL.

4.6 Semantica descritta.

Il linguaggio ODL_I^3 è stato progettato per descrivere schemi provenienti da sorgenti eterogenee. Dovendo realizzare un traduttore per XML 1.0, un linguaggio che fornisce scarsi strumenti per la rappresentazione della conoscenza, era inevitabile che alcuni aspetti semantici andassero perduti. Per limitare la perdita si è cercato di descrivere quei costrutti di cui non è possibile dare una traduzione diretta. Le descrizioni sono state inserite in uno speciale elemento XML.

La trattazione che segue utilizza spesso degli esempi per indicare la struttura di un determinato elemento XML appena introdotto. Per una DTD rigorosa si veda l'Appendice C

Il meta elemento possiede una struttura del tipo:

```
<odli3.meta>
  sorgente 1
     classe locale 1.1
     classe locale 1.2
     …
  sorgente 2
    classe locale 2.1
    classe locale 2.2
    …
  sorgente 3
  …
</odli3.meta>
```

dove sorgente 1, sorgente 2, classe locale 1.1 ecc. sono elementi XML contenenti le meta informazioni.

4.6.1 Descrizione XML di entità ODL_I^3

Sorgenti

Per ogni sorgente che compare nello schema ODL_I^3 viene riportato il nome e la descrizione del tipo: relazionale, ad oggetti, semistrutturata ecc.

Ad esempio:

```
<odli3.source name="University" type="relational" >
```

Il contenuto di questo elemento sono gli elementi relativi ad ogni interfaccia locale.

Classi locali

Le classi locali sono riportate tramite l'elemento `odli3.interface` che ne contiene il nome. Questo elemento ha il solo scopo di identificare il target degli elementi che contiene.

Per ogni attributo di una classe locale esiste un elemento figlio (`odli3.attribute`) che riporta oltre al nome dell'attributo ODL_I^3, il tipo originale e il nome utilizzato dal traduttore nella DTD.

```
<odli3.interface name="CS_Person" extents="CS_Persons">
     . . . .
     <odli3.attribute odli3Name="age"
```

```
                    type_id="integer"
                    DTDName="age"/>
    . . . .
</odli3.interface>
```

Chiavi

Le chiavi di ogni interfaccia sono elencate all'interno dell'elemento odli3.interface corrispondente. La descrizione di una chiave avviene definendo un elemento (`odli3.key`) in cui è riportato il nome della chiave in un attributo; il nome della chiave primaria è "PrimaryKey".

Per ogni attributo ODL_I^3 componente la chiave viene inserito nell'elemento `odli3.key` un elemento `key_field` che ne riporta il nome.

Ad esempio per l'interfaccia:

```
interface CS_Person
 ( source object Computer_Science
   extent CS_Persons
   key (first_name, last_name) )
{  attribute string first_name;
   attribute string last_name;  };
```

si ottiene:

```
<odli3.interface name="CS_Person" extents="CS_Persons" >
         <odli3.key name="PrimaryKey" >
            <key_field attribute="first_name "  />
            <key_field attribute="last_name "  />
         </odli3.key>
          ...
</odli3.interface>
```

Foreign key

Le foreign key sono descritte analogamente alle chiavi ordinarie:

?? viene definito l'elemento `odli3.foreignKey` che contiene il nome della chiave e il nome dell'interfaccia cui la chiave fa riferimento.

?? per ogni attributo ODL_I^3 componente la foreign key è riportato il suo nome nella classe corrente e il nome del corrispondente attributo nella classe riferita.

Ad esempio la seguente foreign key anonima:

```
interface Section
    ( source relational University
      extent Section
      key (section_code)
```

```
      foreign_key ( room_code ) references Room )
   {     ...
           attribute integer room_code;  };
```

genera la descrizione:

```
<odli3.foreignKey references="University.Room" >
   <key_field attribute="room_code "
              references="room_code "/>
</odli3.foreignKey>
```

Relatioship

Le relationship sono riportate per convenienza anche nel meta elemento, la loro descrizione è simile a quella degli attributi. Viene definito l'elemento `odli3.relationship` in cui oltre alle informazioni tipiche per gli attributi ODL_I^3, viene aggiunta la relazione inversa.

Ad esempio la relationship:

```
relationship Department dept inverse Department::r_staff;
```

viene descritta con:

```
<odli3.relationship odli3Name="dept"
                    type_id="University.Department"
                    DTDName="dept"
                    inverse="Department::r_staff; "  />
```

4.7 Semantica perduta di ODL_I^3

ODL_I^3 fornisce vari strumenti per la rappresentazione della conoscenza:

?? Relazioni terminologiche
?? Regole di integrità
?? Relazioni estensionali

Questi costrutti non si riescono a tradurre mediante la definizione di una struttura più o meno complicata.

La conoscenza dello schema in essi contenuta può essere descritta nel meta elemento, in questo caso tuttavia il traduttore si limiterebbe a riportare le stringhe che definiscono le varie regole così come sono riportate in ODL_I^3.

A causa dell'inadeguatezza di XML 1.0 nel rappresentare la conoscenza e dato che ai fini della realizzazione di una sorgente integrata di dati XML, non

sono strettamente necessari, si è deciso di non tradurre gli aspetti puramente semantici.

Capitolo 5

Il traduttore per XML 1.0

La realizzazione pratica del traduttore ha comportato la soluzione di numerosi problemi di carattere tecnico: il sistema MOMIS ha raggiunto una complessità notevole e realizzare un software che sia in grado di integrarsi con i moduli per il supporto ad ODL_I^3, ha richiesto lo studio e una comprensione approfondita dei progetti svolti in altri lavori di tesi.

5.1 Il software

ODL_I^3 è un linguaggio utilizzato nell'ambito di un gruppo di ricerca, anche se negli ultimi tempi si è molto consolidato non sono da escludersi modifiche future, atte a supportare eventuali sviluppi della ricerca.

Questo aspetto ha rappresentato un vincolo stringente per il progetto del traduttore: l'architettura deve facilitare le modifiche al traduttore nel caso si renda necessario modificare il linguaggio ODL_I^3.

La scelta di implementare un software monolitico appariva dunque in netto contrasto con le necessità del progetto MOMIS. Si è quindi cercato di realizzare un'architettura "scalabile" che permettesse di modificare il traduttore anche se non si ha completa conoscenza del suo funzionamento.

5.1.1 Architettura

Il traduttore è realizzato per compartimenti stagni: al progettista delle modifiche è richiesta solo la conoscenza di come tradurre in DTD le proprie novità.

MOMIS supporta ODL_I^3 attraverso una complessa gerarchia di classi che descrivono le varie entità: i tipi, gli attributi, le sorgenti, le interfacce ecc. Il codice di controllo della traduzione è distribuito su ogni entità che deve essere tradotta.

Ogni entità è in grado di tradurre se stessa tramite l'invocazione ricorsiva del processo di traduzione delle sotto entità direttamente dipendenti. Ad esempio

un'interfaccia sa rappresentarsi come elemento DTD e invoca la traduzione dei propri attributi, ma ignora come avvenga la traduzione dei tipi negli attributi. Una classe che descrive un attributo, dovrà considerarne il tipo per giungere ad una corretta dichiarazione DTD, ma i dettagli della traduzione del tipo vengono nascosti dalle classi "tipo".

Il codice che realizza la traduzione di una entità ODL_I^3 è ottenuto implementando una semplice interfaccia Java. In questo modo se si devono aggiungere nuove entità in ODL_I^3 sarà sufficiente implementare l'interfaccia di traduzione delle singole entità. Eventualmente si dovrà modificare anche il codice delle entità esistenti da cui dipendono le nuove entità in modo che le vecchie contemplino le nuove, ma solamente quello.

Il progettista deve dunque conoscere solo come tradurre l'entità che sta modificando e/o aggiungendo senza preoccuparsi di come funziona il processo di traduzione generale.

Lo schema di funzionamento generale del traduttore è il seguente:

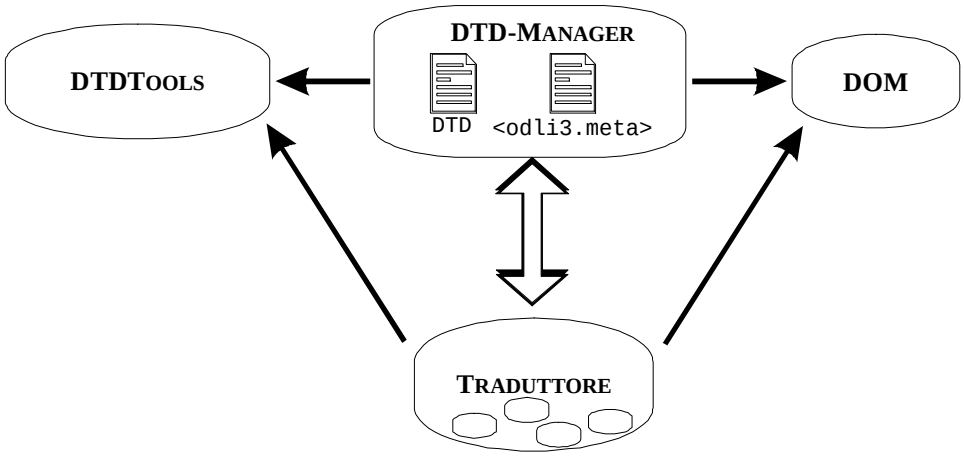

Figura 15: Schema generale del traduttore

DTD Manager

Dovendo distribuire il codice di controllo si rende necessaria l'introduzione di una struttura per il coordinamento della traduzione: il DTD manager. Questa risorsa fornisce servizi e informazioni ai vari comparti del traduttore e gestisce la creazione di una DTD ordinata e leggibile.

DTDTools

Poiché le DOM Level 1&2 specification non introducono una interfaccia standard per la creazione e l'editing di una DTD, si è reso necessario progettare ed implementare una libreria di classi che consentisse di descrivere e generare una DTD valida sintatticamente e semanticamente secondo le specifiche di XML 1.0.

DOM

Come si è visto nel capitolo precedente il traduttore necessita di creare un elemento XML contenente delle meta-informazioni. Per generare questo elemento viene utilizzato un DOM tree.

5.1.2 Il modulo DTDTools

DTDTools è composto da varie classi, ognuna delle quali modella un componente della DTD: dichiarazione di elementi, entity, sezioni condizionali, liste di attributi e commenti.

Il modulo implementa solo le funzionalità di una DTD utili ai fini del traduttore, tuttavia si è progettato il modulo in maniera che risulti facilmente estendibile e manutenibile.

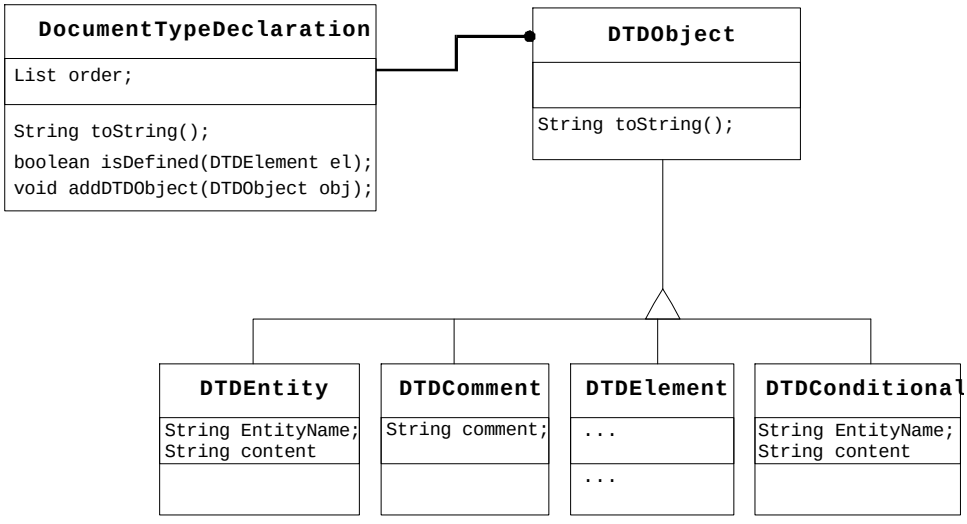

Figura 16: Schema semplificato del modulo DTDTools

DTDConstants

Tutte le classi di DTDTools implementano l'interfaccia DTDConstants, non riportata nello schema, dove sono definite tutte le costanti di uso comune nella creazione di una DTD.

DocumentTypeDeclaration

La classe `DocumentTypeDeclaration` gestisce automaticamente i dettagli per la generazione di una DTD sintatticamente corretta e mantiene al suo interno una lista di `DTDObject` che determina l'ordine con cui le varie dichiarazioni compaiono nell'output.

Se si tenta di inserire un elemento già definito in precedenza (contenuto nella lista) l'oggetto lancia un'eccezione per segnalare la violazione del vincolo di unicità per gli elementi di una DTD.

Su questa classe sono definite varie operazioni tra cui l'aggiunta di una qualunque dichiarazione alla DTD e il controllo se un elemento specificato è già stato definito.

Il metodo `toString()` ritorna la conversione in stringa dell'intera DTD, utile quando si deve produrre un file di testo o inviarla su un generico stream.

DTDObject

Questa classe descrive una generica dichiarazione che può essere inclusa in una DTD. Ogni discendente fornisce un proprio metodo di conversione in stringa in modo che un oggetto DTD possa produrre l'output, sfruttando la natura polimorfica di questa classe.

Discendenti di DTDObject

DTDEntity produce dichiarazioni di entity nella forma:

```
<!ENTITY entity_name SYSTEM "entity_content">
```

La forma con la clausola PUBLIC non è attualmente supportata da DTDTools.

`DTDComment` descrive dei commenti mentre `DTDConditional` descrive sezioni condizionali della DTD e richiede di specificare il nome di un entità che servirà da selettore.

DTDElement

Questa classe descrive le dichiarazioni di elementi e relative liste di attributi. In DTDTools si è separato il concetto di dichiarazione di un elemento dalla descrizione del suo contenuto.

Ad un oggetto `DTDElement` è sempre associato un oggetto `ContentModel` che ne definisce sia il contenuto sia la lista di attributi associata all'elemento.

