UNIVERSITA′ DEGLI STUDI
DI MODENA E REGGIO EMILIA
Sede di Modena-Facoltà di Ingegneria
Corso di Laurea in Ingegneria informatica

Dallo “Human Oriented” HTML al linguaggio ODLi3 per l’integrazione di informazioni

Relatore
Chiar.mo Prof. Maurizio Vincini
Correlatore
Ing. Francesco Guerra

Tesi di
Paola Prampolini

Anno Accademico 2001/2002

Ringraziamenti

Un ringraziamento speciale alla mia famiglia che mi ha permesso di arrivare alla fine di questo lungo percorso, dandomi la serenità necessaria a raggiungere un obbiettivo così importante.

In particolare ringrazio mia mamma, perché è stata capace non solo di starmi vicino in ogni momento dandomi dei consigli preziosi, ma anche di trovare sempre le parole giuste per farmi sentire all'altezza delle situazioni.

Un grazie di cuore a Max, ha "vissuto" questa tesi giorno per giorno, incoraggiandomi nei momenti peggiori, riuscendo a farmi sorridere anche quando sembrava impossibile …… vi assicuro che ha avuto molta pazienza!

Mille grazie agli amici di sempre, lontani, ma comunque pronti a dimostrarmi il loro affetto: Gianluca, Cristina, Elena.

Ringrazio le mie amiche per la "carica" che solo loro hanno e sanno trasmettere: Stefania, Francesca, Chiara, Morena, Marta, Cecilia.

Grazie alla Dottoressa, Giorgia, le nostre lunghe chiacchierate hanno alleggerito molte lezioni altrimenti interminabili!

Grazie a tutti voi, che avete reso piacevoli questi anni: spero che le nostre cene continuino anche quando saremo tutti dei "dott.ing."!

Desidero ringraziare inoltre il Prof. Maurizio Vincini, la Professoressa Sonia Bergamaschi e l' Ing. Francesco Guerra per la disponibilità dimostrata e il prezioso aiuto arrecato.

Paola Prampolini

INDICE

INTRODUZIONE

Il Web, così come lo conosciamo, è stato progettato per essere uno spazio contenente informazioni ed uno strumento per mediare l'accesso alle risorse da parte degli utenti. La necessità di tale mediazione si è tradotta, nel tempo, nello sviluppo di un complesso di tecnologie e strumenti sia ipertestuali, sia ipermediali che hanno condotto fino alla situazione attuale.

Il World Wide Web continua ad avere uno sviluppo straordinario: molte compagnie hanno creato un loro sito Web in cui forniscono cataloghi online e descrizioni di prodotti; agenzie governative hanno creato siti per divulgare le nuove regolamentazioni e informazioni sui servizi; i singoli individui hanno creato siti dedicati ai loro hobbies e interessi. Tale crescita è dovuta alla possibilità offerta dalla rete di rendere disponibile una grande quantità di informazioni, facilmente e in modo abbastanza economico.

Dal punto di vista dell'interscambio di informazioni il Web è un'enorme base informativa, tuttavia i dati sono organizzati e distribuiti in formati molto lontani dall'essere interpretabili e comprensibili, sul piano semantico, in modo automatico.

Ultimamente è diventata di grande interesse la ricerca di quali possano essere le metodologie e le tecniche per passare dall'attuale Web ad uno completamente nuovo, in cui anche le macchine saranno in grado di muoversi, cercare ed interpretare informazioni. Si tratta del Semantic Web, un Web con estensioni semantiche, la cui primaria caratteristica è quella di definire, intrinsecamente, una nuova struttura in cui tutte le risorse e le pagine siano associate a descrizioni, secondo formati standarizzati ed elaborati in modo automatico. In tal modo sarebbe possibile, ad esempio, rendere il Web navigabile tramite agenti software che, opportunamente istruiti, svolgerebbero automaticamente operazioni di ricerca e, più in generale, di accesso ai servizi. Sul piano dell' architettura logica, il Semantic Web non è separato da quello attuale ma, semplicemente, un layer di estensioni in cui alle informazioni è attribuito un significato ben definito ed espresso in modo tale da favorire la cooperazione fra utente e calcolatore.

L'impossibilità attuale di gestire in modo automatico le informazioni presenti nel Web, è dovuta sia all'eterogeneità dei formati in cui i dati sono conservati, sia alla natura del linguaggio HTML.

La maggior parte delle informazioni presenti in rete sono disponibili in HTML, un linguaggio "human oriented", cioè che da luogo a pagine Web disegnate in modo da essere facilmente comprensibili all'uomo e di notevole impatto, ma non adatto all'integrazione e alla manipolazione automatica dei dati, infatti i documenti prodotti non presentano una struttura rigorosa, ma sono semistrutturati.

L'obbiettivo della tesi è quello di introdurre una metodologia che permetta l'estrazione delle informazioni presenti nella rete in formato HTML, ed una loro successiva integrazione.

La prima parte di questo lavoro (capitoli uno e due) consiste nell' analisi di un sistema di generazione di wrapper, chiamato XWRAP, sviluppato presso il"Georgia Institute of Technology", in grado di trasformare i documenti HTML semistrutturati, in documenti XML ben strutturati, con associate le relative DTD. Le DTD (Document Type Definition) hanno la funzione di specificare la struttura e la sintassi dei documenti a cui sono associate. Una volta estratte le informazioni e convertite in un formato adatto all'utilizzo da parte della macchina, è neccessario passare alla loro integrazione. A questo proposito, sulla base delle conoscenze svilluppate in [15], quando ancora l'XML non era una Raccomandazione del W3C, nel capitolo cinque vengono ricavate delle regole di conversione fra DTD e $ODLi_3$. Il lavoro di traduzione riguarderà solamente la DTD, in quanto è da essa che si ricavano informazioni relative alla struttura dei dati, e quindi utili all'integrazione delle informazioni. $ODLi_3$ e' un linguaggio di descrizione dei dati ad alto livello che si avvale dello standard di modelli ad oggetti ODMG-93 [17,18] ed è indipendente dal formato della sorgente dei dati stessi. $ODLI_3$ nasce nell'ambito del progetto MOMIS (**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources) frutto di una collaborazione fra le unita' operative dell' Universita' di Milano e dell' Universita' di Modena con l'obiettivo di realizzare l'integrazione di sorgenti eterogenee e distribuite.

Per far si che il sistema MOMIS possa essere usufruito dal piu' ampio bacino d'utenza e' necessario che esso si apra agli standard di comunicazione attuali. Per questo motivo sono state ricavate anche le regole di traduzione fra XML Schema e $ODLI_3$ (capitolo 6). Uno schema XML è una collezione di definizioni di tipi e dichiarazioni di elementi utilizzati nel documento XML che ne costituisce un'istanza. Gli schemi sono molto più potenti e precisi rispetto alle DTD, infatti sono stati pensati per fornire quel supporto di validazione che le DTD permettono solo parzialmente, in particolare sul contenuto degli elementi e degli attributi dei documenti XML.

Una volta che gli schemi locali sono stati tradotti in $ODLi_3$, il sistema MOMIS è in grado di fornire una vista integrata che permette all'utente la formulazione di interrogazioni,

svincolandolo dal dover conoscere la struttura ed il contenuto delle singole sorgenti. La rappresentazione ottenuta non contiene alcun dato, ma costituisce la base dell'operazione di integrazione dei dati appartenenti alle singole sorgenti. Il risultato di questa fase consiste, quindi, nella produzione di conoscenze intensionali ed estensionali: le prime permettono la risoluzione di conflitti fra schemi, mentre le seconde indicano il modo in cui le entità del dominio applicativo sono rappresentate sull'insieme di sorgenti.

Nell'ultimo capitolo si verifica la completezza e validità delle regole ricavate, applicandole a due file associati a xCML e cXML, due dei più importanti standard utilizzati nell'ambito del commercio B2B. Il successo dei portali di B2B e-commerce è legato all'utilizzo di linguaggi standard di questo tipo, che garantiscono l' interoperabilità fra i partecipanti, permettendo uno scambio di documenti facilmente interpretabili da clienti e fornitori.

Vengono svolte, inoltre, un'analisi del linguaggio $ODLi_3$ (capitolo tre), e un confronto fra DTD, XML Schema e $ODLi_3$.

La tesi comprende, in appendice, il BNF del linguaggio descrittivo $ODLi_3$ (appendice A), mentre nelle appendici B e C si riportano le traduzioni complete dei file Orderstatusrequest.xsd dello standard xCBL, e CXML.dtd di cXML

CAPITOLO 1
Generazione Semiautomatica di Wrapper per Risorse Web: XWRAP

1.1 Introduzione

La maggior parte delle informazioni presenti in rete sono disponibili in HTML, hypertext markup language. Tale linguaggio si basa sui markup tags, elementi che specificano come deve essere interpretato il contenuto dei testi; essi sono disposti in modo gerarchico, e al loro interno è inserito del testo libero, perciò i documenti prodotti non presentano una struttura rigorosa, ma sono semistrutturati. HTML è un linguaggio ”human oriented” che dà luogo a pagine web disegnate in modo da essere facilmente comprensibili dall'uomo e di notevole impatto, ma non adatto all'integrazione e manipolazione automatica delle informazioni. Infatti la natura semistrutturata e statica dei dati HTML rende l'estrazione dei contenuti dalle risorse Web un compito alquanto difficile.

Tuttavia sono stati elaborati vari metodi che rendono possibile l'estrazione di dati dal web dopo averli convertiti in un formato adatto all'utilizzo da parte della macchina. La maggior parte di questi utilizza i Wrapper, programmi in grado di ricavare informazioni da documenti semistrutturati e restituirle sotto forma di dati strutturati. Una volta che le tuple vengono immagazzinate in un database, i contenuti possono essere interrogati e gestiti utillizzando degli standard query languages, come ad esempio SQL.

Inizialmente i Wrapper sono stati costruiti manualmente, ma risultavano troppo costosi in termini di tempo e denaro e poco pratici. Si è passati allora allo sviluppo di Wrapper semiautomatici e automatici utilizzando tecniche di apprendimento per le macchine. I Wrapper vengono istruiti mediante alcune annotazioni da parte dell'utente, impiegate come campioni ed esempi, più numerose nella progettazione dei Wrapper semiautomatici e ridotte per quelli automatici.

Nel caso della presente tesi sono stati analizzati vari metodi di generazione di wrapper; al termine di tale analisi si è scelto di focalizzare l' attenzione su XWRAP, elaborato dal “Georgia Institute of Technology”[1].

Lo scopo del toolkit XWRAP è quello di rendere il contenuto delle risorse Web facilmente accessibile ad ogni tipo di applicazione. Tale obbiettivo è stato raggiunto trasformando i difficili input HTML in output XML "program friendly", i quali possono essere parsati e compresi da sofisticati query services, mediator-based informations systems e agent_based systems.

Ogni volta che troviamo pagine interessanti online e vogliamo integrare i dati disponibili, o estrarli da risorse differenti per darne una rappresentazione complessiva, è necessario incapsulare il set di informazioni per tradurle in XML prima del processo di integrazione.

La tecnologia di XWRAP consiste in XWRAP original e XWRAP elite. Quest'ultimo costituisce un perfezionamento della prima versione, e verrà trattato in dettaglio più avanti. Ci soffermiamo ora su XWRAP original, chiarendo per prima cosa alcuni dettagli che emergono dalla definizione data.

XWRAP è un XML-enabled software system per la generazione semiautomatica di wrappers, applicabili a risorse Web.

Si parla di generazione semiautomatica in quanto XWRAP fornisce un interfaccia interattiva che assiste lo sviluppatore e contribuisce a rendere semplice la costruzione di Wrappers [2].

Per XML-enabled si intende che il programma Wrapper generato è in grado di produrre una rappresentazione XML della risorsa Web, rimuovendo le parti di scarso interesse, come per esempio annunci e immagini pubblicitarie. Praticamente i metadati impliciti nelle pagine Web originali vengono estratti ed esplicitati come tags XML nei documenti wrappati.

I Wrapper programs sono generati utilizzando classi Java, viene fornito inoltre un XML Wrapper Query Language (XML-WQL) come linguaggio di scambio per interpretare le richieste delle applicazioni, e per restituire gli oggetti o i documenti risultanti in formato XML.

XWRAP presenta tre caratteristiche fondamentali:

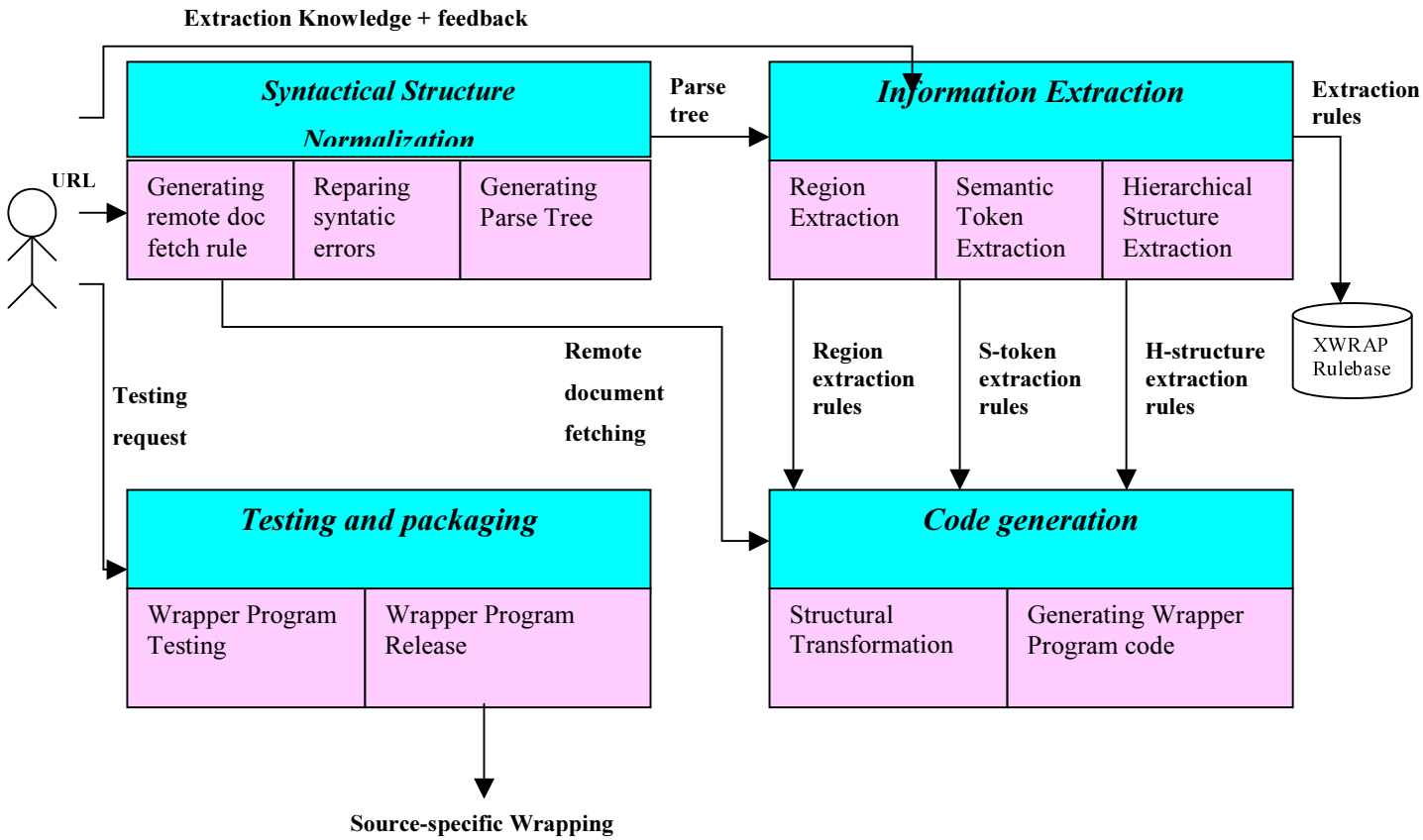
- È in grado di separare le regole di costruzione dei wrappers specifici di una risorsa da quelli ripetitivi propri di ogni risorsa, e fornisce una libreria dove sono raccolti i blocchi base per generare wrapper programs;
- Mette a disposizione un interfaccia interattiva che permette allo sviluppatore di generare Wrapper in pochi "mouse clicks";
- Introduce e sviluppa una struttura di generazione di codice a 2 fasi:
    1. PRIMA FASE: utilizza un'interfaccia interattiva per decifrare la conoscenza espressa dai metadati specifici della risorsa, individuati dallo sviluppatore come significative extraction-rules, cioè criteri per estrarre informazioni. Si tratta di

un elemento distintivo rispetto ad altri approcci che richiedevano all'utente di immettere in modo manuale le extraction-rules usando un linguaggio specifico di quel contesto.

2. SECONDA FASE: combina le extraction-rules ricavate nella prima fase con le librerie a disposizione per costruire ed eseguire wrapper programs per le risorsa data.

1.2 Architettura

L'architettura di XWRAP è costituita da quattro componenti: Sintactical Structure Normalization, Information Extraction, Code Generation, Program Testing and Packaging. La figura sottostante mostra il sistema di generazione dei wrapper.



Analizziamo ora i vari componenti passando in rassegna le fasi di generazione del wrapper.

Syntactical Structure Normalization. Il primo componente ad entrare in gioco ha la funzione di preparare il processo di estrazione delle informazioni, attraverso tre fasi:

1. Riceve dall'utente un URL, inoltra una richiesta http al server identificato dall'URL e ricerca il documento corrispondente, chiamato *page object.* Il page object è utilizzato da XWRAP come campione per interagire con l'utente in modo da apprendere e derivare le extraction rules rilevanti.
2. Elimina i tag HTML inutili e gli errori sintattici
3. Trasforma i page object ricavati in parse tree, detti anche *token tree* sintattici.

Information Extraction. Il secondo componente è responsabile della derivazione delle extraction rules, le quali descrivono come estrarre il contenuto di interesse dal documento HTML. L'information extraction component raggiunge tale obbiettivo in tre fasi:

1. Identifica le regioni di interesse nel documento analizzato.
2. Localizza i token semantici importanti e la loro posizione all'interno del token tree.
3. Ricava la struttura gerarchica del documento.

Tali fasi verranno analizzate nel dettaglio nella sezione 1.3 dedicata all'Information Extraction

Code Generation. Il compito del terzo componente è quello di generare il codice del wrapper basandosi sui tre set di extraction rules ricavati al passo precedente.
Si tratta di un passo molto importante, in quanto viene effettuata la conversione in codice della conoscenza semantica, rappresentata nella forma di extraction rules dichiarative e, successivamente, di XML_template. Il code generator , inoltre, produce la versione XML del page object considerato.

Testing and Packing. Fase finale del processo di data wrapping. L'utente inserisce una serie di URLs della stessa risorsa Web per permettere al modulo di testig di effettuare il debug del programma wrapper generato. Per ogni URL introdotto, il modulo di testing passa automaticamente alla prima e alla seconda fase per controllare se è possibile derivare ulteriori extraction rules o apportare modifiche a quelle esistenti. Inoltre, l'utente può consultare il

resoconto del test grazie alla finestra di test_monitoring.Ogni volta che uno dei tre set di extraction rules viene modificato, il modulo di testing attiva il code generator per produrre una nuova versione del wrapper program. Quando l'utente è soddisfatto dal risultato del test può cliccare il bottone di release e ottenere la versione definitiva.

1.3 Information Extraction

Il compito principale dell'information extraction component è quello di analizzare la struttura del page object e di descriverla mediante un linguaggio dichiarativo, utilizzando regole specifiche. Tale componenete ha come input il parse tree generato dal Syntactical Normalizer, e, per prima cosa, interagisce con l'utente per identificare i tokens semantici e la struttura gerarchica del documento.
I semantic tokens sono gruppi di syntatic tokens collegati tra loro, mentre i syntatic tokens sono sequenze di caratteri che possono essere trattate come singole unità sintattiche.
In seguito, l'information extractor, annota i nodi dell'albero associati a semantic tokens in un comma delimited file.

1.3.1 Preparazione al Processo di Estrazione delle Informazioni

Prima che il processo di information extraction inizi, il syntatical normalizer analizza il documento correggendo gli errori sintattici, successivamente si dedica alla costruzione del parse tree. I blocchi vengono scanditi carattere per carattere, sezionando il documento in una sequenza di unità atomiche, i syntatic tokens definiti precedentemente. Nella struttura ad albero generata ogni nodo rappresenta un syntatic token, e ogni tag node , come ad esempio TR, rappresenta una coppia di tags HTML: uno di inizio,<TR>, e uno di fine, </TR>. Tutti i nodi non terminali sono tags e tutte le foglie sono stringhe di testo, ognuna contenuta tra una coppia di tags.
E' da specificare che in HTML per token si intendono coppie di tags, tag singoli, nomi o valori.

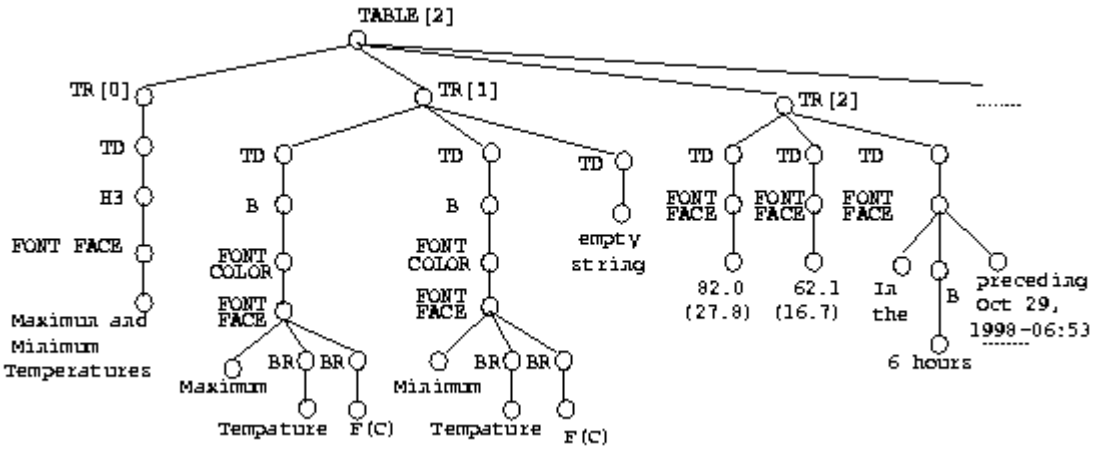
Esempio 1

Consideriamo la pagina relativa alle previsioni del tempo nella città di Savannah del National Weather Service site.
Di seguito viene riportato un frammento di documento HTML relativo a tale pagina.

```
<TABLE>
 <TR>
 <TD COLSPAN=3>
  <H3>
   <FONT FACE="Arial, Helvetica">Maximum end Minimum Temperature</FONT>
  </H3>
 </TD>
 <TR>
 <TD ALIGN=CENTER BGCOLOR="*FFFFFF>
  <B>
  <FONT COLOR=#0000A0">
   <FONT FACE="Arial,Helvetica">Maximum<BR>Temperature<BR>F(C)</FONT>
  </FONT>
 </B>
</TD>
<TD ALIGN=CENTER BGCOLOR==">*FFFFFF>
 <B>
  <FONT COLOR=#0000A0">
   <FONT FACE="Arial,Helvetica">Maximum<BR>Temperature<BR>F(C)</FONT>
  </FONT>
 </B>
</TD>
<TD ALIGN=CENTER>
 <FONT FACE="Arial, Helvetica">82.0(27.8)</FONT>
</TD>
<TD ALIGN=CENTER>
 <FONT FACE=="Arial, Helvetica">62.1(16.7)</FONT>
</TD>
<TD>
 <FONT FACE="Arial, Helvetica">In the<B>6 hours</B>precidingOct 29, 1998-
06:53 PM EST/ 1998.10.29 2353 UTC
```

```
</FONT>
<TD>
</TR>
<TR>
 <TD ALIGN=CENTER>
  <FONT FACE="Arial,Helvetica">80.1(26.7)</FONT>
 </TD>
 <TD ALIGN=CENTER>
   <FONT FACE="Arial, Helvetica">45.0(7.2)</FONT>
</TD>
<TD>
 <FONT FACE="Arial, Helvetica">In the<B>24 hours</B>precidingOct 28, 1998 -
11:53 PM EST/ 1998.10.28 0453 UTC</FONT>
<TD>
</TR>
<TR>
 <TD COLSPAN=3<HR SIZE=1 NOSHADE WIDTH="100%>
 </TD>
</TR>
</TABLE>
```

Segue una porzione dell'albero HTML corrispondente al frammento sovrastante, generato tramite un parser HTML-compliant-tree.



L'albero presenta sei tipi di tag nodes: TABLE, TR, TD, B, H3, FONT; mentre le foglie sono dei semantic tokens: Maximum temperature, Minimum Temperature, 82.0 (27.8), 62.1 (16.7), etc.Xwrap definisce un set di funzioni per ogni oggetto rappresentato come nodo nell'albero.

1.3.2 Estrazione delle regioni

Attraverso un'interfaccia interattiva si permette all'utente di specificare quali siano le regioni di interesse nel documento sorgente. L'output di questa fase è il set di extraction rules per le regioni, grazie alle quali è possibile identificare le regioni importanti nel parse tree.
Nel primo prototipo di XWRAP, la fase di estrazione delle regioni inizia chiedendo all'utente di evidenziare il nodo che costituisce lo start tag di un elemento importante. Il region extractor cercherà l'end tag corrispondente, evidenziando l'intera regione e definendo il tipo e il numero di sotto_regioni ad essa associate. Vengono estratte, inoltre, le extraction rules che ne descrivono la struttura, specifiche di ogni tipo di regione (tabelle, paragrafi,testi..)
Per esempio, per la regione del tipo TABLE, appartenente al documento sopra riportato, si possono ricavare le seguenti extraction rules:

```
Tree_Path(string,Node_ID,string
Node_Path)
      {setTableNode=Node_ID;
```

- *Tree_Path* specifica come definire il path di un nodo della table. Accetta come parametro di ingresso l'identificatore del nodo:Node_ID e restituisce il path relativo ad esso, ricavato mediante la funzione getNodePath. Mediante tale extraction rule è possibile identificare il path della TABLE[2], che sarà: HTML.BODY.TABLE[0].TR[0].TD[4].TABLE[2], come si vede chiaramente osservando il parse tree riportato precedentemente.

```
Table_Area(string Node_ID, string TN, string CN, integer RowMax, integer ColMax)
{setRowTag(Node_ID)=?TN;
 setColTag(Node_ID)=?CN;
 RowMax=GetNumOfRowe(Node_ID);
 ColMax=GetNumOfCole(Node_ID);}
```

- *Table_Area* conta il numero di righe e di colonne della tabella. I parametri accettati sono Node_ID, il tag di riga della tabella, TN, e il tag di colonna, CN. Questi due

vengono richiesti all'utente e sono necessari affinchè le funzioni getNumOfRowe e GetNumOfCole calcolino il numero di righe e di colonne. il region extractor è in grado di dedurre che la tabella TABLE[2] consiste al massimo di 5 righe e 3 colonne.

```
Effective_Area(string Node_ID, string RowSi, string RowKi, string
ColSi, string ColKi)
{SetRowStartIndex(Node_ID)=?RowSi;
 SetRowEndIndex(Node_ID)=?RowKi;
 SetColStartIndex(Node_ID)=?ColSi;
 SetColEndIndex(Node_ID)=?ColKi;
 GetEffectiveArea(Node_ID);
 }
```

- *Effective_Area* definisce l'area effettiva della tabella, cioè la sotto_regione in cui risiedono le righe e le colonne di interesse, eliminando, per esempio, quelle parti occupate solo da spazi. Per poter applicare tale regola è necessario che l'utente identifichi il tag di riga, TR, e quello di colonna, TD. Per calcolare l'area effettiva di TABLE[2] si richiede inoltre all'utente di immettere il row start index, cioè il numero della riga di inizio,(rowSI=2), il row end index, il numero della riga di fine, (rowEI=4), il column start index, colSI=1 e il column end index, colEI=3.

```
TableStyle(Node_ID)
{if(ElementType(Child(Child(Node_ID,1)),1)='Attribute?'
    If (ElementType(Child(Child(Node_ID,1)),2)='Attribute')
        SetVertical(Node_ID)=1,SetHorizontal(node_ID)=0;
    Else
        SetHorizontal(Node_ID)=0, SetVertical(Node_ID)=1;
}
```

- *Table_Style* serve a distinguere le tabelle verticali, nelle quali la prima colonna contiene una lista di nomi di attributi, dalle tabelle orizzontali, in cui tale lista si trova nella prima riga. Applicando la regola Table_style, si può dedurre che TABLE[2] è una tabella orizzontale.

```
GetTableInfo(string Node_ID, string TNN, string TN, string TP)
{SetTableNameNode(Node_ID)=TNN;
 TN=GetTableName(TNN);
 TP=GetNOdePath(TNN);
}
```

- *GetTableInfo* permette di ricavare il nome della tabella fornendo il path e la posizione del nodo relativi nel parse tree.

Si tratta di un metodo di estrazione robusto, nel senso che è possibile ricavare tutte le informazioni più importanti in tempo reale; inoltre le extraction rules vengono scritte in un linguaggio dichiarativo indipendente dall'implementazione del codice del wrapper. Questo alto livello di astrazione permette ai wrapper generati con XWRAP di essere maggiormente adattabili a cambiamenti inaspettati della sorgente.

1.3.3 Estrazione dei Semantic Token

Il Semantic Token Extractor è un programma interattivo che guida lo sviluppatore ad attraversare il parse tree sottolineando i semantic tokens di interesse, dove per semantic token si intendono gruppi di syntatic token legati tra loro logicamente. L'output prodotto da questa fase è un set di extraction rule che permettono l'estrazione dei semantic token di interesse da documenti web appartenenti allo stesso sito, e del comma_delimited file contenente i tipi e i valori degli elementi di interesse. Il formato comma_delimited è detto anche formato text, è utilizzato per file addetti allo scambio di dati fra diverse classi di software o differenti applicazioni. La prima linea di un comma_delimited file contiene i nomi dei campi che racchiudono i dati, elencati nelle righe successive. Sia i nomi dei campi che i dati sono separati da un delimitatore speciale, in particolare il sistema XWRAP supporta 3 tipi di separatori: la virgola (,), il punto e virgola (;), e il pipe (|). Per identificare i semantic token importanti il Semantic token extractor esamina i nodi successivi della pagina, partendo dalla prima foglia ancora non raggruppata in un token.
Tornando all'esempio trattato, il region extractor identifica quattro regioni di interesse, ognuna delle quali è una tabella. In particolare si è dedotto che il nodo denominato Maximum

and Minimum Temperatures è il titolo di una tabella, e, grazie all'interazione con l'utente, si è appreso che le foglie ancorate al sottoalbero TABLE[2].TR[1].TD[0] devono essere trattate come un unico semantic token, denominato Maximum Temperature F(c), con valore 84.9 (29.4). Allo stesso modo è possibile derivare delle extraction rules per i sottoalberi ancorati a TR[3] e a TR[4], utilizzando la funzione getStoken(), la quale preleva il semantic token indicato. Le regole derivate permettono al Token Extractor di generare l'algoritmo seguente, mediante il quale è possibile estrarre i token values associati ad ogni token name della regione.

```
<ST_extract>
ST_extract (String ST_name[], string ST_val[][])
<!--Start of the repetition-- >
<? XG-Iteration-XG "Start"?>
<loop> integer row_i = 3, 4
   <loop> integer col_j =o, 1, 2
        <rule_exp>
          extract ST_val[row_i, col_j]  =
                      TABLE[2].TR[row_i].TD[col_j].getStoken()
          Where TABLE[2].TR[1 ].TD[col_j].getStoken() = ST_name[col_j];
       </rule_exp>
   </loop>
<loop>
</ST_extract>
```

Attraversando l'intero albero della pagina, il Semantic Token Extractor produce il comma delimited file relativo, inserendo in esso i valori estratti nel giusto ordine.
Di seguito viene riportato un frammento del comma-delimited file relativo al nodo TABLE[2]. La prima riga riporta i nomi dei campi usati, la secona e la terza sono due data records.

```
     ....
Maximum temperature F(c); Minimum Temperature F(c);<TD></TD>
82.0(27.8);62.1(16.7); In the <B>6 hours</B> preceding Oct 29,
1998 - 6:53 PM EST / 1998.10.29 2353 UTC
80.1(26.7);46.0(7.2); In the <B>24 hours</B> preceding Oct 28,
1998 - 11:53 PM EST / 1998.10.28 0453 UTC
....
```

1.3.4 Estrazione della Struttura Gerarchica.

L'obbiettivo di questa fase è quello di individuare la struttura gerarchica del documento originale, identificando quali parti delle regioni o quali insiemi di tokens potrebbero essere raggruppati. In pratica viene determinata la struttura della pagina, specificando quale sia il tipo di gerarchia, il livello più alto della pagina,le sottosezioni, etc.

Il risultato è un set di extraction rules, scritte con una grammatica indipendente dal contesto, che descrivono la struttura gerachica del documento.

Le euristiche seguenti vengono utilizzate dall'estrattore per fornire una visione approssimativa di quale possa essere la struttura della risorsa, punto di partenza per un'analisi più approfonita effettuata tramite l'interazione con l'utente. Tali euristiche si basano sull'osservazione che le dimensioni dei caratteri usati per i titoli delle sottosezioni sono in genere più ridotte di quelle dei caratteri utilizzati per le sezioni madri.

- Si individuano le regioni che derivano dallo stesso nodo padre nel parse tree e le si dispongono in modo sequenziale,come appaiono nel documento originale.
- Il titolo della sezione o della tabella è sempre quello racchiuso tra la coppia di tags <H3>, </H3>.
- Si costruisce la gerarchia della sezione considerando la grandezza dei caratteri usati e la struttura dei tags di layout, come <TR>, <TD>, <P>.

Le euristiche utilizzate per identificare sezioni e titoli , in certi casi possono portare il sistema a commettere errori nella definizione della struttura gerarchica. Per esempio, basandosi sull'euristica della dimensione dei caratteri, il sistema potrebbe vedere alcune parole o frasi come titoli,mentre in realtà non lo sono, o viceversa. Per questo si dà all'utente la possibilità di interagire, per correggere eventueli errori, tramite un'interfaccia grafica che permette di evidenziare token omessi o cancellarne altri.

XWRAP arriva così a generare un algoritmo di estrazione della struttura che, data una pagina nella quale sono state identificate tutte le sezioni e i titoli corrispondenti, restituisce un XML template file per la pagina. Si tratta di un file XML ben formato contenente istruzioni che guidano l'XWRAP XML Template Engine a collocare i dati nel template.

1.4 Generazione della versione XML della risorsa

Il generatore XML consiste di un XML Template Engine e di un XML parser. L'XML Template Engine genera degli statements usando sia il comma delimited file sia l'XML template parsato dall'XML parser.
Di seguito viene riportato un frammento dell'XML template corrispondente alla regione della pagina analizzata.

```
 ...
<Maximum_and_Minimum_Temperatures>
<description>Maximum and Minimum temperatures</Description>
<!--Start of the repetition-->
<?XG-Iteration-XG "Start"?>
 <Maximum_and_Minimum_Temperature_Child>
    <Maximum_Temperature>
    <Description>Maximumtemperature F(c)</Description>
    <Value><?XG-InsertField-XG "Maximum Temperature"></Value>
    <Maximum_Temperature>


    <Minimum_Temperature>
    <Deacsription>MinimumTemperature F(c)</Description>
    <value><?XG-InsertField-XG "Minimum Temperature"></Value>
    </Minimum_Temperature>

    <TD>
    <Description></Description>
    <Value>< ?XG-InsertField-XG "TD"></Value>
    </TD
  </Maximum_and_Minimum_Temperatures_Child>
<?XG-Iteration-XG "End"?>
<!-- End Of The Repetition -->
</Maximum_and_Minimum_Temperature>
....
```

La funzione <?XG-InsertField-XG “FieldName”?>, dove XG sta per XWRAP code Generator, ricerca un campo con nome specificato “Fieldname” nel comma_delimited file e inserisce il dato nel punto in cui si trova l’istruzione.
L’istruzione <XG-iteration-XG> determina l’inizio e la fine di una parte ripetitiva; quando il Template Engine ha trovato l’end esrtae un altro record dalcomma delimited file e torna allo start per creare lo stesso set di tags creati al passo precedente, in questo modo vengono inseriti nuovi dati nel file XML risultante.
XWRAP code Generator produce il codice del wrapper applicando il comm_delimited file, le region extraction rules e le hierarchical structure extraction rules, il tutto descritto utilizzando un linguaggio specifico, chiamato XWRAP’s XML template_based extraction specification language.

CAPITOLO 2
Generazione Automatica di Wrapper per Risorse Web: XWRAP ELITE

2.1 Introduzione

XWRAP elite è una nuova versione di XWRAP che permette la generazione di Wrapper in modo automatico [3].

Lobbiettivo è sempre quello di fornire una tecnica per trasformare lo "human-oriented" HTML nel "machine-readable" XML; tuttavia XWRAP elite differisce da XWRAP original per molti aspetti:

- La versione elite permette l'estrazione di oggetti ed elementi in modo automatico, richiedendo un intervento da parte dell'utente estremamente ridotto rispetto al modello precedente. Viene fornito un set di algoritmi ed euristiche che creano un equilibrio fra la richiesta di input semantici e la necessità di automatizzare l'estrazione di informazioni e la generazione di wrapper.
- XWRAP elite è efficiente lavorando su pagine ricche di data object, cioè pagine che presentano oggetti aventi struttura simile e in numero non troppo ridotto (almeno 5 oggetti). Tali pagine sono solitamente il risultato di ricerche effettuate via HTML, appartenenti per esempio a Web sites che forniscono cataloghi di prodotti. XWRAP elite non è stato progettato per lavorare con pagine Web personali, per questi tipi più complessi può essere utilizzato XWRAP original.
- Distinzione saliente è la robustezza dei wrapper generati in relazione ai cambiamenti di presentazione delle corrispondenti Web pages. XWRAP elite è molto più robusto, infatti può lavorare finchè la natura dei documenti in questione è data objects rich, e i cambiamenti di presentazione e di layout, o gli annunci introdotti non hanno alcun effetto sulla sua performance. E' stato testato su migliaia di pagine Web e su centinaia di risorse ottenendo come risultato che, su siti ricchi di data object, gli algoritmi di estrazione di oggetti ed elementi garantivano dal 95% al 100% dell'efficienza. Al

contrario, la robustezza dei Wrapper generati da XWRAP original dipende fortemente dall'esperienza dell'utilizzatore del toolkit e dalla qualità di interazione tra lui e lo stesso toolkit.

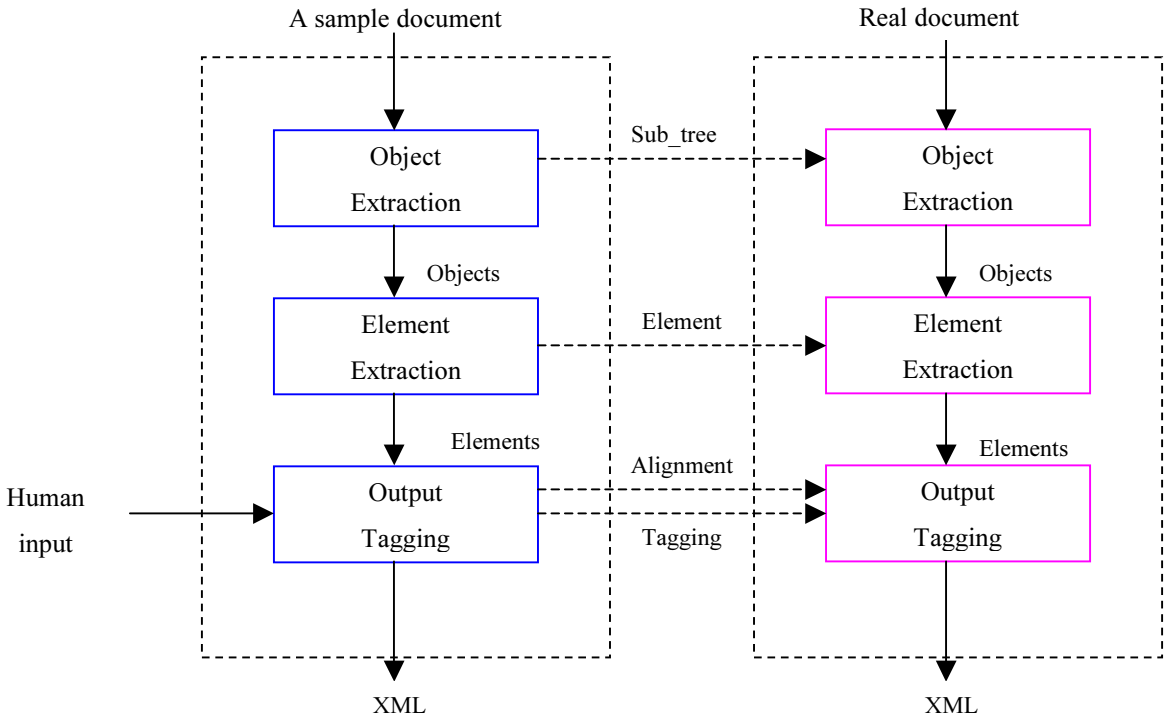
Quanto detto evidenzia l'importanza dell'automatizzazione di questo sistema di estrazione di informazioni, passo fondamentale nel raggiungimento dell'obbiettivo di creare un servizio sicuro di ricerca e aggregazione delle risorse Web.

I Wrapper programs sono generati utilizzando classi Java e ognuno include moduli come: remote page retrieval, object boundary discovery, object extraction, Element extraction, XML generation, XML query and filtering.

Viene fornito inoltre un XML Wrapper Query Language (XML-WQL) come linguaggio di scambio per interpretare le richieste delle applicazioni, e per restituire gli oggetti o i documenti risultanti in formato XML.

2.1 Architettura

Nella figura seguente viene illustrata l'architettura del sistema di XWRAP Elite [4].



XWRAP Elite riceve come input un documento HTML, con le caratteristiche discusse prima, e produce un wrapper per convertire il testo in XML, passando attraverso quattro fasi.

1. Viene generato un albero HTML, si localizza il sottoalbero contenente i data objects, e si ricavano le regole per sezionare tale sottoalbero in singoli oggetti. In tal modo si crea un ***Object Extraction*** component. Vengono estratti gli oggetti e resi disponibili per la fase successiva.
2. Si analizzano gli oggetti estratti per ottenere un gruppo di elementi separatori che li decompongano in elementi. Tale gruppo, che include anche tag HTML, determina un ***Element Extraction component***.
3. XWRAP Elite analizza gli elementi in modo da ricavare regole per raggruppare elementi similari. Lo sviluppatore assegna un nome ad ogni gruppo. Sulla base delle regole di allineamento e di assegnazione dei tags, XWRAP Elite genera un ***Element Tagging component.***
4. XWRAP Elite integra l'Object Extraction component, l'Element Extraction component e l'Output Tagging component, generando un wrapper in grado di convertire in XML una pagina HTML simile a quella del documento campione. La versione XML ottenuta dal documento utilizzato come campione aiuta a validare il processo.

2.2 Estrazione degli Oggetti

Per estrarre data objects da un documento HTML sono necessari quattro passi fondamentali: preparazione del documento, individuazione del contenuto principale, separazione degli oggetti, estrazione degli oggetti.

2.2.1 Document preparation

Si esegue la preparazione del documento HTML al processo di estrazione degli oggetti, cioè:

- la pagina data viene ripulita da un algoritmo di Syntactic normalization, con lo scopo di trasformarla in un documento che segue regole simili a quelle osservate da un documento XML ben formato;

- il documento risultante viene convertito in un tag tree, costituito da una struttura innestata di start e end tags. Esistono molti standard in grado di convertire un testo HTML in un documento ben formato, come Tidy (W3C), e, dopo questa operazione, generare una struttura ad albero è molto semplice.

Di seguito vengono riportati due data objects appartenenti alla Web page Barnes & Noble.com (www.barnesandnoble.com). Si tratta dei risultati della ricerca effettuata immettendo la parola chiave "java".

Core Java 2, Volume 1: Fundamentals
In Stock:Ships within 24 hours .
Cay S. Horstmann,Gary Cornell / Paperback / Prentice Hall PTR / December 2000
Our Price: $35.99, You Save 20%

Java How to Program
In Stock:Ships within 24 hours .
Harvey M. Deitel,Paul J. Deitel,Paul J. Deitel / Paperback / Prentice Hall PTR / August 2001
Our Price: $74.00

2.2.2 Primary content location

A questo punto viene individuato il contenuto di interesse interno alla pagina. Molte pagine contengono altre informazioni oltre a quelle principali, per esempio annunci, links, etc. Data una struttura ad albero, il problema di individuare la regione che racchiude il contenuto di interesse equivale a quello di localizzare il sottoalbero contenente tutti i data objects rilevanti. Come risultato è possibile ottenere più sottoalberi che rispettano soddisfano tale caratteristica, l'obbiettivo finele è quello di individuare tra essi il sub_tree minimo. In XWRAP Elite sono

stati implementati tre metodi per ricavare i sottoalberi corretti, i cui risultati vengono combinati insieme.

- Metodo ***Largest Tag Count*****:** si considera il numero di tags contenuti in ogni sottoalbero. Un elevato numero di tags indica che il sottoalbero corrispondente è molto ricco di contenuto.

- Metodo ***Highest Fanout*****:** si confrontano i sottoalberi in base al numero di figli diretti che hanno. Il numero più alto corrisponde al padre più probabile di tutti i data objects di interesse.

- Metodo ***Largest Size Increase*****:** si confrontano i sottoalberi in base allo sviluppo del contenuto visibile di ognuno. I sottoalberi con lo sviluppo più ampio molto probabilmente contengono le informazioni rilevanti.

2.2.3 Object Separation

Una volta terminata la fase di estrazione, il compito seguente è quello di individuare come i data objects vengono separati fra loro e rispetto ad altre informazioni contenute nella pagina [5]. L'obbiettivo è quello di sviluppare un metodo che trovi, in modo automatico, il tag separatore corretto nel sottoalbero minimo scelto. I passaggi fondamentali sono due:

- Si decide quali tags nel sottoalbero scelto possono essere considerati come tags separatori candidati.
- E' necessario un metodo per identificare il tag separatore corretto fra il set di candidati.

Si possono utilizzare vari metodi per individuare i possibili tags separatori:

- Si considera ogni nodo del sottoalbero scelto come tag candidato;
- Si considerano come candidati i nodi figli del sottoalbero scelto;

XWRAP Elite utilizza la seconda possibilità, e supporta cinque euristiche volte ad individuare il tag corretto, fornendo così un'ampia gamma di scelta fra diversi possibili meccanismi. Tali euristiche scelgono indipendentemente i tags candidati, in fine vengono combinati i risultati ottenuti da ognuna per migliorare l'accuratezza della scelta. In seguito verranno analizzate una ad una.

1) SD: Standard deviation heuristic

L'euristica SD misura la deviazione standard della distanza, in termini di numero di caratteri, fra due occorrenze consecutive di ogni tag candidato; in fine compila una lista di tali tag disposti in ordine ascendente in base alla loro deviazione standard. Tale procedimento è motivato osservando che istanze multiple di uno stesso tipo di oggetto, in un documento Web, solitamente sono poste alla stessa distanza.

2) RP: Repeating Pattern Heuristic

L'euristica RP sceglie i separatori di oggetti contando il numero di occorrenze di ogni coppia di tags candidati fra le quali non è contenuto testo. Si calcola il valore assoluto della differenza fra il numero di occorrenze della coppia e quello di ognuno dei due tag considerati separatamente; le coppie vengono poi elencate in una lista in ordine ascendente in base a tale valore assoluto.
L'intuizione alla base di questa euristica è che l'uso di un tag, da solo, può avere diversi significati, ma l'uso ripetuto di una coppia di tag probabilmente ha un unico scopo. Quando nel sottoalbero scelto le coppie non sono sufficienti, RP produce una lista vuota di tags, ciò significa che l'euristica non è in grado di restituire alcun risultato.

3) IPS: Identifiable Path Separetor Tag heuristic

L'euristica IPS elenca I tags candidati in modo coerente alla lista di sistema degli IPS tags. Questi sono i tags individuati dal sistema come quelli più comunemente usati per separare oggetti nei diversi tipi di sottoalberi appartenenti a documenti Web. Si può notare, infatti, che in una pagina Web possono essere contenuti vari tipi di sottoalberi, corrispondenti a tabelle, liste, ecc, e ognuno di questi tende ad avere una struttura regolare. Per esempio, per le tabelle, vengono usati cominemente il tag di riga TR e quello di colonna TC. In base a questa osservazione, maturata testando una serie di documenti web, è stata creata una lista di tags separator per ogni tipo di sub_tree. Successivamente si è calcolata la frequenza di utilizzo di

tali tags come separatori di oggetti, riportata nella tabella seguente, prodotta sulla base di sperimentazioni effettuate su vari siti Web.

Tag	% of time used as object separator
tr	34
table	18
p	10
li	8
hr	6
dt	6
ul	2
pre	2
font	2
dl	2
div	2
dd	2
blockquote	2
b	2
a	2

Dalla tabella è possibile ricavare la IPS list: {tr, table, p, li, hr, dt, ul, pre, font, dl, div, dd, blockquote, b, a, span, td, br, h4, h3, h2, h1, strong, em, i}. Per i tags con uguale rank l'ordine è arbitrario.

4) SB: Sibling Tag Heuristic

L'euristica SB considera le coppie di tags che discendono in modo diretto dallo stesso nodo padre, e le dispone in ornine discendente in base al numero di occorrenze. Per le coppie che hanno lo stesso numero di occorrenze sis egue l'ordine di apparizione nel documento. La SB e motivata dal fatto che, dato un sottoalbero minimo, i tag separatori compaiono tante volte quanti sono gli oggetti.

5) PP: Partial Path Heuristic

L'euristica PP considera tutti i path da un nodo candidato ad un altro, da esso raggiungibile, e calcola le occorrenze dei path identificati. La lista dei tags è ordinata in modo decrescente in base al numero di occorrenze dei path associati. Se a due path corrisponde lo stesso valore, il path più lungo vale di più perché indica una struttura più articolata. La motivazione di tale euristica è che istanze multiple di uno stesso tipo di oggetto spesso hanno la stessa struttura di tags.

Combinazione delle euristiche

Ognuna delle cinque euristiche lavora indipendentemente per raggiungere il medesimo obbiettivo: trovare il separatore do oggetti corretto fra il set di tags candidati. Ogni euristica restituisce come risultato il primo elemento della lista prodotta, tuttavia non sempre le cinque scelte fatte saranno coerenti fra loro.

Al fine di analizzare le performance delle euristiche su differenti pagine Web, è stata condotta una serie di esperimenti su cinquecento pagine Web, appartenenti a quindici diversi siti. Nella tabella seguente viene riportata una distribuzione empirica della probabilità di successo di ogni euristica. Per ognuna è stato calcolato il numero di volte in cui ha effettuato una scelta corretta, e il tasso di successo relativo ad un sito, normalizzando il numero di volte in cui un tag è stato scelto correttamente in relazione alla quantità di pagine testate appartenenti allo stesso sito.

Heuristic	Rank 1	2	3	4	5
SD	0.78	0.18	0.10	0.00	0.00
RP	0.73	0.13	0.00	0.00	0.00
IPS	0.40	0.46	0.13	0.07	0.00
PP	0.85	0.06	0.02	0.00	0.00
SB	0.63	0.17	0.12	0.06	0.03

Il modo per migliorare l'accuratezza di tale ricerca è, ovviamente, quello di utilizzare un metodo che combini in modo soddisfacente le cinque euristiche indipendenti. Per unire le conclusioni di due osservazioni indipendenti si usa la legge base della probabilità:

- sia P(A) la probabilità associata al risultato di un'euristica A applicata ad un documento Web, e P(B) la probabilità associata al risultato di un'euristica B applicata allo stesso documento Web. La formula $P(A \cup B)=P(A)+P(B)-P(A \cap B)$ produce la probabilità di trovare il tag separatore di oggetti corretto in un documento Web.

Esistono ventisei possibili combinazioni delle cinque euristiche e qulla che le combina tutte ha il maggior successo.

2.2.4 Objects Extraction

L'estrazione degli oggetti di interesse contenuti nella pagina analizzata viene effettuata in due fasi: costruzione degli oggetti candidati e raffinamento dell'estrazione .

1. ***Costruzione degli oggetti candidati.*** Gli oggetti vengono estratti partendo dal nodo radice ed utilizzando il tag separatore individuato precedentemente, il quale a volte si trova tra due oggetti, altre volte è il nodo radice o una parte di un oggetto. Può capitare che un oggetto venga diviso in più parti dal separatore, in questo caso è necessario adottare un meccanismo che lo ricostruisca.
2. ***Raffinamento dell'estrazione.*** E' il processo di eliminazione degli oggetti candidati che non rispettano i criteri minimi, individuati nella fase di estrazione, e soddisfatti dalla maggior parte dei candidati. In pratica vengono rimossi quegli oggetti estranei che non hanno lo stesso tipo di struttura degli altri.

Nella figura seguente viene schematizzato il processo di estrazione degli oggetti nel suo complesso.

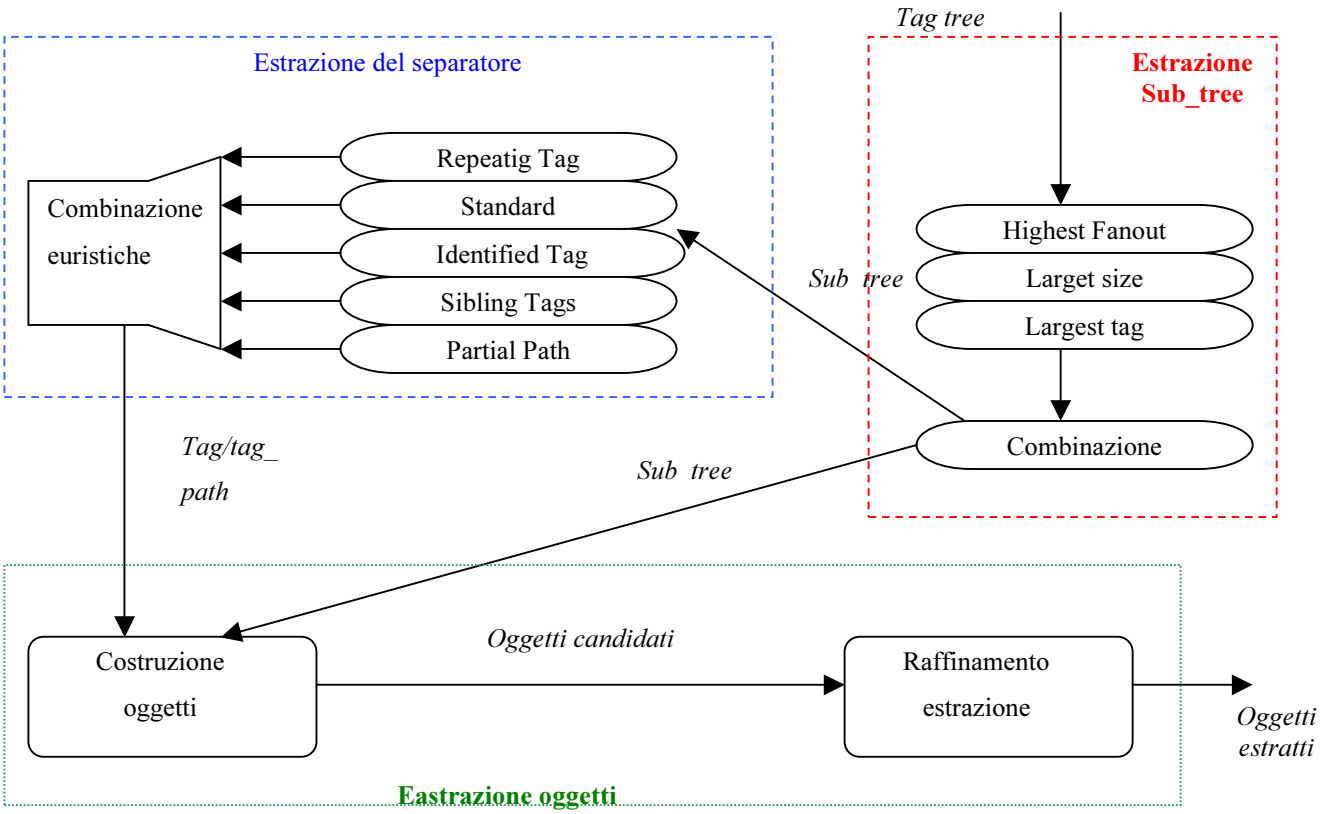


Figura 2.3: processo di estrazione degli oggetti

2.3 Estrazione degli Elementi

Una volta estratti gli oggetti, il passo successivo è quello di individuare gli elementi presenti al loro interno. Ogni data object è costituito da una serie di elementi delimitati da un gruppo di separatori, i quali possono essere sia tags HTML, che segni di punteggiatura. Spesso, i data object di una pagina Web, appartenenti ad una stessa regione di contenuto, sono omogenei, nel senso che presentano la stessa struttura e lo stesso gruppo di separatori. Questo permette, una volta individuato il set di delimitatori di ogni oggetto, di creare un gruppo di separatori riutilizzabile.

2.3.1 Separazione degli Elementi

Si utilizzano due diversi approcci per individuare text sepaparators e tag separators. Per prima cosa si costruisce un HTML tag tree e si applica ad esso un'euristica per ottenere un gruppo di tag separators, in seguito si utilizzano tre euristiche complementari per revisionarli. Mentre, per ottenere i text separators, si estraggono le stringhe di testo dagli oggetti e poi si analizzano con un' euristica.

1)Tag Separators

Ogni albero HTML , corrispondente ad un data object, viene scomposto dai tag separators in sottoalberi che contengono tutti gli elementi di interesse.
Per riuscire ad individuarli, inizialmente si cercano i tag separators comunemente usati, come
 e <a>, in seguito si applicano delle euristiche complementari al fine di scoprirne altri. Tali sono le seguenti:

- *Highest Count Tag Heuristic:* maggiore è il numero di occorrenze di un tag, più alta è la probabilità che sia un separatore;
- *Repeating Pattern Euristica:* se un tag compare di frequente in un pattern ripetitivo è, molto probabilmente, un tag separators;

- *Standard Deviation Heuristic:* la deviazione standard, relativa all'intervallo ta due occorrenze di uno stesso tag, è indicativa della regolarità del tag stesso. Più piccola è la deviazione standard di un tag, più regolari sono le sue occorrenze, quindi maggiore è la probabilità che sia un separatore.

2) Text Separators

Viene messa a disposizione una lista dei text separators usati più di frequente, la quale include i seguenti simboli: "/", ":", ",", ";".
Il simbolo "\n" non rientra nella lista in quanto in HTML viene trattato come spazio, perciò può comparire anche all'interno degli elementi.
Tuttavia i simboli elencati possono essere usati Anche come segni matematici. L'euristica per individuare i text separators opera nel modo seguente:

1. Si scandisce l'oggetto,
2. Quando si trova uno dei simboli della lista, se questo compare nell'oggetto non compreso fra due cifre, viene inserito nel gruppo di text separators candidati.

Una votla ottenuti i tag separators e i text separators per ogni oggetto, si costruisce un gruppo di delimitatori rappresentativo, il quale contiene solo i separatori che compaiono nella maggior parte dei gruppi.

2.4 Output Tagging

Per produrre la versione XML del documento analizzato ,è necessario annotare oggetti ed elementi indicando il loro significato semantico. Si suppone che i data object siano omogenei, perciò vengono denominati tutti allo stesso modo. Per gli elementi la questione è più complessa, infatti un oggetto ne racchiude di diversi tipi, e non è detto che siano sempre tutti presenti. Per questo motivo, prima di assegnare ad essi un nome, è necessario identificarli.
Esistono due metodi per identificare gli elementi:

1. **Usando espressioni regolari**. Sono però difficili da ottenere automaticamente, sooprattutto quando gli elementi sono simili.

2. **In base all'ordine di apparizione nell'oggetto**. Si ottengono dei buoni risultati solo per certi domini di applicazione, inoltre un errore causa un'errata identificazione anche degli altri elementi.

Si è scelto un approccio ibrido, che usa sia il primo che il secondo metodo, costituito dalle seguenti fasi:

1. si assegna un numero ad ogni elemento in base all'ordine in cui appare nell'oggetto;
2. vengono generate, in modo automatico, alcune espressioni regolari per facilitare l'allineamento degli indexnumber nel caso in cui un elemento non compaia un oggetto;
3. Agli elementi con lo stesso numero viene assegnato lo stesso nome.

2.4.1 Individuazione delle espressioni regolari

Un'espressioni regolare può essere di due tipi:

- ***Strong_matching***: implica un'alta probabilità che due elementi siano allineati se entrambi soddisfano tale modello. Un tipico esempio sono gli elementi costituiti da valori costanti, oppure quelli contenenti stringhe costanti asssociate a numeri variabili.
- ***Weak_matching***: indica che due elementi potrebbero non essere allineati se solo uno dei due soddisfa tale modello. Esistono quattro tipi di weak_machine patterns, ognuno corrispondente ad una classe di elementi: i link, le immagini, i prezzi, le stringhe comuni.

2.4.2 Allineamento degli elementi

L'allineamento consiste nello stabilire una corrispondenza fra elementi appartenenti all'oggetto con il maggior numero di componenti, che chiameremo oggetto di riferimento, e quelli di altri oggetti. A ciascuno di questi viene assegnato un numero, in base all'ordine di apparizione dell'elemento corrispondente nell'oggetto di riferimento, creando così un indice; elementi con lo stesso numero condividono lo stesso nome.
Consideriamo un generico elemento E, per individuare il corrispondente nell'oggetto di riferimento si segue il procedimento seguente:

- Se E è un valore costante ed esite un elemento strong_matching nell'oggetto di riferimento, assegnare ad E il numero di indice dell'elemento. Se ci sono più elementi di questo tipo scegliere quello che il cui ordine di apparizione è più simile a quello di E. Se nessuna di queste ipotesi è verificata procedere al passo successivo.

- Se un elemento dell'oggetto di riferimento, il cui numero di indice è maggiore di quello dell'elemento immediatamente prima di E, costituisce un weaking_pattern per E, si stabilisce che esso sia l'elemento corrispondente. Se esistono più elementi corrispondenti, si sceglie quello associato al numeropiù piccolo; nel caso non se ne trovi nemmeno uno si procede al passo successivo.

- Se un elemento dell'oggetto di riferimento, il cui numero di indice è minore o uguale di quello dell'elemento immediatamente prima di E, costituisce un weaking_pattern per E, si stabilisce che esso sia l'elemento corrispondente. Se esistono più elementi corrispondenti, si sceglie quello associato al numeropiù grande; nel caso non se ne trovi nemmeno uno E viene marcato come "unknown".

La tabella riportata di seguito mostra gli elementi dei due oggetti prima dell'allineamento.
Il primo oggetto contiene 14 elementi, mentre il secondo solamente 2; inoltre il decimo elemento dell'object2 non corrisponde al decimo elemento dell'object1, ma è dello stesso tipo del dodicesimo (link), lo stesso vale per l'undicesimo e il tredicesimo rispettivamente dell'oject2 e dell'object1.

order	Object1	Object2
1	<a href="...">	<a href="...">
7	December 2000	August 2001
8	Our price	Our price
9	$35.99	$74.00
10	You save	<a href="morelink">
11	20%	more
12	<a href="morelink">	<img src=".."
13	more	
14	<img src="."	

E' possibile allineare decimo, undicesimo e dodicesimo elemento dell'object2 con dodicesimo tredicesimo e quattordicesimo elemento dell'object1, come avviene nella tabella seguente.

order	Object1	Object2
1	<a href=”..”>	<a href=”..”>
7	December 2000	August 2001
8	Our price	Our price
9	$35.99	$74.00
10	You save	
11	20%	
12	<a href=”morelink”>	<a href=”morelink”>
13	more	more
14	<img src=”.”	<img src=”..”

2.5 Pregi di XWRAP ELITE

L’approccio presentato permette di generare wrapper per risorse Web in modo automatico. La maggior parte dei metodi implementati richiede allo sviluppatore di scrivere manualmente le extraction rules, utilizzando un linguaggio specifico del dominio, oppure immettendo le loro conoscenze tramite interfaccia interattiva. L’automazione dell’estrazione dei dati offre diversi vantaggi:

- non viene richiesto necessariamente un set di documenti campione;
- la generazione dei wrapper richiede molto meno tempo;
- all’utente viene richiesta solo l’immissione dei nomi degli oggetti e degli elementi, in questo modo la robustezza dei wrapper è indipendente dall’esperienza dell’utilizzatore. Infatti, incorporando altre tecnologie, è possibile revisionare automaticamente il wrapper quando i dati della sorgente vengono modificati.

CAPITOLO 3

Il Linguaggio ODLi$_3$

3.1 L'integrazione delle Informazioni

I problemi principali che si presentano nell'integrazione delle informazioni sono legati alla natura dei dati (testi, immagini, suoni, record, ...) ed ai diversi tipi sorgenti, che possono essere pagine HTML, DBMS relazionali o ad oggetti, file system, etc... .

Gli standard attuali (TCP/IP, ODBC, OLE, CORBA, SQL, etc...) non risolvono completamente i problemi legati alle diversita' Hw e Sw dei protocolli di rete e delle comunicazioni fra i moduli, rimangono, infatti, irrisolti quelli specifici della modellazione delle informazioni. I modelli dei dati e gli schemi si differenziano tra loro per quanto riguarda la definizione di una struttura logica per i numerosi generi di dati da immagazzinare e questo crea una eterogeneita' semantica (o logica) non risolvibile da questi standard.

I problemi principali che complicano la generazione di un sistema di integrazione di informazioni sono [6] :

1. problemi ontologici.
2. problemi semantici.

3.1.2 Problemi Ontologici

Per ontologia si intende, in questo ambito, "l'insieme dei termini e delle relazioni usate in un dominio, che denotano concetti ed oggetti". Con ontologia quindi ci si riferisce a quell'insieme di termini che, in un particolare dominio applicativo, denotano in modo univoco una particolare conoscenza e fra i quali non esiste ambiguità poichè sono condivisi dall'intera comunità di utenti del dominio applicativo stesso. Non è certamente l'obiettivo nè di questo paragrafo, nè della tesi in generale, dare una descrizione esaustiva di cosa si intenda per ontologia e dei problemi che essa comporta ma ci si limita a riportare una semplice classificazione delle ontologie (mutuata da Guarino [7, 8]), per inquadrare l' ambiente in cui

ci si muove. I livelli di ontologia (e dunque le problematiche ad essi associate) sono essenzialmente i seguenti:

1. top-level ontology: descrivono concetti molto generali come spazio, tempo, evento, azione, che sono quindi indipendenti da un particolare problema o dominio: si considera ragionevole, almeno in teoria, che anche comunita' separate di utenti condividano la stessa top-level ontology.

2. domain e task ontology: descrivono, rispettivamente, il vocabolario relativo a un generico dominio (come puo' essere un dominio medico, o automobilistico) o a un generico obiettivo (come la diagnostica, o le vendite), dando una specializzazione dei termini introdotti nelle top-level ontology.

3. application ontology: descrivono concetti che dipendono sia da un particolare dominio che da un particolare obiettivo. Come ipotesi semplificativa di questo progetto, si e' considerato 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).

3.1.3 Problemi Semantici

Pur ipotizzando che anche sorgenti diverse condividano una visione similare del problema da modellare, e quindi un insieme di concetti comuni, niente ci dice che i diversi sistemi usino esattamente gli stessi vocaboli per rappresentare questi concetti, nè tantomeno le stesse strutture dati. Poiché infatti le diverse sorgenti sono state progettate e modellate da persone differenti, è molto improbabile che queste persone condividano la stessa concettualizzazione del mondo esterno, ovvero non esiste nella realtà una semantica univoca a cui chiunque possa riferirsi.

Se la persona P1 disegna una fonte di informazioni (per esempio DB1) e un'altra persona P2 disegna la stessa fonte DB2, le due basi di dati avranno sicuramente differenze semantiche: per esempio, le coppie sposate possono essere rappresentate in DB1 usando degli oggetti della classe COPPIE, con attributi MARITO e MOGLIE, mentre in DB2 potrebbe esserci una classe PERSONA con un attributo SPOSA.

La causa principale delle differenze semantiche si può identificare nelle diverse concettualizzazioni del mondo esterno che persone distinte possono avere, ma non e' 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 così 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. eterogeneita' 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. eterogeneita' tra le strutture delle classi: comprendono le differenze nei criteri di specializzazione, nelle strutture per realizzare una aggregazione, ed anche le discrepanze schematiche, quando cioe' valori di attributi sono invece parte dei metadati in un altro schema (come puo' essere l'attributo SESSO in uno schema, presente invece nell'altro implicitamente attraverso la divisione della classe PERSONE in MASCHI e FEMMINE).

3. eterogeneita' 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.

Parallelamente a tutto questo, e' però il caso di sottolineare la possibilità di 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.

3.2 Il Sistema Momis

I problemi esposti sottolineano la complessità degli aspetti che le architetture dedicate all'integrazione devono comprendere. Per facilitare il processo di progettazione e realizzazione di tali moduli dedicati, che siano affidabili, flessibili e tali da far fronte efficacemente ad un panorama in continua evoluzione, si cerca di sviluppare un architettura di moduli che assicuri il riuso e la capacità di interagire con altri sistemi esistenti.

Gli approcci all'integrazione, descritti in letteratura o effettivamente realizzati, presentano diverse metodologie: la reingegnerizzazione delle sorgenti mediante standardizzazione degli

schemi e creazione di un database distribuito; il repository independence, 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, ovvero delle porzioni di sorgenti, replicando fisicamente i dati ed affidandosi a complicati algoritmi di allineamento per assicurare la loro consistenza a fronte di modifiche nelle sorgenti originali.
Tenendo presente tutte le problematiche, e un insieme di progetti preesistenti quali TSIMMISS [9] GARLIC [10] e SIMS [11], si è giunti alla progettazione di un sistema intelligente di Integrazione delle Informazioni. MOMIS (Mediator EnvirOnment for Multiple Information Sources [12][13]), sviluppato con l'obbiettivo di realizzare l'integrazione di sorgenti eterogenee e distribuite, nasce all interno del progetto MURST 40% INTERDATA, come collaborazione fra le unita' operative dell' Universita' di Milano e dell Universita' di Modena.
MOMIS e' il progetto di un sistema I^3 per l'integrazione di sorgenti di dati strutturati e semistrutturati. I^3 è un programma sviluppato dall'ARPA, agenzia che fa capo al Dipartimento di Americano [14], come frutto di un'ambiziosa ricerca, finalizzata ad indicare un architettura di riferimento che realizzi l'integrazione di sorgenti eterogenee in maniera automatica. L'integrazione aumenta il valore dell' informazione ma richiede una forte adattabilità realizzativa: si devono poter gestire i casi di aggiornamento e sostituzione delle sorgenti, dei loro ambienti e/o piattaforme, della loro ontologia e della semantica. Le tecniche sviluppate dall'*Intelligenza Artificiale*, potendo efficacemente dedurre informazioni utili dagli schemi delle sorgenti, diventano uno strumento prezioso per la costruzione automatica di soluzioni integrate flessibili e riusabili.
MOMIS adotta un'architettura a tre livelli con un *Mediatore* che ne occupa la parte centrale ed avente lo scopo di fornire una visione integrata degli schemi locali. Questa vista integrata deve permettere all'utente la formulazione di interrogazioni svincolandolo dal dover conoscere la struttura ed il contenuto delle singole sorgenti. Il *Mediatore* rappresenta dunque il cuore del sistema ed ha il compito di realizzare l'integrazione degli schemi e di provvedere alla gestione delle interrogazioni.
Elementi indubbiamente innovativi di questo progetto sono l'impiego di un approccio *semantico* e di Logiche Descrittive. Questi elementi introducono, infatti comportamenti intelligenti che permettono di sfruttare al meglio le conoscenze intensionali, semantiche ed estensionali sia inter-schema sia intra-schema, per generare una vista globale il più possibile

espressiva. Oltre ad una migliore integrazione delle sorgenti si ottiene dunque un processo di gestione delle interrogazioni più efficiente e funzionale.
Per far si che il sistema MOMIS possa essere usufruito dal piu' ampio bacino d'utenza e' necessario che esso si apra agli standard di comunicazione attuali.
Questo problema è ulteriormente complicato dall'approccio *semantico* e dall'uso di Logiche Descrittive fatto nel sistema MOMIS, in particolar modo sulle modalita' di esportazione della conoscenza che tali meccanismi innovativi permettono di catturare.
Alcune delle problematiche del sistema MOMIS sono oggetto di discussione da parte di molti esperti ed hanno portato alla constatazione della necessita' di una terza generazione del web il cosidetto *Semantic-Web* che mira a far in modo che i dati presenti nella rete siano non solo *machine-readable* ma anche *machineunderstandable*.

3.3 Il Linguaggio ODLi$_3$

L'integrazione delle sorgenti informative strutturate e semistrutturate viene compiuta in modo semi-automatico, utilizzando le descrizioni degli schemi locali in linguaggio ODLi$_3$.
ODLi$_3$ è un linguaggio di descrizione dei dati ad alto livello, si avvale dello standard di modelli ad oggetti ODMG-93 [17,18] ed è indipendente dal formato dalla sorgente dei dati stessi. Esso deriva dal linguaggio ODL con delle estensioni riguardanti:

- l'introduzione di due tipi di rules,
- il costrutto union
- il costrutto optional (*),
- le relazioni terminologiche intra ed inter sorgenti.

Nell'ambito di MOMIS viene appunto sfruttato per le sue caratteristiche che consentono al mediatore di lavorare allo stesso modo per sorgenti diverse, sia per dati semistrutturati che per dati strutturati.
La traduzione di ogni particolare schema in un formato ODLi$_3$ avviene mediante l'utilizzo di wrapper, i quali sono anche responsabili dell'aggiunta di informazioni utili per il mediatore stesso, quali il nome ed il tipo della sorgente originaria. Questa operazione di traduzione avviene direttamente sulle basi della sintassi ODLi$_3$ e sulla definizione del modello ad oggetti, in particolare dato un modello <l,A>,o una relazione di una sorgente relazionale, abbiamo che

una classe ODLi$_3$ corrisponde a l o al nome della relazione, e per ogni etichetta $l^1 \in$ A, o attributo relazionale, un attributo e' definito nella corrispondente classe ODLi$_3$.
Il punto di arrivo dell'acquisizione degli schemi sorgenti e la base di partenza per le funzionalitàa di MOMIS stesso e' il linguaggio ODLi$_3$.
I wrappers dovranno quindi preoccuparsi di leggere gli schemi sorgenti e di tradurre tali schemi tramite la sintassi ODLi$_{3;}$ successivamente verranno passati al Global Schema Builder (GSB), modulo apparteneente al mediatore, che combina, integra e si occupa di dare una rifnitura ai dati provenienti dal Wrapper.