Si è scelto di operare in questa maniera per permettere una facile clonazione del contenuto di un elemento (si pensi alla traduzione dei tipi ODL_I^3) e lo stesso contenuto può essere condiviso da più elementi.

`DTDElement` fornisce una serie di metodi di per fondere altri content model in *and* o in *union* con il contenuto corrente. La fusione genera l'unione degli attributi dei due contenuti, `DTDElement` vigila affinché l'unione abbia senso e non vi siano attributi duplicati.

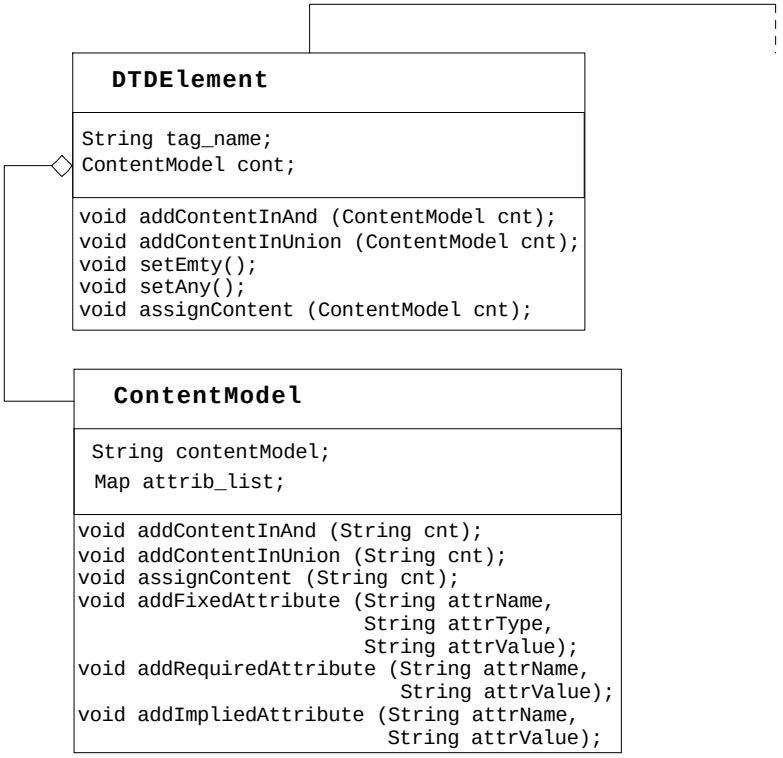

Figura 17: le classi DTDElement e ContentModel

ContentModel

Questa classe descrive il contenuto di una dichiarazione di un elemento della DTD. In ContentModel sono presenti due metodi `addContentInAnd()` e `addContentInUnion()` analoghi a quelli introdotti da DTDElement. I due metodi servono per aggiungere elementi figli in and o in union, la differenza rispetto a DTDElement è che questi metodi lavorano più a basso livello: manipolano delle stringhe e gestiscono in automatico l'aggiunta di parentesi.

Un'altra categoria di operazioni sono quelle legate all'aggiunta di vari tipi di attributi. Ognuno dei metodi riportati in Figura 17 causa l'aggiunta di un attributo ad esempio:

```
addFixedAttribute:

   <!ATTRLIST . . . .
     attrName     attrType    #FIXED “attrvalue”
   >
```

`attrType` può assumere il valore di una delle costanti dichiarate nell'interfaccia `DTDConstants` (“CDATA” “ID” “IDREF” ecc.) oppure dichiarare una lista di possibili valori per l'attributo, come ad esempio “`rosso |verde |blu`”.

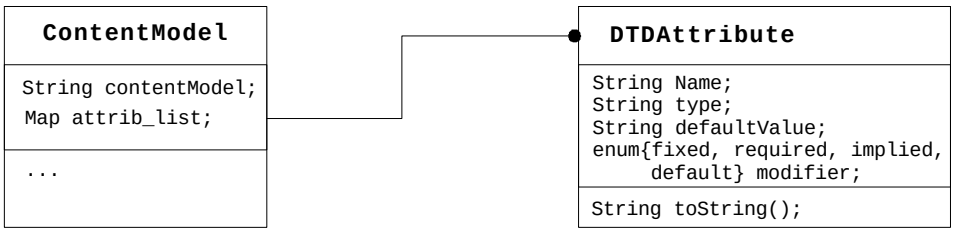

Figura 18: Dettaglio della classe ContentModel

La dichiarazione di attributi XML è affidata alla classe `DTDAttribute`.

5.1.3 Il traduttore

Per realizzare il traduttore è stata definita la seguente interfaccia che le varie classi che descrivono le entità ODL_I^3 da tradurre devono implementare[12].

[12] Si ricorda che Java non supporta l'ereditarietà multipla ma consente di dichiarare l'implementazione di più interfacce.

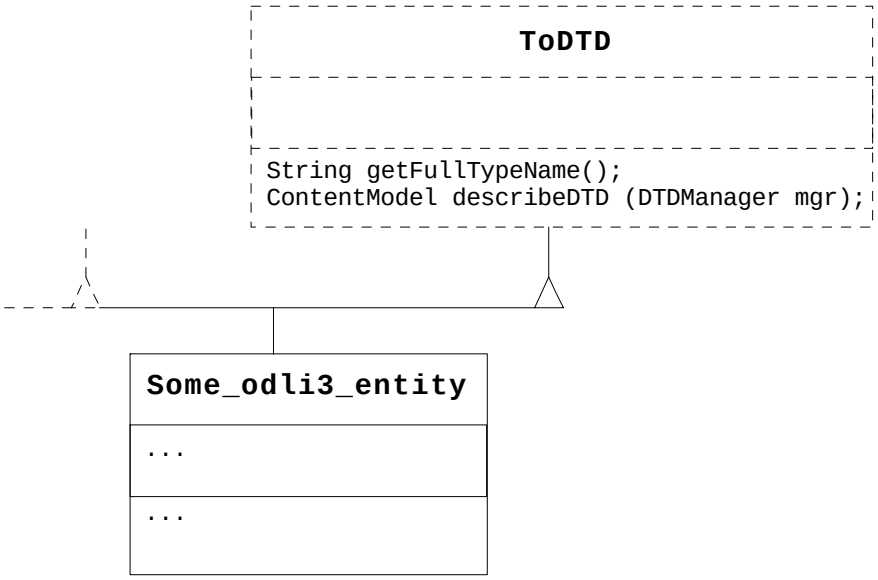

Figura 19: interfaccia ToDTD

L'interfaccia definisce due soli metodi:

getFullTypeName

ritorna una stringa contenente il nome che l'entità ODL_I^3 consiglia di utilizzare per definire l'elemento DTD che conterrà la traduzione. Una classe locale di ODL_I^3 ad esempio ritornerà il suo nome completo

```
sorgente.nome_classe
```

describeDTD

Questo metodo deve generare la traduzione vera e propria dell'entità e ritornare un'istanza di ContentModel. Si noti che l'unico parametro passato è un'istanza del DTD manager a cui andranno comunicati i risultati della traduzione.

ODL_I^3 è supportato a livello software da una complessa gerarchia di classi Java che ne rispecchiano fedelmente tutte le caratteristiche. Sono due i moduli già esistenti che sono stati interessati dall'implementazione del traduttore: il modulo `odli3` e il modulo `globalschema`.

Il primo modulo supporta tutte le principali caratteristiche del linguaggio: classi locali, attributi, tipi ecc., mentre il secondo fornisce il supporto per le classi globali e la mapping table.

L'interfaccia ToDTD è stata implementata in molte classi, di seguito viene descritto il compito dei vari comparti del traduttore. Per orientarsi nella

trattazione che segue fare riferimento alla gerarchia di classi di Figura 20 e Figura 21.

Schema

La classe schema (non riportata nelle figure) rappresenta la radice del grafo di oggetti rappresentante lo schema ODL_I^3 e rappresenta anche il punto di partenza del processo di traduzione.

Questa classe crea una nuova istanza del DTD manager e invoca la traduzione per tutte le classi `Source` rappresentanti le sorgenti locali. Una volta che tutte le classi e attributi locali sono stati correttamente tradotti si passa alla traduzione dello schema globale.

Per ogni classe globale viene definito un elemento DTD con il nome suggerito dalla classe stessa, quindi viene invocata la traduzione affinché ogni classe possa definire i propri attributi globali.

Source

Per prima cosa viene definito l'elemento XML `odli3.source`, relativo alla sorgente corrente, all'interno del meta elemento. Quindi per ogni classe locale:

?? viene dichiarato un elemento DTD utilizzando il suo nome esteso.

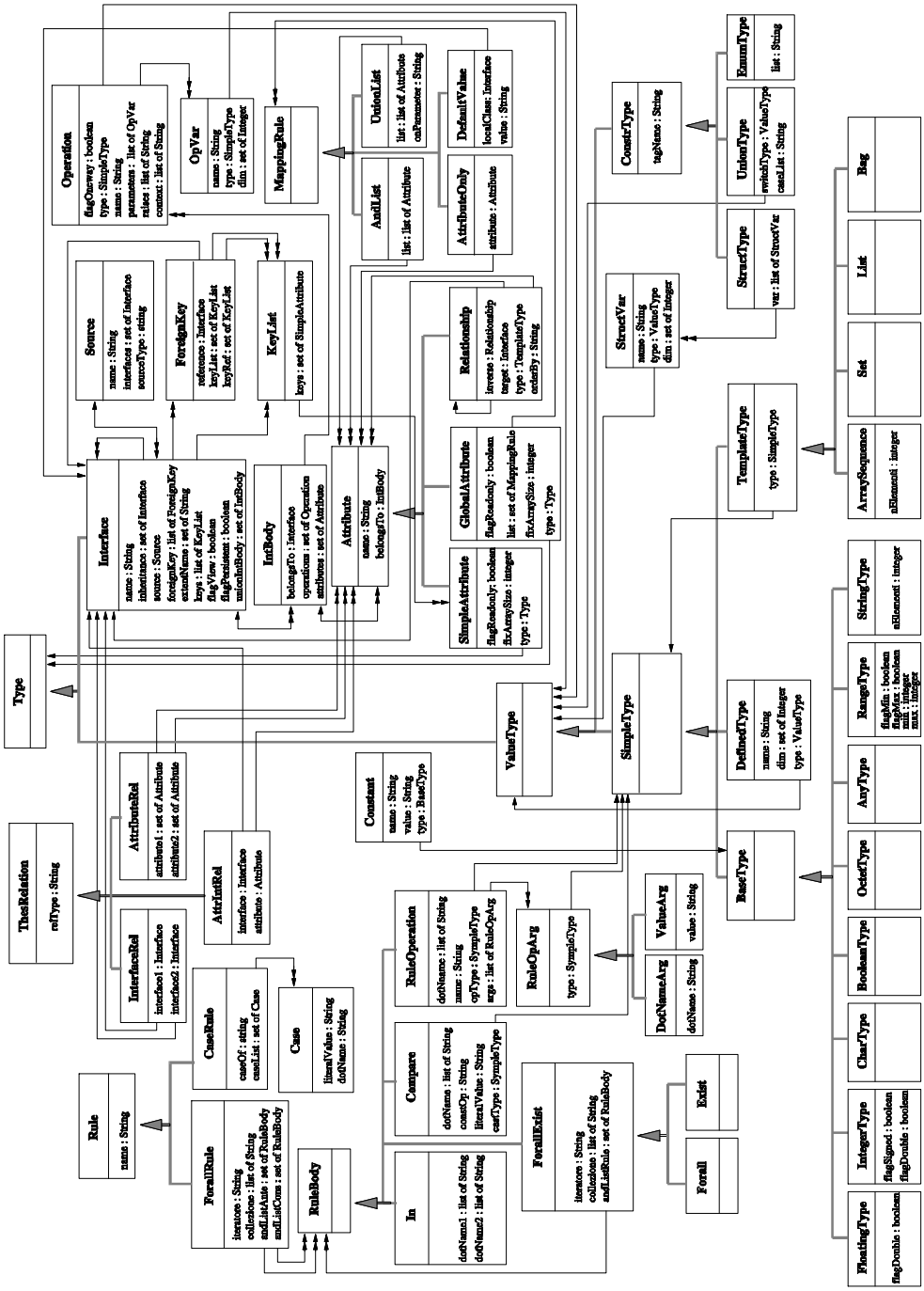

Figura 20: gerarchia di classi per il linguaggio ODL_I^3

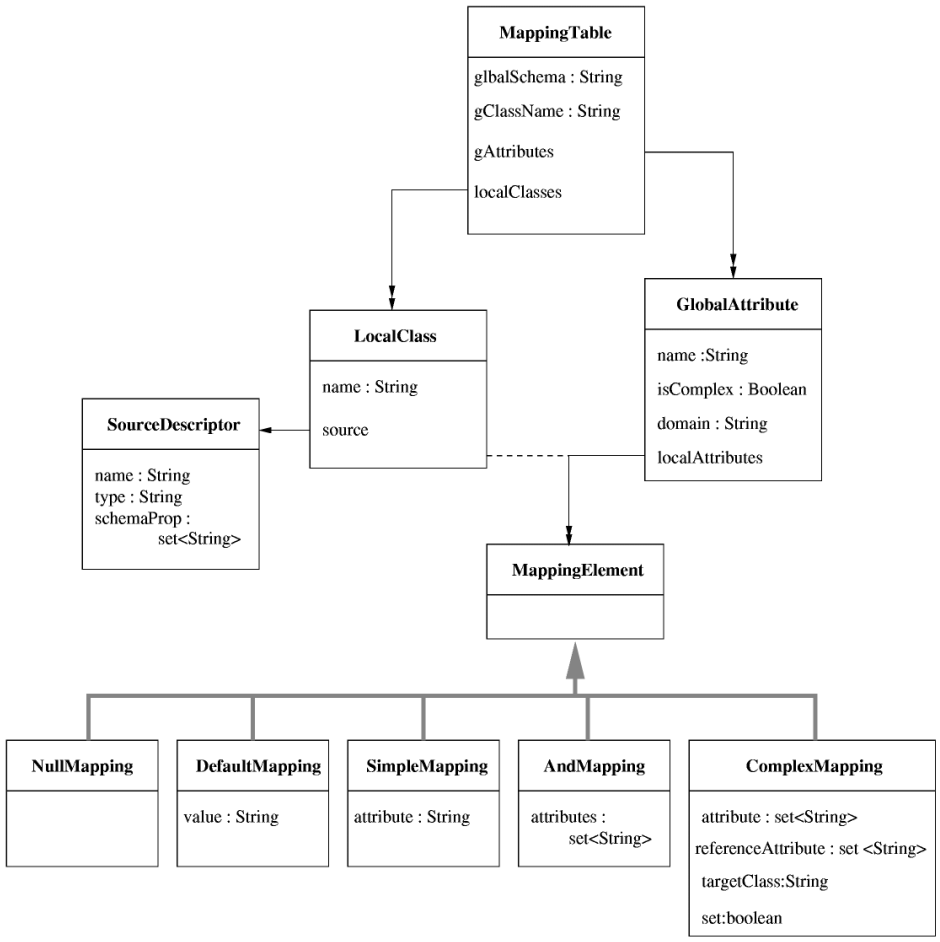

Figura 21: gerarchia di classi della mapping table

?? Viene invocata la sua traduzione in modo che le varie definizioni in union tra loro siano inserite nella DTD.

Interface

Questa classe descrive le classi locali di ODL_I^3, il processo di traduzione comincia con la creazione delle meta informazioni relative alla classe tra cui le chiavi e le foreign key.

Il contenuto di un elemento classe viene ottenuto concatenando l'elemento che traduce l'immediata superclasse l'*union* del contenuto dei vari interface body. Il contenuto di ognuno di questi è ottenuto concatenando in *and* il nome degli attributi contenuti da ciascun body.

Il nome XML di ogni attributo ODL_I^3 è ottenuto con l'ausilio del DTD manager, per ogni attributo è invocata la sua traduzione per dare modo ai vari tipi dichiarati nello schema di venire tradotti nella DTD.

Attribute

Questa classe implementa l'interfaccia `ToDTD` per conferire caratteristiche polimorfiche ai suoi discendenti. I metodi di questa classe tuttavia non fanno nulla.

SimpleAttribute

Viene invocato il DTD manager affinché fornisca la traduzione del tipo dell'attributo prelevandola dalla sua cache interna. Se la traduzione del tipo dell'attributo non è ancora stata effettuata, il DTD manager invoca ricorsivamente il comparto del traduttore responsabile per la traduzione del tipo ed aggiorna la cache interna.

Un elemento `<odli3.attribute>` è aggiunto alle meta informazioni nella sezione relativa alla classe corrente.

Relationship

Il contenuto della traduzione di una relatioship è ottenuto seguendo le indicazioni viste nel paragrafo 4.4.1, il DTD manager è invocato per ottenere la lista dei discendenti da porre in alternativa. Viene controllato inoltre se la relationship è semplice o utilizza un qualche template type.

Un elemento `<odli3.relationship>` è aggiunto alle meta informazioni nella sezione relativa alla classe corrente.

Tipi ODL_I^3

Tutti i tipi ODL_I^3 implementano l'interfaccia `ToDTD` in modo da poter realizzare una traduzione polimorfica dei tipi all'interno degli attributi.

Strutture

Per ogni variabile che compare nella struttura viene invocata la sua traduzione. La definizione DTD di una `StructVar` avviene in maniera analoga agli attributi semplici.

Classi globali

La traduzione avviene in maniera analoga alle classi locali. Per ogni attributo globale viene definito un elemento DTD il cui contenuto è ottenuto invocando la traduzione della mapping table.

Mapping table

Ogni classe MappingElement implementa l'interfaccia ToDTD, la traduzione della mapping table è ottenuta invocando il metodo di traduzione per ogni entry della tabella.

Il contenuto di un attributo globale è ottenuto mettendo in *union* tra loro le traduzioni dei vari MappingElement.

Il codice di MOMIS che implementa la mapping table mantiene solo riferimenti simbolici con le entità ODL_I^3 coinvolte: Il tipo degli attributi globali e gli attributi locali sono riportati per nome.

Questo fatto rappresenta una complicazione: non avendo a disposizione dei riferimenti diretti alle istanze che descrivono le entità ODL_I^3 non si è in grado di recuperare il nome utilizzato dal traduttore per i vari attributi locali.

Per risolvere questo problema il DTD manager mantiene una speciale struttura dati che permette di ottenere il nome cercato partendo dalle informazioni contenute nella mapping table.

5.1.4 Il DTD manager

Il DTD manager è una classe il cui compito è fornire un'interfaccia tra il traduttore e le classi per la descrizione di una DTD. Più in dettaglio i compiti assegnati al DTD manager sono:

- ?? coordinare il processo di traduzione
- ?? fornire varie informazioni al codice di controllo del traduttore
- ?? raccogliere e raggruppare opportunamente l'output del traduttore per produrre una DTD leggibile.
- ?? Cache dei tipi tradotti.
- ?? Implementa il meccanismo di renaming degli identificatori omonimi.

Output

Il compito primario del manager è quello di raccogliere l'output dei vari comparti del traduttore e fonderlo in una DTD organica e ordinata. Il manager quindi è la classe con cui i moduli di MOMIS devono dialogare per ottenere la DTD di uno schema ODL_I^3 o l'elemento XML contenente le meta informazioni non tradotte direttamente.

Per ordinare l'output il manager permette la definizione di "sezioni" della DTD, ossia delle aree che consentano di raggruppare dichiarazioni DTD omogenee: ad esempio la traduzione del global virtual schema genera una sezione dove sono riportate in sequenza tutte e sole le classi globali.

Le sezioni sono definite dinamicamente dagli utenti del manager che le identificano attraverso dei nomi simbolici. Il manager mette a disposizione dei

metodi del tipo “scrivi la dichiarazione di questa classe locale, nella sezione “classi_locali_della_sorgente_X””.

La definizione di sezioni è necessaria in quanto, se si seguisse l’ordine con cui i vari comparti generano le loro traduzioni, si produrrebbero delle DTD illeggibili in cui le definizioni di elementi “classe” sono mischiate a quelle relative agli attributi e ai tipi ecc.

Utilizzando le sezioni si è immuni all’ordine con cui il traduttore genera le dichiarazioni DTD, quindi eventuali modifiche future al linguaggio e al traduttore non causeranno disordini nell’output.

Normalmente una DTD è ottenuta affiancando le dichiarazioni contenute nelle varie sezioni; le sezioni sono riportate in base all’ordine alfabetico del loro nome.

Il manager mantiene un riferimento anche a un DOM tree contenente le meta informazioni non direttamente tradotte e fornisce gli strumenti per editarlo in maniera semplice.

Cache dei tipi tradotti.

Per ogni tipo ODL_I^3 che deve essere tradotto il traduttore genera un `type identifier` (type_id), una stringa che identifica univocamente il tipo in questione. Spesso l’identificatore di tipo coincide con il nome del tipo ma esistono molte eccezioni, ad esempio i template type riportano oltre al tipo di template (set, list, bag, array) anche il tipo dei componenti.

Il manager gestisce una cache, una struttura dati che associa ai type_id le relative traduzioni sotto forma di oggetti `ContentModel`.

Quando deve essere tradotto un tipo ODL_I^3, viene invocato il manager perché sovrintenda il processo: se non esiste ancora una traduzione per il tipo corrente allora il manager ne invoca la traduzione e memorizza il `ContentModel` ottenuto nella cache. Se invece il tipo è già stato tradotto allora il suo content model viene restituito dal manager immediatamente.

Gli utenti di questa funzione sono solitamente i comparti del traduttore relativi agli attributi di classi locali o ai campi di una struttura.

Generazione dei riferimenti a classi.

Il manager implementa l’algoritmo descritto in 4.4.1 che esamina la discendenza di una classe per trovare tutte le classi legate dalla relazione *isa*. I comparti del traduttore relativi alle relationship e agli attributi di tipo classe utilizzano questa funzione per generare le loro traduzioni.

Mapping tra nomi ODL_I^3 e nomi XML.

Spesso nei vari comparti del traduttore si ha la necessità di conoscere quale elemento DTD effettua la traduzione di un attributo ODL_I^3 di cui però è noto solo il nome e il tipo in forma simbolica.

Il manager implementa quindi una struttura dati che date queste informazioni simboliche è in grado di restituire il nome o l'istanza di DTDElement corrispondente alla traduzione dell'attributo cercato.

La struttura è stata realizzata a partire da un requisito introdotto dal linguaggio ODL_I^3: per identificare univocamente un attributo locale occorre conoscere le seguenti informazioni simboliche:

- Nome della sorgente
- Nome della classe locale
- Nome semplice dell'attributo
- Nome del tipo dell'attributo

L'informazione sul tipo è necessaria in quanto una classe locale può avere più definizioni dello stesso attributo che si differenziano solo dal tipo.

La struttura realizzata è una doppia mappa: la prima mappa associa al nome esteso di un attributo (`sorgente.classe_locale.nome_attributo`) una seconda mappa.

Nella seconda mappa ad ogni identificatore di tipo è associato l'elemento DTD (un'istanza di `DTDElement`) corrispondente alla traduzione dell'attributo.

Ai vari comparti del traduttore non è richiesta la conoscenza nei dettagli di questa struttura, il manager implementa tutte le operazioni di aggiornamento e ricerca sulla struttura fornendo una semplice interfaccia.

Questa funzione è utilizzata per la traduzione degli attributi globali e per determinare se un determinato attributo locale è già stato tradotto o meno.

Una classe ODL_I^3 avente molteplici definizioni può riportare più volte gli stessi attributi, ad esempio:

```
interface UN_Person (
   . . .){
   attribute string first_name;
   attribute string last_name;
   attribute string address;
} union second_definition {
  typedef struct location {
                            int number;
                            string street;
                            string city;
                           } location;

   attribute string first_name;
   attribute string last_name;
   attribute location address;};
```

Quando si giunge alla traduzione del secondo corpo di interfaccia gli attributi `first_name` e `last_name` sono già stati tradotti in quanto coincidono con

quelli del primo corpo. L'attributo address invece è l'unico che deve essere tradotto con una definizione differente.

Utilizzando la struttura descritta, il manager è in grado di segnalare al traduttore se un attributo necessita di una nuova traduzione o se è possibile riutilizzare una di quelle già prodotte.

5.2 Output del traduttore

Il traduttore ha come output la produzione di due file: la DTD corrispondente allo schema ODL_I^3 e l'elemento contenente le meta informazioni. A fianco di questi file vi è la DTD per le meta informazioni che però non varia da schema a schema.

Per semplificare l'utilizzo del traduttore è stata realizzata una semplice interfaccia utente che consente di produrre la DTD e le meta informazioni o di tutto lo schema ODL_I^3 o solamente per una sorgente specificata.

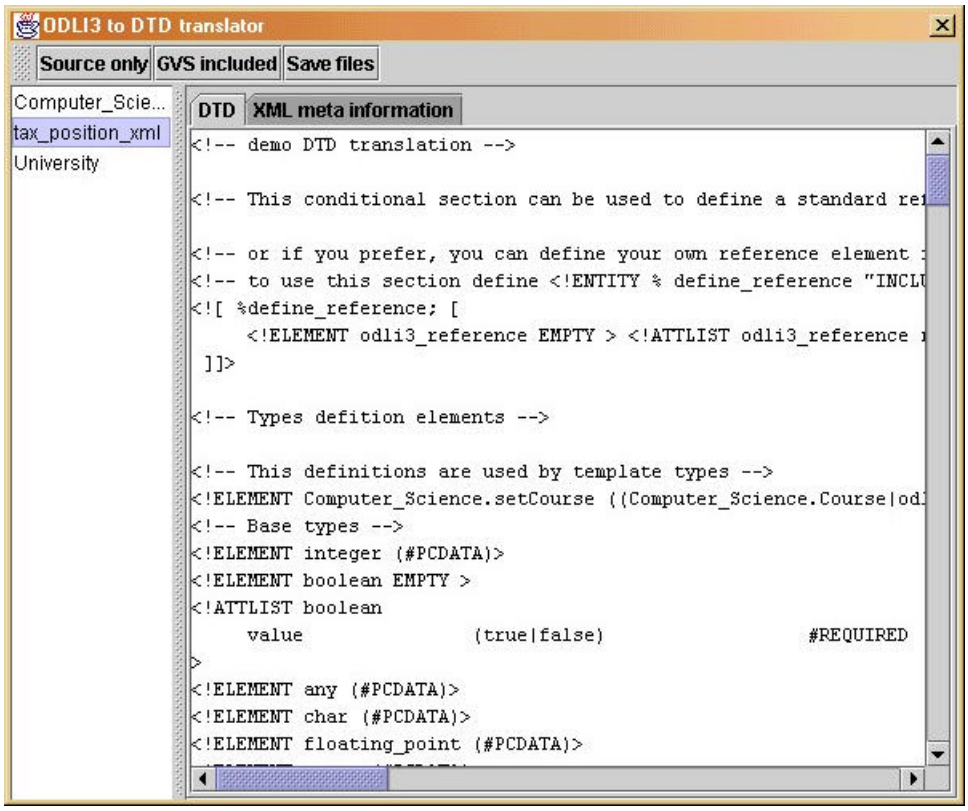

Figura 22: GUI del traduttore

Oltre all'interfaccia grafica esiste anche una versione del traduttore a riga di comando integrata nel parser ODL_I^3 di MOMIS.