3.3.1 Il linguaggio ODL

Il linguaggio ODL (Object Definition Language) e' il linguaggio utilizzato per la specifica di schemi ad oggetti proposto dal gruppo di standardizzazione ODMG-93 [17, 18] e riconosciuto come un standard dalla comunità informatica. Tale linguaggio svolge, nell'ambito degli ODBMS, le funzioni di definizione degli schemi che nei sistemi tradizionali sono assegnate al Data Defnition Language.
Caratteristiche fondamentali di tale linguaggio, come per altro anche degli altri linguaggi che si basano sul paradigma ad oggetti, sono:

- definizione di tipi classe e tipi valore
- distinzione tra intensione ed estensione di una classe di oggetti
- distinzione di attributi tra semplici e complessi
- definizione di attributi atomici e collezione (set, list, bag)
- definizione di relazioni binarie con relazioni inverse
- dichiarazione delle signature dei metodi.

La sintassi di ODL e' un ampliamento della sintassi prevista per il linguaggio IDL (Interface Defnition Language) che e' il linguaggio sviluppato nell'ambito del progetto CORBA (Common Object Request Broker Architecture).

3.3.2 La estensione I^3: il linguaggio ODLI3

ODLI$_3$ deriva direttamente da ODL e ne rappresenta una estensione in accordo con le raccomandazioni della proposta di standardizzazione per i linguaggi di mediazione elaborata presso l'Università del Maryland dal gruppo di lavoro I^3 costituitosi in tale ambito. Infatti il linguaggio ODL, pur essendo ben progettato per la conoscenza relativa ad un singolo schema ad oggetti, risulta insufficiente per la descrizione di un insieme si sorgenti di basi di dati eterogenee quale quello richiesto da MOMIS e quindi può essere considerato solamente come una buona base di partenza nel progetto di integrazione.
Seguendo il lavoro dell'I^3 Workgroup si e' potuto quindi ottenere un linguaggio che fosse in grado di supportare sia dei sistemi complessi, quali possono essere quelli ad oggetti, sia i modelli più semplici, quali i file strutturati.
Malgrado questi obiettivi si e' però sempre cercato di mantenere un certa uniformità rispetto al linguaggio ODL e quindi si è lavorato sulla sintassi in modo da introdurre il meno possibile delle divergenze. Il risultato di questo lavoro ha portato appunto ad ODLI$_3$ che ha aggiunto nuovi scopi, rispetto a quanto previsto in ODL e tutt'ora presenti, di descrizione che possono essere riassunti nei seguenti punti:

- è stata data al wrapper la possibilità di indicare, per ogni classe, il nome del sorgente di appartenenza ed il relativo tipo;
- nel caso di sorgenti relazionali e' possibile definire, per le classi appartenenti a tali sorgenti, le chiavi candidate e le eventuali foreignkey;
- sono stati previsti due nuovi costrutti quali union e optional che servono rispettivamente per avere strutture dati alternative e per indicare la natura opzionale di un attributo. Queste caratteristiche sono in accordo con la strategia utilizzata per la descrizione di dati semistrutturati;
- è stata introdotta la possibilità di definire delle grandezze locali e delle grandezze globali;
- viene supportata la dichiarazione di regole di mapping, dette mapping rule fra grandezze locali e grandezze globali;
- e possibile la definizione di regole di integrità denominate *if then rule* che possono essere applicate sia sullo schema globale che sui singoli schemi locali;
- è stato introdotto il supporto per la definizione di relazioni terminologiche di sinonimia (SYN), ipernimia (BT), iponimia (NT) ed associazione (RT);

- il linguaggio può essere automaticamente tradotto nella logica descrittiva OLCD usata dagli ODB-Tools e quindi utilizzarne le capacità, nei controlli di consistenza e nell'ottimizzazione semantica, delle interrogazioni.

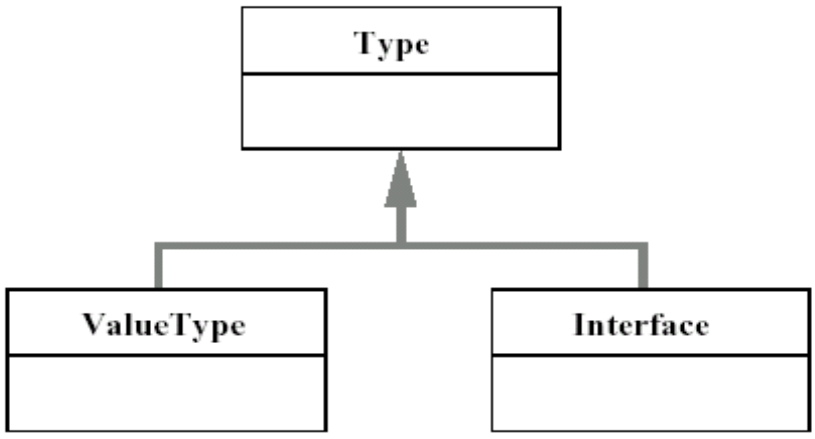
Nell'appendice A è riportata in BNF la sintassi del linguaggio ODLI$_3$.
Andiamo adesso ad illustrarne un po' più in dettaglio le caratteristiche principali e soprattutto le estensioni che sono state introdotte rispetto alla sintassi di ODL [15] [16].

3.3.3 I Tipi di Dati

In ODLI$_3$, secondo quanto presente anche nella sintassi del linguaggio ODL, è prevista una distinzione fondamentale nei tipi di dati e precisamente:

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

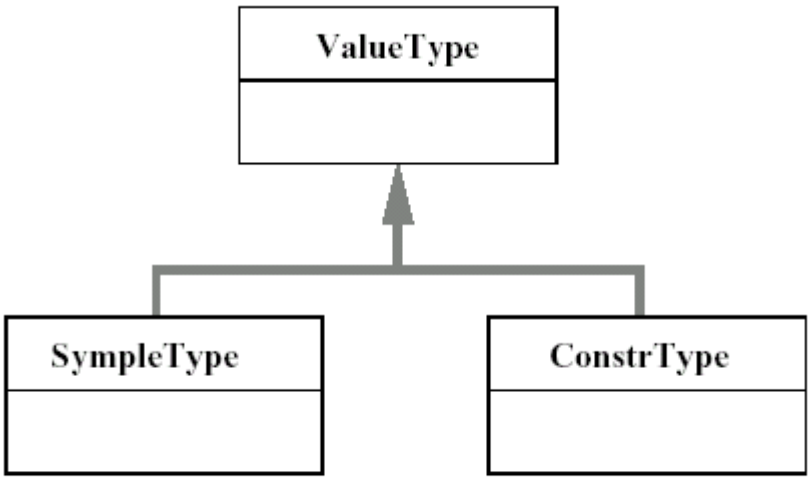
I primi sono caratteristici delle variabili e degli attributi semplici ed ogni istanza di questo tipo non ha un proprio identificatore, ma la sua unica proprietà è il suo valore. Diversamente gli attributi complessi, o comunque gli oggetti in generale, sono istanze di classe caratterizzate da un proprio OID, una propria interfaccia ed un proprio comportamento. Questa differenza è rappresentata anche nella figura seguente:



3.3.3.1 I Tipi Valore

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

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



SYMPLETYPE

Appartengono a questa categoria tutti i dati di tipo atomico quali possono essere:

- **i tipi predefiniti**: integer, string, float, char, boolean, anytype, octettype;
- **i vari tipi di collezione:** TemplateType (set, list, bag, array);
- **i tipi DefinedType** definiti dall'utente.

I tipi predefiniti sono i tipi semplici di base. Fra questi si trova ANYTYPE, che non pone alcun vincolo sulla struttura del documento.

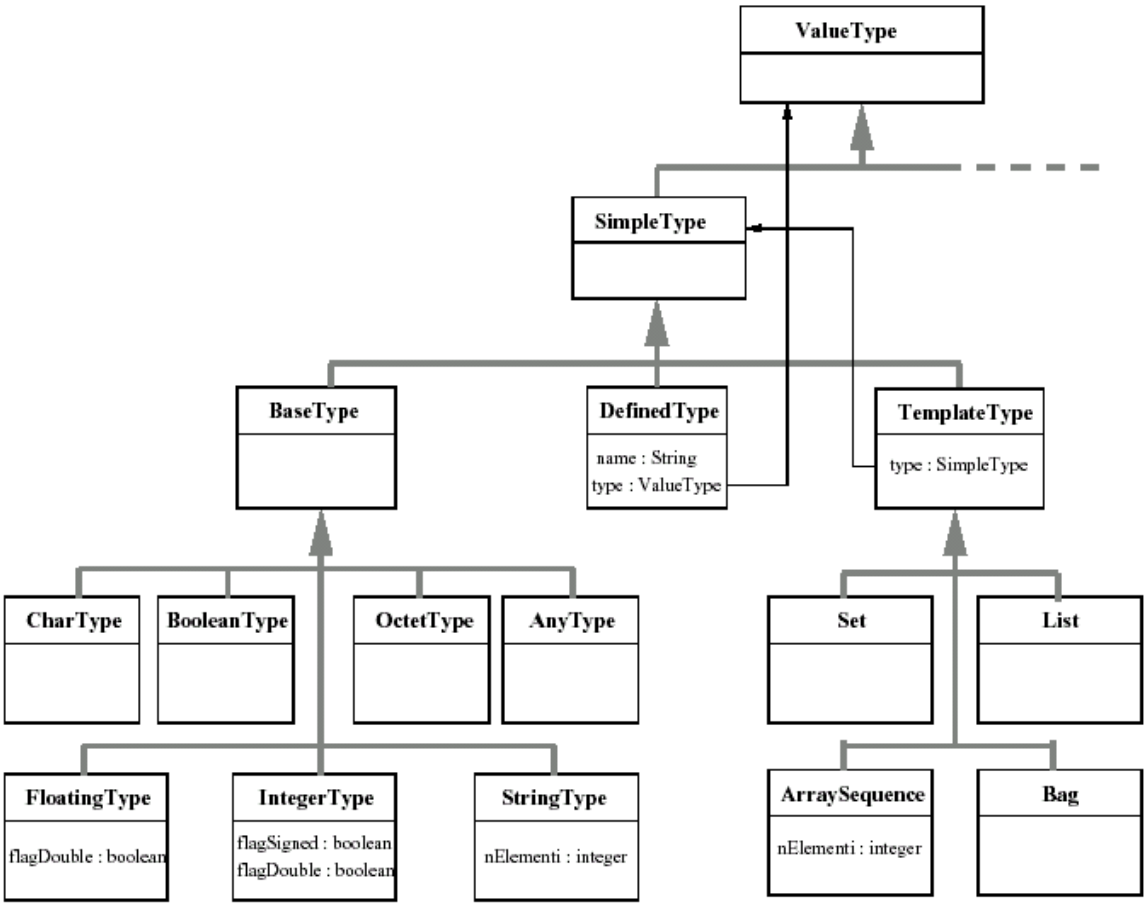
I TemplateType non sono altro che una collezione di istanze tutte omogenee tra di loro di tipo SimpleType.

- **set:** collezione non ordinata di elementi in cui non sono permessi duplicati
- **bag:** collezione non ordinata di elementi che pu`o contenere duplicati
- **list:** collezione ordinata di elementi
- **array:** collezione ordinata di un numero fissato di elementi che possono essere referenziati per posizione

Mentre per i **DefinedType** la sintassi ODL prevede una definizione che segue le stesse regole del linguaggio ANSIC: in particolare un tipo definito possiede un nome proprio ed un tipo mappato (è consentito solo il tipo-valore) ed eventualmente un insieme di valori interi, nel caso si tratti di un array e che stanno ad indicare le dimensione dell'array stesso. La particolarità che il tipo riferito sia un generico ValueType piuttosto che un SimpleType consente la dichiarazione di variabili strutturate, ad esempio, tramite una sintassi semplificata: prima si definisce un tipo nuovo, poi le variabili secondo la sintassi dei tipi atomici. Una conseguenza del fatto che i TemplateType consentano una collezione di istanze solamente SimpleType è quella di non rendere possibile una dichiarazione diretta di una variabile come set di struct. Tale risultato è possibile però ottenerlo grazie ad un artificio che consta di due passi: prima di tutto si crea una definizione di un nuovo tipo e poi la si dichiarazione della collezione desiderata. Il tutto è meglio illustrato nel seguente esempio:

```
typedef struct
{
int a,b;
boolean x;
} tipostruct;
set<tipostruct> coord1, coord2;
```

La struttura dei SimpleType è graficamente illustrata nella figura seguente:



CONSTRTYPE

Anche gli attributi di questo tipo possono essere dettagliati ulteriormente in:

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

dove i primi due sono caratterizzati da un tagName opzionale che da' la possibilità all'utente di dichiarare diverse variabili dello stesso tipo senza essere costretto tutte le volte a ridescrivere l'intera struttura. Non bisogna però cadere nell'errore di considerare il tagName come un nome di tipo in quanto non si tratta di un elemento a se stante, ma piuttosto deve essere sempre accompagnato dal **tipo-struttura**.

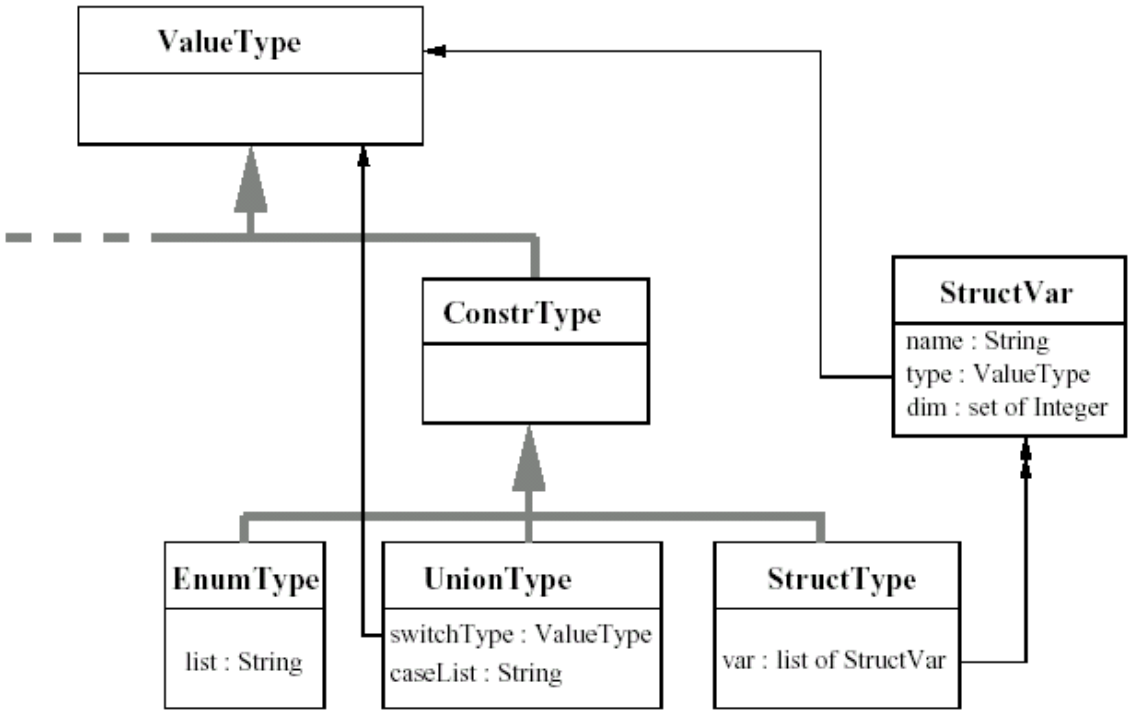
Il tutto è evidenziato nel seguente esempio:

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

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

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

Allo stato attuale il tipo **UnionType** è sensibilmente semplificato. Infatti, a parte il tipo della variabile di switch, tutto ciò che riguarda la definizione non viene interpretato, ma viene solamente archiviato così come è stata definita. Ovviamente la correttezza sintattica della definizione deve sempre essere garantita.
Invece i tipi **EnumType** hanno un comportamento leggermente diverso da quanto appena illustrato, infatti in essi non c'è altro che un semplice elenco definito di valori e la variabile associata dovrà assumere solamente uno dei valori elencati nella dichiarazione.
L'intera struttura dei ConstrType può essere agevolmente illustrata nella figura seguente:

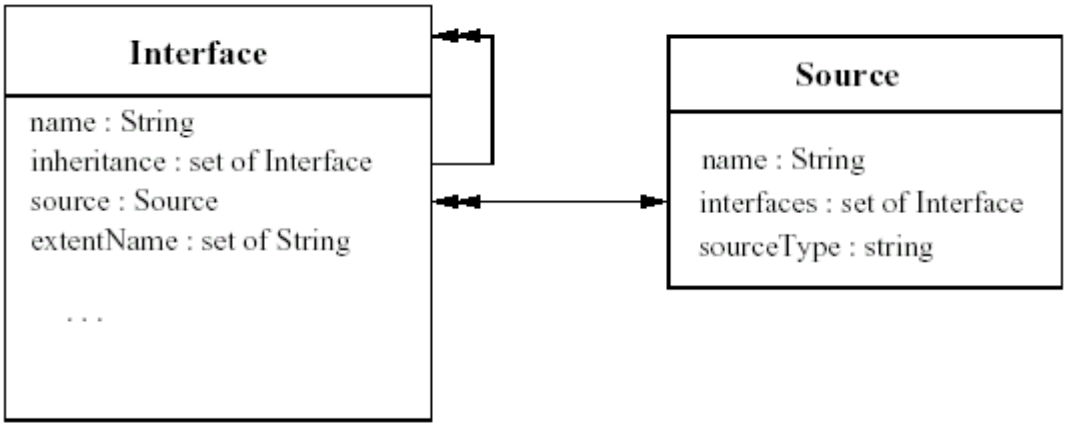


3.3.3.2 I tipi classe

I tipi classi, più brevemente denominati Classi, sono identificati dalla parola chiave interface e non sono altro che un insieme di diverse dichiarazioni. Ogni classe è caratterizzata da un proprio nome, dal riferimento al sorgente che l'ha generata ed eventualmente dal riferimento ad una eventuale superclasse da cui deriva secondo un criterio di ereditarietà ammessa per questo linguaggio.

EREDITARIETA'

Nella figura seguente viene illustrato come viene mappato il discorso della ereditarietà all'interno di una classe ODLI$_3$ ed evidenzia come per ogni Interface sia ammessa l'ereditarietà multipla, cioè sia ammessa la possibilità di avere una o più superclassi. Questa proprietà viene rappresentata attraverso la collezione complessa inheritance che mappa sulla stessa classe Interface.



SOURCE

Nel linguaggio ODLI$_3$ _e stato aggiunto un costrutto per la dichiarazione del sorgente che contiene la descrizione della classe. Tale costrutto mi indica, come visualizzato sempre nella

figura , che ogni classe appartiene ad un sorgente caratterizzato da un nome, e dal tipo del sorgente stesso (relazionale,a oggetti, file, semistrutturato).
Poiche' non viene a priori esclusa la presenza di classi caratterizzate da uno stesso nome per ottenere una identificazione univoca della classe stessa occorre ricorrere alla coppia nomeSource-nomeInterface, cioè al mantenimento anche all'interno dell'oggetto Source, delle lista di tutte le classi che ne fanno parte. Questo meccanismo risulta efficace anche dal punto di vista di una milgiore navigazione nella struttura dei dati.

EXTENT

Per facilitare il reperimento indicizzato, e di conseguenza rapido, delle informazioni da parte del DBMS, il linguaggio ODLI$_3$ da' la possibilità di dichiarare nella classe un'estensione tramite la parola chiave extent. Una estensione rappresenta un insieme di tutte le istanze delle classe all'interno di un particolare database e, nel caso di classe globale, potrebbe tornare utile la dichiarazione di più estensioni associate alla classe stessa attraverso l'uso di un set di nomi in formato stringa.

KEY E FOREIGNKEY

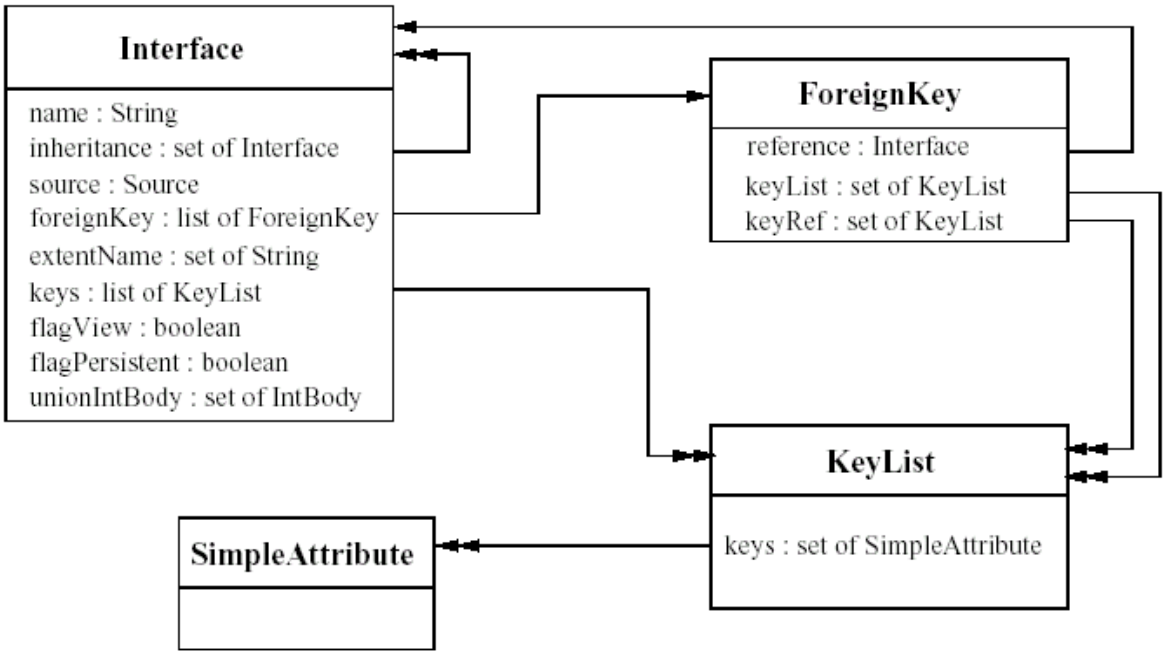
Nella sintassi di ODLI$_3$ è stata inserita anche la possibilità di definire delle chiavi candidate, delle chiavi semplici e delle chiavi composte. Infatti per ogni istanza dell'oggetto KeyList corrisponde una chiave candidata che coinvolge un solo attributo nel caso di chiave semplice, oppure più attributi nel caso sia necessario descrivere una chiave composta. Poichè tale linguaggio deve essere in grado di descrivere anche dei database relazionali è stata inserita una sintassi per la descrizione delle foreign key in modo da non perdere le informazioni che il modello relazionale comporta per la realizzazione delle gerarchie di aggregazione. Tale prerogativa consente quindi di mantenere l'informazione in un oggetto ForeignKey in cui viene riportata la lista degli attributi componenti, la lista dei medesimi attributi appartenenti alla classe riferita e la classe riferita stessa.
Un esempio in ODLI$_3$ potrebbe essere dato dalle seguente linee di codice:

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

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

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

Il tutto è graficamente illustrato nella figura seguente:

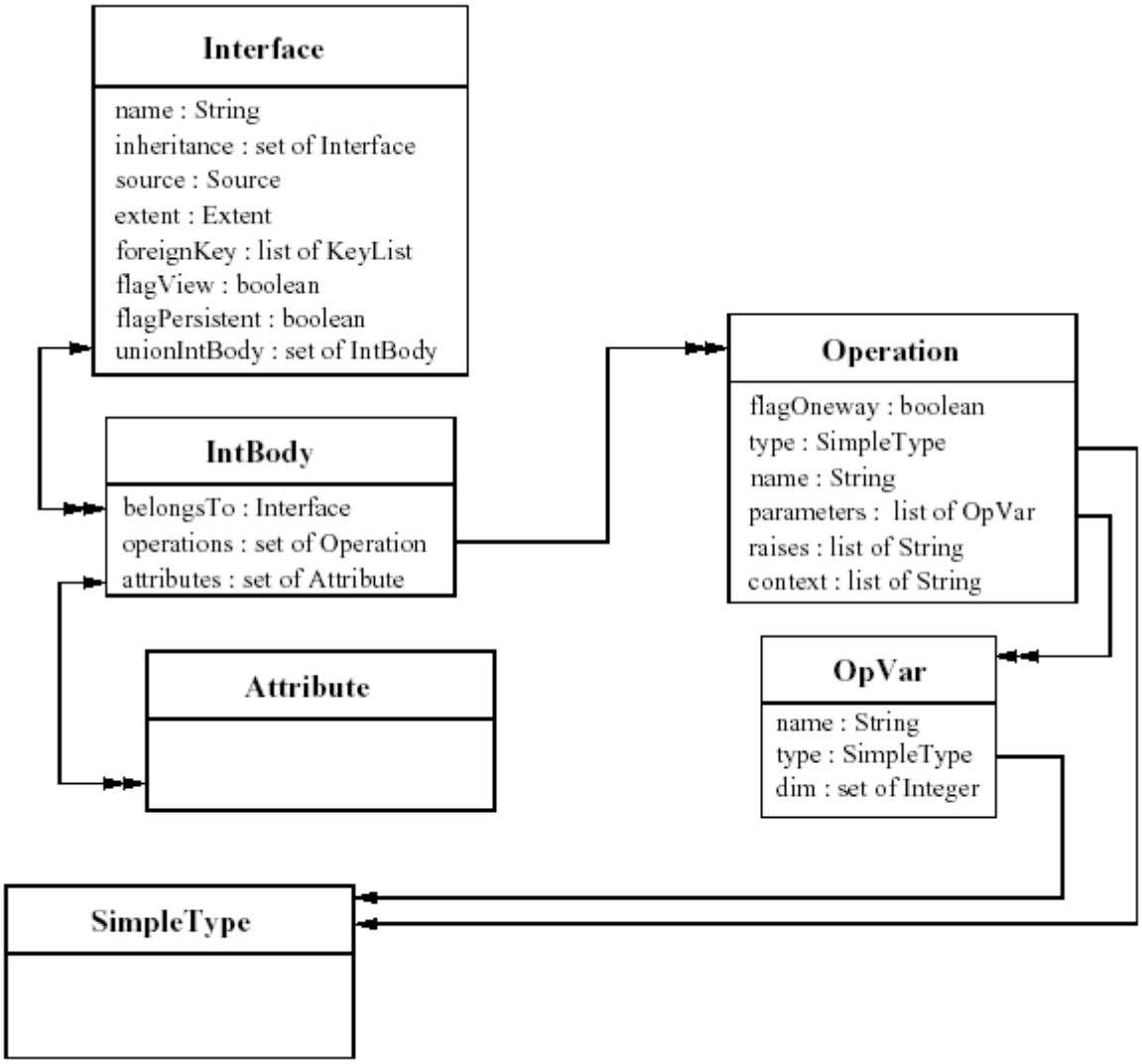


INTERFACE BODY

Nel Body dell'interfaccia vengono descritti tutti i metodi e gli attributi che sono caratteristici della classe (o vista). Diversamente da quanto previsto per il linguaggio ODL, in ODLI3 è prevista la possibilità di definire strutture di dati semistrutturati che, per definizione ed a differenza dei dati strutturati, non devono seguire una rigida formattazione. I dati semistrutturati, infatti, si prestano a rappresentare lo stesso aspetto della realtà attraverso contenuti fortemente differenziati a livello strutturale; attributi che generalmente assumono questa caratteristica potrebbero essere l'indirizzo, l'ora oppure la data. Per quest'ultimo caso, ad esempio, ci si potrebbe trovare di fronte ad un formato dato solamente da una stringa in cui sono racchiuse tutte le informazioni, oppure ad un formato composto da un insieme di campi quali il giorno il mese e l'anno. Il linguaggio ODLI$_3$ mette a disposizione, per questi casi, lo speciale costrutto union che consente di definire più specifiche alternative per una stessa interfaccia. Tale costrutto permette di confrontare degli oggetti che rappresentano due istanze della medesima realtà, ma che sono descritte attraverso scelte strutturali differenti. Riprendendo l'esempio dell'attributo data ci si potrebbe trovare di fronte a delle righe di sorgente come le seguenti:

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

Questo esempio mette anche in evidenza il fatto che ad una stessa classe possono appartenere più attributi che hanno lo stesso nome, uno diverso per ogni implementazione consentita.
La struttura di una InterfaceBody può essere illustrata tramite la figura seguente:



OPERAZIONI

Nella stessa figura precedente sono rappresentate anche le operazioni, che non sono altro che i metodi della classe o meglio la loro signature. Infatti la classe Operation contiene le informazioni riguardanti il nome ed il tipo restituito dal metodo ed una lista di parametri passati (di tipo OpVar) ognuno avente un nome, un tipo ed un elenco di eventuali dimensioni di array. Unica cosa da sottolineare per questa classe riguarda i tipi ammessi per i parametri passati e per il tipo restituito dal motodo stesso che sono previsti dalla sintassi siano istanze della classe SimpleType.

ATTRIBUTI

Sono previsti due categorie di attributi differenziati dal tipo a cui appartengono:

- gli attributi **semplici** di tipo-valore
- gli attributi **composti** di tipo-classe.

Oltre a questo viene data anche la possibilità di dichiarare delle relationships, cioè degli attributi complessi che possiedono anche la relazione inversa, e di utilizzare un costrutto per definire delle mapping rule in grado di legare grandezze appartenenti allo schema integrato con grandezze degli schemi locali.
I tre tipi di attributi possibili sono:

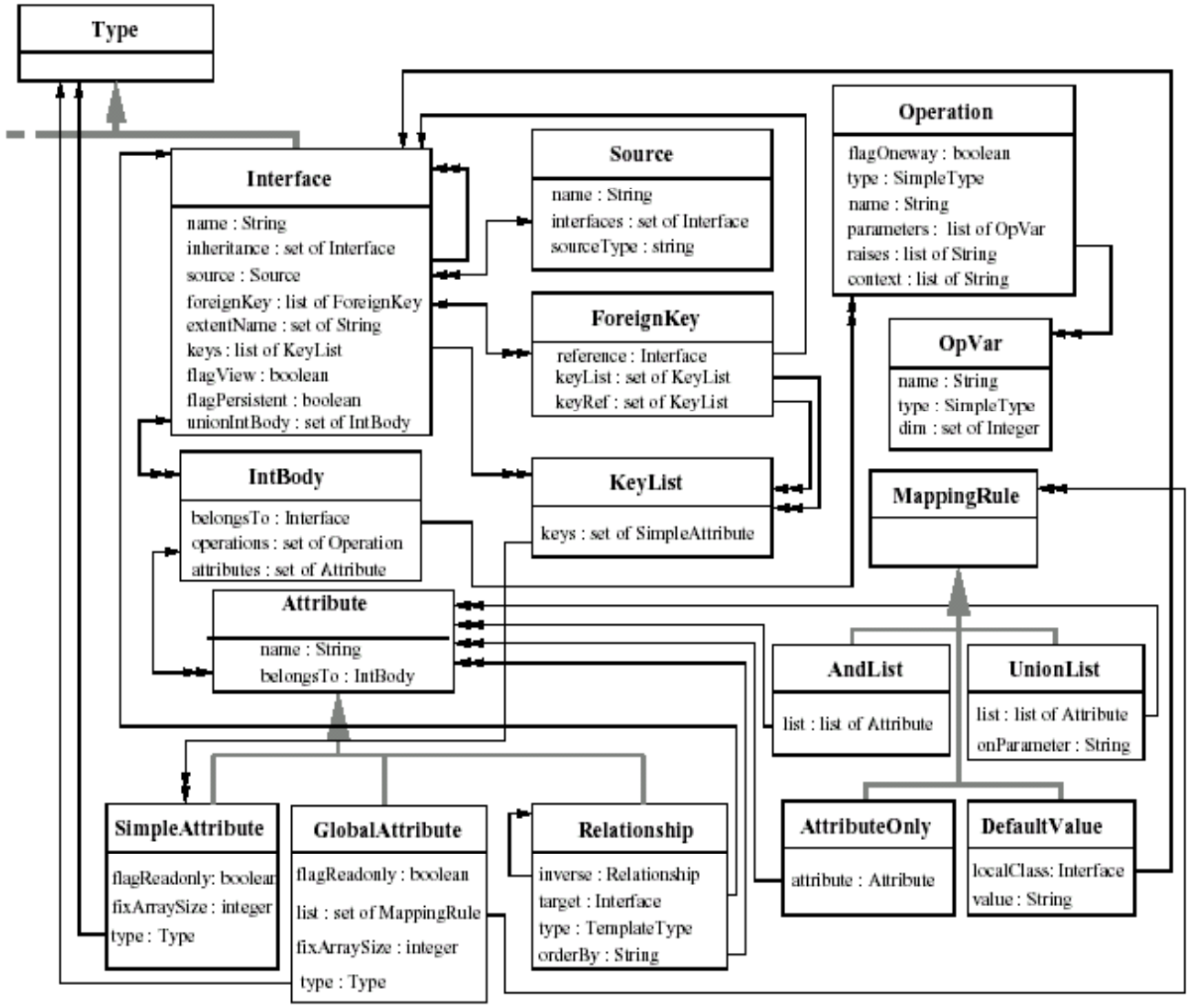
- I SimpleTypeAttribute caratterizzati da un tipo generico denomiato Type, un ag booleano per indicare se si tratta di attributi di sola lettura ed un set di eventuali indici di array.
- Le Relationship che servono per mappare una interfaccia e contengono le informazioni sulla inversa attraverso un puntatore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

MAPPINGRULE

Qualora la dichiarazione di un attributo sia seguito da un insieme di regole di mapping, l'attributo è automaticamente considerato globale.
Preso un attributo globale è possibile definire diverse regole di mapping a seconda della caratteristiche degli attributi stessi che vengono coinvolti. In particolare si possono presentare i seguenti casi:

- corrispondenza tra l'attributo globale ed un solo attributo locale, che è anche il caso più semplice. Se ricade in questa eventualità, infatti, il sistema possiede le informazioni necessarie per realizzare in modo automatico il meccanismo di mapping e fa coincidere il nome con quello dell'attributo locale;
- corrispondenza tra l'attributo globale ed un insieme di attributi locali tutti appartenenti a classi locali differenti, ma in fusione tra di loro. Come nel caso precedente il mapping si ottiene in maniera automatica, ma il nome potrebbe essere assegnato dal progettista per identificare la grandezza con una denominazione appropriata.
- nel caso ci sia una corrispondenza tra l'attributo globale e la fuzione di diversi attributi locali, magari appartenenti alla stessa classe locale, possiamo distinguere due sottocasi:
    1. gli attributi locali sono concatenati tramite una condizione and il che implica che l'attributo globale sia ottenuto dal concatenamento degli attributi locali stessi. Tipico esempio di questa eventualità potrebbe essere data dall'attributo globale Owner.name che mappa in Person.first name e Person.last name.
    2. gli attributi locali sono legati da un costrutto del tipo union il che viene risolto tramite la corrispondenza tra l'attributo globale ed uno solo alla volta degli attributi locali in union. Per poter individuare con esattezza quale di questi parametri locali debba essere mappato si è ricorso ad un terzo attributo locale rappresentato nella struttura di figura successivada onParameter.
- nel caso non esista una corrispondenza tra l'attributo gloable e gli attributi locali potrebbe essere necessario esprimere un metaconcetto tramite un valore di default.

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



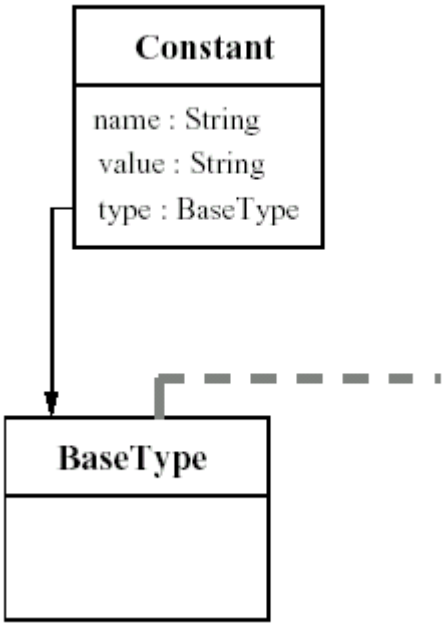
3.3.3.3 Costanti

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

Nella rappresentazione della figura successiva sono identificabili tre attributi:

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

Unica nota da segnalare per questa classe riguarda il formato di archiviazione del valore che è di tipo stringa e quindi dovrà essere convertito nel formato corretto tramite un operazione di cast da chi si occupa della consultazione della struttura dati.