5.2.1 Esempi di traduzione.

Traduzione di un interfaccia locale

Si consideri la seguente interfaccia ODL_I^3:

```
interface Department
 ( source relational University
   extent Department
   key (dept_code) )
{       attribute string dept_name;
        attribute integer dept_code;
        attribute integer budget;
        attribute string location; }
union second_def {
        typedef enum loc_tp {street, square, rue, road,
                             avenue} loc_tp;
        typedef struct address {
          int number;
          string loc_name;
          loc_tp loc_type;
          string city;
        } address;

        attribute string dept_name;
        attribute integer dept_code;
        attribute integer budget;
        attribute address location;
};
```

L'interfaccia viene tradotta definendo l'elemento:

```
<!ELEMENT University.Department
        ((University.Department.dept_code,
          budget,location,dept_name) |
         (University.Department.dept_code,
          budget,University.Department.location,
          dept_name))>
<!ATTLIST University.Department
  view         (true|false)         #FIXED "fa lse"
  persistent   (true|false)         #FIXED "true"
  id_xml           ID               #REQUIRED
>
```

Gli attributi vengono definiti successivamente:

```
<!-- Attributes definition specific for interface
     University.Department -->
<!-- Definition for body: second_def -->
<!ELEMENT locations ((University.address)*)>
<!ELEMENT University.Department.dept_code (#PCDATA)>
<!ELEMENT budget (#PCDATA)>
<!ELEMENT dept_name (#PCDATA)>

<!-- Definition for body: Department -->
<!ELEMENT location (loc_name,loc_type,number,city)>

<!-- Struct variables definition -->
<!ELEMENT loc_name (#PCDATA)>
<!ELEMENT loc_type EMPTY >
<!ATTLIST loc_type
  value   (street|square|rue|road|avenue) #REQUIRED
>
<!ELEMENT number (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

Riferimenti ad oggetti:

Come esempio di riferimenti ad oggetti si considerino le seguenti interfacce, gli attributi in grassetto sono quelli a cui prestare maggior attenzione:

```
interface CS_Person
 ( source object Computer_Science
   extent CS_Persons
   key (first_name, last_name) )
{  attribute string first_name;
   attribute string last_name;  };

interface Professor : CS_Person
 ( source object Computer_Science
   extent Professors )
{  attribute Office belongs_to;
   attribute string rank;     };

interface Employee
 ( source object Computer_Science)
{
   attribute string employement_date;
   attribute CS_Person referent;};

interface Spec_Professor : Employee, Professor
 ( source object Computer_Science
   extent Professors )
{  attribute int salary;};
```

```
interface Student : CS_Person
 ( source object Computer_Science
   extent Students )
{  attribute integer year;
   attribute set<Course> takes;
   attribute string rank;     };

interface Course
 ( source object Computer_Science
   extent Courses
   key (course_name) )
{  attribute string course_name;
   attribute Professor taught_by;  };
[...]
```

Le dichiarazioni DTD degli attributi di queste classi sono le seguenti:

```
[...]
<!-- Attributes definition specific for interface
     Computer_Science.Employee -->

<!ELEMENT employement_date (#PCDATA)>
<!ELEMENT referent (Computer_Science.CS_Person |
                    Computer_Science.Professor |
                    Computer_Science.Spec_Professor |
                    Computer_Science.Student |
                    odli3_reference)>
```

l'attributo `referent` è di tipo `CS_Person` quindi può avere come valore un'istanza di qualunque classe discendente da `CS_Person`.

```
<!-- Attributes definition specific for interface
     Computer_Science.Course -->

<!ELEMENT course_name (#PCDATA)>
<!ELEMENT taught_by (Computer_Science.Professor |
                     Computer_Science.Spec_Professor |
                     odli3_reference)>
```

l'attributo taught_by è di tipo professor, l'algoritmo di esplorazione ha scartato quindi le classi `Student` e `CS_Person`.

Attributi e classi globali

Si consideri la seguente interfaccia globale:

```
interface Person {
   attribute complex belongs_to
```

```
      mapping_rule (Computer_Science.Professor.belongs_to)

   attribute integer dept_code
     mapping_rule (University.Research_Staff.dept_code)

   attribute string e_mail
     mapping_rule (University.Research_Staff.e_mail)

   attribute string full_name
     mapping_rule (Computer_Science.CS_Person.last_name
                AND Computer_Science.CS_Person.first_name),
                   (Computer_Science.Professor.last_name
                AND Computer_Science.Professor.first_name),
                   (Computer_Science.Student.last_name AND
                    Computer_Science.Student.first_name),
                    University.Research_Staff.name,
                    University.School_Member.name,
                    tax_position_xml.Student.name;

   attribute string pers_faculty
      mapping_rule University.School_Member.faculty,
                   tax_position_xml.Student.faculty_name;

   attribute string pers_rank
      mapping_rule Computer_Science.Professor.rank,
                   Computer_Science.Student.rank;

   attribute integer pers_year
      mapping_rule Computer_Science.Student.year,
                   University.School_Member.year;

   attribute integer section_code
      mapping_rule University.Research_Staff.section_code;

   attribute string student_code
      mapping_rule tax_position_xml.Student.student_code;

   attribute array takes;
      mapping_rule Computer_Science.Student.takes;

   attribute string tax_fee
      mapping_rule tax_position_xml.Student.tax_fee;

}
```

La traduzione della classe globale è analoga a quella di una classe locale, tuttavia il contenuto degli attributi globali è ottenuto analizzando la mapping table. Si nota che gli attributi globali contengono, composti in varia maniera, le strutture che definiscono gli attributi locali su cui mappano.

```
<!-- attribute definitions for global class: Person -->

<!ELEMENT Person.belongs_to (belongs_to)>
<!ELEMENT Person.dept_code (dept_code)>
<!ELEMENT Person.e_mail (e_mail)>
<!ELEMENT Person.full_name ((last_name,first_name) |
                            University.Research_Staff.name |
                            University.School_Member.name |
                            name)>
```

In questo attributo si nota che le classi locali `Professor` e `Student` non ridefiniscono la struttura di last_name e first_name introdotta da `CS_person`, pertanto il traduttore riutilizza le definizioni introdotte nella traduzione di `CS_Person`. Ogni elemento che traduce `Professor` o `Student` infatti, conterrà un elemento `CS_Person` per rispecchiare il vincolo di parentela con quella classe.

```
<!ELEMENT Person.pers_faculty (faculty|faculty_name)>
<!ELEMENT Person.pers_rank (Computer_Science.Professor.rank
                            |rank)>
<!ELEMENT Person.pers_year (year |
                            University.School_Member.year)>
<!ELEMENT Person.section_code (section_code)>
<!ELEMENT Person.student_code (student_code)>
<!ELEMENT Person.takes (takes)>
<!ELEMENT Person.tax_fee (tax_fee)>
```

5.2.2 Possibile implementazione del documento istanza.

Supponendo di disporre dei file:

- ?? demo.dtd – contenente la traduzione dello schema ODL_I^3
- ?? demo_meta.xml – contenente il frammento XML dell'elemento con le meta informazioni `<odli3.meta>`
- ?? metainfo.dtd – la DTD delle meta informazioni (si veda l'Appendice C)

Un possibile documento istanza prodotto in risposta ad una interrogazione dell'utente potrebbe essere il seguente:

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

<!-- Declares the DTD reference -->
<!DOCTYPE query_res SYSTEM "demo.dtd"[
   <!-- Includes the default reference definition -->
   <!ENTITY % define_reference "INCLUDE">

   <!-- Defines the instance document structure -->
```

```
   <!ELEMENT query_res (odli3_meta, (Office.address,
             Department.dept_name, Department.dept_code)*)>

   <!-- Includes the meta info DTD -->
   <!ENTITY % meta_def PUBLIC
     "http://dbgroup.dsi.unimo.it/some_path/metainfo. dtd">
   %meta_def;

   <!-- Includes the meta info element -->
   <!ENTITY metainfo SYSTEM "demo_meta.xml">
]>

<query_res>
  <!-- import meta information -->
  &metainfo;
  <Office.address>
     <address>
       <Computer_Science.Location>
          <number> 221</number>
          <city> London </city>
          <county> Great Britain</country>
          <street> Backer street</street>
       </Computer_Science.Location>
     </address>
  </Office.address>
  <Department.dept_name>
    <dept_name>Computer Science 2001</dept_name>
  </Department.dept_name>
  <Department.dept_code>
    <University.Department.dept_code>CS00023
    </University.Department.dept_code>
  </Department.dept_code>

  <!-- Etc -->

</query_res>
```

In questo esempio, il documento istanza non ridefinisce l'elemento `<odli3.reference>` per i riferimenti ad altre classi ed include le meta informazioni.

Conclusioni

Durante lo svolgimento della tesi sono stati studiati, progettati ed implementati due moduli software:

- ?? XmlTools
- ?? Traduttore ODL_I^3 – DTD di XML 1.0

XmlTools

L'impiego di XML come formato di serializzazione per oggetti e classi Java presenta vantaggi e svantaggi: da un lato si facilita lo scambio dei dati contenuti nelle classi anche tra ambienti eterogenei, dall'altro si ha un notevole overhead in termini di ridondanza e potenza di calcolo per la produzione o l'utilizzo del documento XML.

La ridondanza di un grafo di oggetti serializzato in XML diventa un problema se lo si deve inviare attraverso una rete geografica, dove la larghezza di banda è una risorsa preziosa. Per risolvere questo inconveniente si può applicare all'output di XmlTools un algoritmo di compressione.

Esistono algoritmi molto efficienti per comprimere un file XML: nel caso di XmlTools la dimensione del documento prodotto è determinata principalmente dal gran numero di elementi innestati, piuttosto che da informazioni testuali o numeriche. Si potrà quindi pensare, come lavoro futuro, di studiare vari metodi per ridurre la ridondanza e permettere la trasmissione di documenti XML anche di grandi dimensioni. Ad esempio, si potrà applicare all'output di XmlTools un algoritmo sul modello seguente:

- ?? associare ad ogni stringa un identificatore numerico: nomi di tag, di attributo, nomi delle classi Java ecc.
- ?? costruire una tabella nomi – identificatori numerici
- ?? produrre uno stream binario che descriva il documento XML in base agli identificatori prodotti.

L'algoritmo di compressione potrebbe essere agevolmente integrato nella classe `XmlWriter` sfruttando le sue strutture dati. Un algoritmo inverso dovrebbe essere implementato in XmlReader: ricostruire il DOM tree partendo

dallo stream binario e quindi procedere normalmente alla ricostruzione delle classi.

Traduzione di schemi ODL_I^3

In questa tesi si sono poste le basi per consentire al futuro query manager di poter esportare i risultati delle query globali in formato XML. Tuttavia molta della conoscenza e delle informazioni semantiche contenute in uno schema ODL_I^3 vengono perse dal processo di traduzione. Lo speciale elemento XML contenente le metainformazioni può essere utile solamente nella misura in cui l'utilizzatore esterno della DTD è in grado di interpretarlo.

Inoltre i tipi di ODL_I^3 e la gerarchia di classi rappresentata nello schema, vengono riportati solo in maniera implicita dalla DTD (si pensi agli espedienti utilizzati per tradurre l'ereditarietà o i riferimenti a classi). Partendo da questo lavoro di tesi, come possibile lavoro futuro si può integrare XML Schema in modo da produrre un traduttore in cui i tipi e la gerarchia di classi saranno riportate fedelmente.

XML Schema è in grado di supportare efficacemente ereditarietà, chiavi primarie e candidate, foreign key, in modo da produrre uno schema XML più simile all'originale rispetto a quello attualmente prodotto dal traduttore in DTD.

Anche utilizzando XML Schema però non tutta la conoscenza presente in uno schema ODL_I^3 potrà essere tradotta. Si pensi alle regole di integrità, alle relazioni intensionali ed estensionali.

In futuro si pensa di utilizzare XML Schema per descrivere schemi ODL_I^3 ai fini delle interrogazioni e utilizzare RDF (Resource Description Framework) e RDF schema per descrivere gli aspetti semantici di ODL_I^3.

RDF è una recommendation emanata dal W3C progettata per descrivere i metadati e facilitare il reperimento delle informazioni sul web. Il meccanismo introdotto da RDF si basa sulla definizione di proprietà, ossia tuple contenenti tre campi (Resource, PropertyType, Value).

Si dovrà dunque sviluppare un traduttore per XML Schema partendo dal lavoro di questa tesi e si dovranno esplorare le potenzialità di RDF per riuscire a trasferire in XML la conoscenza contenuta in uno schema ODL_I^3.

RDF gode delle proprietà:

- *Indipendenza* : Ogni organizzazione o individuo può definire le sue proprietà indipendentemente dalle altre.
- *Interscambio*: Le proprietà RDF sono espresse in XML, quindi sono facilmente intercambiabili.
- *Scalabilità*: ottenuta grazie all'utilizzo delle proprietà.

RDF schema invece è una proposta, che permette di definire regole di consistenza e informazioni addizionali che descrivono come le varie statement di RDF devono essere interpretate.

Utilizzando RDF sarà possibile gestire i vari aspetti semantici di ODL_I^3: le regole di integrità, gli extent ecc. Si potranno definire come risorse RDF lo schema, inteso come insieme delle interfaccie globali e locali, le interfaccie e le sorgenti. Per ognuna di queste si dovranno definire le proprietà che le caratterizzano.

Sia XML Schema che RDF Schema sono giunti allo stadio di "candidate recommendation" e hanno quindi raggiunto un certo grado di stabilità, tuttavia sino a quando non saranno emanate le rispettive "proposed recommendation" e "recommendation" non si potrà considerarli dei veri e propri standard.

Appendice A

Glossario

Le sezioni da 2 a 5 di questo glossario ed il vocabolario sul quale si basa sono stati originariamente sviluppati durante l'I^3 Architecture Meeting in Boulder CO, 1994, sponsorizzato dall'ARPA, e rifiniti in un secondo incontro presso l'Università di Stanford, nel 1995. Il glossario è strutturato logicamente in diverse sezioni:

Sezione 1: Termini specifici di questa tesi
Sezione 2: Architettura
Sezione 3: Servizi
Sezione 4: Risorse
Sezione 5: Ontologia

Nota: poiché la versione originaria del glossario usa una terminologia inglese, in alcuni casi è riportato, a fianco del termine, il corrispettivo inglese, quando la traduzione dal termine originale all'italiano poteva essere ambigua o poco efficace.

A.1 Termini specifici

?? Classe = Entità caratterizzata da un insieme di proprietà e di operazioni che agiscono sulle proprietà. Una classe cattura le caratteristiche generali di un'entità astratta o appartenente al mondo reale. In questa tesi si usa questo termine in due diversi contesti: classi Java e classi ODL_I^3 spesso riportate anche come interfacce ODL_I^3.

?? Mappa = struttura dati caratterizzata da un insieme di entità dette chiavi e da un insieme di valori. Ad ogni chiave è associato un unico valore. La mappa viene interrogata per chiave e restituisce il valore corrispondente. Esistono anche mappe in cui ad ogni chiave sono associati più valori.