3.3.4 Relazioni Terminologiche

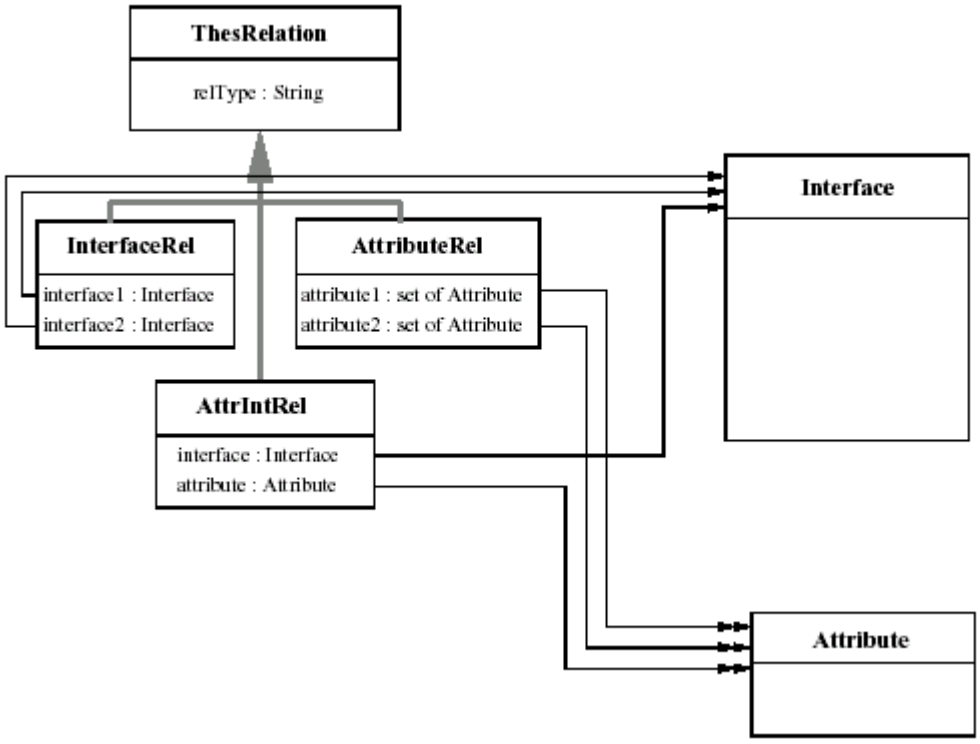
Altra aggiunta nella sintassi del linguaggio ODLI$_3$ riguarda la possibilità di inserire delle relazioni terminilogiche che possono intercorre tra classi, tra attributi, oppure miste tra gli attributi e le classi. Sarà poi il parser che si occuperà di interpretare e capire quali siano gli attributi e/o le classi coinvolte in tali relazioni.
Fondamentalmente i tipi di relazione possibili sono :

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

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

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

Il tutto è mostrato nella figura seguente:



3.3.5 Regole di Integrità

Ultima particolarità della sintassi di ODLI$_3$ è data dalle regole di integirtà di tipo if-then che devono essere verificate per ogni istanza del database. Tale sintassi contiene due forme distinte e precisamente:

- rule *nomerule*

forall *iteratore* in *collezione* : *antecedente* then *conseguente*

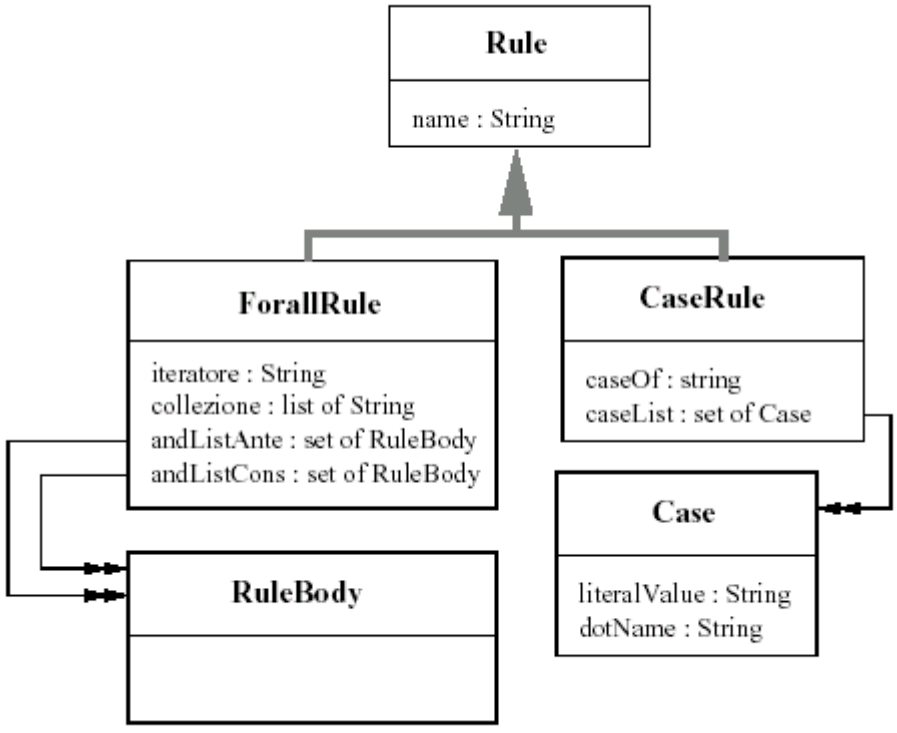
oppure:

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

Dove i termini che rappresentano gli attributi della classe Rule assumono il seguente significato:

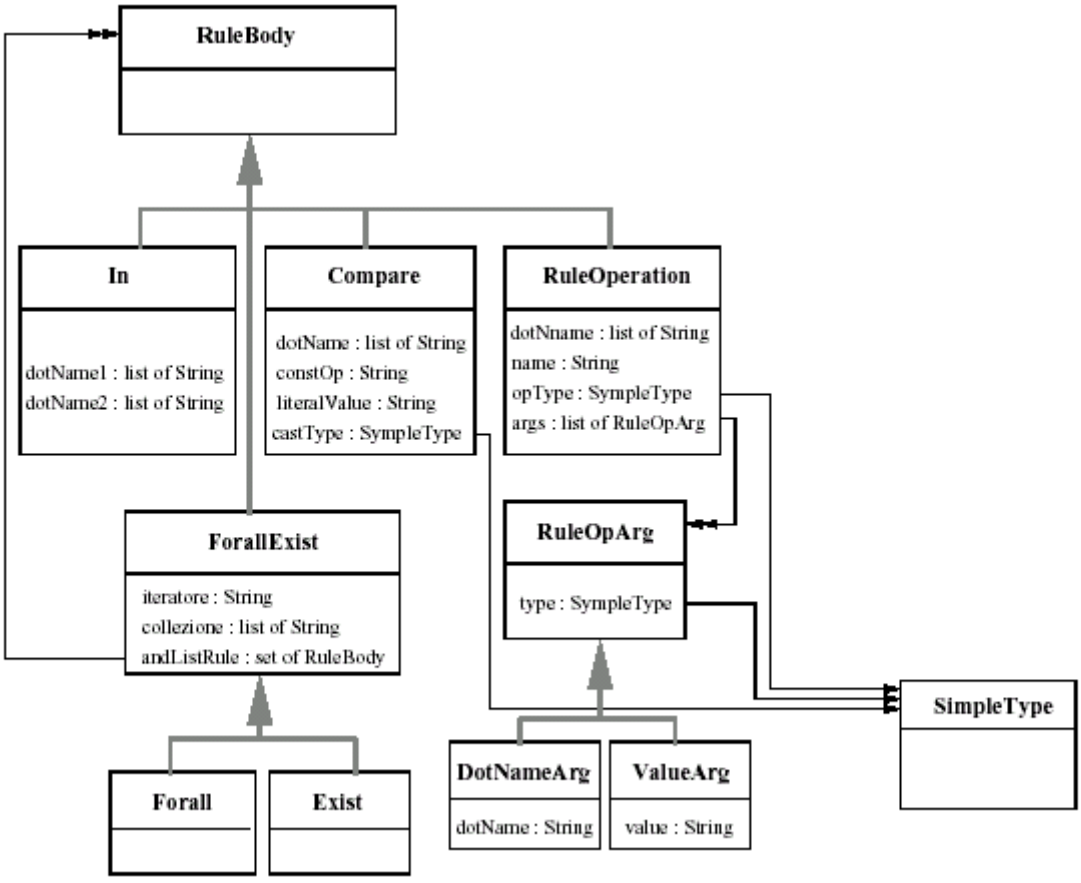
- nomerule: è il nome della regola
- iteratore: è in identificatore che rappresenta l'istanza.
- collezione: è un insieme di istanze e può essere una classe o parte di essa.
- antecedente: è la condizione di if che si ottiene ponendo in and una serie di predicati booleani.
- conseguente: è l'affermazione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

Tutto ciò è illustrato nella figura seguente:



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

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



Ora vengono descritte un po' in dettaglio le particolarità di ognuna di queste condizioni.

In

Confronta due oggetti e ritorna un valore vero se dotName1 (ad esempio un oggetto) è contenuto nella collezione dotName2. Viene utilizzato questa denominazione per gli attributi in quanto la sintassi ODLI$_3$ prevede che contengano una lista di stringhe e che siano dei

cammini esprimili tramite la dot-notation caratteristica di alcuni linguaggi orientati agli oggetti.

Forall e Exist

Per questo predicato è prevista una sintassi identica a quanto già visto per la prima parte della rule in quanto è previsto che si faccia una chiamata ricorsiva a predicati booleani della classe RuleBody. La differenza tra questi due predicati consiste nel numero di match necessari perchè la condizione ritornata sia vera, in particolare:

- Forall *iteratore* in *collezione* : condizioni in and ritorna vero se per tutte le istanze valgono tutte le condizioni in and
- Exist *itaratore* in *collezione* : condizioni in and ritorna vero se esiste almeno una istanza per cui valgono tutte le condizioni in and.

Compare

Si tratta del classico operatore di confronto descritto in più di un linguaggio e che ritorna il valore vero o falso a seconda che il confronto tra il valore di una varaibile (ad esempio un attributo) ed il valore di un letterale sia rispettivamente verificato o meno. Gli operatori di confronto sono i classici segni matematici =, <, ≤, >, ≥, mentre l'attributo CastType serve per fare, quando richiesto, il cast del valore letterale in modo da rendere possibile il confronto con la variabile specificata. In questi casi contiene il tipo della variabile che entrerà nel confronto.

RuleOperation

Questo confronto coinvolge una variabile (dotName) e il valore ritornato da una funzione invocata (chiamata operatore) e ritorna vero se l'uguaglianza tra questi due operandi risulta verificata.
Gli altri attributi della classe RuleOperation sono:

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

CAPITOLO 4
Confronto fra DTD, XML Schema e ODLI$_3$

4.1 L'importanza di XML: Extensible Markup Language

La definizione di XML [19,20] aveva come abbiettivo originario il superamento dei limiti di HTML relativamente al markup per il Web: nella sua natura di metalinguaggio, (cioè di linguaggio per la definizione di altri linguaggi), ha la possibilità di esprimere nuovi tag senza bisogno di ridefinire una nuova versione del linguaggio. In questo consiste, fondamentalmente, la sua caratteristica di estensibilità.

Tuttavia XML ha avuto una diffusione ed un ambito di applicazione più vasto di quanto, probabilmente, avessero immaginato i suoi progettisti. La sua flessibilità e il suo supporto, praticamente su ogni piattaforma, lo hanno portato ad essere utilizzato in ambiti diversi da quello originario: XML è diventato uno standard universale per lo scambio di dati nel Web, offrendo un semplice metodo per integrare applicazioni distribuite che forniscono sofisticati servizi, come l'E-commerce.

La libertà con cui possiamo inventare nuovi tags e la netta separazione dei dati dalla loro rappresentazione, fa si che XML sia il linguaggio ideale per la descrizione di informazioni di vario tipo. E' da sottolineare che la libertà di invenzione di nuovi tags non crea "situazioni anarchiche", in quanto è una libertà prevista dalla definizione del linguaggio stesso ed è regolata da norme ben precise, elencate nel riquadro sottostante.

Regole per documenti XML ben formati e validi.

- Un documento XML è **VALIDO** se fa riferimento ad una DTD (Document Type Definition), o ad uno Schema, e ne segue le regole grammaticali.

- Un documento XML è **BEN FORMATO** se segue le seguenti regole:
  1. Unica radice. Ogni documento deve includere un unico elemento di massimo livello (radice) che contenga tutti gli altri elementi nel documento
  2. Chiusura dei tag. Ogni elemento deve avere un tag di chiusura o utilizzare una sintassi abbreviata per i tag vuoti (/>)
  3. Nidificazione dei tag. I tag di apertura e chiusura degli elementi annidati non devono sovrapporsi.
  4. Distribuziuone tra maiuscole e minuscole. XML considera diversi i caratteri minuscoli e maiuscoli; di conseguenza se si usano le maiuscole per un tag di apertura bisognerà usarle anche per quello di chiusura
  5. Valore degli attributi. Il valore degli attributi deve essere sempre racchiuso tra apici singoli (‘) o doppi (“).

La precedente osservazione sulla "libertà controllata" e sulla possibilità di utilizzare strumenti standard per interpretare documenti XML è il punto fondamentale in base al quale si completa l'universalità del linguaggio. Non solo XML consente di descrivere dati liberamente, ma anche di interpretarli in maniera automatica tramite tool standard. Per analizzare un documento che contiene i tag più fantasiosi, utilizziamo, sempre, parser XML standard, non abbiamo bisogno di un nuovo strumento per ogni nuovo gruppo di tag o per ogni nuova descrizione di dati.

Questo aspetto è molto importante in quanto consente di vedere la capacità descrittiva di XML nell'ottica della portabilità dei dati e dell'interoperabilità tra le applicazioni , indipendentemente dalla piattaforma utilizzata.

In questo senso XML può essere visto come una sorta di lingua franca per l'interscambio di dati tra le applicazioni e, quindi, per una prima forma di interoperabilità.

La capacità descrittiva di XML non è limitata ai dati: la potenza espressiva di questo linguaggio ci consente di descrivere anche invocazioni di procedure e di funzioni. Questa possibilità consente non solo lo scambio di dati tra applicazioni diverse, ma, addirittura, l'invocazione di procedure remote tra applicazioni diverse. Quest'ultime possono girare su piattaforme incompatibili e possono essere scritte in linguaggi di programmazione diversi.

Tale possibilità ha dato origine al concetto di servizio Web (Web Service), concetto di cui si sentirà parlare sempre più spesso, soprattutto con l'avvento della .NET Platform. Il web viene trasformato in una vera e propria piattaforma di sviluppo. In un futuro non molto remoto potremo scrivere la nostra applicazione nel linguaggio che preferiamo sfruttando una o più librerie pubblicate sul Web e raggiungibili tramite HTTP: i dati da scambiare o le procedure da invocare saranno descritte in XML, e, quindi, in maniera standard.

4.2 Confronto fra DTD e ODLi3

I documenti XML la cui sintassi è stata verificata con successo sono chiamati documenti *validi*; in particolare, come si è detto in precedenza, un documento XML è considerato valido se è collegato con una DTD (*Document Type Definition,*[21,19]) o con uno schema XML e se soddisfa o la DTD o lo schema. Varie organizzazioni possono condividere una DTD per mettere in produzione un'applicazione XML; la funzione delle DTD è quella di specificare la struttura e la sintassi dei documenti, ma non il loro contenuto.

Nel capitolo seguente verranno analizzate alcune regole di traduzione per convertire una DTD in linguaggio ODLi3, ma prima ci si sofferma sulle differenze esistenti fra la struttura dei due linguaggi.
Essi sono nati con obbiettivi diversi, infatti mentre ODLi3 è nato come linguaggio in grado di descrivere una base di dati archiviata in sorgenti dal formato differente, XML nasce con l'intento di descrivere dati in modo che siano facilmente veicolati, in particolare tramite la rete Internet.
I principali elementi di diversità sono i seguenti:

- ***Strutture dati***:ODLi3 distingue fra *attribute,* quando si vuole definire un singolo nodo, e *interface* quando si vogliono definire degli insiemi di nodi. In ODLi3 è poi possibile esprimere delle strutture complesse, definire dei tipi e delle enumerazioni. In XML esistono invece solo due strutture: *element e attribute*. Inoltre non si utilizza una sintassi diversa per distinguere tra la rappresentazione del singolo nodo e la rappresentazione di insiemi di nodi. Tale distinzione viene effettuata sulla base della definizione del contenuto dell'elemento.

- ***Tipi di dato:*** il linguaggio XML non supporta tipi di dato, ma ogni nodo terminale ha un unico tipo di valore denominato PCDATA.

- ***Chiavi primarie, esterne, candidate:*** in ODLI3 ogni elemento può assumere funzione di chiave primaria, esterna o candidata, invece XML non applica il concetto di chiave, ma di identificatore (ID) e di riferimento ad un identificatore (IDREF), qualificazioni che non vengono assegnate agli elementi ma a loro specifici attributi. Inoltre la sintassi XML non consente di definire in modo specifico le coppie ID, IDREF, ma unicamente impone che ad ogni attributo IDREF corrisponda un attributo ID, non indicando quale.

- ***Composizione di strutture:*** XML permette di creare una stuttura dati che sia il risultato dell'unione di strutture precedentemente create. In questo modo una DTD può essere ottenuta come unione di molteplici DTD e può essere dunque collocata su molteplici file separati. Tutto ciò è reso possibile utilizzando il costrutto entità, non supportato in ODLi3, che non può svolgere tali funzioni.

- ***Concetto di ordine:*** in XML è possibile esprimere il concetto di ordine per quel che riguarda gli elementi, in ODLi$_3$ non è previsto.

In sintesi si può affermare che, mentre XML permette una migliore rappresentazione della struttura del dato, ODLi$_3$ consente una tipizzazione più precisa ed una referenziazione fra gli oggetti più accurata.
Chiaramente lo scopo che risiede dietro alla definizione di ODLi$_3$, appunto quello di descrivere basi di dati in formato differenti, non può non far pensare che esistano punti in comune tra i due linguaggi. Infatti, sempre in ODLi$_3$, è prevista la descrizione di dati semistrutturati, ed un file XML è appunto un formato che risiede in tale categoria.
Malgrado la diversa filosofia abbia portato alla creazione di sintassi differenti per perseguire ognuna il proprio scopo, i punti in comune fra le due strutture che le loro sintassi descrivono sono sufficientemente numerosi. Questi punti di contatto ci permettono di affermare che un lavoro di traduzione non sia possibile solo da un punto di vista teorico, ma anche in grado di produrre un interscambio di dati fra un linguaggio e l'altro sufficientemente completo.

4.3 Confronto fra DTD e XML Schema

Uno schema XML [19,22,23,24] è una collezione di definizioni di tipi e dichiarazioni di elementi utilizzati nel documento XML che ne costituisce un'istanza. Gli schemi sono molto più potenti e precisi rispetto alle DTD, infatti sono stati pensati per fornire quel supporto di validazione che le DTD permettono solo parzialmente, in particolare sul contenuto degli elementi e degli attributi dei documenti XML.
Le DTD hanno una sintassi particolare, diversa dall'XML, perciò sono richiesti strumenti appositi per la validazione; inoltre non distinguono tra nome del tag e tipo del tag, ed hanno solo due due tipi: complesso (cioè strutturato) e semplice (cioè CDATA o #PCDATA).
Tramite gli schemi, invece, è possibile:

- ***Utilizzare applicazioni XML per la verifica della validità dei dati espressi***; infatti gli schemi sono scritti in XML.

- ***Specificare il tipo di dati del contenuto di ciascun elemento***: viene fornita cioè una tipizzazione specifica per ogni singolo elemento, partendo dai tipi di dato primitivi e arrivando ad un ulteriore caratterizzazione dei tipi primitivi stessi. Inoltre è possibile distinguere fra elementi di tipo complesso, contenenti altri sottoelementi, e di tipo semplice, cioè terminali.

- ***Esprimere maggiori vincoli in merito al numero di occorrenze di un elemento in una struttura, e alla sequenza con la quale avvengono tali occorrenze.*** XML Schema prevede la definizione del numero di occorrenze minime e massime di un elemento e, per i tipi complessi, una sintassi che serva a dichiarare in quale ordine devono comparire nelle istanze gli elementi. Anche nelle DTD si può ottenere tale possibilità, però in maniera meno puntuale, utilizzando la sintassi definita per gli elementi di tipo children. Tali elementi, infatti, possono consistere di strutture innestate a più livelli, composte da elementi in sequenza o in alternativa. Nelle specifiche introdotte in quel contesto non è possibile controllare niente di più se non una corretta corresponsione fra le istanze e la DTD. Con le specifiche XML Schema è invece possibile caratterizzare in maniera più puntuale la struttura, in particolare definendo insiemi di elementi che possano sia essere istanziati in alternativa, sia seguendo l'ordine definito nello schema.

- ***Fornire un approccio object_oriented, permettendo di ampliare i tipi disponibili e di estenderne e precisarne le proprietà.*** E' prevista la possibilità di costruire nuovi elementi sulla base di elementi già definiti nello stesso documento o anche in altri documenti. Nella fattispecie è possibile sia estendere le caratteristiche di un elemento definito precedentemenete, aumentando la capacità espressiva, sia derivare per restrizione un elemento a partire dalla definizione di un altro elemento. Inoltre è possibile implementare il concetto di nodo equivalente, creando degli elementi analoghi sotto il profilo strutturale, ma aventi differente denominazione. In analogia con la programmmazione ad oggetti poi, il concetto di equivalenza e di derivazione può essere arricchito anche attraverao l'introduzione di elementi di tipo astratto, cioè di elementi ai quali non corrisponda un'effettiva istanza nella parte datui del documento XML, ma che vengano utilizzati per definire a partire da essi nuovi elementi.

- ***Implementare il concetto di Key e Foreign_Key*** e, a diferenza di quanto avviene per le DTD, è possibile dichiarare a quale Key un Foreign_Key si riferisce.

- ***Utlizzare il concetto di namespace*** che consente di definire dei domini di elementi, e quindi di realizzare strutture modulari che non entrino in conflitto fra loro. Ciò non è previsto nelle DTD.

4.4 Confronto fra XML Schema e Odli$_3$

In precedenza si è osservato che, per alcuni aspetti le DTD permettono una migliore rappresentazione della struttura del dato rispetto ad ODLi$_3$, tuttavia sono stati individuati anche alcuni limiti di XML rispetto a questo linguaggio. Tali limitazioni sono state superate con le innovazioni introdotte da XML Schema, e si può affermare che la semantica rappresentabile in ODLi$_3$ è un sottoinsieme di quella rappresentabile con XML Schema. Le differenze riguardano soprattutto la definizione precisa della struttura del dato che non trova analogo potere di espressione in ODLi$_3$. Si analizzano ora nel dettaglio gli elementi di distinzione .
XML Schema prevede:

- La distinzione fra elementi e attributi: in ODLi$_3$ sono definiti entrambi come attribute.

- La possibilità di specificare l'ordine in cui gli elementi devono apparire all'interno dell'elemento padre: non prevista in ODLi$_3$.

- La possibilità di creare schemi complessi che utlizzano definizioni incluse in altri: in ODLi$_3$ non è possibile fare riferimento ad oggetti esterni al documento.

- La possibilità di specificare il numero massimo e minimo di occorrenze: in ODLi$_3$ viene espressa solo la cardinalità (0,n) o (0,1)

- L'utilizzo di namespace, per indicare al processore quali schemi utilizzare per la validazione del documento: non previsto in ODLi$_3$

- La possibilità di derivare tipi da altri tipi già esistenti, siano essi semplici o complessi, mediante estensione o restrizione. ODLi$_3$ prevede solo la derivazione di tipi complessi da tipi complessi mediante ereditarietà, ma non la derivazione di tipi complessi da tipi semplici.

- La definizione di elementi a contenuto misto, che possono contenere sia altri elementi figli che testo semplice: non previsti in ODLi$_3$.

- l'utilizzo di Key e Keyref, analoghe alle Key e Foreign_Key di ODLi$_3$.

Qualora si voglia tradurre da ODLi$_3$ a XML Schema, tale traduzione potrà conservare totalmente i vincoli che sono stati definiti, senza l'ausilio di algoritmi di traduzione particolarmente complessi. Traducendo invece da XML Schema a ODLi$_3$ si verificherà una perdita di informazioni, poiché la potenzialità espressiva di ODLI$_3$ è meno puntuale.
Nel capitolo 6 verranno esposte delle regole ricavate per effettuare tale tipo di traduzione in modo da perdere la minore quantità di informazioni possibile, ricorrendo a volte a degli artifici, dove ODLi$_3$ non consentiva un atraduzione immediata. Tuttavia, nell'ottica del progetto MOMIS, qualora si vogliano integrare diversi schemi strutturali, si può osservare che la predita di informazioni che un Wrapper XML Schema basato su tali regole genererebbe, nel passaggio fra i due linguaggi, non provoca forti problemi. Lo stesso linguaggio ODLI$_3$ viene infatti utilizzato, non per descrivere meticolosamente delle strutture, ma per descrivere delle modellazioni di strutture dati.

CAPITOLO 5

Esportazione DTD In ODLi3

5.1 La Traduzione

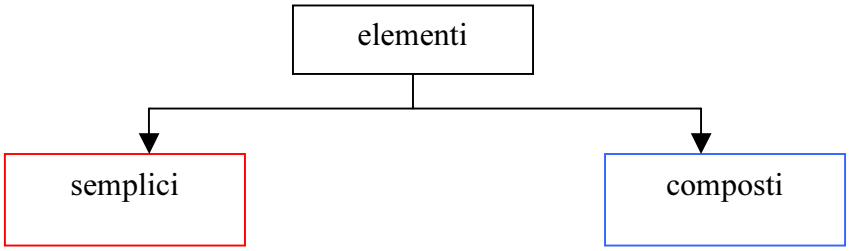
IL lavoro di traduzione riguarderà solamente la DTD, in quanto è da essa che si ricavano informazioni relative alla stuttura dei dati, ed avrà il compito di riprodurre, dove possibile, un'analoga struttura ODLi3 in grado di descrivere senza ambiguità i dati in formato XML.
L'obbiettivo è quello di approfondire il lavoro svolto in [15] quando ancora XML non era una Recommendation del W3C, analizzando le innovazioni introdotte recentemente e fornendo una loro possibile traduzione in $ODLi_3$.
La prima operazione sarà quella di prevedere il nome del file, che verrà lasciato inalterato, modificando l'estensione.
Per quanto riguarda il documento, l'intera struttura può essere descritta tramite un modello DOM, caratterizzato da una radice di origine da cui si diramano i vari nodi. Tale radice è la base di partenza della traduzione, infatti può essere convertita nell'interfaccia principale di ODLi3 e tutti i nodi successivi che si incontrano possono essere tradotti come dei tipi definiti dall'utente, in cascata, fino al raggiungimento dei vari tipi atomici, che non sono altro che le foglie dell'albero associato al DOM.

5.1.2 Elementi Nelle DTD

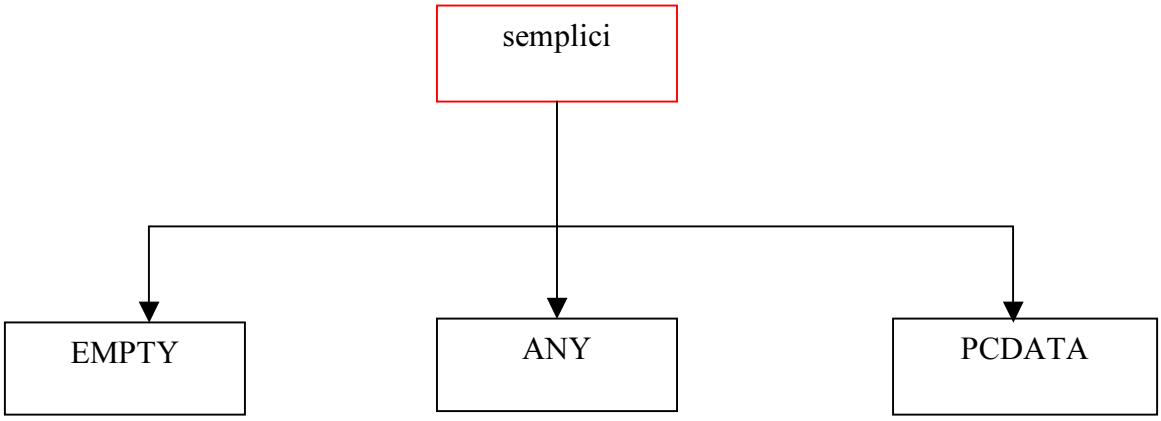
Per dichiarare la sintassi di un elemento in una DTD si usa il costrutto seguente:

```
<!ELEMENT NAME CONTENT_MODEL>
```

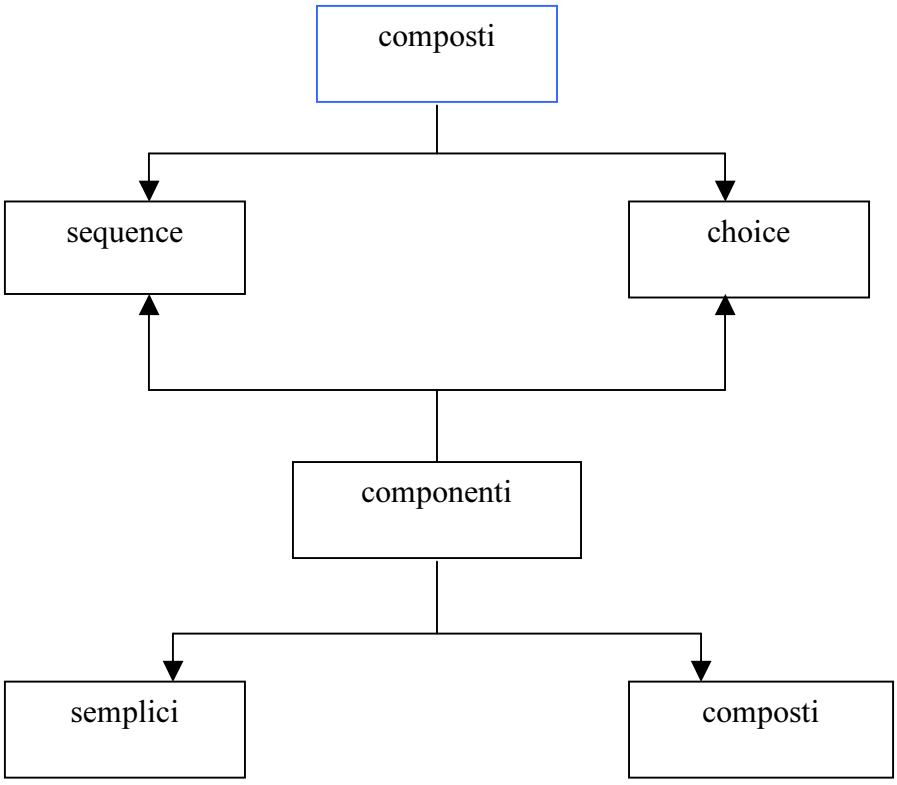
Name è il nome dell'elemento che si sta dichiarando;
content_model può essere impostato a EMPTY o ANY, può avere un contenuto misto (altri elementi e anche dati di tipo carattere), oppure degli elementi figli.
I tipi di elementi previsti nelle DTD sono visualizzati nello schema seguente:



Descrizione elementi semplici



Descrizione elementi composti



Chiaramente i componenti non sono altro che elementi, che possono essere perciò di tipo semplice o composto.

Poiché si è detto che per prima cosa di deve effettuare la traduzione dell'elemento radice, verranno ora analizzati gli elementi composti.

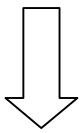
5.2 Traduzione Elementi Composti

La regola da seguire nella traduzione degli elementi composti è la seguente:

- Per ogni elemento composto di nome Ename generare un'interfaccia di nome Ename che comprenda i vari componenti. La descrizione della sorgente prevede l'assegnazione del termine semistructured, dal momento che si tratta di un file XML, seguito dal nome del file.

ES:

```
<!ELEMENT CUSTOMER(NAME,DATE,ORDERS)
 <!ELEMENT ORDERS (#PCDATA)>
 <!ELEMENT PRODUCT (#PCDATA)
 <!ELEMENT DATA (#PCDATA)>
```



```
 Interface costumer
 (source semistructured docum_file
  )
 {attribute string name;
   attribute string date;
   attribute string orders;
  }
```

La sintassi della DTD permette di esprimere elementi innestati a più livelli di profondità. Effettuando la traduzione in ODLI3 è necessario creare nuove interfacce, una per ogni livello, nelle quali rappresentare i termini dichiarati nella specifica di contenuto.

E' opportuno stabilire ulteriori regole a seconda del tipo di componente dell'elemento composto.

5.3 Traduzione Componenti/Elementi Semplici

Gli elementi semplici possono essere di tre tipi: PCDATA, ANY e EMPTY.

5.3.1 PCDATA

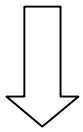
Non esiste in XML una specifica dei tipi di dati che possono essere contenuti in un elemento. Il massimo del dettaglio raggiungibile è il tipo stringa, dichiarato come PCDATA, associato ad ogni nodo terminale dell'albero.
In ODLI3 invece è possibile esprimere il tipo di dato utilizzato. Tale informazione è estremamente utile ai fini di una successiva integrazione dei concetti, per questo ritengo sia importante aggiungerla. Propongo di associare al Wrapper sviluppabile sulla base di tali regole, un'interfaccia che permetta all'utente di interagire in modo da aggiungere informazioni mancanti.
In questo caso può essere utilizzata richiedendo all'utente di selezionare, per ogni dato definito come PCDATA, il tipo più adatto fra quelli disponibili in ODLI3. Nel caso non si trovi una specificazione più conforme, l'utente può lasciare il tipo string, inserito di default.
La regola di traduzione da osservare sarà la seguente:

- Per ogni componente semplice di tipo PCDATA e con nome Ename, appartenente ad un elemento composto tradotto come interfaccia, inserire nell'interfaccia un attribute del tipo specificato dall'utente e di nome Ename.

ES:

```
<!ELEMENT CUSTOMER(NAME,DATE,ORDERS)
<!ELEMENT ORDERS (#PCDATA)>
<!ELEMENT PRODUCT (#PCDATA)
```

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



```
Interface customer
(source semistructured docum_file
 )
{attribute string name;
  attribute string date;
  attribute integer orders;
 }
```

In questo esempio l'elemento orders viene dichiarato come integer dall'utente, mentre per name e date si è scelto di confermare l'opzione di default.

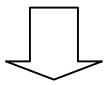
Gli elementi di tipo **EMPTY** e **ANY** verranno verranno analizzati più avanti all'interno dello stesso capitolo.

5.4 Traduzione Componenti Complessi

Viene considerato inizialmente il caso semplice di un singolo componente complesso che non presenta informazioni riguardo la sua occorrenza, e quindi di default ha cardinalità (1,1).

- Un componente complesso di nome Ename, appartenente ad un elemento tradotto come interfaccia, viene convertito in un attribute di none Ename e tipo Ename. Il tipo Ename verrà tradotto come una nuova interfaccia a parte di nome Ename.

Es: <!ELEMENT DOCUMENT (CUSTOMER)>
<!ELEMENT CUSTOMER (NAME)>
<!ELEMENT NAME (#PCDATA)>



```
interface document
(source semistructured docum_file
 )
{attribute customer customer;}

 Interface customer
 (source semistructured docum_file
  )
 {attribute string name;}
```

Analizziamo ora il caso più complesso dei figli multipli.

5.4.1 Figli Multipli

Quando si deve dichiarare un elemento affinchè possa contenere figli multipli, si hanno diverse possibilità. Le DTD in questo caso usano una sintassi analoga all'uso delle espressioni regolari nel linguaggio Perl.
Siano a e b gli elementi figli che si stanno dichiarando, le possibilità sono le seguenti:

- a+ :una o più occorrenze di a
- a* :zero o più occorrenze di a
- a? : una o nessuna occorrenza di a
- a,b : a seguito da b. Indica una **sequence**
- a|b :a o b, ma non entrambi. Indica una **choice**

- (expression) : se si racchiude un'espressione fra parentesi significa che è considerata come un'unità e può usare ?,*,+.

Di seguito verranno analizzate in dettaglio tali opportunità, alla base della definizione delle sequences e delle choices.

5.4.2 Sequence

Una sequenza permette di specificare esattamente quali figli può contenere un particolare elemento e in quale ordine. E' costituita da un elenco di nomi di elementi separati da virgole e specifica al processore XML come devono apparire gli elementi.
In questo caso la traduzione in ODLi$_3$ non ha bisogno di molte spiegazioni.

- Ogni componente di una sequence deve essere tradotto come fosse un unico componente.

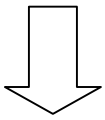
Un tipo particolare di sequence è rappresentato dagli **elementi ripetuti.**
Si tratta di elementi composti costituiti da più elementi di uno stesso tipo, definiti facendo uso dei caratteri +,*,?.

"?" indica che un particolare elemento figlio può essere presente, ma non necessariamente. Anche in ODLI3 esiste la possibilità di esprimere l'opzionalità . La regola di traduzione sarà la seguente:

- Ogni componente di nome Ename con suffisso ?, appartenente ad un elemento tradotto come interfaccia, viene convertito in un attribute dell'interfaccia con nome Ename e cardinalità (1,0), quindi con suffisso ?. Per il tipo si fa riferimento alle regole sopra citate.

ES:

```
<!ELEMENT DOCUMENT (CUSTOMER)?>
<!ELEMENT CUSTOMER (#PCDATA)>
```



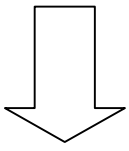
```
interface document
(source semistructured…)
{attribute string customer?} ;
```

"*" indica che un elemento può essere presente da zero a n volte. La traduzione in ODLi$_3$ avviene mediante i tipi di collezione, in particolare tramite la dichiarazione dei set.

- Ogni componente di nome Ename con suffisso * , appartenente ad un elemento tradotto come interfaccia, viene convertito in un attribute dell'interfaccia con nome Ename e cardinalità (0,N), di tipo set<type_set>. Type_set indica il corrispondente del tipo di dato DTD, quindi verrà scelto dall'utente nel caso sia un simple type, oppure sarà un tipo definito in un'altra interface.

ES:

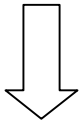
```
<!ELEMENT DOCUMENT (CUSTOMER)*>
<!ELEMENT CUSTOMER (#PCDATA)>
```



```
interface document
(source semistructured…)
{attribute set<string> customer} ;
```

"+" non trova traduzione diretta in ODLi$_3$, infatti non è prevista la possibilità di distinguere tra ricorrenze di cui è obbligatorio specificare almeno un valore, oppure ricorrenze in cui tale specificazione non è obbligatoria.
Per questo si è pensato ad una traduzione in cui vengono creati due tipi di attribute, uno di tipo set per indicare la cardinalità (0,N), e l'altro con cardinalità (1,1) .