?? RTTI (Run Time Type Information) = insieme di informazioni che descrivono i tipi e altre meta informazioni che il compilatore di un

linguaggio (ad es. il C++) rende disponibili nell'ambiente di esecuzione di un programma.

?? Identificatore di classe = in questa tesi, con questo termine ci si riferisce a classi Java. Un identificatore di classe è stringa che caratterizza completamente la classe Java cui si riferisce. Le informazioni riportate sono il tipo (primitivo o classe), uno o più flag di array e il nome della classe. Ad esempio: `[Ljava.lang.String;` è l'identificatore di classe per un array di oggetti stringa. Per una spiegazione dettagliata sul formato di questi identificatori si rimanda alla documentazione del linguaggio.

?? Object Pattern = Per maggiori dettagli si veda [31]. La definizione formale di object pattern è la seguente. Dato un insieme di oggetti set_l definito sulla sorgente semistrutturata S. L'object pattern dell'insieme set_l è una coppia nella forma $<l,A>$, dove l è l'etichetta degli oggetti correlati all'insieme set_l, ed A= label *so'* tale che esiste almeno un oggetto *so* ? set_l con *so* ? *so'*.

?? Recommendation (terminologia del W3C) = stadio finale dell'iter che una nuova proposta di standardizzazione deve seguire in ambito del W3C. Gli stadi intermedi sono i seguenti:

   ?? Working draft – documento che descrive lo stadio evolutivo di una nuova tecnologia, viene aggiornato ogni tre mesi

   ?? Last call working draft – ultimo working draft pubblicato per ricevere commenti dalla comunità degli sviluppatori.

   ?? Candidate recommendation – gli sviluppatori sono invitati ad implementarla e risolti i problemi implementativi si passa allo stadio di Proposed recommendation.

   ?? Recommendation – la tecnologia è abbastanza stabile da essere impiegata globalmente.

A.2 Architettura

?? Architettura = insieme di componenti.

?? architettura di riferimento = linea guida ed insieme di regole da seguire per l'architettura.

?? componente = uno dei blocchi sui quali si basa una applicazione o una configurazione. Incorpora strumenti e conoscenza specifica del dominio.

- ?? applicazione = configurazione persistente o transitoria dei componenti, rivolta a risolvere un problema del cliente, e che può coprire diversi domini.
- ?? configurazione = istanza particolare di una architettura per una applicazione o un cliente.
- ?? collante (glue) = software o regole che servono per collegare i componenti o per interoperare attraverso i domini.
- ?? strato = grossolana categorizzazione dei componenti e degli strumenti in una configurazione. L'architettura I^3 distingue tre strati, ognuno dei quali fornisce una diversa categoria di servizi:
  1. Servizi di Coordinamento = coprono le fasi di scoperta delle risorse, distribuzione delle risorse, invocazione, scheduling
  2. Servizi di Mediazione = coprono la fase di query processing e di trattamento dei risultati, nonché il filtraggio dei dati, la generazione di nuove informazioni, etc.
  3. Servizi di Wrapping = servono per l'utilizzo dei wrappers e degli altri strumenti simili utilizzati per adattarsi a standards di accesso ai dati e alle convenzioni adoperate per la mediazione e per il coordinamento.

?? agente = strumento che realizza un servizio, sia per il suo proprietario, sia per un cliente del suo proprietario.

?? facilitatore = componente che fornisce i servizi di coordinamento, come pure l'instradamento delle interrogazioni del cliente.

?? mediatore = componente che fornisce i servizi di mediazione e che provvede a dare valore aggiunto alle informazioni che sono trasmesse al cliente in risposta ad una interrogazione.

?? cliente (customer) = proprietario dell'applicazione che gestisce le interrogazioni, o utente finale, che usufruisce dei servizi.

?? risorsa = base di dati accessibile, server ad oggetti, base di conoscenze…

?? contenuto = risultato informativo ricavato da una sorgente.

?? servizio = funzione fornita da uno strumento in un componente e diretta ad un cliente, direttamente od indirettamente.

?? strumento (tool) = programma software che realizza un servizio, tipicamente indipendentemente dal dominio.

?? wrapper = strumento utilizzato per accedere alle risorse conosciute, e per tradurre i suoi oggetti.

?? regole limitative (constraint rules) = definizione di regole per l'assegnamento di componenti o di protocolli a determinati strati.

?? interoperare = combinare sorgenti e domini multipli.

?? informazione = dato utile ad un cliente.

?? informazione azionabile = informazione che forza il cliente ad iniziare un evento.

?? dato = registrazione di un fatto.

?? testo = dato, informazione o conoscenza in un formato relativamente non strutturato, basato sui caratteri.

?? conoscenza = metadata, relazione tra termini, paradigmi…, utili per trasformare i dati in informazioni.

?? dominio = area, argomento, caratterizzato da una semantica interna, per esempio la finanza, o i componenti elettronici…

?? metadata = informazione descrittiva relativa ai dati di una risorsa, compresi il dominio, proprietà, le restrizioni, il modello di dati, …

?? metaconoscenza = informazione descrittiva relativa alla conoscenza in una risorsa, includendo l'ontologia, la rappresentazione…

?? metainformazioni = informazione descrittiva sui servizi, sulle capacità, sui costi

A.3 Servizi

?? Servizio = funzionalità fornita da uno o più componenti, diretta ad un cliente.

?? instradamento (routing) = servizio di coordinamento per localizzare ed invocare una risorsa o un servizio di mediazione, o per creare una configurazione. Fa uso di un direttorio.

?? scheduling = servizio di coordinamento per determinare l'ordine di invocazione degli accessi e di altri servizi; fa spesso uso dei costi stimati.

?? accoppiamento (matchmaking) = servizio che accoppia i sottoscrittori di un servizio ai fornitori.

?? intermediazione (brokering) = servizio di coordinamento per localizzare le risorse migliori.

?? strumento di configurazione = programma usato nel coordinamento per aiutare a selezionare ed organizzare i componenti in una istanza particolare di una configurazione architetturale.

?? servizi di descrizione = metaservizi che informano i clienti sui servizi, risorse…

?? direttorio = servizio per localizzare e contattare le risorse disponibili, come le pagine gialle, pagine bianche…

?? decomposizione dell'interrogazione (query decomposition) = determina le interrogazioni da spedire alle risorse o ai servizi disponibili.

?? riformulazione dell'interrogazione (query reformulation) = programma per ottimizzare o rilassare le interrogazioni, tipicamente fa uso dello scheduling.

?? contenuto = risultato prodotto da una risorsa in risposta ad interrogazioni.

?? trattamento del contenuto (content processing) = servizio di mediazione che manipola i risultati ottenuti, tipicamente per incrementare il valore delle informazioni.

?? trattamento del testo = servizio di mediazione che opera sul testo per ricerca, correzione…

?? filtraggio = servizio di mediazione per aumentare la pertinenza delle informazioni ricevute in risposta ad interrogazioni.

?? classificazione (ranking) = servizio di mediazione per assegnare dei valori agli oggetti ritrovati.

?? spiegazione = servizio di mediazione per presentare i modelli ai clienti.

?? amministrazione del modello = servizio di mediazione per permettere al cliente ed al proprietario del mediatore di aggiornare il modello.

?? integrazione = servizio di mediazione che combina i contenuti ricevuti da una molteplicità di risorse, spesso eteorgenee.

?? accoppiamento temporale = servizio di mediazione per riconoscere e risolvere differenze nelle unità di misura temporali utilizzate dalle risorse.

?? accoppiamento spaziale = servizio di mediazione per riconoscere e risolvere differenze nelle unità di misura spaziali utilizzate dalle risorse.

?? ragionamento (reasoning) = metodologia usata da alcuni componenti o servizi per realizzare inferenze logiche.

?? browsing = servizio per permettere al cliente di spostarsi attraverso le risorse.

?? scoperta delle risorse = servizio che ricerca le risorse.

?? indicizzazione = creazione di una lista di oggetti (indice) per aumentare la velocità dei servizi di accesso.

?? analisi del contenuto = trattamento degli oggetti testuali per creare informazioni.

?? accesso = collegamento agli oggetti nelle risorse per realizzare interrogazioni, analisi o aggiornamenti.

?? ottimizzazione = processo di manipolazione o di riorganizzazione delle interrogazioni per ridurne il costo o il tempo di risposta.

?? rilassamento = servizio che fornisce un insieme di risposta maggiore rispetto a quello che l'interrogazione voleva selezionare.

?? astrazione = servizio per ridurre le dimensioni del contenuto portandolo ad un livello superiore.

?? pubblicità (advertising) = presentazione del modello di una risorsa o del mediatore ad un componente o ad un cliente.

?? sottoscrizione = richiesta di un componente o di un cliente di essere informato su un evento.

?? controllo (monitoring) = osservazione delle risorse o dei dati virtuali e creazione di impulsi da azionare ogniqualvolta avvenga un cambiamento di stato.

?? aggiornamento = trasmissione dei cambiamenti dei dati alle risorse.

?? instanziazione del mediatore = popolamento di uno strumento indipendente dal dominio con conoscenze dipendenti da un dominio.

?? attivo (activeness) = abilità di un impulso di reagire ad un evento.

?? servizio di transazione = servizio che assicura la consistenza temporale dei contenuti, realizzato attraverso l'amministrazione delle transazioni.

?? accertamento dell'impatto = servizio che riporta quali risorse saranno interessate dalle interrogazioni o dagli aggiornamenti.

?? stimatore = servizio di basso livello che stima i costi previsti e le prestazioni basandosi su un modello, o su statistiche.

?? caching = mantenere le informazioni memorizzate in un livello intermedio per migliorare le prestazioni.

?? traduzione = trasformazione dei dati nella forma e nella sintassi richiesta dal ricevente.

?? controllo della concorrenza = assicurazione del sincronismo degli aggiornamenti delle risorse, tipicamente assegnato al sistema che amministra le transazioni.

A.4 Risorse

?? Risorsa = base di dati accessibile, simulazione, base di conoscenza, comprese le risorse ”legacy”.

?? risorse ”legacy” = risorse preesistenti o autonome, non disegnate per interoperare con una architettura generale e flessibile.

?? evento = ragione per il cambiamento di stato all’interno di un componente o di una risorsa.

?? Oggetto = istanza particolare appartenente ad una risorsa, al modello del cliente, o ad un certo strumento.

?? valore = contenuto metrico presente nel modello del cliente, come qualità, rilevanza, costo.

?? proprietario = individuo o organizzazione che ha creato, o ha i diritti di un oggetto, e lo può sfruttare.

?? proprietario di un servizio = individuo o organizzazione responsabile di un servizio.

?? database = risorsa che comprende un insieme di dati con uno schema descrittivo.

?? warehouse = database che contiene o da accesso a dati selezionati, astratti e integrati da una molteplicità di sorgenti. Tipicamente ridondante rispetto alle sorgenti di dati.

?? base di conoscenza = risorsa comprendente un insieme di conoscenze trattabili in modo automatico, spesso nella forma di regole e di metadata; permettono l’accesso alle risorse.

?? simulazione = risorsa in grado di fare proiezioni future sui dati e generare nuove informazioni, basata su un modello.

?? amministrazione della transazione = assicurare che la consistenza temporale del database non sia compromessa dagli aggiornamenti.

?? impatto della transazione = riporta le risorse che sono state coinvolte in un aggiornamento.

?? schema = lista delle relazioni, degli attributi e, quando possibile, degli oggetti, delle regole, e dei metadata di un database. costituisce la base dell'ontologia della risorsa.

?? dizionario = lista dei termini, fa parte dell'ontologia.

?? modello del database = descrizione formalizzata della risorsa database, che include lo schema.

?? eterogeneità = incompatibilità trovate tra risorse e servizi sviluppati autonomamente, che vanno dalla piattaforma utilizzata, sistema operativo, modello dei dati, alla semantica, ontologia,…

?? costo = prezzo per fornire un servizio o un accesso ad un oggetto.

?? database deduttivo = database in grado di utilizzare regole logiche per trattare i dati.

?? regola = affermazione logica, unità della conoscenza trattabile in modo automatico.

?? sistema di amministrazione delle regole = software indipendente dal dominio che raccoglie, seleziona ed agisce sulle regole.

?? database attivo = database in grado di reagire a determinati eventi.

?? dato virtuale = dato rappresentato attraverso referenze e procedure.

?? stato = istanza o versione di una base di dati o informazioni.

?? cambiamento di stato = stato successivo ad una azione di aggiornamento, inserimento o cancellazione.

?? vista = sottoinsieme di un database, sottoposto a limiti, e ristrutturato.

?? server di oggetti = fornisce dati oggetto.

?? gerarchia = struttura di un modello che assegna ogni oggetto ad un livello, e definisce per ogni oggetto l'oggetto da cui deriva.

?? network = struttura di un modello che fa uso di relazioni relativamente libere tra oggetti.

?? ristrutturare = dare una struttura diversa ai dati seguendo un modello differente dall'originale.

?? livello = categorizzazione concettuale , dove gli oggetti di un livello inferiore dipendono da un antenato di livello superiore.

?? antenato (ancestor) = oggetto di livello superiore, dal quale derivano attributi ereditabili.

?? oggetto root = oggetto da cui tutti gli altri derivano, all'interno di una gerarchia.

?? datawarehouse = deposito di dati integrati provenienti da una molteplicità di risorse.

?? deposito di metadata = database che contiene metadata o metainformazioni.

A.5 Ontologia

?? Ontologia = descrizione particolareggiata di una concettualizzazione, i.e. l'insieme dei termini e delle relazioni usate in un dominio, per indicare oggetti e concetti, spesso ambigui tra domini diversi.

?? concetto = definisce una astrazione o una aggregazione di oggetti per il cliente.

?? semantico = che si riferisce al significato di un termine, espresso come un insieme di relazioni.

?? sintattico = che si riferisce al formato di un termine, espresso come un insieme di limitazioni.

?? classe = definisce metaconoscenze come metodi, attributi, ereditarietà, per gli oggetti in essa istanziati.

?? relazione = collegamento tra termini, come *is*-a, *part-o*f,…

?? ontologia unita (merged) = ontologia creata combinando diverse ontologie, ottenuta mettendole in relazione tra loro (mapping).