ES: <!ELEMENT DOCUMENT (CUSTOMER)+>
<!ELEMENT CUSTOMER (#PCDATA)>



interface document
(source semistructured…)
{attribute customer customer,
attribute set<customer> customer_1} ;
typedef customer
{attribute string strvalue}

In questo caso è stato definito un nuovo tipo con typedef, ma se CUSTOMER non fosse stato di tipo PCDATA avremmo potuto creare una nuova interface.
La regola di traduzione è la seguente.

- Ogni componente di nome Ename con suffisso + , appartenente ad un elemento tradotto come interfaccia, viene convertito creando un attribute di nome Ename e tipo Ename e un attribute di tipo set con set_type Ename e nome Ename_1. Si definisce poi un nuovo tipo o una nuova interfaccia per specificare Ename.

Se si desidera, è possibile usare lo stesso elemento in una sequenza un certo numero di volte. Nell'esempio che segue l'elemento CUSTOMER dovrà contenere esattamente tre elementi NAME:

<!ELEMENT CUSTOMER (NAME,NAME,NAME)

La traduzione in ODLi$_3$ è immediata.

Un'altra nota importante è che è possibile creare sottosequenze con le parentesi, anche in questo caso non ci sono particolari problemi di traduzione, basta seguire per ogni componente le regole sopra citate.

5.4.3 Choice

Nelle DTD oltre alle sequenze è possibile usare anche le scelte. Una scelta consente di specificare che uno fra più elementi possa apparire in quella determinata posizione.
Quando il processore incontra un'espressione del tipo (a | b | c) sa che potrà essere presente solo uno degli elementi "a" o "b" o "c".
Tale opzione permette di avere nei dati la possibilità di inserire delle informazioni che possono avere significati diversi anche a seconda della struttura con cui tali informazioni sono rappresentate.

```
ES: <!ELEMENT  PRODUCT ( (NAME | NAME_COMP ) , QUANTITY )>
      <!ELEMENT NAME #PCDATA>
      <!ELEMENT NAME_COMP ( CODE, COLOUR)
```

Questo significa che il nome di un prodotto può essere o una semplice stringa, oppure una struttura più articolata comprendente codice e colore del prodotto.

La sintassi ODLi$_3$ ha possibilità di definire quali possono essere le forme che un attribute può assumere, perciò è necessario utilizzare il costrutto union e il costrutto typedef.

Le righe ODLi3 da generare sono le seguenti:

```
interface product
(source semistructured…)
{attribute product_union_1 product_union_1;
  attribute string quantity ;
 }

typedef product_union_1
{ string name;
}
union
{string code;
  string colour;
}
```

Enunciamo di seguito la regola di traduzione.

- Per ogni scelta che costituisce un componente di un elemento Ename bisogna introdurre nell'interfaccia Ename un attribute di nome Sname, composto da Ename, la parola chiave _union, ed infine "_" seguito da un numero che indica la posizione del componente nella lista, e tipo Sname. Il tipo Sname viene definito tramite typedef al cui interno le scelte sono separate da union.

La sintassi XML prevede anche dei casi più articolati di quello trattato nell'esempio consentendo diversi livelli di dettaglio. Per la traduzione è necessario applicare ad ogni livello la regola sopra esposta.

Un caso particolare di scelta si ha con gli elementi di contenuto misto

5.5 Elementi Misti

E' possibile specificare che un elemento possa contenere sia l'elemento PCDATA sia altri elementi; questo modello di contenuto è chiamato misto. Per specificare un modello di contenuto misto si dovrà elencare #PCDATA insieme agli elementi figli che si desidera consentire, separati da "|".

ES: <!ELEMENT PRODUCT (#PCDATA | PRODUCT_ID)>
<!ELEMENT PRODUCT_ID (#PCDATA)>

La traduzione è simile a quella delle choice, però il costrutto union viene utilizzato nella interface. Inoltre viene inserito un nodo fittizio PCDATA_NODE nel quale memorizzare l'informazione contenuta nell'opzione di tipo PCDATA.

- Ogni elemento misto viene tradotto come una choice fra un PCDATA e un altro elemento. L'opzione PCDATA viene indicata inserendo un attribute di nome PCDATA_NODE e tipo scelto dall'utente. Gli elementi che costituiscono le scelte vengono tradotti in base alle regole esposte prima.

La traduzione in ODLi3 è la seguente.

```
Interface product
(source semistructured….)
{attribute string product_id;
}
union
{attribute string PCDATA_NODE;
}
```

Tuttavia esistono restrizioni all'uso di tale contenuto: è possibile specificare solo i nomi degli elementi figli che possono essere presenti; non è possibile impostare l'ordine degli elementi figli o il numero di occorrenze.

A causa di tali restrizioni l'uso del modello di contenuto misto è sconsigliato, si preferisce dichiarare un nuovo elemento che possa contenere PCDATA e includerlo in un modello di contenuto standard.
Nel nostro caso però è utile per comprendere una possibile traduzione di un elemento di tipo ANY, intendendolo come un elemento misto particolare.

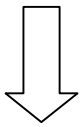
5.6 Elementi Di Tipo ANY

Un elemento di tipo ANY, può avere qualsiasi tipo di contenuto, cioè qualsiasi elemento che compare nel documento o un PCDATA..
In pratica il contenuto di tali elementi non viene verificato dai validatori XML.
Per non eliminare la verifica della sintassi è preferibile specificare un modello di contenuto effettivo, definendo un elenco di elementi figli.
Analizzando tali tipi di dati ho individuato tre possibilità di traduzione.

1)

- Ogni elemento semplice di nome Ename e di tipo ANY viene tradotto come se fosse un semplice elemento PCDATA..

ES: <!ELEMENT DOCUMENT (CUSTOMER)
< !ELEMENT CUSTOMER ANY>



interface document
(source semistructured…)
{attribute string customer}

2)

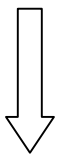
- Ogni elemento semplice di nome Ename e di tipo ANY viene tradotto come fosse un elemento misto, costituito da una choice fra PCDATA ed altri elementi definiti nel documento scelti dall'utente .

In pratica, tramite interfaccia interattiva, l'utente può specificare quali tipi di dato l'elemento in questione potrà assumere, fra quelli semplici previsti da ODLi$_3$ e quelli complessi definiti nel documento.

```
ES:   <!ELEMENT DOCUMENT (CUSTOMER, ORDERS)
      <!ELEMENT CUSTOMER ANY>
      <!ELEMENT ORDERS (PRODUCT, NUMBER, PERSON)
      <!ELEMENT PRODUCT  (#PCDATA)>
      <!ELEMENT NUMBER (#PCDATA)>
      <!ELEMENT PERSON (NAME, DATE, ADDRESS)
     <!ELEMENT NAME (#PCDATA)>
     <!ELEMENT DATE (#PCDATA)>
     <!ELEMENT ADDRESS (#PCDATA)>
```

L'utente dovrà scegliere quali tipi di struttura customer potrà assumere. In questo caso è ragionevole pensare che l'elemento in questione possa essere o una stringa, o del tipo person. Quindi la traduzione verrà effettuata come se la situazione fosse la seguente :

```
          ….
<!ELEMENT CUSTOMER (#PCDATA | PERSON)
          ……
```



```
Interface customer (source semistructured….)
 {attribute person person}
   union
 {attribute string PCDATA_NODE;
  }
 ……..
```

```
 Interface person (source semistructured….)
{attribute string name;
 attribute string date;
 attribute string address
}
```

3)

- Ogni elemento semplice di nome Ename e di tipo ANY viene tradotto come un elemento del tipo Any previsto da ODLi3.

  Il tipo Any proprio di ODLi3 lascia ancora più libertà al contenuto degli elementi, permettendo ad essi di assumere qualsiasi tipo di struttura, anche non definita nel documento di cui fanno parte.

```
Attribute any Customer
```

A mio avviso la seconda soluzione è la più corretta, in quanto permette di non rimuovere completamente il controllo della sintassi per questo tipo di elementi, limitando i tipi di valori ammissibili. Tuttavia nei documenti di grandi dimensioni diventa molto laboriosa, in questi casi, perciò, è preferibile la traduzione tramite il costrutto ANY di ODLi3.

5.7 Attributi Nelle DTD

Gli attributi sono coppie nome valore che è possibile utilizzare per inserire informazioni aggiuntive su un elemento.

La dichiarazione degli attributi e dei tipi è molto utile nel linguaggio XML. Affinchè i documenti siano validi, è necessario dichiarare gli attributi prima dell'uso. E' possibile assegnare ad essi valori predefiniti e richiedere agli autori XML che usano le DTD di assegnare valori agli attributi.

Per dichiarare l'elenco degli attributi di un elemento si utilizza <!ATTLIST>, la cui forma generale è:

```
<!ATTLIST  ELEMENT_NAME
      ATTRIBUTE_NAME TYPE DEFAULT_VALUE
     ATTRIBUTE_NAME TYPE DEFAULT_VALUE
       ….
     ATTRIBUTE_NAME TYPE DEFAULT_VALUE
```

ELEMENT_NAME : nome dell'elemento per cui si effettua la dichiarazione degli attributi.
ATTRIBUTE_NAME : nome dell'attributo che si sta dichiarando.
TYPE : tipo attributo
DEFAULT_VALUE :specifica il valore predefinito e può assumere diverse forme.
Segue l'elenco dei possibili valori di TYPE utilizzabili

Tipo	***Descrizione***
CDATA	Dati di tipo carattere (testo escluso caratteri di markup)
ENTITIES	Nomi di entità (che devono essere dichiarati nelle DTD) separati da spazi
ENTITY	Entità che deve essere dichiarata nelle DTD
Enumerated	Elenco di valori;qualsiasi voce dell'elenco è un valore di attributo appropriato e deve essere usato un elemento della lista
ID	Corrisponde ad un nome proprio di un nome XMLche deve essere univoco, non condiviso,cioè, da qualsiasi altro attributo di tipo ID
IDREF	Contiene il valore di un attributo ID di un elemento, normalmente un altro elemento a cui è correlato quello corrente
IDREFS	Mostra più ID di elementi separati da spazi
NMTOKEN	Mostra un nome proprio XML
NMTOKENS	Mostra più nomi XML propri in un elenco, separati da spazi
NOTATION	Mostra il nome di una notation che deve essere dichiarata nella DTD

Segue l'elenco delle possibili impostazioni DEFAULT_VALUE che possono essere utilizzate.

Metodo	*Descrizione*
VALUE	Mostra un valore di tipo testo racchiuso fra " "
#IMPLIED	Non esiste un valore predefinito per l'attributo che non deve essere necessariamente usato
#REQUIRED	Non esiste un valore predefinito, ma un valore deve essere assegnato a questo attributo
#FIXED VALUE	Value è il valore dell' attributo e l'attributo deve avere sempre questo valore

5.8 Traduzione degli Attributi

In ODLi$_3$ non esiste un corrispondente di ATTLIST, infatti la parola chiave attribute ha significato differente. Per questo è necessario a volte l'utilizzo di qualche artificio per descrivere tale informazione, come per esempio la promozione di un attributo al rango di elemento.

Consideriamo in primo luogo la traduzione degli attributi associati all'elemento radice, il quale non è componenete di nessun altro elemento
Una prima **regola generale** di traduzione è la seguente.

- Dato un set di attributi $a_1,\ldots,a_n$ riferiti ad un elemento Ename senza indicazioni per il DEFAULT VALUE e di tipo CDATA
  1) si crea l'interfaccia Ename
  2) si generano dei sottoelementi attribute denominati utilizzando come prefisso il nome dell'elemento a cui è riferito l'attlist, seguito da "_" e dal nome stesso dell'attributo
  3) ogni attribute è indicato come opzionale e il tipo viene scelto dall'utente come per gli elementi semplici.

ES: <!ATTLIST DOCUMENT TYPE CDATA>

…….

⇩

Interface document

(….)

{attribute string document_type*;

….}

Vengono esaminati ora i vari casi in cui il DEFAULT VALUE è specificato

5.8.1 #REQUIRED

Il valore dell'attributo deve comparire obbligatoriamente.

Vale la regola generale prima esposta, con la differenza che l'attribute creato non è opzionale

ES: <!ATTLIST DOCUMENT TYPE CDATA #REQUIRED>

⇩

Interface document (….)

{attribute string document_type;

….}

5.8.2 Valori Immediati

Nelle DTD è possibile specificare un valore predefinito per un attributo, semplicemente assegnando il valore fra doppi apici nella dichiarazione dell'attributo

ES: <!ATTLIST DOCUMENT TYPE CDATA "excellent">

Il valore preimpostato rappresenta una sceltra tra i valori di tipo CDATA che l'attributo OWES può assumere. Si tratta di una peculiarità non molto significativa dal punto di vista della struttura, quindi non è necessaria la traduzione, tuttavia è possibile esprimere tale informazione in ODLi3 creando una nuova interface contenente il costrutto union.

```
Interface document
(….)
{attribute document_type document_type*
 ….}
interface document_type
(…)
{const string document_type_value "excellent" ;}
  union
{attribute string type_node}
```

In questo caso l'attributo viene promosso al rango di interfaccia.
Esponiamo la regola:

- Quando un elemento Ename ha un attributo di nome Aname con valore preimpostato l'attributo viene tradotto inserendo nell'interfaccia Ename un attribute di tipo Aname e nome Aname. Il nuovo tipo Aname viene tradotto creando un'altra interfaccia Aname comprendente una costante di nome Aname_value, tipo scelto dall'utente fra quelli base e

valore uguale a quello preimpostato, e un semplice attribute di nome Aname_node e tipo generico string, separati dal costrutto union.

5.8.3 #IMPLIED

A livello di traduzione non implica nessuna attenzione particolare, vale la regola generale.

5.8.4 #FIXED VALUE

Viene impostato e specificato un valore fisso per l'attributo. La costante, in ODLi3, esprime un concetto analogo.

- Un attributo Aname di un elemento Ename con DEFAULT_VALUE di tipo FIXED viene tradotto inserendo nell'interfaccia Ename una costante di nome Ename_Aname, tipo scelto dall'utente e valore specificato.

<!ATTLIST DOCUMENT TYPE CDATA #FIXED " excellent" >

Interface document
(….)
{const string document_type ='excellent'
….}

5.8.5 Caso Particolare: Attributi di Sottoelementi PCDATA

Consideriamo ora il caso in cui l'attributo da tradurre sia associato ad un elemento che fa parte della specifica di contenuto di un altro elemento ed è del tipo PCDATA..

ES:<!DOCUMENT (CUSTOMER)>
<!ELEMENT CUSTOMER (#PCDATA)
<!ATTLIST CUSTOMER OWES CDATA

Se non avesse attributi , un elemento di questo tipo verrebbe tradotto normalmente come un attribute di tipo base e non come una nuova interfaccia.
In questo caso però la situazione è diversa,in quanto l'attributo non può essere tradotto come un nuovo nodo posto sotto all'elemento in questione, in quanto questo non è un'interface.
La soluzione migliore è tradurre ogni elemento dotato di attributi come interface, introducendo un nodo fittizio PCDATA_NODE nel quale memorizzare l'informazione contenuta nel campo PCDATA relativo all'elemento in questione.
L'esempio tradotto in ODLi3 diventa:

```
interface document
(…)
{attribute customer customer,}


interface customer
(..)
{attribute string PCDATA_NODE ;
  attribute string customer_owes;
}
```

Fino ad ora abbiamo analizzato solo attributi CDATA, ma esistono vari tipi di attributi che forniscono alcune funzionalità per consentire la verifica della sintassi di un documento.

5.8.6 Attributi Enumerati

Gli attributi enumerati forniscono un elenco dei valori possibili, ciascuno dei quali deve essere un nome XML valido. Sono molto utili nel caso si desideri impostare gli intervalli consentiti per i valori di un attributo.
Nell'esempio che segue si dichiara un attributo CREDIT_OK che può avere solo uno dei due valori possibili: "TRUE" o "FALSE" e che ha il valore predefino FALSE.

<!ATTLIST CUSTOMER CREDITOK (TRUE | FALSE) "TRUE">

In ODLi3 è possibile esprimere un concetto di questo tipo tramite il costrutto *enum.*
La traduzione sarà:

```
enum creditok_type
{ 'true';
   'false';
}
```

interface customer
(…)
{attribute creditok_type creditok_type;
}

La regola generale è la seguente.

- Un attribute di nome Aname di tipo enumerato viene tradotto come un attribute di nome Aname_type e tipo Aname_type. Il nuovo tipo di nome Aname_ type viene craeto mediante il costrutto enum, elencando al suo interno i valori possibili.

La stessa regola può essere utilizzata per la traduzione di un elemento di tipo empty.

5.8.7 Elementi Di Tipo EMPTY

Un elemento di tipo empty può essere visto come un possibile valore di un attributo enumerato.

```
Es: <!ELEMENT CUSTOMER (CREDITWARNING?,NAME,DATE)>
      <!ELEMENT CREDITWARNING EMPTY>
       …….
```

Questo elemento empty, in effetti, ha il significato di un attributo associato all'elemento CUSTOMER. La stessa informazione viene trasmessa anche nel modo seguente:

```
<!ELEMENT CUSTOMER (NAME,DATE)>
<!ATTLIST CUSTOMER
       CUSTOMER_INFO (CREDITWARNING | NO_CREDITWARNING)>
```

In ODLi3 diventa:

```
enum customer_creditwarning
{ 'creditwarning';
  'no_creditwarning';
}
  interface customer
(…)
{attribute customer_creditwarning customer_creditwarning;
}
```

Gli elementi di tipo Empty possono contenere attributi, come nel caso seguente:

```
<!ELEMENT CUSTOMER (CREDITWARNING?,NAME,DATE)>
    <!ELEMENT CREDITWARNING EMPTY>
    <!ATTLIST CREDITWARNING CREDIT CDATA>
```

In ODLi3 un elemento contenente solo attributi è un elemento complesso, quindi ,in questo caso, la traduzione viene effettuata generando una nuova interfaccia :

```
Interface customer
(…)
{attribute customer_creditwarning customer_creditwarning;
}

Interface customer_creditwarning
(…)
{attribute string customer_creditwarning_credit?;}
```

Perciò la regola di traduzione di un elemento di tipo **EMPTY** è la seguente.

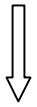
- Ogni elemento semplice di nome Ename e di tipo EMPTY ,se non contiene attributi, viene tradotto come valore possibile di un attributo fittizio dell'elemento padre, di tipo enumerato e nome Ename_info, contenente due valori possibili : 'Ename' e 'no_Ename'; altrimenti viene tradotto generando una nuova interfaccia contenente i suoi attributi.

5.8.8 ID

XML assegna un significato particolare al valore ID di un elemento, pochè si tratta del valore usato generalmente dalle applicazioni per l'identificazione degli elementi. Non possono esistere due elementi con lo stesso valore di ID all'interno di un documento ed è possibile

assegnare agli elementi solo un attributo di questo tipo. Il valore assegnato ad ID deve essere un nome XML corretto, quindi non può essere un numero iniziare con una cifra.
Non è possibile usarlo con gli attributi di tipo FIXED, poiché hanno tutti lo stesso valore,si usa invece, normalmente,la parola chiave REQUIRED.
Sono evidenti le analogie con il concetto di primary key di $ODLi_{3,}$, dove per primary key si intende appunto l'identificatore della classe.
Per la traduzione è necessario tuttavia l'intervento dell'utente che deve specificare se l'attributo ID può essere usato come primary Key.

ES:<!ATTLIST CUSTOMER CUSTOMER_ID ID #REQUIRED>



Interface customer
(source semistructured docum_file
Key customer_id
{attribute string customer_id
…..}

- Un attributo di nome Aname e tipo ID, associato ad un elemento Ename , deve essere tradotto come un attribute di tipo string e nome Aname, e , se l'utente lo consente, nell'interfaccia Ename deve essere indicata la primary key Aname

L'attribute creato sarà di tipo string a causa delle restrizioni a cui sono soggetti i valori che possono essere aseegnati agli attributi di tipo ID, di cui abbiamo già parlato.

5.8.9 IDREF/IDREFS

Tali attributi contengono il valore ID di qualche altro elemento nel documento.

L'attributo IDREF costituisce un tentativo di consentire l'uso degli attributi per specificare un aspetto della struttura del documento, in particolare per quanto riguarda le relazioni fra gli elementi. Si supponga per esempio di voler impostare una relazione genitore figlio tra elementi non evidenziata e non rappresentata dalla normale struttura di annidamento del documento; in questo caso si potrebbe impostare l'attributo IDREF di un elemento all' ID del genitore.
Sono evidenti le analogie con il concetto di foreign key di ODLi3, tuttavia la traduzione presenta dei problemi, risolvibili solo con l'intervento dell'utente.

Analizziamo in primo luogo un caso semplice in cui nel documento sono presenti un solo attributo di tipo ID e un solo attributo di tipo IDREF.

```
<!ELEMENT DOCUMENT (CUSTOMER,PRODUCT)
<!ELEMENT CUSTOMER(NAME,DATE,ORDERS)
   …
   <!ELEMENT ORDERS (#PCDATA)>
 <!ATTLIST ORDERS PROD_ID  IDREF #IMPLIED >
<!ELEMENT PRODUCT (#PCDATA)
  <!ATTLIST PRODUCT PROD_CODE ID #REQUIRED>
```

In questo caso è evidente che all'attributo PROD_ID di tipo IDREF verrà assegnato l'ID dell'elemento PRODUCT, quindi la traduzione non presenta problemi.

```
Interface costumer
(source semistructured docum_file
     )
{attribute string name;
 attribute string date;
 attribute orders orders;
  }

interface orders
```

```
(source semistructured docum_file
 foreign_Key (prod_id ) references product  )
{attribute string PCDATA_NODE;
attribute string prod_id ;
}

interface product
(source semistructured docum_file
 key prod_code)
{attribute string PCDATA_NODE;
   attribute string prod_code ;
}
```

Nel caso in cui all'interno della DTD vengono utilzzati più attributi di tipo ID le cose si complicano. Osserviamo l'esempio seguente:

```
<!ELEMENT DOCUMENT (CUSTOMER,PRODUCT)
<!ELEMENT CUSTOMER(NAME,DATE,ORDERS)
    ……..
  <!ATTLIST CUSTOMER CUST_ID ID #REQUIRED>
   <!ELEMENT ORDERS (#PCDATA)>
 <!ATTLIST ORDERS  PROD_ID  IDREF #IMPLIED >
 <!ELEMENT PRODUCT (#PCDATA)>
  <!ATTLIST PRODUCT PROD_CODE ID #REQUIRED>
```

Intuitivamente possiamo pensare che all'attributo di tipo IDREF venga assegnato l'ID di PRODUCT, ma non abbiamo nessuna garanzia. Infatti notiamo che non viene specificato in nessun modo quale la sia la primary key a cui la foreign key PRO_ID referenzia, quindi nulla vieta di assegnare all'attributo IDREF l'ID di customer.

Tuttavia, anche se non è possibile stabilirlo con precisione, nella maggior parte dei casi ad un attributo di tipo IDREF viene associato l'ID di uno stesso elemento (o degli stessi elementi) all'interno di un documento. Si potrebbe quindi prevedere una funzione che va a confrontare i

valori degli attributi IDREF nel documento con i vari ID, estrapolando così una serie di relazioni fra gli elementi.
A questo punto può intervenire l'utente, tramite interfaccia, per confermare le relazioni corrette ed eliminare quelle che in realtà non sussistono.

Un caso particolare si ha quando un elemento contiene un IDREF al quale è associato l'ID dell'elemento stesso, come avviene nell'esempio seguente.

```
ES: <!ELEMENT CUSTOMER(NAME,DATE)
        ……..
      <!ATTLIST CUSTOMER
        CUST_ID ID #REQUIRED
       EMPLOYER IDREF #IMPLIED>
```

In questo esempio viene effettuata la dichiarazione di due attributi, un CUSTOMER_ID di tipo ID e un EMPLOYER_ID di tipo IDREF che corrisponde al valore ID del datore di lavoro del cliente.
Praticamente nel documento avremo una situazione di questo tipo:

```
<DOCUMENT>
  <CUSTOMER CUSTOMER_ID =’C1234’>
    <NAME>…….</NAME>
    <DATE>……..</DATE>
 </CUSTOMER>
<CUSTOMER CUSTOMER_ID =’C175434’ EMPLOYER_ID=’C1234’>
    <NAME>…….</NAME>
    <DATE>……..</DATE>
 </CUSTOMER>
```

Traducendo in ODLi$_3$ dovremmo impostare una foreign_key che autoreferenzi l'elemento di appartenenza, in questo caso CUSTOMER. Per evitare questo si propone di creare un nuovo elemento EMPLOYER che abbia le stesse caratteristiche di CUSTOMER e in più l'attributo di tipo IDREF.

Praticamente l'interface CUSTOMER sarà una superclasse dell' interface EMPLOYER e verrà tradotta nel modo seguente:

```
interface customer
(source semistructured docum_file
key cust_id)
{attribute string name ;
  attribute string date ;
  attribute string cust_id ;}


interface employer  : customer
(source semistructured docum_file
 foreign_key (employer_id) references customer)
{  attribute string employer_id ;}
```

Si riporta la regola generale:

- Un attributo di tipo IDREF e nome Aname, associato ad un elemento Ename, deve essere tradotto come attributo di tipo string e nome Aname nell'interfaccia Ename. Se l'utente lo consente verrà indicato come foreign key nell'interface Ename , riferita all'elemento a cui appartiene l'ID ad esso associato, individuato dalla funzione specifica. Fa eccezione il caso in cui l'attributo IDREF, di nome IDREFname, e il relativo ID, di nome IDname, appartengono al medesimo elemento; se si verifica tale condizione deve essere creata una nuova interface denominata IDREFname, che eredita da Ename e ha in più l'attribute IDREFname_id di tipo string, indicato anche come foreign_key riferita ad Ename.

Per quanto riguarda IDREFS le cose non cambiano, l'unica differenza è relativa ai dati che possono essere costituiti da diverse stringhe separate da spazi bianchi.

5.8.10 NMTOKEN/NMTOKENS

Un attributo di questo tipo può assumere solo valori corrispondenti a nomi propri XML. Si tratta di restrizioni riguardanti il contenuto senza implicazioni importanti dal punto di vista strutturale. Attributi di questo tipo verranno tradotti come semplici attribute di tipo string.

- Un attributo di tipo NMTOKEN di nome Aname, associato all'elemento Ename, deve essere tradotto come attribute di tipo string e nome Aname nell'interfaccia Ename.

Usando NMTOKENS il valore dell'attributo può essere composto da più attributi NMTOKEN separati da spazi.

5.8.11 NOTATION

Quando si dichiara un attributo di questo tipo è possibile assegnare nei documenti XML valori che sono stati dichiarati come notazioni,e pecificano il formato di dati non XML. I tipi MIME (*Multipurpose Internet Mail Extension*) sono esempi diffusi di NOTATION come image/gif, application/xml, text/html..

Per dichiarare una notazione XML si usa:
<!NOTATION NAME SYSTEM "EXTERNAL_ID">

NAME corrisponde al nome della notazione ed *EXTERNAL_ID* è l' ID esterno che si desidera usare per la notazione, spesso un tipo MIME.
E' possibile usare la parola chiave *public* per le notazioni pubbliche se si fornisce un identificatore FPI nel modo seguente:

<!NOTATION NAME PUBLIC FPI "EXTERNAL_ID">

Nell'esempio che segue si dichiarano due notazioni, GIF e JPG che denotano i tipi MIME denominati: image/gif e image/jpeg. Si imposterà quindi un atrributo a cui può essere assegnato uno di questi valori.

```
<!ELEMENT DOCUMENT (CUSTOMER)>
<!ELEMENT CUSTOMER (NME, DATE, ORDERS)>
…….
<!NOTATION GIF SYSTEM "image/gif">
<!NOTATION JPG SYSTEM "image/jpeg">
<!ATTLIST CUSTOMER
                    IMAGE NMTOKEN #IMPLIED
                    IMAGE_TYPE  NOTATION (GIF | JPG) #IMPLIED>
```

Non esistono strutture analoghe in ODLi$_3$, però possiamo tentare di fornire una traduzione ricorrendo a degli artifici.
Bisognerebbe prevedere una funzione in grado di individuare le notation e inserire il loro contenuto come valore degli attributi di tipo notation.
A questo punto l'attributo IMAGE_TYPE diventerebbe un semplice attributo di tipo enumerato, traducibile secondo le regole elencate.

```
Interface document
(….)
{…….}

enum image_type
{'image/gif'
'image/jpeg'}


interface customer
(….)
{attribute string name,
attribute string date
```

…..
attribute string image*;
attribute image_type image_type*;}

In questo caso è meglio non esporre una regola generale poichè le notazioni possono essere usate in vari modi e alcune volte una loro traduzione è inutile, in quanto a livello strutturale non aggiungono alcuna informazione.
Il caso trattato ,comunque, costituisce una delle più usate e importanti applicazioni delle notazioni.

5.8.12 ENTITY

Gli attributi di tipo ENTITY verranno analizzati più avanti, dopo aver trattato le ENTITA'.

5.8.13 XML:lang

Si tratta di un attributo particolare che serve a specificare in che linguaggio deve essere scritto il contenuto dell'elemento a cui è associato.

ES: <!ELEMENT Name (#PCDATA)>
<!ATTLIST Name xml:lang %xmlLangCode; #REQUIRED>
……

La traduzione sarà la seguente:

Interface Name
(…..)
{attribute string Name_lang;
…….}

- L'attributo xml:lang. Associato ad un elemento Ename, viene tradotto come un attribute di tipo string e nome Ename_lang, all'interno dell'interfaccia Ename.

5.9 Entità

Tramite queste strutture è possibile il riferimento a porzioni di dati che, per vari motivi, possono venire rappresentati da una parola chiave definita appunto tramite la dichiarazione di un'entità. Sono generalmente testo, ma possono anche essere di tipo binario.
Esistono due tipi di entità:

- ***generali***: si utilizzano nel contenuto dei documenti XML. I riferimenti ad entità generali iniziano con & e terminano con ; .
- ***parametriche***: si usano all'interno delle DTD. I riferimenti ad entità parametriche iniziano con % e terminano con ; .

Le entità poi possono essere :

- ***interne:*** completamente definite all'interno del documento XML che fa riferimento ad essa (infatti lo stesso documento è considerato un'entità nel linguaggio XML)
- ***esterne:*** derivano il loro contenuto da una forma esterna, per esempio un file, e un riferimento ad esse include generalmente un URI dove possono essere reperite.

Inoltre si possono avere entità:

- ***analizzate***: il loro contenuto è testo XML ben formato
- ***non analizzate:*** contengono dati che non si desidera analizzare, come semplice testo o dati di tipo binario.

Di seguito verranno analizzati tutti i tipi di entità e le possibili traduzioni.

3.9.1 Entità Generali Interne

L'esempio che segue definisce un'entità generale denominata TODAY che memorizza la data April 1, 2002. Quando si inserisce nel documento un riferimento ad entità, ovvero &TODAY, questo sarà sostituito con il testo April 1, 2002 dal processore XML.

ES:

```
<!ELEMENTO DOCUMENT (CUSTOMER)*>
<!ELEMENT CUSTOMER (NAME,DATE,ORDERS)>
<!ELEMENT NAME (LAST_NAME,FIRST_NAME)>
<!ELEMENT LAST_NAME (#PCDATA)>
<!ELEMENT FIRST_NAME (#PCDATA)>
<!ELEMENT DATE (#PCDATA)>
……
<!ENTITY TODAY “April 1, 2002”>
```

L’informazione data dall’entità, in questo caso, riguarda il contenuto e non tanto la struttura del documento.

Tuttavia è possibile aggiungere nella traduzione anche questa particolarità, traducendo il testo memorizzato nell’entità come una costante. Sarà necessaria una funzione che vada ad analizzare il documento XML per scoprire dove viene utilizzato il riferimento a tale entità, in modo tale da inserire la costante nelle interfacce giuste.

In questo caso, all’interno dell’interfaccia CUSTOMER, verrà definita una costante di tipo string e nome date.

```
Interface customer
(…)
{attribute name name;
const string date=’April 1, 2002’;
…}
```

Non è possibile usare i riferimenti ad entità generali per l’inserimento di testo che deve essere usato solo nella DTD e non nel contenuto del documento; tuttavia le definizioni di tipo generale possono essere annidate.

ES:

```
<!ENTITY NAME “Alfred Hitchcock”>
<!ENTITY SIGNATURE “&NAME; 14 Mystery Drive”>
```

In questo caso è necessario creare due costanti, una il cui valore sarà Alfred Hitchcock, corrispondente all'entità NAME, e l'altra il cui valore sarà Alfred Hitchcock; 14 Mystery Drive, corrispondente all'entità SIGNATURE. Per la traduzione della prima entità ci si comporta come per il primo esempio, per la seconda, invece, è nacessario sostituire il contenuto di NAME al suo riferimento all'interno di SIGNATURE.
E' necessaria una funzione che vada a ricercare i riferimenti ad entità generali interni alla DTD, i quali possono essere riconosciuti individuando stringhe che iniziano con il carattere &, comprese fra gli apici, nelle dichiarazioni delle entità; una volta trovati deve inserire il contenuto che sottointendono come valore nella costante corrispondente.
Come regola generale può essere adottata la seguente:

- Un'entità generale interna ENTname deve essere tradotta come costante di tipo string avente come valore il testo memorizzato nell'entità. Si distinguono due casi:
    2. l'elemento Ename all'interno del quale viene usato il riferimento ad essa non contiene altri sottoelementi o attributi. In questo caso tale elemento non viene tradotto, e la costante viene inserita all'interno dell'interfaccia dell'elemento padre di Ename con nome Ename.
    3. l'elemento Ename all'interno del quale viene usato il riferimento ad essa contiene altri sottoelementi o attributi.In questo caso la costante viene inserita nell'interfaccia Ename con nome ENTname.

  Se ENTname contiene un riferimento ad un'altra entità ENT1name, entrambe vengono tradotte in base a quanto detto sopra, ma il testo di ENTname conterrà anche quello di ENT1name.

Va notato che i riferimenti ad entità non possono essere circolari.

5.9.2 Entità Generali Esterne

Le entità, oltre che interne, possono essere anche esterne; questo implica che si dovrà fornire un URI che indirizzi il processore XML all'entità. E' possibile usare i riferimenti ad entità esterne per inserirle nei documenti. Possono essere costituite da semplici stringhe di testo, da documenti completi o da sezioni di documenti. Quando le entità sono inserite all'interno del

contenuto dei documenti, il processore XML si limita a controllare che il documento sia ben formato e valido.

E' possibile dichiarare le entità esterne usando le parole chiave SYSTEM e PUBLIC. Le entità destinate all'uso privato da parte di organizzatori o privati sono dichiarate con la parola chiave SYSTEM, mentre le entità dichiarate con la parola chiave PUBLIC sono pubbliche e, quindi, richiedono l'uso di un identificatore pubblico formale o FPI *(Formal Public Identifier).*

Tali parole chiavi vanno usate nel modo seguente:

<!ENTITY NAME SYSTEM URI>

<!ENTITY NAME PUBLIC FPI URI>

Tramite le entità esterne è possibile creare documenti composti a loro volta da altri documenti.Questo è utile, per sempio, quando si desidera usare lo stesso testo come firma o quando si usano testi che cambiano con frequenza che si desidera modificare centralmente in un'unica posizione.

Consideriamo il caso in cui l'entità esterna è costituita da una stringa di testo.

Si assuma che una data sia stata memorizzata con la stringa di testo April 1, 2002 in un file denominato date.xml. Si potrà quindi creare un'entità denominata TODAY correlata al file:

<!ENTITY TODAY SYSTEM "date.xml">

Si potrà poi usare un riferimento a questa entità per inserire la data nel contenuto di un documento.

La traduzione in ODLi$_3$ verrà effettuata mediante una costante, come per le entità generali interne. In questo caso sarà necessario disporre di una funzione che apra il file e inserisca il suo contenuto come valore della costante; tale funzione sarà in grado di riconoscere che l'entità contiene il nome di un file grazie alla presenza dell'estensione.

Nel caso in cui il contenuto del file sia un intero documento, troppo esteso per essere considerato una stringa di testo, la traduzione verrà omessa. A livello strutturale questo non comporterà comunque una grave perdita di informazioni, in quanto tali entità vengono utilizzate solo per specificare il contenuto dei documenti.

5.9.3 Entità Generali di Tipo Predefinito

Esistono cinque entità predefinite in XML che indicano caratteri interpretabili come markup o caratteri di controllo.

- &amp: indica il carattere &;
- &apos: indica il carattere ‘;
- &gt: indica il carattere >;
- &lt: indica il carattere <;
- &quot: indica il carattere “;

Normalmenete è difficile gestire questi caratteri nei documenti XML, poiché i processori XML assegnano ad essi un significato particolare. L’utilizzo dei riferimenti ad entità permette una gestione sicura, poiché il processore effettuerà la sostituzione con il carattere corrispondente quando verrà eleborato il documento.
Sebbene esistano solo cinque riferimenti ad entità predefinite, è possibile creare altri riferimenti ad entità per specifici caratteri XML, specificando il codice corrispondente del carattere nella codifica usata.
ES:<!ENTITY at_new “&#64”>

In questo modo si definisce un’entità denominata at_new affinchè i riferimenti ad at_new siano sostituiti con il carattere corrispondente al codice #64.

Tali entità non aggiungono alcuna informazione dal punto di vista strutturale, per questo non verrà effettuata alcuna traduzione in ODLi3.

5.9.4 Entità Parametriche

Come è stato visto, nei documenti si usano i riferimenti a entità affinchè il processore XML possa effettuare la sostituzione con le entità corrispondenti. Nelle DTD, tuttavia , è possibile usare le entità generali solo in modo limitato, cioè per inserire testo nel documento, ma non nelle dichiarazioni dei documenti all’interno delle DTD.

Per poter usare le dichiarazioni degli elementi e degli attributi si usano le entità parametriche. I riferimenti ad entità parametriche possono essere usati solo nelle DTD, con una restrizione ulteriore: qualsiasi riferimento parametrico che si usa in una qualsiasi dichiarazione DTD dovrà apparire solo nel sottoinsieme esterno della DTD. E' possibile usare anche quelle interne, ma con restrizioni ulteriori che vederemo in seguito.
A differenza delle entità generali, i riferimenti ad entità parametriche iniziano con % e non con &. Per crere un' entità parametrica si usa la seguente espressione:
<!ENTITY % NAME DEFINITION>.
E'anche possibile utilizzare le parole chiave SYSTEM e PUBLIC.

5.9.4.1 Entità Parametriche Interne

Segue un esempio che usa un'entità parametrica interna; in questo caso si dichiara un'entità parametrica denominata BR che rappresenta il testo <!ELEMENT BR EMPTY> all'interno della DTD:

```
<!ENTITY %BR "<ELEMENT BR EMPTY>"
<!ELEMENTO DOCUMENT (CUSTOMER)*>
<!ELEMENT CUSTOMER (NAME,DATE,ORDERS)>
<!ELEMENT NAME (LAST_NAME,FIRST_NAME)>
<!ELEMENT LAST_NAME (#PCDATA)>
<!ELEMENT FIRST_NAME (#PCDATA)>
<!ELEMENT DATE (#PCDATA)>
……
%BR;
```

Il riferimento all'entità parametrica consente di includere la dichiarazione dell'elemento <!ELEMENT BR EMPTY> all'interno della DTD.
Sarebbe stato possibile inserire la dichiarazione <!ELEMENT BR EMPTY> direttamente nella DTD. D'altra parte non è possibile fare molto di più con le entità interne parametriche, definite nel sottoinsieme interno della DTD, poiché non è possibile usarle all'interno di altre dichiarazioni.
Per la traduzione in ODLi$_3$ sarà necessario disporre di una funzione in grado di individuare il riferimento all'entità parametrica nella DTD, associarla all'entità corrispondente, ed effettuare

la traduzione degli elementi o attributi in essa contenuti, collocandoli nella stessa posizione in cui si trova il riferimento.
Per la traduzione degli elementi e degli attributi si seguono le regole già analizzate.

5.9.4.2 Entità Parametriche Esterne

Quando si usa un'entità parametrica nel sottoinsieme esterno della DTD si può fare riferimento all'entità dovunque nella DTD, comprese le dichiarazioni dell'elemento. Nell'esempio che segue si usa la DTD esterna denominata order.dtd per gestire un documento.
Il sottoinsieme esterno della DTD order.dtd sarà impostato affinchè l'elemento DOCUMENT possa contenere non solo gli elementi CUSTOMER, ma anche BUYER e DISCOUNTER. Ciascuno di questi nuovi elementi ha lo stesso modello di contenuto dell'elemento CUSTOMER, cioè può contenere gli elementi NAME, DATE e ORDERS; per risparmiare tempo si assegna il modello di quel contenuto all'entità parametrica denominata record:
<!ENTITY %record "(NAME,DATE,ORDERS)
<!ELEMENT DOCUMENT (CUSTOMER | BUYER | DISCOUNTER)*)

E' ora possibile far riferimento all'entità parametrica dove si desidera; in questo caso per dichiarare gli elementi CUSTOMER BUYER e DISCOUNTER:

```
<!ENTITY %record "(NAME,DATE,ORDERS)
<!ELEMENT DOCUMENT ((CUSTOMER | BUYER | DISCOUNTER)*)
<!ELEMENT CUSTOMER %record>
<!ELEMENT BUYER %record >
<!ELEMENT DISCOUNTER %record>
```

Questo esempio illustra il motivo principale per cui gli sviluppatori usano entità parametriche che è quello di gestire del testo ripetuto nelle dichiarazioni degli elementi nelle DTD. In questo caso è stato specificato il modello di contenuto di tre elementi che usano tutti la stessa entità parametrica, ma si poteva anche impostare un'entità parametrica per specificare una lista degli attributi dello stesso tipo con il numero di elementi desiderato.

E' possibile, in questo modo, controllare la dichiarazione di molti elementi ed attributi, anche in una DTD di grandi dimensioni. Inoltre, se bisogna modificare una dichiarazione, è sufficiente farlo solo per l'entità parametrica e non per ciascuna singola dichiarazione.
Si consiglia per esempio di suddividere nei vari titpi gli attributi di una DTD di grandi dimensioni.
Consideriamo il caso relativo all'esempio trattato, in cui l'entità costituisce il modello di contenuto di elementi complessi, traducibili come interfaccia. Dal momento che gli elementi BUYER e DISCOUNTER sono identici, come struttura, all'elemento CUSTOMER, si genereranno tre interfacce, una di nome customer, una buyer e una discounter, aventi gli stessi componenti.

5.9.5 Attributo di Tipo Entity

Un attributo di tipo ENTITY può essere può essere impostato al nome dell'entità che è stata dichiarata. Si supponga per esempio che sia stata dichiarata un'entità denominata SNAPSHOT1 che fa riferimento ad un file esterno contenente un'immagine.si può creare un nuovo attributo denominato IMAGE impostato al nome dell'entità SNAPSHOT1

```
<!ATTLIST CUSTOMER
  IMAGE ENTITY #IMPLIED>
<!ENTITY SNAPSHOT1 SYSTEM "image.gif">
```

La traduzione in ODLi$_3$ equivale a quella di un attributo di tipo CDATA a cui è stato assegnato il valore "image.gif". L'attributo avrà nome IMAGE.
Consideriamo invece il caso in cui l'entità definita si riferisca ad una stringa memorizzata in un file esterno, per esempio una firma o un indirizzo. L'attributo questa volta non verrà tradotto, si effettuerà invece la normale traduzione dell'entità generale.
Verrà utilizzata una funzione in grado di distinguere i due casi in base all'estensione del file esterno, grazie alla quale si può dedurre se il contenuto è un'immagine o un testo.

5.10 Include e Ignore

INCLUDE e IGNORE. Sono due direttive, spesso usate con le entità parametriche, la cui funzione è quella di includere o rimuovere sezioni di una DTD personalizzandola.
Si utilizzano nel modo seguente:
<![INCLUDE [DTD section]]>
<![IGNORE [DTD section]]>
Cambiando il valore di un'entità parametrica da IGNORE a INCLUDE o viceversa è possibile includere o ignorare più sezioni di una DTD, si parla in questo caso di parametrizzazione della DTD.

Segue un esempio in cui si consentirà agli autori di includere o ignorare sezioni di una DTD semplicemente cambiando il valore di un'entità parametrica denominata *includer.* Per utilizzare un'entità parametrica nelle sezioni INCLUDE e IGNORE, è necessario usare il sottoinsieme esterno della DTD.

```
<!ENTITY %includer "INCLUDE">
<!ELEMENTO DOCUMENT (CUSTOMER)*>
<!ELEMENT CUSTOMER (NAME,DATE,ORDERS)>
<!ELEMENT NAME (LAST_NAME,FIRST_NAME)>
<!ELEMENT LAST_NAME (#PCDATA)>
<!ELEMENT FIRST_NAME (#PCDATA)>
<!ELEMENT DATE (#PCDATA)>
…….
<![ %includer; [
  <!ELEMENT PRODUCT_ID   (#PCDATA)>
  <!ELEMENT SHIP_DATE  (#PCDATA)>
   <!ELEMENT SKU (#PCDATA)>
  ] ]>
```

L'entità parametrica includer viene impostata di default al valore INCLUDE, ma variando il suo valore è possibile poi includere o ignorare la sezione finale.

L'uso di una tecnica di questo tipo consente di semplificare la centralizzazione delle entità che si desidera usare contemporaneamente per personalizzare una DTD.
In ODLI3 non è previsto l'uso di direttive con tale significato, per questo motivo, in fase di traduzione, sarà necessario produrre un file in ODLi$_3$ per ogni documento XML descritto da tale DTD, che presenti diverse impostazioni di include e ignore. Nei file prodotti verranno tradotte le sezioni INCLUDE e tralasciate le sezioni IGNORE.

5.11Tabella Riassuntiva

Riassumiamo gli algoritmi di traduzione illustrati finora in una tabella conclusiva, che mostra nella colonna sinistra le combinazioni di righe che si possono incontrare in una DTD XML, mentre sulla destra viene riportato, ove possibile, la corrispondente traduzione in ODLi3.

DTD XML	ODLi3

<!ELEMENT [name] [list]	Interface [name]
	(Source semistructured ….)
	{
	[…]
	}

<!ELEMENT[name] (#PCDATA)	Attribute string [name]

<!ELEMENT [name] ANY>
….
<!ELEMENT [l_x] [list]>
	Interface [name]
	(Source semistructured ….)
	{attribute [lx] [lx]
	};
	union
	{
	attribute string PCDATA_NODE
	}

<!ELEMENT [name] EMPTY>	enum [name]_info] { ‘[name]’; ‘no_[name]’; }
<!ELEMENT [name] ([l1],…[l_X]?,., [ln])> <!ELEMENT [l_x] (#PCDATA)	Attribute string [l_x]?;
<!ELEMENT [name] ([l1],…[l_X]*,., [ln])> <!ELEMENT [l_x] (#PCDATA)	Attribute set <string> [l_x];

<!ELEMENT [name] ([l1],…[l_X]+,., [ln])> <!ELEMENT [l_x] (#PCDATA)	Interface [name] (source semistructured…) {attribute [l_x] [l_x]; attribute set<[l_x] >[l_x] _1} ; typedef [l_x] {attribute string strvalue}

<!ELEMENT [name]
(#PCDATA | [other]) >
<!ELEMENT [other] [list]>
	Interface [name]
	(Source semistructured ….)
	{attribute [other] [other];
	};
	union
	{
	attribute string PCDATA_NODE
	}

<!ELEMENT [father] ([childA] [childB])> <!ELEMENT [childA] (#PCDATA)> <!ELEMENT [childB] [list]>	Interface [father] (source semistructured…) {attribute [father]_union_1 [father]_union_1; } typedef [father]_union_1 { string [childA]; } union { […]; }
<!ELEMENT [name][list] <!ATTLIST [name] [att1] CDATA [att2] CDATA ‘[val2]‘ [att3] CDATA #REQUIRED [att4] CDATA #FIXED [val4]	Interface [name] (source semistructured…) {[…]; attribute string [name]_[att1]*; attribute [name]_[att2] [name]_[att2]*; attribute string [name]_[att1]; const string [att4] =’ [val4]’ } interface [name]_[att2] (source semistructured…) {const string [name]_[att2]_value‘[val2]‘ ;} union {attribute string [name]_[att2]_node}

<!ATTLIST [name] [att1] CDATA #IMPLIED	attribute string [name]_[att1]*; Viene tradotto come un attributo normale
<!ATTLIST [name] [att1] ID #REQUIRED	Interface [name] (source semistructured … Key [name]_[att1]) {attribute string [name]_[att1]; …..} Tale traduzione è possibile solo se l'utente consente di trattare l'attributo come primary_key
<!ATTLIST [name] [att1] IDREF #IMPLIED	Interface [name] (source semistructured … foreign_Key [name]_[att1] references [interface]) {attribute string [name]_[att1]; …..} [interface] rappresenta il nome dell'interfaccia a cui il progettista ha indicato che tale foreign key referenzi, scegliendola tra varie interfacce estratte come candidate da una specifica funzione.
<!ATTLIST [name] [att1] NMTOKEN	attribute string [name]_[att1]*
<!ATTLIST [name] [att1] ([l1] |…| [ln]) ' [l_x] '>	enum [name]_[att1] { '[l_1]'; '[…]'; '[l_n]';} … attribute [name]_[att1] [name]_[att1];
<!ELEMENT [name] ([l1],.., [ln])> …….. <!ELEMENT [l_x] (#PCDATA)> ……. <!ENTITY [entname] '[value]'>	Interface [name] (source semistructured…) {…… const string [l_x]='[value]'; …} [l_x] è uno degli elementi all'interno dei quali viene usato il riferimento all'entità generale interna nel documento. Ognuno di questi viene trasformato in una costante come [l_x].

<!ELEMENT [name] ([l1],.., [ln])> …….. <!ELEMENT [l_x] (#PCDATA)> ……. <!ENTITY [entname] SYSTEM '[file.xml]' >	Interface [name] (source semistructured…) {…… const string [l_x]='[value]'; …} Tale traduzione è possibile solo se il contenuto del file xml, [value], può essere considerato come stringa di testo. Una funzione apre il file e inserisce il suo contenuto come valore della costante;
<!ENTITY %[entname] "<ELEMENT [name] (..)>" …… %BR;	Per la traduzione in ODLi$_3$ sarà necessario disporre di una funzione in grado di individuare il riferimento all'entità parametrica nella DTD, associarla all'entità corrispondente, ed effettuare la traduzione degli elementi o attributi in essa contenuti, collocandoli nella stessa posizione in cui si trova il riferimento. Per la traduzione degli elementi e degli attributi si seguono le regole già analizzate.
<!ATTLIST [name] [att1] ENTITY #IMPLIED> <!ENTITY [entname] SYSTEM '[value]'>	Const string [name]_[att1]='[value]
<!ENTITY % [entname] ([l1],.., [ln])> <!ELEMENT [name] (([f1] |..| [fm])*) <!ELEMENT [f_1] %[entname] .. <!ELEMENT [f_m] %[entname]	Interface [f_1] (..) {attribute ….[l1]; ….. attribute …[ln]; } …. Interface [f_m] (..) {attribute ….[l1]; ….. attribute …[ln]; }
<!ENTITY %[includer] [INCLUDE/IGNORE]'> …. <![%includer; [DTD section]]>	In ODLI3 non è previsto l'uso di direttive con tale significato, per questo motivo, in fase di traduzione, sarà necessario produrre un file in ODLi$_3$ per ogni documento XML descritto da tale DTD, che presenti diverse impostazioni di include e ignore. Nei file prodotti verranno tradotte le sezioni INCLUDE e tralasciate le sezioni IGNORE.

CAPITOLO 6
Esportazione di XML Schema in ODLi$_3$

6.1 Introduzione a XML Schema

Uno schema XML è una collezione di definizioni di tipi e dichiarazioni di elementi utilizzati nel documento XML che ne costituisce un'istanza. Gli schemi sono molto più potenti e precisi rispetto alle DTD, infatti sono stati pensati per fornire quel supporto di validazione che le DTD permettono solo parzialmente, in particolare sul contenuto degli elementi e degli attributi dei documenti XML.

6.1.1 Documento XML Schema

Un documento di XML Schema è racchiuso in un elemento **<schema>** e può contenere i seguenti elementi:

- **<element> e <attribute>** per la definizione di elementi e attributi globali;
- **<simpleType> e <complexType>** per la definizione di tipi;
- **<attributeGroup> e <group>** per definire serie di attributi e gruppi di content model complessi;
- **<annotation>** per esprimere commenti per l'utente o per applicazioni diverse dal parser XML;
- **<import> e <include>** per inserire frammenti di schema da altri documenti.

I namespace ricoprono un ruolo molto importante negli XML schema, nella sezione seguente verranno analizzati nel dettaglio.

6.1.2 XML Namespace

XML offre molta libertà ai programmatori e consente la definizione di tag personalizzati. Al crescere, tuttavia, del numero di applicazioni XML, è sorto un problema che non era stato previsto dai creatori delle specifiche originali XML: i conflitti dei nomi dei tag. La soluzione

è usare i namespace. Il concetto di namespace in XML è molto simile a quello usato in linguaggi di programmazione tipo java e C++: permette di evitare collisioni tra i nomi di elementi e attributi tramite l'uso di nomi qualificati, consistenti in un prefisso che indica il namespace cui l'elemento appartiene seguito da un nome locale a tale namespace.
Per definire un namespace, si assegna l'attributo *xmlns:prefix* ad un identificatore univoco che, in XML, è generalmente un URI (Uniform Resource Identifier).
ES: xmlns:book=http://www.starpowder.com/book
Dopo aver definito il namespace book è possibile premettere book: ad ogni tag e nome di attributo che appartiene a tale namespace:

```
<book:library
 xmlns:book="http://www.starpowder.com/book">
        <book:book>
                <book:title>
                    Learning Java
                </book:title>
        </book:book>
</book:library>
```

In questo modo tag e attributi vengono modificati, per esempio il tag <library> è stato sostituito da <book:library> per quanto riguarda il processore XML.
In particolare, quando si usa l'attributo xmlns autonomamente, senza specificare un prefisso, si definisce un namespace predefinito e si assume che tutti gli elementi racchiusi nel documento appartengano al namespace.
Un'osservazione importante riguardo gli XML namespace, che li distingue da quelli Java e C++, è che non implicano, di per sé, l'esistenza di alcun file o risorsa contenete la dichiarazione degli elementi e attributi che appartengono ad essi. L'URI usato come valore dell'attributo xmlns, in genere, non si risolve in alcuna risorsa effettivamente esistenete su qualche server, identifica semplicemente il namespace in modo univoco, così che le applicazioni possano interpretare correttamente i tag che si riferiscono a quel namespace.
Secondo quanto stabilito dal W3C, l'associazione fra schema e documento avviene proprio tramite i namespace. Sarà necessario, perciò, dichiarare un namespace per ogni documento che fa riferimento ad uno schema nel modo seguente:

```
<?xml version=”1.0” ?>
<transaction borrowDate=”2002-04-18”
      xmlns= “http://www.starpowder.com/schema”>
…….
```

Dipenderà quindi dal processore XML trovare lo schema a partire da questa descrizione. All'interno di uno schema è possibile dichiarare un namespace *target.* Quando il processore trova lo schema e verifica che il namespace target è identico a quello del documento, può validare il documento stesso.
Quando si utilizzano gli schemi del W3C si usa convenzionalmente il prefisso *xsd:*.
ES: `xmlns:xsd=”http://www.W3.org./2000/08/XMLSchema`

6.1.3 Tipi ed Elementi in XML Schema

Per dichiarare un elemento si usa l'espressione XML seguente:

<[prefix:]**element name**=[name element] **type**=[type element]>

dove **name** e **type** sono seguiti rispettivamente da nome e tipo dell'elemento, e sono suoi attributi.
Emerge perciò una differenza fondamentale rispetto alle DTD, la necessità di specificare il tipo degli elementi che si dichiarano con gli schemi. Ciò significa che il primo passo nel dichiarare gli elementi è accertarsi di avere i tipi che si desidera, e questo spesso richiede di definire nuovi tipi complessi. I tipi complessi possono racchiudere elementi e attributi, a differenza dei tipi semplici.
Nella tabella seguente vengono indicati i tipi semplici più significativi predefiniti negli schemi XML. (Quando si specificano questi tipi negli schemi bisogna anteporre il prefisso dello schema W3C, normalmente xsd:)

TIPO	DESCRIZIONE
binary	Valori binari
boolean	Valori come true, false, 1, 0
byte	Valore di un byte, massimo 255
date	Data, come 2002-04-18
decimal	Valori decimali
double	Numero a virgola mobile a 64 bit
ENTITY	Attributo ENTITY XML
float	Numero a mobile a 32 bit
ID	Attributo ID XML
IDREF	Attributo IDREF XML
int	Intero
language	Contiene un identificatore di lingua XML
long	Intero lungo
Name	Tipo Name XML
NegativeInteger	Intero negativo
NMTOKEN	Attributo NMTOKEN XML
NOTATION	Attributo NOTATION XML
PositiveInteger	Intero positivo
QName	Tipo XmlNamespace Qualified name
short	Intero breve
string	Stringa di testo
time	Contiene un'ora come 12:00:00.000
UnsignedByte/int/long/short	Valore senza segno
uriReference	Contiene un URY
Year	Contiene un anno come 2002

Per assicurare la compatibilità fra schemi XML e DTD, i tipi ID, IDREF, ENTITY, NOTATION e NMTOKEN vengono usati quando si dichiarano gli attributi.

6.1.4 Attributi

Gli attributi vengono dichiarati mediante la seguente espressione:

<[prefix:]**attribute name**=[name_attribute] **type**=[type_attribute]>

A differenza degli elementi, gli attributi devono essere sempre di tipo semplice, cioè non possono contenere struttura, e possono comparire al massimo una volta.
L'elemento attribute ha due attributi: use e value.
Use specifica se l'attributo è :

- **required**:è obbligatorio e può avere qualsiasi valore;
- **optional**:è facoltativo e può avere qualsisi valore;
- **prohibited**: non deve apparire.

Un attributo, di default, è optional.
Se l'attributo è optional Use può indicare anche se il suo valore è:

- **fixed**: ha un valore fisso che può essere impostato con **value;**
- **default**: se l'attributo non viene usato, ha un valore di default impostato con **value**, altrimenti ha il valore che gli è stato assegnato nel documento;

6.2 Traduzione Concetti di Base

IL lavoro di traduzione avrà il compito di riprodurre, dove possibile, un'analoga struttura ODLi$_3$ in grado di descrivere, senza ambiguità, i dati in formato XML. E' importante sottolineare che, poiché XML Schema fornisce un approccio object oriented, si presenta molto più vicino al linguaggio ODLi$_3$ di quanto non lo sia una DTD.
Prima operazione da fare è quella di prevedere il nome del file, che verrà lasciato inalterato, modificando l'estensione.
ES: book.xml $\Longrightarrow$ book.odl

6.3 Traduzione Elementi Semplici Predefiniti

Per poter effettuare la traduzione è necessario stabilire come convertire i tipi predefiniti di XML Schema nei tipi previsti dal linguaggio ODLi3.
Nella tabella seguente vengono riportate le regole di conversione adottate per i tipi di elementi (esclusi ID, IDREF, ENTITY, NOTATION e NMTOKEN, relativi agli attributi, che verranno trattati in seguito).

XML SCHEMA		ODLi3
boolean	⇨	**boolean**
Binary, byte, int, long, short, unsignedshort, unsignedlong, negativeinteger, positiveInteger, year	⇨	**integer**
Float, double, decimal	⇨	**float**
String, date, time, uriReference, Qname, Name	⇨	**string**

6.4 Traduzione Tipi Complessi

La traduzione di uno schema XML in ODLi3 inizia dall'elemento radice. Essendo questo un elemento complesso ci si soffermerà, per prima cosa, ad analizzare tali tipi.
E' possibile creare nuovi tipi complessi usando l'elemento:

<complexType name="[nametype]">

Una definizione di tipo complesso normalmente contiene dichiarazioni di elementi, riferimenti ad altri elementi e dichiarazioni di attributi.
Segue un esempio tratto da book.xsd, schema del documento book.xml, relativo alla registrazione di libri prestati. In questo caso si dichiara un tipo complesso chiamato address che contiene gli elementi che costituiscono l'indirizzo di una persona.

```
<xsd:complexType name="address">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        </xsd:complextype>
```

La traduzione di un complexType in ODLi$_3$ sarà effettuata generando un'interfaccia i cui attribute corrispondono ai vari componenti:

```
Interface address
(source semistructured book.xml)
{attribute string name;
  attribute string street;
  attribute string city;
  attribute string state;
  }
```

La definizione di address contiene solo dichiarazioni basate su tipi semplici, come xsd:string, perciò gli attribute prodotti saranno anche essi di un tipo semplice, dedotto dalle regole di conversione citate in precedenza.
Una volta definito un tipo è possibile dichiarare nuovi elementi di questo tipo, quindi un complexType può contenere anche elementi basati su altri complexType, come nell'esempio che segue:

```
<xsd:complexType name="transactionType">
        <xsd:element name="Lender" type="address">…..
```

In questo caso si genererà un attribute del tipo espresso da type, che dovrà essere tradotto tramite un'ulteriore interfaccia corrispondente al complextype address.

Interface transactiontype
(source semistructured book.xml)
{attribute address lender;
…
}
interface address
(..)
{…}

La regola generale di traduzione sarà la seguente:

- Ogni complexType di nome Cname viene tradotto generando un'interfaccia di nome Cname. Per ogni elemento Ename di tipo Typename, appartenente a tale complexType, si crea un'attribute di tipo Typename e nome Ename all'interno dell'interfaccia Cname. Se Typename è semplice si seguiranno le regole di traduzione fra tipi semplici, altrimenti si dovrà creare un'ulteriore interfaccia di nome Typename per tradurre il complextype relativo. La descrizione della sorgente prevede l'assegnazione del termine semistructured, dal momento che si tratta di un file XML, seguito dal nome del file.

6.4.1 Traduzione dell'Elemento radice

L'elemento radice si distingue in quanto non è contenuto in nessun altro elemento e racchiude tutti gli altri. Essendo di tipo complesso, alla sua dichiarazione sarà associata la definizione del complexType relativo, come nell'esempio:

<xsd:schema xmlns: xsd="http://www.W3.org./2000/08/XMLSchema"/>
<xsd:element name ="transaction" type="transactionType"/>
<xsd:complexType name="transactionType">
<xsd:element name="Lender" type="address"/>…….

La traduzione sarà effettuata creando un'interfaccia corrispondente al complexType associato, che diverrà radice del file ODLi$_3$, mentre la riga relativa alla dichiarazione dell'elemento non verrà tradotta.

```
Interface transactionType
(source semistructured book.xml)
{attribute address lender;
…
}
```

La regola generale è la seguente:

- L'elemento radice dello schema di nome Ename e tipo Typename viene tradotto generando l'interfaccia corrispondente al complexType di nome Typename, ignorando la dichiarazione dell'elemento stesso.

6.5 Traduzione Elementi di Riferimento

Si consideri la dichiarazione dell'elemento note nell'esempio seguente:

```
<xsd:element name="note" type="xsd:string">
……
<xsd:complexType name="transactionType">
        <xsd:element name="Lender" type="address"/>
        <xsd:element name="Borrower" type="address/">
        <xsd:element ref="note" >
        <xsd:element name="books" type="book">
</xsd:complexType>
```

In questo esempio non è stato dichiarato un nuovo elemento, è stato incluso, invece, un elemento *di riferimento* già esistente all'interno della definizione di un elemento complesso. Questo è possibile tramite l'attributo *ref.*

L'elemento di riferimento deve essere stato dichiarato in modo globale, cioè deve essere figlio diretto dell'elemento <schema>. L'uso dell'attributo ref risulta una tecnica potente, poiché consente di evitare di ripetere la definizione di elementi che esistono già globalmente.
Poiché in ODLi3 non esistono espressioni di questo tipo, sarà necessario disporre di una funzione che, quando trova un attributo ref, vada ad individuare la dichiarazione dell'elemento a cui fa riferimento. Ogni volta che in un complextype si trova un riferimento ad esso lo si tradurrà come un normale attribute interno alla relativa interfaccia.

```
Interface transactionType
(source semistructured book.xml)
{attribute address lender;
  attribute address borrower:
  attribute string note;
  attribute book books;
}
```

La regola generale adottata è la seguente:

- Un elemento definito globalmente, di nome Refname e tipo Typename, viene tradotto come attribute di nome refname e tipo Typename nell'interfaccia corrispondente ad ogni complexType in cui si trova il riferimento ad esso.

6.6 Ricorrenza degli Elementi

Negli XML Schema è possibile specificare il numero minimo e massimo di volte che un elemento può apparire con gli attributi minOccurs e maxOccurs.
Il valore predefinito per entrambi è 1. Per indicare che non esiste un limite superiore per l'attributo maxOccurs è necessario impostarlo come *unbounded.*
Non sempre si può effettuare una traduzione precisa in ODLi3, perciò vengono analizzati i vari modi di utilizzo di questi attributi.

1.

```
<xsd:complexType name="transactionType">
        <xsd:element ref="note" minOccurs="0" >
        ……
```

In questo caso l'elemento note può apparire una volta o non apparire in transactionType, infatti maxOccurs sarà uguale ad 1 di default.

In ODLi$_3$ esiste la possibilità di esprimere tale informazione con il simbolo ?, che indica cardinalità (0,1).

```
Interface transactionType
 (source semistructured book.xml)
{…..
  attribute string note?;
  }
```

2.

```
<xsd:complexType name="transactionType">
        <xsd:element ref="note" minOccurs="0" maxOccurs="unbounded">
        ……
```

In questo caso note può apparire da 0 ad n volte. La traduzione in ODLi$_3$ avviene mediante i tipi di collezione, in particolare tramite la dichiarazione dei set.

```
Interface transactionType
 (source semistructured book.xml)
{…..
  attribute set<string> note;
  }
```

3.

```
 <xsd:complexType name="transactionType">
<xsd:element ref="note" maxOccurs="unbounded"          ……
```

Questa volta note deve apparire in transactioType almeno una volta, cioè avrà cardinalità (1,n). Non è possibile una traduzione diretta in ODLi$_3$come nei casi precedenti, infatti non è prevista la possibilità di distinguere tra ricorrenze di cui è obbligatorio specificare almeno un valore oppure ricorrenze in cui tale specificazione non è obbligatoria.
Per questo si è pensato ad una traduzione in cui vengono creati due tipi di attribute, uno di tipo set per indicare la cardinalità (1,N), e l'altro con cardinalità (1,1) .

```
interface transactionType
(source semistructure book.xml)
{attributenote note,
 attribute set<note> note_1;
} ;
typedef note
{attribute string strvalue}
```

4.

```
<xsd:complexType name="transactionType">
    <xsd:element ref="note" minOccurs="2" maxOccurs="unbounded">
```

In questo caso note ha cardinalità (2,n). La tecnica di traduzione è simile alla precedente, verrà creato un numero di attribute di tipo note e nome note uguale al valore di minoccurs, e uno di tipo set.

```
interface transactionType
(source semistructure book.xml)
{attribute note note,
 attribute note note;
 attribute set<note> note_1;
} ;
typedef note
{attribute string strvalue}
```

5.

```
<xsd:complexType name=”transactionType”>
        <xsd:element ref=”note” maxOccurs=”5”
            ……
```

In questo caso non è possibile fornire una traduzione precisa in ODLi$_3$, infatti tale linguaggio non prevede la possibilità di indicare il numero massimo di occorrenze ammissibili per un elemento. Note verrà tradotto come un elemento con cardinalità (1,n), o (x,n) nel caso si abbia minOccurs=”x”.

```
interface transactionType
(source semistructure book.xml)
{attributenote note,
 attribute set<note> note_1;
} ;
typedef note
{attribute string strvalue}
```

Tutto questo viene riassunto nella regola generale sulle occorrenze:

- Ogni elemento Ename di tipo Typename appartenente ad un tipo complesso Cname, con:
  - minOccurs=0, viene convertito in un attribute dell’interfaccia Cname con nome Ename, tipo Typename e cardinalità (1,0), quindi con suffisso *.
  - minOccurs=0 e maxOccurs=unbounded, viene convertito in un attribute dell’interfaccia con nome Ename, e cardinalità (0,N), di tipo set<type_set>. Type_set indica il corrispondente del tipo di dato in XML.
  - minOccurs=x e maxOccurs=unbounded, viene convertito creando x attribute di nome Ename e tipo Ename, e un attribute di tipo set con set_type Ename e nome Ename_1. Si definisce

poi un nuovo tipo o una nuova interfaccia per specificare Ename.

- MinOccurs=x e maxOccurs=y, viene tradotto come se maxOccurs fosse indeterminato, perciò si ricade nei 2 casi precedenti.

6.7 Traduzione Attributi

ODLi3, come è stato detto, non differenzia tra attributi e sottoelementi, perciò anche i primi verranno tradotti come attribute.
In seguito vengono analizzati separatamente i vari modi di utilizzare gli attributi.

6.7.1 Attributi Optional

Si consideri l'esempio seguente, in cui al tipo address è stato aggiunto un attributo denominato phone:

```
<xsd:complexType name="address">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:attribute name="phone" type="xsd:string" use="optional""/>
 </xsd:complextype>
```

In questo caso l'attributo use poteva anche essere omesso, perché il suo valore è optional di default. La traduzione sarà la seguente:

```
Interface address
(source semistructured book.xml)
{attribute string name;
```

```
attribute string street;
attribute string city;
attribute string state;
attribute string address_phone?;
}
```

- Dato un attributo Aname di tipo Typename opzionale, riferito ad un tipo complesso Cname:
  1. si genera un attribute denominato utilizzando come prefisso il nome dell'elemento a cui è riferito, seguito da "_" e dal nome stesso dell'attributo;
  2. l'attribute è indicato come opzionale ed il tipo assegnato, nel caso Typename non sia già stato definito, è Cname_Aname.

E' possibile avere anche una situazione di questo tipo:

```
<xsd:complexType name="transactionType">
……
<xsd:attribute name="counter" type="xsd:int" use="fixed" value="400"/>
……
```

Questa dichiarazione crea un attributo di tipo intero denominato counter il cui valore è sempre 400. Un concetto di questo tipo viene espresso in ODLi$_3$ tramite le costanti, quindi la traduzione sarà:

```
interface transactionType
(source semistructure book.xml
{…..
const integer transactionType_counter='400';
}
```

I valori delle costanti ODLi$_3$ devono essere di tipo Basetype, perciò potrebbero nascere dei problemi se l'attributo dichiarato non fosse di un tipo predefinito. Tuttavia, si può affermare

che un attributo con valore fisso difficilmente sarà di un tipo non predefinito, in quanto la funzione dei nuovi tipi semplici è quella di aggiungere vincoli ai valori che l'elemento o l'attributo di quel tipo possono assumere. Per evitare problemi, anche se poco probabili, è necessaria una funzione che vada ad estrarre il valore associato a *base* nel simpleType relativo all'attributo da tradurre, ed indicare questo come tipo della costante prodotta.

La regola generale sarà:

- Un attributo Aname di tipo FIXED, riferito ad un tipo complesso Cname, viene tradotto inserendo nell'interfaccia Cname una costante di nome Cname_Aname, tipo Typename (o Cname_Aname se Typename non è ancora stato definito) e valore specificato da value. Nel caso Typename non sia un tipo predefinito, ma un nuovo tipo semplice, la costante prodotta sarà del tipo indicato nell'attributo base del sympleType relativo.

Si consideri ora la seguente dichiarazione di attributo:

```
<xsd:complexType name="transactionType">
……
<xsd:attribute name="counter" type="xsd:int" use="default" value="400"/>
……
```

Questo significa che l'attributo counter ha un valore di default se non è usato,e se è usato ha il valore che gli è stato assegnato.

Il valore preimpostato rappresenta una sceltra tra i valori di tipo int che l'attributo counter può assumere. Si tratta di una peculiarità non molto significativa dal punto di vista della struttura, quindi non è necessaria la traduzione, tuttavia è possibile esprimere tale informazione in ODLi3 creando una nuova interface contenente il costrutto union.

```
Interface transactionType
(….)
{attribute transactionType_counter transactionType_counter*
….}
```

```
 interface transactionType_counter
(…)
 {const integer transactionType_counter_value “excellent” ;}
  union
 {attribute integer counter_node}
```

In questo caso l’attributo viene promosso al rango di interfaccia
Esponiamo la regola

- Un attributo di nome Aname e tipo Typename, appartenente al complextype Cname, con use = default, viene tradotto inserendo nell’interfaccia Cname un attribute di tipo Cname_Aname (o Typename se è gia stato definito) e nome Cnam_Aname. Il nuovo tipo Cname_Aname viene tradotto creando un’altra interfaccia comprendente una costante di nome Cname_Aname_value, valore uguale a quello preimpostato e tipo Typename, e un semplice attribute di nome Aname_node e tipo Typename, separati dal costrutto union. Nel caso Typename non sia un tipo predefinito, ma un nuovo tipo semplice, la costante prodotta sarà del tipo indicato nell’attributo base del sympleType relativo.

6.7.2 Attributi Required

Si consideri l’esempio seguente:

```
<xsd:complexType name=”address”>
        <xsd:element name=”name” type=”xsd:string”/>
        <xsd:element name=”street” type=”xsd:string”/>
        <xsd:element name=”city” type=”xsd:string”/>
        <xsd:element name=”state” type=”xsd:string”/>
        <xsd:attribute name=”phone” type=”xsd:string” use=”required””/>
 </xsd:complextype>
```

In questo caso l'attributo phone deve necessariamente apparire e può assumere qualunque valore intero. La traduzione sarà la seguente:

```
Interface address
(source semistructured book.xml)
{attribute string name;
 attribute string street;
 attribute string city;
 attribute string state;
 attribute string address_phone;
 }
```

- Dato un attributo Aname di tipo Typename opzionale, riferito ad un tipo complesso Cname, si genera un attribute denominato utilizzando come prefisso il nome dell'elemento a cui è riferito, seguito da "_" e dal nome stesso dell'attributo;il tipo assegnato, nel caso Typename non sia già stato definito, è Cname_Aname.

6.7.3 Attributi Prohibited

Tali attributi non devono comparire, perciò la loro traduzione è inutile.

6.8 Traduzione Tipi Semplici

I tipi semplici racchiudono solo testo semplice, come numeri, stringhe, ma non possiedono alcun sottoelemento.
Per dichiararli si usa l'espressione seguente:

<**simpleType name**="[nametype]" **base**="[type]" >

Un nuovo tipo semplice deve essere basato su un tipo semplice già esistente (incorporato o creato), indicato tramite l'attributo **base.**

Per descrivere le proprietà dei nuovi tipi semplici, gli schemi XML usano le sfaccettature, le quali consentono di restringere i dati che possono essere contenuti nei tipi. Queste, insieme all'attributo base, sono indicate nell'elemento **<restriction>.** Con alcuni tipi è possibile utilizzare solo certe sfaccettature.

6.8.1 Sfaccettature applicabili ai tipi ordinati

Consideriamo, inizialmente, le sfaccettature applicabili ai tipi ordinati: maxInclusive, minInclusive, MaxExclusive, minExclusive, totalDigits e fractionDigits.