?? ontologia condivisa = sottoinsieme di diverse ontologie condiviso da una molteplicità di utenti.

?? comparatore di ontologie = strumento per determinare relazioni tra ontologie, utilizzato per determinare le regole necessarie per la loro integrazione.

?? mapping tra ontologie = trasformazione dei termini tra le ontologie, attraverso regole di accoppiamento, utilizzato per collegare utenti e risorse.

?? regole di accoppiamento (matching rules) = dichiarazioni per definire l'equivalenza tra termini di domini diversi.

?? trasformazione dello schema = adattamento dello schema ad un’altra ontologia.

?? editing = trattamento di un testo per assicurarne la conformità ad una ontologia.

?? algebra dell’ontologia = insieme delle operazioni per definire relazioni tra ontologie.

?? consistenza temporale = è raggiunta se tutti i dati si riferiscono alla stessa istanza temporale ed utilizzano la stessa granularità temporale.

?? specifico ad un dominio = relativo ad un singolo dominio, presuppone l’assenza di incompatibilità semantiche.

?? indipendente dal dominio = software, strumento o conoscenza globale applicabile ad una molteplicità di domini.

Appendice B

Protocollo di serializzazione XML

Nota: i nomi indicati con ***`questo stile`***, indicano un termine che può essere un qualunque identificatore XML.

Per semplificare la trattazione, nella definizione del protocollo di serializzazione sarà utilizzata una dichiarazione DTD *non rigorosa*, con l'aggiunta di commenti o ripetizioni dei nomi di elemento.

Definizione di alcune entità di utilità generale:

```
<!ENTITY % boolvalue "true|false">
<!ENTITY % primitive_type
"int|float|double|short|long|byte|char|boolean">
<!ENTITY % object_name
"object|short_class_name|class_name">
```

A causa delle varie funzionalità di personalizzazione presenti in XmlTools, non è possibile dare una DTD rigorosa che valga per qualunque documento prodotto da `XmlWriter`, qui i nomi ***`short_class_name`*** e ***`class_name`*** indicano rispettivamente un identificatore di classe normalizzato in forma semplice o estesa, ad esempio:

```
String
java.lang.String
```

"`object`" è il nome del tag che identifica la serializzazione di un oggetto nel caso non siano usati gli identificatori di classe come nomi di tag.

Quando si parla di identificatori di classe si intende la codifica utilizzata dal linguaggio Java per nominare una classe, ad esempio:

`[Ljava.lang.String;` è l'identificatore di classe per un array di stringhe, si veda la documentazione Java per i dettagli di questo tipo di codifica.

Radice

Un grafo di oggetti serializzato in XML con XmlTools ha la seguente struttura:

```
<!ELEMENT root_name (meta?,
(null|%object_name;|%primitive_type;
      | array)*)>
<!ATTLIST root_name
  ObjectArchive               (%boolvalue;) #REQUIRED
  ClassNameAsTagName          (%boolvalue;) #REQUIRED
  FullJavaClassName           (%boolvalue;) #REQUIRED
  FullClassNameAsAttribute    (%boolvalue;) #REQUIRED
  Balanced                    (%boolvalue;) #DEFAULT “false”
>
```

L’elemento “meta” è facoltativo e contiene la tabella degli SUID, come impostazione di default questo elemento è presente.

L’archivio è costituito da una sequenza di oggetti, array, valori null o tipi primitivi.

Attributi:

?? `ObjectArchive` – se vale true, il documento rappresenta un archivio di oggetti valido e può essere deserializzato.

?? `ClassNameAsTagName` – se vale true, gli identificatori di classe sono utilizzati come nomi di tag.

?? `FullJavaClassName` – se vale true, gli identificatori di classe sono in forma estesa, questo attributo viene considerato solo se `ClassNameAsTagName` vale true.

?? `FullClassNameAsAttribute` – se vale true gli elementi che contengono la serializzazione di un oggetto possiedono un attributo class con il rispettivo identificatore di classe in forma estesa.

?? `Balanced` – se vale true, il documento XML è stato bilanciato con l’algoritmo di `XmlWriter`.

SUID Table

La tabella degli SUID associa ad ogni identificatore di classe che compare nel documento il rispettivo SUID.

```
<!ELEMENT meta (suid.entry)*>
<!ATTLIST meta
  xmlns     CDATA #FIXED “momis.project.dsi.unimo.it”
>
<!ELEMENT suid.entry EMPTY>
```

```
<!ATTLIST suid.entry
  class     CDATA       #REQUIRED
  suid      CDATA       #REQUIRED
>
```

Ogni `suid.entry` ha due attributi,

?? `class` – identificatore di classe in forma estesa.

?? `suid` – intero in base 10 rappresentante lo suid.

Definizione di oggetti

```
<!ELEMENT %object_name; (object |%primitive_type;|
      end_default_data|superclass|array|null)*>
<!ATTLIST %object_name;
  field     CDATA #IMPLIED
  objectID  CDATA #REQUIRED
  class     CDATA #IMPLIED
>
```

Il nome di un tag rappresentante un oggetto può assumere il nome dell'identificatore di classe cui appartiene, sia in forma estesa che semplice, oppure si usa la stringa `object`.

Un oggetto contiene la lista dei valori dei suoi campi specifici e la definizione della sua immediata superclasse.

I valori dei campi possono essere oggetti, tipi primitivi, array o valori nulli.

Se una classe dichiara un metodo `writeObject()` personalizzato nel quale invoca `defaultWrite()` (per `XmlWriter`) o `defaultWriteObject()` (per uno dei wrapper), al termine della sequenza dei campi serializzati compare l'elemento `end_default_data` con il compito di indicare che da quel punto in poi seguono degli elementi XML definiti dall'utente.

```
<!ELEMENT end_default_data EMPTY>
<!ATTLIST end_default_data
 xmlns      CDATA #FIXED "momis.project.dsi.unimo.it"
>
```

Attributi:

?? `field` – contiene il nome di un campo serializzato. Se presente indica che l'oggetto in questione deve essere assegnato al campo indicato dal valore dell'attributo. Solitamente `field` compare negli oggetti innestati dentro altri oggetti ed è assente negli oggetti più vicini alla radice. Il valore di `field` si riferisce ai campi dell'oggetto che contiene questo elemento.

?? `objectID` – contiene il valore numerico in base 10 che identifica univocamente l'oggetto all'interno del documento.
?? `class` – se presente, contiene l'identificatore di classe in forma estesa cui appartiene l'oggetto serializzato. Se i nomi di classe non sono usati come nomi di tag, questo attributo deve essere presente se si desidera poter deserializzare il documento.

Definizione di superclassi

La definizione di una superclasse contiene a sua volta la definizione delle proprie superclassi. Il tag utilizzato per indicare una definizione di superclasse è il seguente:

```
<!ELEMENT superclass (object |%primitive_type;|
end_default_data|superclass|array|null)*>
<!ATTLIST superclass
  xmlns     CDATA #FIXED "momis.project.dsi.unimo.it"
  class     CDATA #REQUIRED
>
```

Ogni elemento `superclass` è relativo ad un'unica classe e viene scritto nel documento solamente se la superclasse in esame contiene dei campi serializzabili. Il contenuto di questo elemento è lo stesso che può comparire in una definizione di oggetto.

L'attributo `class` contiene l'identificatore di classe in forma estesa.

Riferimenti ad oggetti

Il nome di un tag che rappresenta un riferimento ad un oggetto, definito altrove nel documento, è il medesimo utilizzato nella definizione vera e propria. L'elemento deve essere vuoto. Se l'attributo `balanced` della radice è false allora è garantito che questi elementi compaiono solo dopo le rispettive definizioni.

```
<!ELEMENT %object_name; EMPTY>
<!ATTLIST %object_name;
  field     CDATA #IMPLIED
  objectREF CDATA #REQUIRED
>
```

Attributi:

?? `field` – contiene il nome di un campo serializzato. Se presente indica che l'oggetto in questione deve essere assegnato al campo indicato dal valore dell'attributo. Solitamente `field` compare negli oggetti innestati dentro altri oggetti ed è assente negli oggetti più vicini alla radice. Il valore di `field` si

riferisce ai campi dell'oggetto che contiene questo elemento. Se l'oggetto è contenuto in un array questo attributo può contenere il valore numerico della posizione occupata dall'oggetto nell'array.

?? `objectREF` - contiene il valore numerico in base 10 che referenzia un oggetto all'interno del documento.

Array

Gli array vengono serializzati mediante il seguente tag:

```
<!ELEMENT array (%object_name;*|%primitive_type;*)>
<!ATTLIST array
  field     CDATA #IMPLIED
  xmlns     CDATA #FIXED "momis.project.dsi.unimo.it"
  objectID  CDATA #IMPLIED
  objectREF CDATA #IMPLIED
  class     CDATA #REQUIRED
  length    CDATA #REQUIRED
>
```

Il contenuto di un `array` è uniforme: o compaiono solo oggetti o compaiono solo tipi primitivi omogenei (solo `int` oppure solo `float` ecc.).

Gli elementi che descrivono i componenti di un array hanno nell'attributo `field` il valore numerico dell'indice che occupa l'elemento. Questo valore viene scritto perché un elemento `array` contiene solamente i suoi componenti non nulli e il valore dell'indice è necessario per ricostruire l'array originario.

Attributi:

?? `field` – nome del campo a cui questo array va assegnato. Se l'array è contenuto in un altro array questo attributo può contenere il valore numerico della posizione occupata da questo array dentro l'array contenitore.

?? `objectID` e `objectREF` – contengono un valore numerico in base 10. La loro funzione è definire o riferire un oggetto, questi attributi si escludono a vicenda, se compare objectREF l'elemento array sarà vuoto.

?? `class` – contiene l'identificatore di classe dell'array: il suo valore identifica anche il tipo dei componenti dell'array. Questo attributo è obbligatorio.

?? `length` – contiene il valore numerico in base 10 che rappresenta la dimensione totale dell'array. Il numero degli elementi contenuti nell'elemento `array` in generale non è pari a questo valore: eventuali valori `null` non vengono scritti nel documento se sono componenti di un array.

Tipi primitivi

Un tipo primitivo è descritto da un elemento vuoto. Il valore di un tipo primitivo è contenuto dentro l'attributo `value`.

```
<!ELEMENT a_primitive_type EMPTY>
<!ATTLIST a_primitive_type
  field     CDATA #IMPLIED
  value     CDATA #REQUIRED
>
```

a_primitive_type indica uno qualunque dei tipi primitivi : `int`, `short`, `double`, `boolean`, `long`, `float`, `char`, `byte`.

Stringhe

Le stringhe java sono serializzate in XML mediante delle CDATA section, questo perché i dati contenuti potrebbero essere qualunque e si deve essere in grado di ricostruire esattamente la lunghezza della stringa originaria, anche gli spazi sono significativi. Una stringa ha la stessa intestazione di un oggetto ma può contenere solamente delle sezioni CDATA.

Valori null

Un riferimento nullo viene tradotto con il seguente elemento:

```
<!ELEMENT null EMPTY>
<!ATTLIST null
  field CDATA     #IMPLIED
>
```

Appendice C

Una DTD per la semantica descritta.

Quella che segue è la DTD utilizzata per definire il meta elemento. Le definizioni dei nomi dei tag sono state accentrate in alcune entity all'inizio del documento, in modo da facilitare la modifica.

```
<!-- DTD to define some elements that mantain the -->
<!-- ODLi3 semantic information lost in the translation
process -->

<!-- Customizable entities -->
<!ENTITY % metaelement "odli3_meta" >
<!ENTITY % odli3_source "odli3.source" >
<!ENTITY % odli3_interface "odli3.interface" >
<!ENTITY % odli3_key "odli3.key" >
<!ENTITY % odli3_foreignKey "odli3.foreignKey" >
<!ENTITY % key_field "key_field" >
<!ENTITY % odli3_attribute "odli3.attribute" >
<!ENTITY % odli3_relationship "odli3.relationship" >


<!-- Elements definition -->
<!ELEMENT %metaelement; (%odli3_source;)*>
<!ELEMENT %odli3_source; (%odli3_interface;)*>
<!ATTLIST %odli3_source;
   name                 NMTOKEN         #REQUIRED
   type                 CDATA           #IMPLIED
>

<!-- Interface element definition -->
<!ELEMENT %odli3_interface; ((%odli3_key;)*,
 (%odli3_foreignKey;)*, (%odli3_attribute; |
  %odli3_relationship; )*)>
<!ATTLIST %odli3_interface;
   name                 NMTOKEN         #REQUIRED
   extents              NMTOKENS        #IMPLIED
```

```
>


<!-- Key elements definition -->


<!ELEMENT %odli3_key; (%key_field;)+ >
<!ATTLIST %odli3_key;
  name          NMTOKEN         #IMPLIED
>
<!ELEMENT %key_field; EMPTY>
<!ATTLIST %key_field;
    attribute           NMTOKENS         #REQUIRED
    references          NMTOKENS         #IMPLIED
>

<!ELEMENT %odli3_foreignKey; (%key_field;)+ >
<!-- here 'references' must contain a odli3
     interface name -->
<!ATTLIST %odli3_foreignKey;
    references          NMTOKEN         #REQUIRED
>

<!-- meta attributes definitions -->
<!ELEMENT %odli3_attribute; EMPTY >
<!ATTLIST %odli3_attribute;
    odli3Name           NMTOKEN         #RE QUIRED
    type_id             CDATA           #REQUIRED
    DTDName             NMTOKEN         #REQUIRED
>

<!ELEMENT %odli3_relationship; EMPTY >
<!ATTLIST %odli3_relationship;
    odli3Name           NMTOKEN         #REQUIRED
    type_id             CDATA           #REQUIRED
    DTDName             NMTOKEN         #REQUIRED
    inverse             CDATA           #REQUIRED
>
```

Appendice D

Sintassi del linguaggio ODL_I^3.

Viene di seguito riportata la BNF del linguaggio descrittivo ODL_I^3. Sono riportati solamente i frammenti sintattici che differiscono dalla grammatica ODL originale. La sintassi completa del linguaggio può essere trovata in [21]

⟨interface_dcl⟩ : : = ⟨interface_header⟩ {[⟨interface_body⟩]};
⟨interface_header⟩ : : = **interface** ⟨Identifier⟩
[⟨inheritance_spec⟩]
[⟨type_property_list⟩]
⟨inheritance_spec⟩ : : = : ⟨scoped_name⟩ [,⟨inheritance_spec⟩]

Definizioni relative al modello di uno schema locale: il wrapper deve indicare il tipo e il nome della sorgente per ogni modello.