- ***maxInclusive, minInclusive/ MaxExclusive, minExclusive*** :

Nell'esempio seguente si crea un tipo semplice chiamato dayOfMonth che può contenere solo valori fra 1 e 31, compresi. In questo caso lo si può definire nel modo che segue:

```
<xsd:compexType name="book">
……..
<xsd:element name="day" type="dayOfMonth">
……..
</xsd:complexType>

<xsd:simpleType name="dayOfMonth" >
<xsd:restriction base="xsd:int">
<xsd:minInclusive value="1"/>
<xsd:maxInclusive value="31"/>
</xsd:restriction>
</xsd:simpleType>
```

In ODLi$_3$ esiste il tipo **RANGE,** che permette di definire variabili intere con un dominio limitato, perciò riesce ad esprimere il significato di tale simpleType. La traduzione sarà:

```
Interface book
(source semistructured book.xml)
{…….;
  attribute range 1 , 31 day;
    }
```

Utilizzando maxExclusive e minExclusive vengono indicati gli estremi esclusi, perciò nella traduzione il primo va diminuito di uno e il secondo aumentato di uno, in quanto range indica sempre gli estremi inclusi.

attribute range 0, 32 day

La regola generale adottata è la seguente:

- Un Elemento o un attributo di nome Name, appartenente ad un complexType Cname, di tipo Typename, dichiarato come simpleType, a cui sono associate le sfaccettature minInclusive e MaxInclusive, viene tradotto come un attribute denominatoName, di tipo range, i cui estremi sono quelli indicati dagli attributi value di minInclusive, il primo, e di maxInclusive, il secondo. Nel caso vengano usate minExclusive e maxExclusive la traduzione viene effettuata nello stesso modo, ma prima di inserire gli estremi in range bisogna diminuire di uno il primo e aumentare di uno il secondo.

- ***totalDigits e fractionDigits***

Il concetto espresso dalle due sfaccettature totalDigits e fractionDigits non è esprimibile in ODLi3, quindi un elemento o un attributo relativo ad un un simpleType che ne fa uso, viene tradotto come un attribute del tipo espresso da base. Se nell'esempio seguente si fossero usate tali sfaccettature la traduzione sarebbe stata: attribute int day.

6.8.2 Sfaccettature applicabili a tipi generici

Tali sfaccettature sono: leght, minlenght, maxlenght, pattern, enumeration e whitespace.

- ***Enumeration:***

La sfaccettatura enumeration consente di fornire un elenco dei valori che un tipo semplice può assumere, esattamente come si può fare nelle DTD. Per esempio, per impostare un tipo semplice denominato weekday, i cui valori possono essere i giorni della settimana, si dovrà definire il tipo nel modo seguente:

```
<xsd:complextype name="book">
  …..
   <xsd:element name="weekday" type="weekday">
 …….
</xsd:complexType>
<xsd:simpleType name="weekday">
 <xsd:restriction  base="xsd:string">
   <xsd:enumeration value="Sunday"/>
   <xsd:enumeration value="Monday"/>
   <xsd:enumeration value="Tuesday"/>
   <xsd:enumeration value="Wednesday"/>
   <xsd:enumeration value="Thursday"/>
   <xsd:enumeration value="Friday"/>
   <xsd:enumeration value="Saturday"/>
 </xsd:restriction>
</xsd:simpletype>
```

Per la traduzione in ODLi$_3$, si utilizza il costrutto enum:

```
enum weekday
{ 'Sunday';
  'monday';
  'Tuesday';
   …….
}
Interface book
(source semistructured book.xml)
{…….;
 attribute weekday weekday;
   }
```

- Un elemento di nome Ename di tipo semplice enumerato viene tradotto come un attribute di nome Ename e tipo Ename. Il nuovo tipo di nome Ename viene creato mediante il costrutto enum, elencando al suo interno i valori possibili.

- ***Pattern:***

  tale sfaccettatura viene utilizzata per specificare un'espressione regolare che i valori delle stringhe di testo di questo tipo devono soddisfare.

  ES:

```
<xsd:simpleType name="catalogID" base"xsd:string">
  <xsd:pattern value="\ d{3} -\ d{4} -\ d{3} "/>
<xsd:simpleType>
```

Il concetto espresso da pattern non è esprimibile in $ODLi_3$, quindi un elemento o un attributo relativo ad un simpleType che ne fa uso, viene tradotto come un attribute del tipo espresso da base.

- ***Whitespasce:***

  vale il discorso fatto per la sfaccettatura pattern.

- ***Lenght, minlenght, maxlenght:***

  non sono molto usati, ma il più delle volte vengono applicati ai List Types trattati nella sezione seguente, per aggiungere informazioni sulla lunghezza.

6.9 List Types

Il tipo Lists permette di generare liste di tipi. XML Schema comprende tre List Type predefiniti: NMTOKENS, IDREFS, e ENTITIES; per esempio la lista NMTOKENS è costituita da singoli NMTOKEN delimitati da spazi.
Oltre a questi, è possibile creare nuovi List Types che si basano sui tipi semplici esistenti.
(Non possono essere basati su List Type già esistenti o su tipi complessi).

ES:

```
 <xsd:compexType name="book">
```

```
……..
<xsd:element name=”pubPlace” type=”USStateList”>
……..
</xsd:complexType>

<xsd:simpleType name=”USState”>
<xsd:restriction base=”xsd:string”>
  <xsd:enumeration value=”AK”/>
  <xsd:enumeration value=”AL”/>
  ……
</restriction>
</simpleType>

<xsd:simpleType name=”USStateList”>
<xsd:restriction itemType=”USState”/>
</ xsd:simpleType>
```

Il valore dell'attributo **itemtype** è il tipo semplice degli elementi della lista.

ODLi$_3$ utilizza due tipi da collezione per generare liste: **list** e **set,** al primo è associato il concetto di ordine, al secondo no. Si tratta di un'informazione aggiuntiva rispetto ad XML Schema, che può essere introdotta dall'utente tramite interfaccia, indicando se effettuare una traduzione con list o con set.
Nell'esempio trattato l'ordine degli elementi non sembra importante, perciò è più appropriato l'uso di set:

```
Interface book
(source semistructured book.xml)
{…….;
  attribute set<USState> pubplace;
    }
enum USState
{ ‘AK’;
   ‘AL’;….}
```

Si supponga di aggiungere l'informazione che la lista deve essere composta esattamente da sei elementi USState, per fare questo si usa la sfaccettatura lenght:

ES:

```
 <xsd:compexType name="book">
……..
<xsd:element name="pubPlace" type="SixUSState">
……..
</xsd:complexType>

<xsd:simpleType name="USState">
<xsd:restriction base="xsd:string">
  <xsd:enumeration value="AK"/>
  <xsd:enumeration value="AL"/>
  ……
</restriction>
</simpleType>

<xsd:simpleType name="USStateList">
<xsd:restriction itemType="USState"/>
</ xsd:simpleType>

<xsd:simpletype name="SixUSState"
 <xsd:restriction base:="USStateList">
    <xsd:length value="6"/>
    </xsd:restriction>
</xsd:simpleType>
```

Praticamente length sta ad indicare la lunghezza di un vettore, perciò la traduzione sarà la seguente:

```
Interface book
(source semistructured book.xml)
{…….;
  attribute SixUSState pubplace;
    }
enum USState
{ 'AK';
   'AL';
 …….
}

typedef USState  SixUSState[6]
```

La regola generale da adottare è la seguente:

- Un elemento Ename il cui tipo è un List Type, di nome ListName, viene tradotto come un attribute di nome Ename utilizzando il tipo da collezione scelto dall'utente fra set e list. Il tipo base di set o list è quello indicato dall'attributo itemType del List Type e viene tradotto in base alle regole specifiche per quel tipo,trattate precedentemente.Nel caso venga utilizzata la sfaccettatura lenght, o maxlengh, l'elemento viene tradotto come un attribute di nome Ename e tipo ListName. Questo viene tradotto come vettore della lunghezza indicata dalla sfaccettatura, e tipo base uguale a quello indicato nell'itemType.

E' da notare che con maxlenght si può effettuare la stessa traduzione.
L'informazione che può essere aggiunta con minlenght, invece, non è traducibile in ODLi$_3$.

6.10 Union Types

Gli Union types consentono di creare nuovi tipi definiti come unione di tipi atomici o di List Types.

Nell'esempio seguente si utilizza uno Union Type per rappresentare gli Stati americani mediante abbreviazioni o liste di codici numerici. Lo Union Type zipUnion è costituito da un tipo atomico e da un List Type :

```
<xsd:complexType name="address">
    ……..
    <xsd:element name="zip" type="zipunion"/>
   .....................
</xsd:complexType>

<xsd:simpleType name="zipunion">
<xsd:union memberTypes="USState listOfInt"/>
</xsd:simpleType>

<xsd:simpleType name="USState">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="AK"/>
      <xsd:enumeration value="AL"/>
         ……..
  </restriction>
</simpleType>


<xsd:simpletype name="listOfInt"/>
  <xsd:list itemTpe="myInt"/>
</xsd:simpleType>
<xsd:simpletype name="myInt"/>
   <xsd:restriction base="xsd:int">
      <xsd.minInclusive value="10000"/>
      <xsd:maxInclusive value="99999"/>
   </xsd:restriction>
</xsd:simpleType>
```

Il valore dell'attributo **memberTypes** è una lista dei tipi dell'unione.

Esempi di istanze valide dell'elemento zip sono :

```
<zip>CA</zip>
<zip>9563 95977 95945</zip>
```

Anche il linguaggio ODLi$_3$ mette a disposizione, per questi casi, lo speciale costrutto **union**, che consente di definire più specifiche alternative per una stessa interfaccia. La traduzione sarà la seguente:

```
interface address
(…)
{…..
   attribute zipunion zip,
  ……
}

Interface zipunion
(…)
{ attribute USState USState;
};
union
{ attribute set<myInt> listOfInt;
}

enum USState_type
{ 'AK';
   'AL';
 …….
}

 typedef range 10000, 99999  myInt
```

La regola generale da adottare è la seguente:

- Dato un elemento Ename di tipo Union Type, con nome UnionName, viene generato un attribute Ename di tipo UnionName. Questo viene tradotto mediante una nuova interfaccia all'interno della quale vengono definiti i vari elementi che compaiono come valore di memberTypes, separati dal costrutto Union.

6.11 Definizioni di tipo Anonimo

Le definizioni di tipo anonimo consentono di evitare le definizioni complete di nuovi tipi a cui viene fatto riferimento una sola volta.
Si effettuano racchiudendo l'elemento <xsd:simpleType> o <xsd:complexType> all'interno della dichiarazione <xsd:element>. In questo caso non si dovrà assegnare un valore esplicito all'attributo type di element, poiché il tipo anonimo che si sta usando non possiede un nome.
Segue un esempio che in cui viene effettuata una definizione di tipo anonimo per l'elemento <book>.

```
<xsd:complexType name="books">
  <xsd:element name="book" >
     <xsd:complexType>
           <xsd:element name="bookTitle" type="xsd:string"/>
           <xsd:element name="pubdate" type="xsd:string" minoccurs="0"/>
           ……..
    </xsd:complextype>
  </xsd:element>
</xsd:complextype>
```

E' possibile usare anche i tipi semplici anonimi; nell'esempio seguente l'elemento maxdaysOut contiene il numero massimo di giorni che si potrà tenere un libro in prestito. Per impostare a 14 il numero massimo di giorni, si userà un nuovo tipo semplice anonimo con la sfaccettatura maxexclusive:

```
<xsd:complexType name=”books”>
  <xsd:element name=”book” >
    <xsd:complexType>
          <xsd:element name=”bookTitle” type=”xsd:string”/>
          <xsd:element name=”pubdate” type=”xsd:string” minoccurs=”0”/>
          …….
          <xsd:element name=”maxdaysOut” >
                <xsd:simpleType>
                  <xsd:restriction base=”xsd:int”>
                    <xsd:maxEclusive value=”14”/>
                  </xsd:restriction>
                </xsd:simpleType>
          </xsd:element>
    </xsd:complextype>
  </xsd:element>
</xsd:complextype>
```

In ODLi$_3$ non è possibile effettuare delle definizioni di tipo anonimo, perciò ad ogni tipo complesso verrà associata un interfaccia separata e per i tipi semplici si seguiranno le regole esposte.
In generale:

- Per ogni complexType definito in modo anonimo all’interno di un elemento Ename, si genera un’ interfaccia di nome Ename seguendo le regole precedenti, mentre l’elemento viene tradotto come un normale attribute di nome Ename e tipo Ename .
- Per ogni simpleType definito in modo anonimo all’interno di un elemento Ename, si genera un attribute Ename in corrispondenza dell’elemento, applicando le regole esposte riguardo i simpleType.

6.12 Contenuto degli Elementi

6.12.1 Tipi complessi derivanti da SimpleType.

Si consideri l'elemento seguente:

```
<xsd:element name="USPrice" type ="decimal"/>
```

Poichè decimal è un tipo semplice, tale elemento non può contenere attributi. La soluzione potrebbe essere definire un complextype che supporti la dichiarazione di un attributo, ma si vuole mantenere un contenuto di tipo semplice.
Per fare questo è necessario derivare il ComplexType da un SimpleType, come nell'esempio seguente:

```
<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:simpleContent>
       <xsd:extension base="xsd:decimal">
            <xsd:attribute name="currency" type="xsd:string"/>
       </xsd:exstension>
    </xsd:simpleContent>
  </xsd:complexType >
</xsd:element>
```

Il complexType viene utilizzato per definire un nuovo tipo anonimo; l'elemento **<simpleContent>** sta ad indicare che il suo modello di contenuto è costituito solo da dati di tipo carattere e non da altri elementi. Questo nuovo tipo si ottiene estendendo il simpleType decimal, cioè aggiungendo l'attributo currency.
Un'istanza di questo elemento può essere:

```
<internationalPrice currency="EUR">423.46</internationalPrice>
```

La traduzione in ODLi3 prevede la creazione di un'interfaccia per InternationalPrice che contenga due attributi, quello di tipo decimale e currency:

```
Interface internationalPrice
(…..)
{attribute string internationalPrice_currencY;
 attribute float  internationalPrice_node ;   }
```

La regola generale è la seguente:

- Un Complextype definito in modo anonimo all'interno di un elemento Ename, e derivante da un simpletype, viene tradotto generando un'interfaccia di nome Ename. Ogni attributo Aname e di tipo Typename è tradotto come un'attribute di nome Ename_Aname e tipo Typename, infine si genera un attribute di nome Ename_node del tipo indicato da base.

6.12.2 Elementi a Contenuto Misto

Con gli schemi, come con le DTD, è possibile creare elementi a contenuto misto, i quali sono in grado di contenere sia testo sia altri elementi figli.
Segue un esempio che mostra come si presentano gli elementi a contenuto misto usando lo schema book.xsd; in questo caso si è creato un nuovo elemento, chiamato <reminder>, che racchiude una lettera di sollecito a chi ha preso in prestito un libro affinchè lo restituisca:

```
<?xml version="1.0">
<reminder>
  Dear <name>Brigitta Regensburg</name>:
  the book <bookTitle>Snacking on Volcanoes</boohTitle>
  was supposed to be out for only <maxDaysOut>14</maxDaysOut>
  days. Please return it or pay $<replacementValue>17.99</replacementValue>.
 Thank you.
</reminder>
```

L'elemento <reminder> ha un modello di contenuto misto; per dichiararlo in uno schema si inizierà creando un nuovo tipo anonimo complesso all'interno della dichiarazione di <reminder>. L'elemento <complexType> ha un attributo, **mixed**, settato a true se il modello

di contenuto è misto**.** All'interno del tipo complesso si dichiarano gli elementi utilizzabili in reminder:

```
<xsd.element name="reminder">
  <xsd:complexType mixed="true">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="booktitle" type="xsd:string"/>
        <xsd:element name="maxDaysout">
                    <xsd:simpleType >
                        <xsd:restriction base="xsd:int">
                            <xsd:maxEclusive value="14"/>
                        </xsd:restriction>
                    </xsd:simpleType>
        </xsd:element>
        <xsd:element name="replacementValue" type="xsd:decimal"/>
  <xsd:complexType>
</xsd:element>
```

L'ordine e il numero di elementi figli deve corrispondere all'ordine e numero di elementi specificati nello schema.

Vengono fornite specifiche di sintassi molto più complete rispetto alle DTD, infatti in esse non è possibile vincolare il numero di elementi figli che devono comparire in un elemento a modello misto.

La traduzione in ODLi3 sarà la seguente:

```
Interface reminder
(…)
{attribute string node_1;
  attribute string name;
  attribute string node_2,
  attribute string bookTitle;
 attribute string node_3;
 attribute range 0, 15 maxdaysOut;
 attribute string node_4;
```

```
 attribute float replacementValue;
attribute string node_5
}
```

I nodi di tipo stringa introdotti servono a contenere il testo che può essere presente tra un elemento e l'altro. Una funzione dovrà incrementare di uno il numero associato ad ogni nodo introdotto.
La regola generale è la seguente:

- Per un elemento a contenuto misto si seguono le regole di traduzione di un normale elemeto definito in modo anonimo, però all'interno dell'interfaccia si introducono, prima e dopo ogni attribute corrispondente ad un elemento figlio, degli attribute di tipo string e nome "node_ " seguito da un numero, incrementato di uno per ogni nuovo attribute.

6.12.3 AnyType

Il tipo Anytype rappresenta un'astrazione chiamata *ur_type* , base da cui derivano sia i complexType che i simpleType.
ES: <xsd:element name="anithing" type="xsd:anytype"/>

Il contenuto di questo tipo di elementi non è vincolato in nessun modo.
Quando non viene specificato nulla ,il tipo di default è anyType, perciò l'esempio di prima può anche essere scritto nel modo seguente:
<xsd:element name="anithing"/>

Anche ODLi$_3$ mette a disposizione il tipo any per non vincolare in alcun modo la struttura di un elemento, quindi la traduzione è immediata:
attribute any anithing

Quando però si deve effettuare la traduzione di un elemento di tipo anonimo si segue la regola relativa esposta in precedenza.

6.12.4 Elementi Vuoti

Gli elementi vuoti sono elementi che non hanno contenuto, anche se possono avere attributi.
ES: <internationalPrice currency="EUR" value="423.46"/>

Nello schema è necessario definire un tipo che possa contenere solo elementi, senza però dichiararne nessuno, in modo che il contenuto del tipo risulti vuoto:

```
<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
          <xsd:attribute name="currency" type="xsd:string"/>
          <xsd:attribute name="value" type="xsd:decimal"/>
      </xsd:restriction >
    </xsd:complexContent>
  </xsd:complexType >
</xsd:element>
```

In questo esempio si definisce un tipo anonimo avente un contenuto di tipo **complexContent,** ciò significa che al suo interno possono comparire solo elementi. Il modello di contenuto viene ristretto agli elementi del tipo anyType, senza però introdurne nessuno, mentre vengono dichiarati i due attributi.
E' possibile dichiarare l'elemento internationalPrice in modo più compatto:

```
<xsd:element name="internationalPrice">
  <xsd:complexType>

          <xsd:attribute name="currency" type="xsd:string"/>
          <xsd:attribute name="value" type="xsd:decimal"/>
  </xsd:complexType >
</xsd:element>
```

Questo è possibile in quanto, un tipo complesso, definito senza specificare se il contenuto è un simpleContent o un complexContent, viene interpretato automaticamente come un complexContent ristretto al tipo anyType.

La traduzione in ODLi$_3$ sarà la seguente:

```
Interface internationalPrice
(…..)
{attribute string internationalPrice_currencY;
 attribute float  internationalPrice_value ;}
```

Se l'elemento di tipo empty non ha nessun attributo può essere visto come un possibile valore di un elemento enumerato. Si consideri l'esempio seguente:

```
<xsd:complexType name="book">
  <xsd:element name="lastVersion"
  <xsd:complexType>
    <xsd:complexContent>
       <xsd:restriction base="xsd:anyType">
         <xsd:minoccurs="0">
       </xsd:restriction >
    </xsd:complexContent>
   </xsd:complexType >
  </xsd:element>
  <xsd:element name="bookTytle" type="xsd:string"/>
   …….
</xsd:complexType>
```

Praticamente, in book.xml si può presentare una situazione di questo tipo:

```
<book>
   <lastVersion/>
   <bookTitle>Snaccking on volcanoes</bookTitle>
….
```

```
</book>
```

Questo elemento empty, in effetti, ha il significato di un attributo associato all'elemento book di un tipo semplice enumerato, in quanto la stessa informazione può essere trasmessa nel modo seguente:

```
<xsd:complexType name="book">
  <xsd:attribute name="lastVersion" type="lastVersion_type"/>
  <xsd:element name="bookTytle" type="xsd:string"/>
   .....
</xsd:complexType>

<xsd:simpleType name="lastVersion_type">
  <xsd:restriction  base="xsd:string">
    <xsd:enumeration value="lastVersion"/>
    <xsd:enumeration value="no_lastVersion"/>
</xsd:restriction>
</xsd:simpleType>
```

La traduzione sarà la seguente:

```
interface book
(…)
{attribute book_lastVersion  book_lastVersion;
  attribute string bookTitle;
……
 }
enum book_lastVersion
{'lastversion';
    'no_lastVersion';
}
```

La regola generale adottata è la seguente:

- Ogni elemento di nome Ename vuoto non contenente attributi viene tradotto come valore possibile di un attributo fittizio dell'elemento padre, di tipo enumerato e nome Ename_info, contenente due valori possibili : 'Ename' e 'no_Ename'. Se invece Ename contiene attributi viene tradotto come un normale complexType.

6.12.5 Creazione delle Scelte

Utilizzando l'elemento **<choice>,** è possibile specificare un numero di elementi, di cui solo uno sarà scelto.
Si consideri l'esempio seguente:

```
 <xsd:complextype name="book">
      <xsd:choice>
       <xsd:element name="booktitle" type="xsd:string"/>
       <xsd:element name="bookID" type="bookID"/>
      </xsd:choice>
   ……….
</xsd:complextype>

<xsd:complexType name=" bookID">
  <xsd:element name="publisher" type="string"/>
  <xsd:element name="code" type="int"/>
 </xsd:complexType>
```

La sintassi ODLi$_3$ ha possibilità di definire quali possono essere le forme che un attribute può assumere, per fare questo è necessario utilizzare il costrutto union e il costrutto typedef.
Le righe ODLi$_3$ da generare sono le seguenti:

```
interface book
(source semistructured…)
{attribute book_union_1 book_union_1;
```

```
}

typedef book_union_1
{ string bookTitle;
}
union
{string publisher;
  int code;
}
```

L'esempio che segue, tratto da book.xsd, costituisce un caso particolare di utilizzo delle choice. Il tipo transactionType viene impostato in modo che chi chiede il prestito può scegliere alcuni libri o solo uno. Si creerà l'elemento <xsd:choice> che contiene l'elemento <books> oppure <book>. Si noti che, in questo caso, book deve essere trasformato in un elemento globale, affinchè sia possibile fare riferimento ad esso nella scelta, per rimuoverlo dalla dichiarazione dell'elemento <books>.

```
<xsd:complexType name="transactionType">
        <xsd:element name="Lender" type="address"/>
        <xsd:element name="Borrower" type="address/">
        ……
        <xsd:choice>

                <xsd:element name="books" type="books">
                <xsd:element ref="book"/>
        </xsd:choice>
</xsd:complexType>

<complexType name="books">
     <xsd:element ref="book" minoccurs="0" maxoccurs="10"/>
</xsd:complextype>

<xsd:element name="book">
```

```
 <xsd:complextype>
        <xsd:element name=”booktitle” type=”xsd:string”/>
        <xsd:element name=”pubdate” type=”xsd:string”/>
              ……
 </xsd:complextype>
</xsd:element>
```

In questo caso la scelta è fra due elementi aventi la stessa struttura, infatti books è un riferimento a book, ciò che li distingue è solamente la cardinalità.
In ODLi$_3$ si produrrano le seguenti righe:

```
Interface transactiontype
(source semistructured book.xml)
{attribute address lender;
attribute address borrower;
attribute transactionType_union_3 transactionType_union_3;
}

interface transactionType_union_3
(..)
{attribute set<book> books;
}union
{attribute book book;
}

interface book
(..)
{attribute string booktitle;
 attribute string pubdate;
……
}
```

Il numero associato a transactionType_union_3 verrà calcolato da una specifica funzione, in base all'ordine di apparizione dell'elemento choice nella sequenza di sottoelementi del relativo complexType.

La regola generale è la seguente:

- Per ogni scelta che costituisce un componente di un elemento Ename bisogna introdurre nell'interfaccia Ename un attribute di nome Sname, composto da Ename, la parola chiave _union, ed infine "_" seguito da un numero che indica la posizione del componente nella lista, e tipo Sname. Il tipo Sname viene definito tramite typedef, nel caso gli elementi che compongono la scelta abbiano struttura semplice, tramite interface in caso contrario. All'interno del corrispondente di Sname le scelte sono separate da union.

6.12.6 Creazione delle Sequenze

Si consideri il solito esempio book.xsd, e si supponga di poter prendere in prestito non solo libri, ma anche riviste. Per fare questo, è possibile creare un nuovo gruppo chiamato booksAndMagazine.
Un gruppo riunisce insieme gli elementi. E' possibile assegnare un nome ai gruppi e quindi includere un gruppo in altri elementi usando **<xsd:group>** e facendo riferimento al gruppo con il nome.
Per essere certi che gli elementi all'interno del gruppo appaiano in un determinato ordine, si usa l'elemento **<xsd:sequence**>, nel seguente modo:

```
<xsd:group name=" booksAndMagazine">
  <xsd:sequence>
        <xsd:element ref="books"/>
        <xsd:element ref="magazine"/>
  </xsd:sequence>
</xsd:group>
<xsd:complexType name="transactionType">
         <xsd:element name="Lender" type="address"/>
         <xsd:element name="Borrower" type="address/">
```

```
……
<xsd:choice>
        <xsd:element name=”books” type=”books”>
        <xsd:element ref=”book”/>
        <xsd:group ref =”booksAndMagazine”/>
</xsd:choice>
</xsd:complexType>
```

La traduzione non presenta particolari problemi, un group può essere tradotto come un normale complexType,tuttavia l’informazione riguardante l’orine non viene mantenuta:

```
Interface booksAndMagazine
(..)
{attribute books books;
 attribute magazine magazine;
……
}
Interface transactiontype
(source semistructured book.xml)
{attribute address lender;
attribute address borrower;
attribute transactionType_union_3 transactionType_union_3;
}

interface transactionType_union_3
(..)
{attribute set<book> books;
}union
{attribute book book;
}union
{attribute booksAndMagazine booksAndMagazine;
}
```

Un gruppo di elementi Gname viene tradotto come fosse un normale complextype, quindi creando un'interfaccia Gname contenente i componenti del gruppo nell'ordine in cui compaiono nella sequence, e definendo, nell'interfaccia corrispondente all'elemento a cui è associato il gruppo, un attribute di nome Gname e tipo Gname.

6.13 Gruppi di Attributi

Usando l'elemento **<attributeGroup>** è possibile creare gruppi di attributi. Per esempio, si supponga di voler aggiungere un certo numero di attributi all'elemento <book> che descrive il libro. Per fare questo, si può creare un gruppo chiamato bookDescription che li contenga, e poi fare riferimento ad esso nella dichiarazione di <book>:

```
<xsd:element name="book">
  <xsd:complextype>
        <xsd:element name="booktitle" type="xsd:string"/>
        <xsd:element name="pubdate" type="xsd:string"/>
              ……
        <xsd:attributeGroup ref="bookDescription"/>
  </xsd:complextype>
</xsd:element>

<xsd:attributeGroup name="bookDescription">
      <xsd:attribute name="bookID" type="xsd:string"/>
      <xsd:attribute name="numberPages" type=xsd:int"/>
       ……
</xsd:attributeGroup>
```

Il processo di creare un gruppo di elementi o attributi, e quindi far riferimento ad esso, è simile all'uso delle entità parametriche nelle DTD lo scopo è quello di gestire del testo ripetuto nelle dichiarazioni. Nel caso non si voglia mantenere tale accorgimento nella traduzione, è sufficiente inserire gli attributi del gruppo nell'interfaccia book. Se invece si vuole evitare la ripetizione di dichiarazioni, e accelerare le correzioni in caso di modifiche,

soprattutto in schemi di grandi dimensioni, sarà necessario tradurre il gruppo come un interfaccia contenente tutti gli attributi:

```
interface bookDescription (..)
{attribute string bookDescription_ bookID;
  attribute int bookDescription_numberPages;
……}
interface book
(..)
{attribute string booktitle;
 attribute string pubdate;
……
attribute bookDescription bookDescription;}
```

- Un gruppo di attributi Gname viene tradotto come fosse un normale complextype, quindi creando un'interfaccia Gname contenente gli attributi componenti il gruppo, e definendo, nell'interfaccia corrispondente all'elemento a cui è associato il gruppo, un attribute di nome Gname e tipo Gname.

6.14 Gruppi All

Gli schemi supportano un altro tipo di gruppo: **all**. Tutti i suoi elementi possono essere presenti una volta o non esserlo del tutto, e apparire in qualsiasi ordine. Questo gruppo va usato al livello più alto del modello di contenuto e i figli del gruppo devono essere singolo elementi, cioè questo gruppo non può contenerne altri. In tale modello di contenuto, qualsiasi elemento non può comparire più di una volta (minOccurs=0 e maxOccurs=1).
ES:

```
<xsd:complextype name="transactionType">
    <xsd:all>
        <xsd:element name="Lender" type="address";
        <xsd:element name="borrower" type="address";
            ……
    </xsd:all>
```

</xsd:complexType>

Se lo si usa, <all> deve contenere tutte le dichiarazioni dell'elemento in un modello di contenuto, cioè non è possibile dichiarare elementi aggiuntivi al suo interno, ma solo all'esterno del gruppo.

Per quanto riguarda il fatto che gli elementi del gruppo possano comparire in qualunque ordine, tale informazione non può essere tradotta in ODLi3, tuttavia è possibile specificare la cardinalità (0,1):

```
Interface transactiontype
(source semistructured book.xml)
{attribute address lender*;
attribute address borrower*;
……
}
```

- Gli elementi di un gruppo all vengono tardotti come normali attribute di tipo opzionale, con cardinalità (0,1).

6.15 Note negli Schemi

Nelle DTD è possibile usare i commenti XML per aggiungere note e fornire documentazione. Gli schemi definiscono tre nuovi elementi che si usano per aggiungere annotazioni: <xsd:annotation>, <xsd:documentation>, <xsd:appInfo>.
L'elemento <**xsd:annotation**> contiene <xsd:documentation> e <xsd:appInfo>.
<**xsd:documentation**> contiene il commento per gli utenti, <**xsd:appInfo**> contiene note adatte alle applicazioni che effettuano letture del documento.
Quest'ultimo elemento è significativo se il documento è scritto in XML, ma ,una volta effettuata la traduzione in ODLi3, le informazioni in esso contenute non hanno più particolare importanza, per questo possono essere omesse.

Si consideri l'esempio seguente, in cui viene aggiunto un commento di spiegazione all'inizio dello schema:

```
<xsd:annotation>
  <xsd:documentation>
    Book borrowing transaction schema
  </xsd:documntation>
</xsd:annotation>
```

La traduzione di tale nota verrà effettuata generando una costante all'interno dell'interfaccia corrispondente all'elemento radice, in questo caso transactionType. Tale costante conterrà il testo del commento:

```
Interface transactiontype
(source semistructured book.xml)
{const string annotation_1 "Book borrowing transaction schema"

attribute address lender*;
attribute address borrower*;
……
}
```

La costante è stata chiamata annotation_1 per distinguerla da un eventuale commento interno al complexType transactionType, il quale verrà tradotto come annotation_2.
Il W3C raccomanda di usare l'attributo **xml:lang** all'interno di documentation, per specificare in che lingua è stato scritto il commento. Questo verrà tradotto come un'altra costante, contenuta nella nuova interfaccia annotation_1:

```
interface annotation_1
(..)
{const string annotation_1_value "Book borrowing transaction schema";
  const string annotation_1_lang "en";
}
```

La regola generale è la seguente:

- Dato un elemento annotation, viene tradotto solo l'elemento documentation. Si distinguono due casi:
    1. Se la nota si riferisce allo schema in generale, viene tradotta nell'interface dell'elemento radice. Per fare questo, nel caso a documentation non siano associati attributi, si genera una costante di nome annotation_ 1, tipo stringa, il cui valore è il testo del commento. Se documentation contiene l'attributo xml:lang, nell'interfaccia dell'elemento radice viene inserito un attribute di nome annotation_1 e tipo annotation_1, poi si genera un'interface annotation_1 contenente due costanti di tipo stringa, una, annotation_1_value, riporta il commento, e l'altra, annotation_1_lang, la lingua specificata.
    2. Se la nota è interna ad un complexType il procedimento è identico, ma invece di annotation_ 1, si utilizzerà annotation_ seguito dal numero relativo alla nota che si sta traducendo per quell'elemento.

6.16 Valore Nil

Quando un elemento viene indicato come opzionale, può accadere che non compaia in tutte le istanze del complexType in cui è definito. L'assenza di un elemento può essere spiegata in vari modi, per esempio può indicare la non conoscenza di un'iformazione. A volte si preferisce indicare esplicitamente la mancanza di contenuto dell'elemento, piùttosto che non farlo comparire. Per fare questo si utilizza, nel documento, l'attributo **xsi:nil** settato a true per indicare che l'elemento ha contenuto nullo, e l'attributo **nillable** nello schema, settato anche esso a true.
ES:
In book.xsd si avrà:

```
<xsd.element name=”pubDate“ type=“xsd:date“ nillable=“true“/>
```

mentre in book.xml:

<pubDate xsi:nil="true"></pubDate>

L'attributo nil viene definito come parte dell'XML schema per le istanze, http://www.w3.org/2001/XMLschema-istances , il cui prefisso convenzionale è xsi. E' da sottolinerare che tale metodo non può essere utilizzato per il valore degli attributi, e che un elemento nil non può contenere altri elementi, ma può avere attributi.

In ODLi$_3$ non può essere fatta tale distinzione tra elementi opzionali ed elementi che possono avere contenuto nullo, per questo entrambi vengono tradotti con cardinalità (0,1), o (0,n), a seconda del valore di maxOccurs.

Attribute string pubDate?

- Un elemento con attributo nillable=true viene tradotto come un attribute opzionale.

6.17 Traduzione di Schemi Complessi

Si analizzeranno ora alcuni concetti più complessi utilizzabili negli schemi, quali: l'uso di Namespace, la qualificazione degli oggetti, la costruzione di schemi in più documenti.

6.17.1 Schemi e Namespace

Una caratteristica molto importante degli schemi è quella di consentire ai processori XML di validare i documenti che usano i namespace, non gestibili dalle DTD. Per questo scopo l'elemento <schema> ha un attributo, targetNamespace, che specifica il namespace a cui si rivolge lo schema, cioè quello a cui è destinato. Ciò significa che se un processore valida un documento e verifica gli elementi in un particolare namespace, saprà quale schema verificare. L'idea alla base dei namespace target è che è possibile indicare al processore quali schemi utilizzare per la validazione del documento. Permettono, per esempio, di distinguere tra la dichiarazione di un *element* nel vocabolario di XML Schema, corrispondente al target namespace http://www.w3.org/2001/XMLshema, e la dichiarazione di un element in un altro vocabolario con un target namespace differente.

Nel caso in cui venga costruito uno schema senza targetNamespace, il W3C raccomanda che tutti i tipi e gli elementi vengano qualificati esplicitamente con un prefisso come xsd:, associato al namespace dell'XMLSchema, in modo da distinguere i tipi definiti dall'utente da quelli propri dell'XML Schema. Questo è il meccanismo adottato nell'esempio analizzato fin ora.

Nell'esempio che segue si dichiara un namespace di default, quello dello schema XML del W3C, in modo da non dover qualificare con il prefisso xsd i suoi elementi, come <annotation>, <complexType>, e i suoi tipi, come string, int, ecc. Http://www.starpowder.com/namespace, viene indicato come target namespace, a cui si associa il prefisso t, che deve essere assegnato ai tipi definiti in questo schema

```
<schema xmlns=http://www.w3.org/2001/XMLschema
        xmlns:t="http://www.starpowder.com/namespace"
        targetNamespace="http://www.starpowder.com/namespace"
.....
```

Non è possibile tradurre queste righe in $ODLi_3$, in quanto tale linguaggio non supporta l'uso dei namespace.

6.17.2 Qualificazione degli Oggetti

Lo sviluppatore può decidere se elementi e attributi dichiarati localmente debbano essere qualificati nel documento, specificando il prefisso del namespace corrispondente. Per indicare se gli elementi devono essere qualificati, si usa l'attributo **ElementFormDefault** dell'elemento <schema>, mentre per gli attributi si usa **AttributeFormDefault**; entrambi possono essere impostati a **"qualified"** o **"unqualified"**.

6.17.2.1 Oggetti Locali Non Qualificati

Nell'esempio che segue si indica che sia gli attributi che gli elementi locali non devono essere qualificati nel documento:

```
<schema xmlns=http://www.w3.org/2001/XMLschema
        xmlns:t="http://www.starpowder.com/namespace"
        targetNamespace="http://www.starpowder.com/namespace"
        elementFormDefault="unqualified"
        attributeFormDefault="unqualified">

    <annotation>
        ……..
    </annotation>
    <element name="transaction" type="t:transactionType"/>
    <complexType name="transactionType">
        <element name="Lender" type="t:address"/>
            ……
    </complextype>
……..
</schema>
```

In ODLi$_3$ non è previsto l'uso dei namespace, quindi le modalità di traduzione di elementi ed attributi saranno le stesse, indipendentemente dal prefisso assegnato. I valori di elementFormDefault e attributeFormDefault hanno implicazione a livello di istanza dello schema, cioè di documento, perciò non è necessario riportare il concetto da essi espresso nel file ODLi$_3$.