⟨type_property_list⟩ : : = **(** [⟨source_spec⟩] [⟨extent_spec⟩]
[⟨key_spec⟩] [⟨f_key_spec⟩] **)**
⟨source_spec⟩ : : = **source** ⟨source_type⟩ ⟨source_name⟩
⟨source_type⟩ : : = **relational | nfrelational** | **object** | **le**
⟨source_name⟩ : : = ⟨identier ⟩
⟨extent_spec⟩ : : = **extent** ⟨extent_list⟩
⟨extent_list⟩ : : = ⟨string⟩ | ⟨string⟩ **,** ⟨extent_list⟩
⟨key_spec⟩ : : = **Key**[**s**] ⟨key_list⟩
⟨f_key_spec⟩ : : = **Foreign_key** ⟨F_key_list⟩
. . .

Regole per la definizione del modello globale usate per mappare gli attributi tra le definizioni globali e le corrispondenti definizioni nelle sorgenti locali.

⟨attr_dcl⟩ : : = **[readonly] attribute**
⟨domain_type⟩ ⟨attribute_name⟩
[⟨x ed_array_size⟩] [⟨mapping_rule_dcl⟩]
⟨mapping_rule_dcl⟩ : : = **mapping_rule** ⟨rule_list⟩
⟨rule_list⟩ : : = ⟨rule⟩ | ⟨rule⟩**,**⟨rule_list⟩
⟨rule⟩ : : = ⟨local_attr_name⟩ | `⟨identifier⟩'
⟨and_expression⟩ | ⟨or_expression⟩
⟨and_expression⟩ : : = **(** ⟨local_attr_name⟩ **and** ⟨and_list⟩ **)**
⟨and_list⟩ : : = ⟨local_attr_name⟩ | ⟨local_attr_name⟩ **and** ⟨and_list⟩
⟨union_expression⟩ : : = **(** ⟨local_attr_name⟩ **union**⟨union _list⟩ **on**⟨identifier⟩ **)**
⟨union _list⟩ : : = ⟨local_attr_name⟩ | ⟨local_attr_name⟩ **union**⟨union _list⟩
⟨local_attr_name⟩ : : = ⟨source_name⟩**.**⟨class_name⟩**.**⟨attribute_name⟩

Relazioni utilizzate per definire il Common Thesaurus

⟨relationships_list⟩ : : = ⟨relationship_dcl⟩**;** | ⟨relationship_dcl⟩**;** ⟨relationships_list⟩
⟨relationships_dcl⟩ : : = ⟨local_Attr_name⟩ ⟨relationship_type⟩ ⟨local_attr_name⟩
⟨relationship_type⟩ : : = ⟨intensional_relationship⟩ |
⟨extensional_relationship⟩

⟨intensional_relationship⟩ : : = **SYN**| **BT**| **NT**| **RT**

⟨extensional_relationship⟩ : : = **SYN_E**| **BT_E**| **NT_E**

Definizione dei vincoli di integrità OLCD: dichiarazioni di regole (usando definizioni *if then*) valide per ogni istanza dei dati; specifiche delle mapping rule (specifica per le regole *and* e *union*)

⟨rule_list⟩ : : = ⟨rule_dcl⟩**;** | ⟨rule_dcl⟩**;** ⟨rule_list⟩
⟨rule_dcl⟩ : : = **rule** ⟨identier ⟩⟨rule_Spec⟩
⟨rule_Spec⟩ : : = ⟨rule_pre⟩ **then** ⟨rule_post⟩ | {⟨case_dcl⟩}
⟨rule_pre⟩ : : = ⟨forall⟩ ⟨identier ⟩ **in** ⟨Identifier⟩ **:** ⟨rule_body_list⟩
⟨rule_post⟩ : : = ⟨rule_body_list⟩
⟨case_dcl⟩ : : = **case of** ⟨identier ⟩ **:** ⟨case_list⟩
⟨case_list⟩ : : = ⟨case_spec ⟩|⟨case_spec ⟩⟨case_list⟩
⟨case_spec ⟩ : : = ⟨identifier⟩:⟨identifier⟩

⟨rule_body_list⟩ : : = **(** ⟨rule_body_list⟩ **)** | ⟨rule_body⟩ |
⟨rule_body_list⟩ **and** ⟨rule_body⟩ |
⟨rule_body_list⟩ **and (** ⟨rule_body_list⟩ **)**

⟨rule_body⟩ : : = ⟨dotted_name⟩ ⟨rule_const_op⟩ ⟨literal_value⟩ |
⟨dotted_name⟩ ⟨rule_const_op⟩ ⟨rule_cast⟩ ⟨literal_value⟩
⟨dotted_name⟩ **in** ⟨dotted_name⟩ |
⟨forall⟩ ⟨Identifier⟩ **in** ⟨dotted_name⟩ **:** ⟨rule_body_list⟩ |
exists ⟨identifier⟩ **in** ⟨dotted_name⟩ **:** ⟨rule_body_list⟩

⟨rule_const_op⟩ : : = = | ≥ | ≤ | > | <

⟨rule_cast⟩ : : = **(**⟨simple_type_spec⟩**)**

⟨dotted_name⟩ : : = ⟨identifier⟩ | ⟨identifier⟩**.**⟨dotted_name⟩

⟨forall⟩ : : = **For all** | **forall**

Appendice E

La grammatica di XML 1.0

La presente appendice è stata tratta da [12] e viene inserita come completamento alle informazioni su XML riportate nel Capitolo 2

In questa appendice si propone la sintassi XML attraverso la descrizione fornita dalla BNF con l'aggiunta dei vincoli di well-formdness e di validity. In questo modo si vuole fornire una guida esaustiva, anche se di difficile lettura in quanto priva di commenti, del linguaggio XML.

Document

```
[1] document::= prolog element Misc*
```

Character Range

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

White Space

```
[3] S::= (#x20 | #x9 | #xD | #xA)+
```

Names and Tokens

```
[4] NameChar::= Letter | Digit | '.' | '-' | '_' | ':' |
CombiningChar | Extender
[5] Name::= (Letter | '_' | ':') (NameChar)*
[6] Names::= Name (S Name)*
[7] Nmtoken::= (NameChar)+
[8] Nmtokens::= Nmtoken (S Nmtoken)*
```

Literals

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

Character Data

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

Comments

```
[15] Comment::= '<!--' ((Char - '-') | ('-' (Char - '-')))*
'-->'
```

Processing Instructions

```
[16] PI::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))?
'?>'
[17] PITarget::= Name - (('X' | 'x') ('M' | 'm') ('L' |
'l'))
```

DATA Sections

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

Prolog

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

Document Type Definition

```
[28] doctypedecl::= '<!DOCTYPE' S Name (S ExternalID)? S?
('[' (markupdecl
| PEReference | S)* ']' S?)? '>'
[VC: Root Element Type ]
[29] markupdecl::= elementdecl | AttlistDecl | EntityDecl |
                   NotationDecl| PI | Comment
[VC: Proper Declaration/PE Nesting ]
[WFC: PEs in Internal Subset ]
```

Validity Constraint: Root Element Type

Il Name nel "document type declaration" deve coincidere con "element type" dell' elemento radice.

Validity Constraint: Proper Declaration/PE Nesting

Il testo di sostituzione delle entità-parametriche deve essere annidato propriamente con le dichiarazioni dei markup. Cioè a dire, se il primo carattere o l'ultimo di una dichiarazione di markup (vedi sopra markupdecl) è contenuto nel testo di sostituzione associato ad un riferimento di entità parametrica, entrambi devono essere contenuti nel medesimo testo di sostituzione.

Well-Formedness Constraint: PEs in Internal Subset

Nei sottoinsiemi dei DTD interni, i riferimenti alle entità-parametriche possono capitare solo dove capitano le dichiarazioni dei markup, non all'interno delle dichiarazioni di markup. (Questo non si applica ai riferimenti che capitano nelle entità-parametriche esterne o al sottoinsieme esterno.)

External Subset

```
[30] extSubset::= TextDecl? extSubsetDecl
[31] extSubsetDecl::= ( markupdecl | conditionalSect |
                        PEReference | S )*
```

Standalone Document Declaration

```
[32] SDDecl::= S 'standalone' Eq (("'" ('yes' | 'no') "'")
                                 |('"' ('yes' | 'no') '"'))
[VC: Standalone Document Declaration ]
```

Validity Constraint: Standalone Document Declaration

La dichiarazione di documento "standalone" deve avere il valore "no" se qualsiasi dichiarazione esterna di markup contenga dichiarazioni di:

- attributi con valori di default, se gli elementi a cui questi attributi si applicano sono presenti nel documento senza le specifiche di valore per quegli attributi, o

- entità (oltre che amp, lt, gt, apos, quot), se i riferimenti a quelle entità sono presenti nel documento, o
- attributi con valori soggetti a normalizzazione, dove l'attributo è presente nel documento con un valore che cambierà come risultato della normalizzazione, o
- tipi di elemento con "element content", se spazi bianchi compaiono direttamente all'interno di qualche istanza di quei tipi.

Language Identification

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

Element

```
[39] element::= EmptyElemTag
| STag content ETag
[WFC: Element Type Match ]
[VC: Element Valid ]
```

Well-Formedness Constraint: Element Type Match

Il Name nel tag-di-fine di un elemento deve essere uguale all' "element type" contenuto nel tag-di-inizio.

Validity Constraint: Element Valid

Un elemento è valido se esiste una dichiarazione che rispetta la produzione elementdecl dove il Name corrisponde al tipo di elemento, e si verifica una delle seguenti condizioni:

1. La dichiarazione corrisponde a EMPTY e l'elemento non ha nessun contenuto.
2. La dichiarazione corrisponde a children e la sequenza di elementi figlio appartiene al linguaggio generato dall'espressione regolare che si trova nel modello di contenuto, con gli opzionali spazi bianchi (caratteri che corrispondono al non terminale S) tra ciascuna coppia di elementi figlio.
3. La dichiarazione corrisponde a Mixed e il contenuto consiste di character data e elementi figlio il cui tipo corrisponde ai nomi presenti nel modello di contenuto.
4. La dichiarazione corrisponde a ANY, e sono stati dichiarati i tipi di qualsiasi elemento figlio.

Start-tag

```
[40] STag::='<' Name (S Attribute)* S? '>'
[WFC: Unique Att Spec ]
```

```
[41] Attribute::= Name Eq AttValue
[VC: Attribute Value Type ]
[WFC: No External Entity References ]
[ WFC: No < in Attribute Values ]
```

Well-Formedness Constraint: Unique Att Spec

Nessun nome di attributo può apparire più di una volta nello stesso tag-di-inizio o nel tag di elemento vuoto

Validity Constraint: Attribute Value Type

L'attributo deve essere stato dichiarato; il valore deve essere del tipo dichiarato. (Per i tipi degli attributi, vedere il paragrafo 2.1.1.)

Well-Formedness Constraint: No External Entity References

I valori degli attributi non possono contenere riferimenti diretti o indiretti a entità esterne.

Well-Formedness Constraint: No ¡ in Attribute Values

Il testo di sostituzione di qualsiasi entità riferita direttamente o indirettamente in un valore di attributo (a parte "<") non deve contenere il carattere ¡.

End-tag

```
[42] ETag::= '</' Name S? '>'
```

Content of Elements

```
[43] content::= (element | CharData | Reference | CDSect |
                 PI | Comment)*
```

Tags for Empty Elements

```
[44] EmptyElemTag::= '<' Name (S Attribute)* S? '/>'
                     [WFC: Unique Att Spec ]
```

Element Type Declaration

```
[45] elementdecl::= '<!ELEMENT' S Name S contentspec S? '>'
[VC: Unique Element Type Declaration ]
[46] contentspec::= 'EMPTY' | 'ANY' | Mixed | children
```

Validity Constraint: Unique Element Type Declaration

Nessun element type può essere dichiarato più di una volta.

Element-content Models

```
[47] children::= (choice | seq) ('?' | '*' | '+')?
[48] cp::= (Name | choice | seq) ('?' | '*' | '+')?
```

```
[49] choice::= '(' S? cp ( S? '|' S? cp )* S? ')'
[VC: Proper Group/PE Nesting ]
[50] seq::='(' S? cp ( S? ',' S? cp )* S? ')'
[VC: Proper Group/PE Nesting ]
```

Validity Constraint: Proper Group/PE Nesting

Il testo di sostituzione di una entità parametrica deve essere propriamente annidato con gruppi racchiusi tra parentesi. In altre parole, se una delle parentesi di apertura o di chiusura di un costrutto tipo choice, seq, o Mixed è contenuta nel testo di sostituzione di una entità parametrica, allora entrambi devono essere contenute nel medesimo testo di sostituzione. Per interoperabilità, se un riferimento ad una entità-parametrica compare in un costrutto tipo choice, seq, o Mixed, il suo testo di sostituzione non dovrebbe essere vuoto, inoltre né il primo né l'ultimo carattere diverso da "blank" del testo di sostituzione dovrebbe essere un connettore (— or ,).

Mixed-content Declaration

```
[51] Mixed::= '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*'
           | '(' S? '#PCDATA' S? ')'
[VC: Proper Group/PE Nesting ]
[VC: No Duplicate Types ]
```

Validity Constraint: No Duplicate Types

Lo stesso nome non deve apparire più di una volta in una singola dichiarazione "mixed content".

Attribute-list Declaration

```
[52] AttlistDecl::= '<!ATTLIST' S Name AttDef* S? '>'
[53] AttDef::= S Name S AttType S Default Decl
```

Attribute Types

```
[54] AttType::= StringType | TokenizedType | EnumeratedType
[55] StringType::= 'CDATA'
[56] TokenizedType::= 'ID'
                              [VC: ID ]
                              [VC: One ID per Element Type ]
                              [VC: ID Attribute Default ]
                     | 'IDREF'
                              [VC: IDREF ]
                     | 'IDREFS'
                              [VC: IDREF ]
                     | 'ENTITY'
                              [VC: Entity Name ]
                     | 'ENTITIES'
                              [VC: Entity Name ]
```

```
| 'NMTOKEN'
      [VC: Name Token ]
| 'NMTOKENS'
      [VC: Name Token ]
```

Validity Constraint: ID

I valori di tipo ID devono rispettare la produzione Name. Un nome non deve apparire più di una volta in un documento XML come un valore di questo tipo: cioè i valori ID devono identificare univocamente gli elementi che li portano.

Validity Constraint: One ID per Element Type

Nessun "element type" può avere specificato più di un attributo di tipo ID.

Validity Constraint: ID Attribute Default

Un attributo ID deve avere un default che sia dichiarato #IMPLIED o #REQUIRED.

Validity Constraint: IDREF

Valori di tipo IDREF devono rispettare la produzione Name, e i valori di tipo IDREFS devono rispettare la produzione Names; a ciascun Name deve corrispondere il valore di un attributo ID di qualche elemento del documento XML; cioè i valori IDREF devono essere uguali al valore di qualche attributo ID.

Validity Constraint: Entity Name

Valori di tipo ENTITY devono rispettare la produzione Name, valori di tipo ENTITIES devono rispettare la produzione Names; a ciascun Name deve corrispondere il nome di una entità unparsed dichiarata nel DTD.

Validity Constraint: Name Token

Valori di tipo NMTOKEN devono rispettare la produzione Nmtoken, valori di tipo NMTOKENS devono rispettare la produzione Nmtokens.

Enumerated Attribute Types

```
[57] EnumeratedType::= NotationType | Enumeration
[58] NotationType::= 'NOTATION' S '(' S? Name (S? '|' S?
                  Name)* S? ')' [VC: Notation Attrib utes ]
[59] Enumeration::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S?
')' [VC: Enumeration ]
```

Validity Constraint: Notation Attributes

Valori di questo tipo devono essere uguali a uno dei nomi di notazione inclusi nella dichiarazione; tutti i nomi di notazione presenti nella dichiarazione devono essere dichiarati.

Validity Constraint: Enumeration

Valori di questo tipo devono essere uguali a uno dei token Nmtoken presenti nella dichiarazione.

Attribute Defaults

```
[60] DefaultDecl::= '#REQUIRED' | '#IMPLIED'
                  | (('#FIXED' S)? AttValue)
                  [VC: Required Attribute]
                  [VC: Attribute Default Legal]
                  [WFC: No < in Attribute Values]
                  [VC: Fixed Attribute Default ]
```

Validity Constraint: Required Attribute

Se la dichiarazione di default è la parola chiave #REQUIRED, allora l'attributo deve essere specificato per tutti gli elementi di tipo specificato nella dichiarazione "attribute-list".

Validity Constraint: Attribute Default Legal

Il valore di default dichiarato deve rispettare i vincoli lessicali del tipo di attributo lessicale.

Validity Constraint: Fixed Attribute Default Se un attributo ha un valore di default

dichiarato con la parola chiave #FIXED, allora tutte le istanze di quell'attributo devono contenere il valore di default.

Conditional Section

```
[61] conditionalSect::= includeSect | ignoreSect
[62] includeSect::= '<![' S? 'INCLUDE' S? '[' extSubsetDecl
']]>'
[63] ignoreSect::= '<![' S? 'IGNORE' S? '['
ignoreSectContents* ']]>'
[64] ignoreSectContents::= Ignore ('<![' ignoreSectContents
']]>' Ignore)*
[65] Ignore::= Char* - (Char* ('<![' | ']]>') Char*)
```

Character Reference

```
[66] CharRef::= '&#' [0-9]+ ';'
                | '&#x' [0-9a-fA-F]+ ';'
                  [WFC: Legal Character ]
```

Entity Reference

```
[67] Reference::= EntityRef | CharRef
[68] EntityRef::= '&' Name ';'
```

```
            [WFC: Entity Declared ]
            [VC: Entity Declared ]
            [WFC: Parsed Entity ]
            [WFC: No Recursion ]
[69] PEReference::= '%' Name ';'
            [VC: Entity Declared ]
            [WFC: No Recursion ]
            [WFC: In DTD ]
```

Well-Formedness Constraint: Entity Declared

In un documento senza alcun DTD, in un documento con soltanto il sottoinsieme interno del DTD che non contenga riferimenti ad entità parametrica, o in un documento con "standalone='yes'", il Name dato nel riferimento all'entità deve essere uguale a quello presente in una dichiarazione di entità, ad eccezione di quanto detto i documenti well-formed non hanno bisogno di dichiarare nessuna delle seguenti entità: amp, lt, gt, apos, quot.

La dichiarazione di una entità parametrica deve precedere qualsiasi riferimento ad essa. Similmente, la dichiarazione di una entità generale deve precedere qualsiasi riferimento ad essa che appaia in un valore di default all'interno di una dichiarazione di attribute-list.

Notare che se le entità sono dichiarate nel sottoinsieme esterno o in una entità parametrica esterna, un processore non-validante non è obbligato a leggere e ad elaborare le loro dichiarazioni; per tali documenti, la regola che un'entità deve essere dichiarata è un vincolo well-formedness solo se standalone='yes'.

Validity Constraint: Entity Declared

In un documento con un sottoinsieme esterno o un'entità parametrica esterna con "stan-dalone='no'", il Name dato nel riferimento di entità deve essere uguale a quella in una dichiarazione di entità. La dichiarazione di un'entità parametrica deve precedere qualsiasi riferimento ad essa. Similmente, la dichiarazione di una entità generale deve precedere qualsiasi riferimento ad essa che appaia in un valore di default all'interno di una dichiarazione di attribute-list.

Well-Formedness Constraint: Parsed Entity

Qualsiasi riferimento a entità non deve contenere il nome di una entità unparsed. Entità unparsed possono essere riferite solo nei valori di attributo che sono stati dichiarati di tipo ENTITY o ENTITIES.

Well-Formedness Constraint: No Recursion

Una entità parsed non deve contenere un riferimento ricorsivo su se stessa, sia direttamente che indirettamente.

Well-Formedness Constraint: In DTD

Riferimenti ad entità parametrica possono apparire solamente nel DTD.

Entity Declaration

```
[70] EntityDecl::= GEDecl | PEDecl
[71] GEDecl::= '<!ENTITY' S Name S EntityDef S? '>'
[72] PEDecl::= '<!ENTITY' S '%' S Name S PEDef S? '>'
[73] EntityDef::= EntityValue | (ExternalID NDataDecl?)
[74] PEDef::= EntityValue | ExternalID
```

External Entity Declaration

```
[75]ExternalID::= 'SYSTEM' S SystemLiteral
      | 'PUBLIC' S PubidLiteral S SystemLiteral
[76] NDataDecl::= S 'NDATA' S Name
      [VC: Notation Declared ]
```

Text Declaration

```
[77] TextDecl::= '<?xml' VersionInfo? EncodingDecl S? '?>'
```

Well-Formed External Parsed Entity

```
[78] extParsedEnt::=TextDecl? content
[79] extPE::= TextDecl? extSubsetDecl
```

Encoding Declaration

```
[80] EncodingDecl::= S 'encoding' Eq ('"' EncName '"' | "'"
                     EncName "'" )
[81] EncName::= [A-Za-z] ([A-Za-z0-9._] | '-')*
```

Bibliografia

[1] A. Rabitti. Architettura di un Mediatore per un Sistema di Sorgenti Distribuite ed Autonome. Tesi di Laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[2] A. Zaccaria. MOMIS: Il componente Query Manager. Tesi di Laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[3] A. Zanoli. SI-Designer, un tool di ausilio all'integrazione di sorgenti di dati eterogenee distribuite: progetto e realizzazione. Tesi di Laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[4] A.G. Miller. WordNet: A Lexical Database for English. *Communications of the AC*M, 38(11):39–41, 1995.

[5] Alon Levy, Dana Florescu, Jaewoo Kang, Anand Rajaraman, and Joanne J. Ordille. The Information Manifold Project. Available at http://www.research.att.com/ levy/imhome.html.

[6] Bourret Ronald "XML Namespaces FAQ" November, 2000 Aviable at http://www.rpbourret.com/xml/NamespacesFAQ.htm

[7] D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. A Description Logics Based Tool for Schema Validation and Semantic Query Optimization in Object Oriented Databases. Technical report, sesto convegno AIIA, 1997.

[8] Daniel P.Miranker and Vasilis Samoladas. Alamo: an Architecture for In-tegrating Heterogenous Data Sources. In *Proceedings of the 4th KRDB Worksho*p, Athens, Greece, August 1997.

[9] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. ODB-QOptimizer: a tool for semantic query optimization in OODB. In *Proc. of Int. Conf. on Data Engineering, ICDE'97*, Birmingham, UK, April 1997.

[10] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. ODB-Tools: a description logics based tool for schema validation and semantic query optimization in Object Oriented

Databases. In *Proc. of Int. Conference of the Italian Association for Artificial Intelligence (AI*IA97*), Rome, 1997.

[11] Eric Amstrong "Working with XML" JAXP tutorial aviable at http://java.sun.com/

[12] F. Guerra – "*MOMIS: il wrapper per sorgenti di dati XML*" Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1999-2000.

[13] F. Saltor and E. Rodriguez. On intelligent access to heterogeneous information.In *Proceedings of the 4th KRDB Worksho*p, Athens, Greece, August 1997.

[14] G. P. Grifa. Analisi di Affinità Strutturali fra classi ODL_I^3 nel Sistema MOMIS. Tesi di Laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1998-1999.

[15] Gio Wiederhold et al. *Integrating Artificial Intelligence and Database Technology*, volume 2/3. Journal of Intelligent Information Systems, June1996.

[16] Gio Wiederhold. Mediators in the architecture of Future InformationSystems. *IEEE Compute*r, 25:38–49, 1992.

[17] H. Garcia-Molina et al. The TSIMMIS Approach to Mediation: DataModels and Languages. In *NGITS worksho*p, 1995. Available ftp://db.stanford.edu/pub/garcia/1995/tsimmis-models-languages.ps.

http://www.sparc20.dsi.unimo.it/Momis/documents/ODL_I^3_syntax.pdf

[18] M. Franceschi. Il componente Query Manager di MOMIS: utilizzo della Conoscenza Estensionale. Tesi di Laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1999-2000.

[19] M.J.Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams, and E.L. Wimmers. Object Exchange Across Heterogeneous Information Sources. Technical report, Stanford University, 1994.

[20] M.T. Roth and P. Scharz. Don't Scrap It, Wrap it! A Wrapper Architecture for Legacy Data Sources. In *Proc. of the 23rd Int. Conf. on Very Large Database*s, Athens, Greece, March 1995.

[21] MOMIS – "Sintassi completa del linguaggio ODL_I^3" Aviable at:

[22] N.Guarino. Semantic Matching: Formal Ontological Distinctions for Infor-mation Organization, Extraction, and Integration. Technical

report, Summer School on Information Extraction, Frascati, Italy, July 1997.

[23] N.Guarino. Understanding, Building, and Using Ontologies. A commentary to 'Using Explicit Ontologies in KBS Development', by van Heijst, Schreiber, and Wielinga.

[24] ODMG "The Object Database Standard: ODMG 2.0" aviable at http://www.omg.org/

[25] Oliver M.Duschka and Micheal R.Genesereth. Infomaster - An Information Integration Toolkit. Technical report, Department of Computer Science. Stanford University, 1996.

[26] R. Hull and R. King et al. Arpa I^3 reference architecture, 1995. Available at http://www.isse.gmu.edu/I3 Arch/index.html.

[27] R. Hull. Managing Semantic Heterogeneity in Databases: A Theoretical Perspective. Technical report, Bell Laboratories, 1996.

[28] S. Castano and V. De Antonellis. Deriving Global Conceptual Views from Multiple Information Sources. In *preProc. of ER'97 Preconference Symposium on Conceptual Modeling, Historical Perspectives and Future Direction*s, 1997.

[29] S. Montanari. Un approccio intelligente all'Integrazione di Sorgenti Etero-genee di Informazione. Tesi di Laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[30] Silvia Zanni. Il componente Query Manager di MOMIS: esecuzione di in-terrogazioni. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1999-2000.

[31] Sonia Bergamaschi, Silvana Castano, Maurizio Vincini, Domenico Beneventano "*Semantic integration of heterogeneus information sources*" Data & Knowledge Engineering 36 (2001) 215-249

[32] Sun Microsystems - "Java object serialization specification" – aviable at http://java.sun.com/

[33] Sun Microsystems "Java Native Interface Specification" – aviable at http://java.sun.com/

[34] V.S. Subrahmanian, Sibel Adali, Anne Brink, J.ames, J. Lu, Adil Ra-jput, Timothy J. Rogers, Robert Ross, and Charles Ward. HER-MES: A Heterogeneous Reasoning and Mediator System. Available at http://www.cs.umd.edu/projects/hermes/overview/paper/index.html.

[35] World Wide Web Consortium "*Document Object Model (DOM) Level 1 Specification (Second Edition)"* Aviable at http://www.w3.org/DOM

[36] World Wide Web Consortium "*Document Object Model (DOM) Level 2 Core Specification "* Aviable at http://www.w3.org/DOM

[37] World Wide Web Consortium "Extensible Markup Language (XML) 1.0" W3C recommendation 6 October 2000 aviable at http://www.w3.org/XML

[38] World Wide Web Consortium "XML Names specification" – W3C recommendation aviable at http://www.w3.org/XML

[39] XML.org "XML cover pages" Aviable at http://www.xml.org/

[40] Y. Arens, C. A. Knoblock, and C. Hsu. Query Processing in the SIMS Information Mediator. *Advanced Planning Technolog*y, 1996.

[41] Y. Arens, C.Y. Chee, C. Hsu, and C. A. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. *International Journal of Intelligent and Cooperative Information System*s, 2(2):127–158, 1993.

[42] Y.Papakonstantinou H.Garcia-Molina and J.Ullman. Med-maker: a mediation system based on declarative specification. Technical report, Stanford University, 1995. ftp://db.stanford.edu/pub/papakonstantinou/1995/medmaker.ps.

[43] Y.Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *VLDB Int. Conf.*, Bombay, India, September 1996.