6.17.2.2 Oggetti Locali Qualificati

Per indicare che gli oggetti locali siano qualificati si impostano elementFormDefault e attributeFormDefault a qualified.

Mediante l'attributo **form** è possibile controllare la qualificazione di ogni singola dichiarazione. Nell'esempuio che segue tutti gli oggetti locali saranno non qualificati tranne bookID, che dovrà esserlo:

```
<schema xmlns=http://www.w3.org/2001/XMLschema
        xmlns:t="http://www.starpowder.com/namespace"
        targetNamespace="http://www.starpowder.com/namespace"
        elementFormDefault="unqualified"
        attributeFormDefault="unqualified">
    <annotation>
        ……..
    </annotation>
    <complexType name="books">
      <element name="book" minoccurs="o" maxoccurs="10"/>
     <complexType>
        <element name="bookTitle" type="string"/>
         …….
         <attribute name="bookID" type"t:catalogID" form="qualified">
    </complextype>
         ……
</schema>
```

Anche in questo caso, a proposito della traduzione, vale quanto detto per gli oggetti non qualificati, quindi anche l'attributo form verrà ignorato.

6.17.3 Schemi in Documenti Multipli

L'esempio analizzato fin ora è contenuto in un singolo documento, in realtà esiste la possibilità di comporre schemi utilizzandone altri, collocati in diversi documenti. Questo meccanismo è particolarmente utile nella costruzione di schemi di grandi dimensioni, in quanto è utile suddivere il contenuto fra più documenti, per facilitare il processo di controllo e di aggiornamento.

Si supponga ,per esempio, di collocare la definizione del complexType books in un nuovo file chiamato books.xsd, mentre chiameremo lo schema modificato starpowder.xsd:

Starpowder.xsd:

```
<schema xmlns=http://www.w3.org/2001/XMLschema
        xmlns:t="http://www.starpowder.com/namespace"
        targetNamespace="http://www.starpowder.com/namespace">
            <annotation>
             ……..
    </annotation>

    <include schemalocation="http://www.starpowder.com/schemas/books.xsd/>
    <element name="transaction" type="t:transactionType"/>
        <complexType name="transactionType">
            <element name="Lender" type="t:address"/>
            <element name="borrower" type="t:address"/>
            <element ref="t:note"/>
            <element name="books" type="t:books"/>
        </complexType>
     <element name="note" type="string"/>
    <complexType name="address" >
        <element name="name" type="string"/>
          ……
  </complexType>
</schema>
```

books.xsd:

```
<schema xmlns=http://www.w3.org/2001/XMLschema
        xmlns:t="http://www.starpowder.com/namespace"
        targetNamespace="http://www.starpowder.com/namespace">

    <annotation>
          ……..
```

```
    </annotation>


      <complexType name="books">
       <element name="book" minoccurs="o" maxoccurs="10"/>
          <complexType>
             <element name="bookTitle" type="string"/>
             <element name="pubDate" type="date" minoccurs="0"/>
             <element name="replacementValue" type="decimal"/>
             <element name="maxDaysOut"/>
                  <simpleType>
                      <restriction   base="integer">
                         <maxExclusive value="14"/>
                      </restriction>
                  </simpleType>
       </element>
    </complextype>
</schema>
```

L'elemento **include** ha la funzione di rendere disponibili le dichiarazioni e definizioni interne a book:xsd come parte di http://www.starpowder.com/namespace, utilizzato come namespace target per entrambi i file.
Esiste la possibilità di includere anche più schemi, che a loro volta ne contengono altri, tuttavia i documenti innestati sono corretti solo se i vari componenti hanno tutti lo stesso target namespace.

Per effettuare una corretta traduzione in ODLi3 sarà necessaria una funzione che vada ad aprire il file indirizzato dal valore dell'attributo schemalocation, il cui contenuto verrà tradotto come fosse interno allo schema principale.

```
Interface transactionType
(source semistructured book.xml)
{attribute address lender;
 attribute address borrower:
```

```
  attribute string note;
  attribute books books;
}
Interface book
  (source semistructured book.xml)
  {attribute string booktitle;
   attribute string pubdate;
   attribute range 0, 15 maxdaysOut;}
```

6.17.4 Derivazione di Tipi Complessi da altri Tipi complessi

Si consideri l'esempio seguente:

```
<complexType name="address">
    <element name="name" type="string"/>
    <element name="street" type="string"/>
    <element name="city" type="string"/>
</complexType>

<complexType name="USAaddress">
  <complexContent>
      <extension base="t:address">
          <element name="state" type="t:USState"/>
          <element name="zip" type="positiveInteger"/>
      </exstension>
  </complexcontent>
</complexType>
```

Il complextype USAaddress definito in questo modo contiene tutti gli elementi definiti in address, più degli elementi che sono specifici dell'indirizzo USA. La tecnica di derivare nuovi tipi per estensione di tipi già esistenti, è la stessa usata nella sezione 2.8.1, qui però il tipo

base è un complexType. Il fatto che il contenuto di USAaddress sia complesso è indicato dall'elemento **complexcontent,** mentre l'attributo base indica il tipo che è stato esteso.
In ODLi3 è possibile tradurre tale concetto creando due interfacce, una per address, l'altra per USAaddress, indicando che quest'ultima deriva dalla prima:

```
Interface address
(source semistructured book.xml)
{attribute string name;
   ……. }
Interface USAaddress: address
(source semistructured book.xml)
{attribute USState state;
 attribute int zip;
 }
Interface USState
  …..
```

Mediante l'uso dell'elemento restriction è possibile derivare tipi complessi da altri tipi complessi per restrizione, cio' modificando la definizione di alcuni elementi. In questo caso, però, la traduzione prevede che si generino interfacce distinte per ciascun complexType, in quanto non è previsto tale tipo di ereditarietà.

6.17.5 Ridefinizione di tipi e Gruppi

Il meccanismo di redefine permette di ridefinire tipi semplici e complessi, gruppi, attributi e gruppi di attributi contenuti in uno schema collocato in un file esterno. Come l'elemento include, redefine incorpora i componenti esterni nel target namespace dello schema di ridefinizione, e permette di ridefinirne alcuni se necessario.
Nell'esempio che segue viene modificata la definizione del complexType books, appartenente a book.xsd, utilizzando redefine in starpowder.xsd

```
<schema xmlns=http://www.w3.org/2001/XMLschema
```

```
        xmlns:t="http://www.starpowder.com/namespace"
        targetNamespace="http://www.starpowder.com/namespace">

            ……
    <redefine schemaLocation=http://www.starpowder.com/schemas/books.xsd>
      <complexType name="books">
        <complexContent>
          <exstension base="t:books">
            <element name="numberOfPages" type="int">
          </exstension>
        </complexContent>
      </complexType>
……
</schema>
```

L'elemento ridefine include tutte le dichiarazioni e definizioni del file books.xsd, modificando il complextype books. Il fatto di definire un tipo complesso con lo stesso nome della base da cui deriva potrebbe causare un errore, ma in questo caso la nuova definizione di books diventa l'unica, perciò non possono sorgere dei problemi.

La traduzione in ODLi3 viene effettuata aprendo il file books.xsd e traducendo tutte le dichiarazioni e definizioni che esso contiene, inserendo, però, anche le modifiche apportate da ridefine.

```
Interface books
(…)
(source semistructured book.xml)
{attribute string booktitle;
 attribute string pubdate;
    attribute range 0, 15 maxdaysOut;
   ……
 attribute int numberOfPages;}
```

6.17.6 Substitution Groups

Il meccanismo di Substitution Group consente l'assegnazione di elementi a speciali gruppi, definiti come sostitutivi per tali elementi.
Nell'esempio che segue vengono dichiarati due elementi, authorComment e readerComment, assegnati al gruppo sostitutivo comment, in modo da poter essere usati ogni volta che può essere usato comment:

```
<complexType name="books">
    <element name="book" minoccurs="o" maxoccurs="10"/>
        <complexType>
            <element name="bookTitle" type="string"/>
            <element name="pubDate" type="date" minoccurs="0"/>
            <element name="replacementValue" type="decimal"/>
            <element name="maxDaysOut"/>
                <simpleType>
                    <restriction   base="integer">
                        <maxExclusive value="14"/>
                    </restriction>
                </simpleType>
            <element name="Comment" type="string"/>
    </element>
  </complextype>
<element name="authorComment" type="string"
        substituitionGroup="t:comment"/>
<element name="readerComment" type="string"
        substituitionGroup="t:comment"/>
```

Poiché in ODLi3 non è previsto un costrutto di questo tipo, nella traduzione sarà necessario definire due nuovi attribute di tipo string, uno di nome readerComment e l'altro authorComment, entrambi opzionali, come Comment. In questo modo viene data a tutti e tre la possibilità di comparire nell'istanza dello schema.

```
Interface books
   (source semistructured book.xml)
{attribute string booktitle;
 attribute string pubdate;
     ……
    attribute string comment*;
attribute string authorcomment*;
attribute string readerComment*;}
```

6.17.7 Elementi e Tipi Abstract

XML Schema prevede un meccanismo per forzare la sostituzione di particolari tipi ed elementi. Un elemento o un tipo dichiarato come abstract non può essere usato in un'istanza: se si tratta di un elemento, nell'istanza deve comparire un membro del suo substitution group, se si tratta di un tipo di un elemento, tutte le istanze di quell'elemento devono usare xsi:type per indicare un tipo derivato da esso che non sia abstract.
Nell'esempio trattato nella sezione precedente, può essere utile non permettere l'uso dell'elemento comment, e forzare quello di authorComment e readerComment. La dichiarazione di comment deve essere la seguente:

```
<element name="comment" abstract="true"/>
```

In questo modo le istanze dello schema saranno valide solo se conterranno authorComment e readerComment.
In questo caso, nella traduzione, l'interfaccia book dovrà contenere solo questi due attribute, e non comment. L'opzionalità sta ad indicare che non devono necessariamente presenti entrambi.

```
Interface books
   (source semistructured book.xml)
{attribute string booktitle;
 attribute string pubdate;
```

```
……
attribute string authorcomment*;
attribute string readerComment*;}
```

6.18 Traduzione Concetti Avanzati

Per analizzare i concetti avanzati utilizzati in XML Schema, facciamo riferimento al seguente schema, denominato report.xsd:

```
<schema targetNamespace="http://www.example.com/Report"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:r="http://www.example.com/Report"
        xmlns:xipo="http://www.example.com/IPO"
        elementFormDefault="qualified">

 <import namespace="http://www.example.com/IPO"/>
 <annotation>
  <documentation xml:lang="en">
   Report schema for Example.com
   Copyright 2000 Example.com. All rights reserved.
  </documentation>
 </annotation>

 <element name="purchaseReport">
  <complexType>
   <sequence>
    <element name="regions" type="r:RegionsType">
     <keyref name="dummy2" refer="r:pNumKey">
      <selector xpath="r:zip/r:part"/>
      <field xpath="@number"/>
     </keyref>
    </element>
    <element name="parts" type="r:PartsType"/>
```

```
 </sequence>
 <attribute name="period"       type="duration"/>
 <attribute name="periodEnding" type="date"/>
</complexType>

<unique name="dummy1">
 <selector xpath="r:regions/r:zip"/>
 <field xpath="@code"/>
</unique>

<key name="pNumKey">
 <selector xpath="r:parts/r:part"/>
 <field xpath="@number"/>
</key>
</element>

<complexType name="RegionsType">
 <sequence>
  <element name="zip" maxOccurs="unbounded">
   <complexType>
    <sequence>
     <element name="part" maxOccurs="unbounded">
      <complexType>
       <complexContent>
        <restriction base="anyType">
         <attribute name="number"   type="xipo:SKU"/>
         <attribute name="quantity" type="positiveInteger"/>
        </restriction>
       </complexContent>
      </complexType>
     </element>
    </sequence>
    <attribute name="code" type="positiveInteger"/>
   </complexType>
```

```
  </element>
 </sequence>
</complexType>

<complexType name="PartsType">
 <sequence>
  <element name="part" maxOccurs="unbounded">
   <complexType>
    <simpleContent>
     <extension base="string">
      <attribute name="number" type="xipo:SKU"/>
     </extension>
    </simpleContent>
   </complexType>
  </element>
 </sequence>
</complexType>

</schema>
```

Si tratta dello schema del file Report.xml, che elenca i tipi di prodotti venduti in ogni regione, indicandone il numero identificatore e la quantità. Risulta chiaro che ogni zip code può comparire una volta sola, come la descrizione di ogni vendita, però ogni prodotto può essere venduto in diversi zip code:

```
<purchaseReport
  xmlns="http://www.example.com/Report"
  period="P3M" periodEnding="1999-12-31">

 <regions>
  <zip code="95819">
   <part number="872-AA" quantity="1"/>
   <part number="926-AA" quantity="1"/>
   <part number="833-AA" quantity="1"/>
   <part number="455-BX" quantity="1"/>
```

```
 </zip>
 <zip code="63143">
  <part number="455-BX" quantity="4"/>
 </zip>
</regions>

<parts>
 <part number="872-AA">Lawnmower</part>
 <part number="926-AA">Baby Monitor</part>
 <part number="833-AA">Lapis Necklace</part>
 <part number="455-BX">Sturdy Shelves</part>
</parts>

</purchaseReport>
```

6.18.1 Unicità

Utilizzando l'elemento **unique** è possibile specificare che il valore di un attributo o di un elemento particolare deve essere unico. Prima di tutto si seleziona un set di elementi, poi si identifica il campo che deve essere unico.
Si consideri l'esempio seguente tratto da report.xsd:

```
 <unique name="dummy1">
  <selector xpath="r:regions/r:zip"/>
  <field xpath="@code"/>
 </unique>
```

L'attributo xpath dell'elemento selector contiene un'espressione Xpath, regions/zip, che seleziona una lista di tutti gli elementi zip presenti nell'istanza, mentre l'attributo xpath dell'elemento field specifica che il valore dell'attributo code di questi elementi deve essere unico.

Si ha anche la possibilità di indicare l'unicità di una combinazione di campi, per esempio si supponga che ogni prodotto debba essere listato una volta sola all'interno di uno zip code. Per fare questo si aggiungono degli elementi field per identificare tutti i campi coinvolti:

```
<unique name="dummy1">
  <selector xpath="r:regions/r:zip"/>
  <field xpath="@code"/>
  <field xpath="r:part/@number"/>
 </unique>
```

In ODLi3 esiste la possibilità di indicare delle chiavi candidate, le quali possono essere semplici, nel caso coinvolgano un solo attributo, o composte, nel caso coinvolgano più attributi. Tale vincolo di unicità dà agli elementi la possibilità di essere identificati dal campo specificato, quindi è traducibile facendo uso proprio del costrutto candidate_key. Tramite l'elemento selector si ricava il contesto, cioè in quale interfaccia collocare la chiave candidate, mentre field indica quale attribute dovrà svolgere tale funzione.

```
Interface zip
(source semistructured Report.xml
  candidate_key code)
{attribute part part;
  attribute int code;
}
```

6.18.2 Chiavi e Referenze

In Report.xml la descrizione di ogni vendita appare una volta sola. Esiste la possibilità di rinforzare tale vincolo di unicità assicurando che ogni elemento part-quantity, listato in uno zip code, sia associato ad una descrizione. Per fare questo si utilizzano gli elementi key e Keyref, la cui sintassi e' uguale a quella di unique. La dichiarazione di number come chiave significa che il suo valore deve essere unico e non nullo, e il nome associato alla chiave la rende referenziabile a se stessa.

```
<element name="purchaseReport">
  <complexType>
   <sequence>
    <element name="regions" type="r:RegionsType">
     <keyref name="dummy2" refer="r:pNumKey">
      <selector xpath="r:zip/r:part"/>
      <field xpath="@number"/>
     </keyref>
    </element>
…….
</element>
……..
<key name="pNumKey">
   <selector xpath="r:parts/r:part"/>
   <field xpath="@number"/>
  </key>
 </element>
…….
<complexType name="PartsType">
  <sequence>
   <element name="part" maxOccurs="unbounded">
    <complexType>
     <simpleContent>
      <extension base="string">
       <attribute name="number" type="int"/>
      </extension>
     </simpleContent>
    </complexType>
   </element>
  </sequence>
 </complexType>

……
```

Per specificare che ogni elemento part-quantity deve avere una descrizione corrispondente, si indica che l'attributo number di quell'elemento deve referenziare la chiave pNumKey.La dichiarazione di number come Keyref non implica che il suo valore debba essere unico, ma che deve esistere una pNumKey con lo stesso valore.

Nel caso della traduzione fra DTD e ODLi$_3$ le chiavi avevano costituito un problema, poichè i tipi ID e IDREF costituiscono un tentativo poco efficace di esprimere i vincoli di integrità referenziale. Il tipo IDREF, infatti, vincola ad avere il valore di un qualsiasi attributo ID, ma non specifica quale, per questo nella traduzione è richiesto l'intervento dell'utente. XML schema mantiene queste definizioni per ragioni di compatibilità con il passato, ma introduce nuovi meccanismi che rendono la traduzione abbastanza semplice, in quanto ODLi$_3$ prevede due costrutti analoghi: key e foreig_key.

```
Interface parts_part
(source semistructured Report.xml
  Key (number))
{…….
attribute int number;
}

Interface zip_parts
(source semistructured Report.xml
  foreign_key (dummy2) references parts_part)
{ attribute int number;
 attribute iny quantity ;
}
```

Tramite le espressioni xpath interne alla definizione della chiave, è possibile individuare l'attributo (o gli attributi) da cui è costituita e l'interfaccia di cui deve far parte.

Mentre, tramite le espressioni xpath interne alla definizione di Keyref, si individua a quale interfaccia deve appartenere la foreign_key e a quale primary_key deve fare riferimento.

Nello schema il nome della chiave è pNumKey, ma l'attributo che la costituisce è number, perciò, per non creare ambiguità, nell'interfaccia viene tradotta con questo nome, che sarà lo stesso dell'attribute corrispondente.

6.18.3 Tipi Importati

Il meccanismo di **import** permette di utilizzare insieme componenti appartenenti a target namespace diversi.In report.xsd viene usato il tipo xipo:SKU, definito in un altro schema. L'elemeneto import identifica il target namespace di appartenenza del tipo SKU, e associa ad esso il prefisso xipo, in modo tale che possa essere referenziato in questo schema.

```
<schema targetNamespace="http://www.example.com/Report"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:r="http://www.example.com/Report"
        xmlns:xipo="http://www.example.com/IPO"
        elementFormDefault="qualified">

 <import namespace="http://www.example.com/IPO"/>
.....
      <element name="part" maxOccurs="unbounded">
       <complexType>
        <complexContent>
         <restriction base="anyType">
          <attribute name="number"   type="xipo:SKU"/>
          <attribute name="quantity" type="positiveInteger"/>
         </restriction>
        </complexContent>
       </complexType>
```

Per la traduzione sarà necessaria una funzione che vada ad aprire il file contenente lo schema indirizzato da import e ad estrarre la definizione del tipo SKU. Tale definizione verrà tradotta come se facesse parte di Report.xsd.

6.18.4 Elemento Any

Supponiamo che in Report.xml sia contenuta una descrizione delle vendite in formato HTML. Il contenuto HTML è racchiuso dall'elemento htmlExample, e tutti i suoi elementi appartengono al namespace http//www.w3.org/1999/xhtml.

Per permettere l'inserimentodel testo HTML, è necessario modificare Report.xsd, aggiungendo htmlExample, il cui contenuto è definito dall'elemento any. Generalmente un elemento any specifica che ogni testo xml ben formato può essere inserito in un modello di contenuto. In questo esempio si richiede che il testo XML appartenga al namespace sopra indicato, cioè che sia HTML.

```
<element name="purchaseReport">
 <complexType>
  <sequence>
   <element name="regions" type="r:RegionsType"/>
   <element name="parts"   type="r:PartsType"/>
   <element name="htmlExample">
    <complexType>
     <sequence>
      <any namespace="http://www.w3.org/1999/xhtml"
          minOccurs="1" maxOccurs="unbounded"
          processContents="skip"/>
     </sequence>
    </complexType>
   </element>
  </sequence>
  <attribute name="period"       type="duration"/>
  <attribute name="periodEnding" type="date"/>
 </complexType>
</element>
```

I namespace possono essere utilizzati per permettere o proibire che un elemento abbia un certo tipo di contenuto, a seconda del valore assegnato all'attributo **namespace:**

Valore dell'attributo namespace	Element Content permessi
##any	Qualunque XML ben formato
##local	Qualunque XML non qualificato(cioè privo di dichiarazione di namespace)
##other	Qualunque XML ben formato che non deriva dal target namespace
"http://www.w3.org/1999/xhtml ##targetNamespace"	Qualunque XML ben formato che appartenga ad un namespace della lista

Oltre ad any, esiste un attributo **anyAttribute** che consente agli attributi di apparire negli elementi. Per esempio,aggiungendo anyAttribute all'interno di htmlexample, si permette ad ogni attributo HTML di essere inserito in questo elemento:

```
<element name="htmlExample">
 <complexType>
  <sequence>
   <any namespace="http://www.w3.org/1999/xhtml"
        minOccurs="1" maxOccurs="unbounded"
        processContents="skip"/>
  </sequence>
  <anyAttribute namespace="http://www.w3.org/1999/xhtml"/>
 </complexType>
</element>
```

Dal momento che il linguaggio ODLi$_3$ non prevede l'uso di namespace, non è possibile fornire una traduzione di tali concetti. L'elemento htmlExample verrà tradotto come un attribute di tipo any, in modo da non dover apecificare alcun modello di contenuto:

```
 Interface purchaseOrder
(……..)
{attribute regionstype region,
  attribute partstype parts;
  attribute any htmlexample ;
}
```

6.19 Tabella Riassuntiva

Riassumiamo gli algoritmi di traduzione illustrati finora in una tabella conclusiva, che mostra nella colonna sinistra le combinazioni di righe che si possono incontrare in una XML schema, mentre sulla destra viene riportato, ove possibile, la corrispondente traduzione in ODLi3. Per questioni di chiarezza nel caso delle chiavi si riporta un esempio.

XML SCHEMA	ODLi$_3$
<complexType name=”[nametype]”> <element name=”[nameel]”type=”[type]”>	Interface [nametype] (Source semistructured...) {attribute [type] [nameel];} Se type è semplice si applicano le regole di traduzione dei tipi semplici o predefiniti, altrimenti si crea anche: Interface [type] (Source semistructured...) {.....;}
<element name=”[nameel]”type=”[type]”> ... <complexType name=”[nametype]”> <element ref=”[nameel]”	Interface [nametype] (Source semistructured...) {attribute [type] nameel;} Necessaria una funzione che quando trova un attributo ref, va ad individuare la dichiarazione dell’elemento a cui fa riferimento.
<complexType name=”[nametype]”> <element name=”[nameel]”type=”[type]” minOccurs=”0” >	Interface [nametype] (Source semistructured...) {attribute [type] [nameel]*;}
<complexType name=”[nametype]”> <element name=”[nameel]”type=”[type]” minOccurs=”0” maxOccurs=”unbounded”>	Interface [nametype] (Source semistructured...) {attribute set<[type]> [nameel];}
<complexType name=”[nametype]”> <element name=”[nameel]”type=”[type]” maxOccurs=”unbounded”>	Interface [nametype] (Source semistructured...) {attribute [type] [nameel] attribute set<[type]> [nameel_1];}
<xsd:complexType name=”[nametype]”> <xsd:element name=”[nameel]”type=”[type] minOccurs=”2” maxOccurs=”unbounded”>	Interface [nametype] (Source semistructured...) {attribute [type] [nameel]; attribute [type] [nameel]; attribute set<[type]> [nameel_1];}
<xsd:complexType name=”[nametype]”>> <xsd:element name=”[nameel]”type=”[type] maxOccurs=”5”>	Tradotto con cardinalità (1,n) o (x,n)

<complexType name=”[nametype]”> <attribute name=”[nameatt]”type=”[type]” use=”optional”>	Interface [nametype] (Source semistructured...) {attribute [type] [nametype_nameatt]?;}
<complexType name=”[nametype]”> <attribute name=”[nameatt]”type=”[type]” use=”fixed” value=”200”>	Interface [nametype] (Source semistructured...) {const [type] [nametype_nameatt]=’200’;}
complexType name=”[nametype]”> <attribute name=”[nameatt]”type=”[type]” use=”default” value=”200”>	Interface [nametype] (Source semistructured...) { attribute [nametype_nameatt] [nametype_nameatt]?;} Interface [nametype_nameatt] {const [type] [nametype_nameatt_value]=“200” ;} union {attribute [type] [nameatt_node];}
complexType name=”[nametype]”> <attribute name=”[nameatt]”type=”[type]” use=”required”>	Interface [nametype] (Source semistructured...) {attribute [type] [nametype_nameatt];}
<complexType name=”[nametype]”> <element name=”[nameel]”type=”[type]” > </complexType> <simpletype name=”[type]”> <xsd:restriction base=”xsd:int”> <xsd:minInclusive value=”1”/> <xsd:maxInclusive value=”31”/> </xsd:restriction> </xsd:simpleType>	Interface [nametype] (Source semistructured...) {attribute range 1,31 [nameel];}
<xsd:complexType name=”[nametype]”> <element name=”[nameel]”type=”[type]” > </complexType> <xsd:simpleType name=”[type]” > <xsd:restriction base=”xsd:string”> <xsd:enumeration value=”Sunday”/> <xsd:enumeration value=”Monday”/> ... </xsd:restriction> </xsd:simpletype>	enum [type] { ‘Sunday’; ‘monday’; ‘Tuesday’; …….} Interface [nametype] (Source semistructured...) {attribute [type] [nameel];}

<xsd:simpleType name=”[nametype]” base”xsd:string”> <xsd:pattern value=”\ d{3} -\ d{4} -\ d{3} “/> <xsd:simpleType>	Si traduce come un attribute del tipo indicato da base
<xsd:complexType name=”[nametype]”> <element name=”[nameel]”type=”[list]” > </complexType> <xsd:simpleType name=”[type]” > <xsd:restriction base=”xsd:string”> <xsd:enumeration value=”[value_1]”/> <xsd:enumeration value=”[value_2]”/> ... </xsd:restriction> </xsd:simpletype> <xsd:simpleType name=”[List]”> <xsd:restriction itemType=”[type]”/> </ xsd:simpleType>	Interface [nametype] (source semistructured ....) { attribute set<[type]> [nameel]; } enum [type] { ‘[value_1]’; ‘[value_2]’; ……. }

<xsd:complexType name=”[nametype]”>
<element name=”[nameel]”type=”[name_union]” >
</complexType>
<xsd:simpleType name=”[name_union]”>
<xsd:union memberTypes=”[type_1],[type_2]”/>
</xsd:simpleType>
 	
<xsd:simpleType name=”[type_1]” >
....
</xsd:simpletype>
 	
<xsd:simpleType name=”[type_2]” >
....
</xsd:simpletype>
	interface address
	(…)
	{attribute [name_union] [nameel],
	}
	
	Interface [name_union]
	(…)
	{ attribute [type_1] [type_1];
	};
	union
	{ attribute [type_2] [type_2];
	}

<element name=”[nameel]”> <complexType> <simpleContent> <extension base=”decimal”> <attribute name=”[nameatt]” type=”[type]”/> </exstension> </simpleContent> </complexType > </element>	Interface [nameel] (…..) {attribute [type] [nameel_nameatt]; attribute float [nameel_node] ;}

<element name=”reminder”> <complexType mixed=”true”> <element name=”[name_1]”type=”[type_1]”/> <element name=”[name_2]”type=”[type_2]”/>/> <complexType> </element>	Interface reminder (…) {attribute string node_1; attribute [type_1] [name_1]; attribute string node_2, attribute [type_2] [name_2];} I nodi di tipo stringa introdotti servono a contenere il testo che può essere presente tra un elemento e l’altro. Una funzione dovrà incrementare di uno il numero associato ad ogni nodo introdotto.
<element name=”[nameel]” type=”anytype”/>	attribute any [nameel]
<complextype name=”[nametype]”> <choice> <element name=”[name_1]” type=”[type_1]”/> <element name=”[name_2]” type=”[type_2]”/> </xsd:choice> </xsd:complextype>	interface [nametype] (source semistructured…) {attribute [nametype_union_1] [nametype_book_union_1]; } interface [nametype_book_union_1] {attribute [type_1] [name_1]; } union {attribute [type_2] [name_2]; }
<group name=”[namegroup]”> <sequence> <element ...../> <element ...../> </sequence>	Interface [namegroup] (Source semistructured...) {attribute...; attribute...; }
<attributeGroup name=”[namegroup]”> <attribute name=”[name_1]” type=”[type_1]”/> <attribute name=”[name_2]” type=”[type_2]”/> ……. </attributeGroup>	Interface [namegroup] (Source semistructured...) {attribute [type_1] [name_1];; attribute [type_2] [name_2]; }
<complexType> <all> <element name=”[name_1]”type=”[type_1]”/> <element	Interface [namegroup] (Source semistructured...) {attribute [type_1] [name_1]?; attribute [type_2] [name_2]?;}

name=”[name_2]”type=”[type_2]”/> </all> <complexType>	
<xsd:annotation> <xsd:documentation> [note] </xsd:documntation> </xsd:annotation>	Interface [el_radice] (Source semistructured...) {const string anotation_1=[note]; }
<element name=”[nameel]“ type=“[type]“ nillable=“true“/>	Attribute [type] [nameel]?
<schema xmlns=http://www.w3.org/2001/XMLschema xmlns:t=“http://www.starpowder.com/namespace“ targetNamespace=“http://www.starpowder.com/namespace”> …….. <include schemalocation=”http://www.starpowder.com/schemas/books.xsd/> <element ...../> ..... <element name=”books” type=”t:books”/> </schema> **books.xsd:** <schema xmlns=http://www.w3.org/2001/XMLschema xmlns:t=“http://www.starpowder.com/namespace“ targetNamespace=“http://www.starpowder.com/namespace”> <complexType name=”books”>	Per effettuare una corretta traduzione in ODLi$_3$ sarà necessaria una funzione che vada ad aprire il file indirizzato dal valore dell’attributo schemalocation, il cui contenuto verrà tradotto come fosse interno allo schema principale.

<complexType name=”[nametype]> <complexContent> <extension base=”[type]”> <element name=”[nameel]” type=”[typeel]”/> </exstension> </complexcontent> </complexType> <complextype name=[type]> ..... </complexType>	interface [nametype] : [type] (source semistructured…) {attribute [typeel] [nameel]; } interface [type] (source semistructured…) {.....; }
<unique name="dummy1"> <selector xpath="r:regions/r:zip"/> <field xpath="@code"/> </unique>	Interface zip (source semistructured Report.xml candidate_key code) {attribute part part; attribute int code;}
<element name="purchaseReport"> <complexType> <sequence> <element name="regions" type="r:RegionsType"> <keyref name="dummy2" refer="r:pNumKey"> <selector xpath="r:zip/r:part"/> <field xpath="@number"/> </keyref> </element> ……. </element> …….. <key name="pNumKey"> <selector xpath="r:parts/r:part"/> <field xpath="@number"/> </key> </element>……. <complexType name="PartsType"> <sequence> <element name="part" maxOccurs="unbounded"> <complexType> <simpleContent> <extension base="string"> <attribute name="number" type="int"/> </extension> </simpleContent> </complexType> </element> </sequence> </complextype>	Interface parts_part (source semistructured Report.xml Key (number)) {…….. attribute int number; } Interface zip_parts (source semistructured Report.xml foreign_key (dummy2) references parts_part) { attribute int number; attribute iny quantity ; } Tramite le espressioni xpath interne alla definizione della chiave, è possibile individuare l’attributo (o gli attributi) da cui è costituita e l’interfaccia di cui deve far parte. Mentre, tramite le espressioni xpath interne alla definizione di Keyref, si individua a quale interfaccia deve appartenere la foreign_key e a quale primary_key deve fare riferimento. Nello schema il nome della chiave è pNumKey, ma l’attributo che la costituisce è number, perciò, per non creare ambiguità, nell’interfaccia viene tradotta con questo nome, che sarà lo stesso dell’attribute corrispondente.

<element name="htmlExample"> <complexType> <sequence> <any namespace="http://www.w3.org/1999/xhtml " minOccurs="1" maxOccurs="unbounded" processContents="skip"/> </sequence> <anyAttribute namespace="http://www.w3.org/1999/xhtml "/> </complexType> </element>	Interface purchaseOrder (……..) {attribute regionstype region, attribute partstype parts; attribute any htmlexample ; } Dal momento che il linguaggio ODLi$_3$ non prevede l'uso di namespace, non è possibile fornire una traduzione di tali concetti. L'elemento htmlExample verrà tradotto come un attribute di tipo any, in modo da non dover apecificare alcun modello di contenuto

CAPITOLO 7
Applicazione delle Regole di Traduzione a Standard per E-commerce: xCBL e cXML

7.1 Introduzione

Le innumerevoli transazioni commerciali, che costituiscono il cuore dell'economia mondiale, si stanno trasferendo sulla Rete, con significative riduzioni dei costi e incrementi consistenti dei margini di guadagno. Svariate stime indicano il B2B come il settore del commercio elettronico che sembra destinato al maggior sviluppo in un futuro molto prossimo.

B2B è un acronimo che stà per **business to business** (coommercio fra imprese) e che indica tutte quelle iniziative tese a integrare l'attività commerciale di un'azienda con quella dei propri clienti o dei propri fornitori «dove però il cliente non sia anche il consumatore finale del bene o del servizio venduti ma un partner attraverso il quale si raggiungono i consumatori finali». (*Dizionario della New Economy*, a cura di Rinaldo Pianola, Baldini e Castoldi 2000).

Il successo dei portali di B2B e-commerce è legato all'utilizzo di linguaggi standard che garantiscono l'interoperabilità fra i partecipanti, permettendo uno scambio di documenti facilmente interpretabili da clienti e fornitori.

Analizziamo ora due standard utilizzati in tale ambito: cXML e xCBL, entrambi applicazioni di XML.

7.2 cXML: commerce eXtensible Markup Language

cXML [25] è un linguaggio generato appositamente per il B2B e-commerce, per effettuare transazioni commerciali.
Si tratta di un'implementazione di XML che consente la comunicazione fra le imprese e gli intermediari mediante un unico linguaggio standard.
Le transazioni in cXML vengono effettuate utilizzando documenti, semplici file di testo contenenti valori racchiusi fra tag predefiniti, ad ognuno dei quali è associata una DTD che ne definisce il modello di contenuto, specificando l'ordine gerarchico degli elementi e i tipi utillizzati.[26]
I tipi più comuni di documenti cXML utilizzati sono:

- Catalogs: descrivono prodotti e servizi offerti dai fornitori riportando i prezzi relativi; costituiscono il principale canale di comunicazione fra clienete e fornitore.
- Punchout: si tratta di un protocollo per la gestione di sessioni interattive attraverso Internet.
- Purchase Orders: costituiscono gli ordini di acquisto.

In questa sede ci si è soffermati in particolare ad analizzare il file cXML.dtd, nel quale sono definiti gli elementi base con cui vengono generati documenti cXML.
Questo file è stato internamente tradotto in $ODLi_3$ utilizzando le regole esposte nel quinto capitolo, dimostarndone l'effettiva validità e completezza. L'intera traduzione e la sorgente vengono riportati in appendice, di seguito vengono analizzati alcuni aspetti rilevanti del lavoro svolto.

7.2.3 Traduzione di cXML.dtd

Il lavoro di traduzione è iniziato individuando l'elemento radice, cioè l'elemento padre di tutti gli altri, in questo caso *Response,* per poi tradurre i suoi componenti scendendo sempre più in basso nella gerarchia, arrivando a ricondurre tutti i tipi utilizzati a tipi semplici predefiniti.
Si noti che la struttura gerarchica innestata della DTD viene perfettamente mantenuta nella traduzione in ODLi$_3$.

Di seguito vengono analizzati alcuni punti significativi della traduzione

- Elemento **Request**:

```
<!ENTITY % cxml.requests "(ProfileRequest |
OrderRequest |
MasterAgreementRequest|
PunchOutSetupRequest |
ProviderSetupRequest |
StatusUpdateRequest |
GetPendingRequest |
SubscriptionListRequest |
SubscriptionContentRequest |
SupplierListRequest |
SupplierDataRequest |
CopyRequest|
CatalogUploadRequest)"
>
……
<!ELEMENT Request (%cxml.requests;)>
<!ATTLIST Request
   deploymentMode  (production | test)  "production"
>
……
<!ELEMENT OrderRequest (OrderRequestHeader, ItemOut+)>…….
```

Interface Request

(Source semistructured OrderRequest.xml)

```
{ attribute any ProfileRequest;
}union
{ attribute OrderRequest OrderRequest;
}union
{ attribute MasterAgreementRequest MasterAgreementRequest;
}union
{ attribute PunchOutSetupRequest PunchOutSetupRequest;
}union
{ attribute ProviderSetupRequest ProviderSetupRequest;
}union
{ attribute StatusUpdateRequest StatusUpdateRequest;
}union
{ attribute GetPendingRequest GetPendingRequest;
}union
{attribute enum {' SubscriptionListRequest ' ;
                 'NO_ SubscriptionListRequest ;
                }Request_SubscriptionListRequest SupplierListRequest;
} union
{attribute enum {' SupplierListRequest ' ;
                 'NO_ SupplierListRequest ;
                }Request_SupplierListRequest;

}union
{ attribute SupplierDataRequest SupplierDataRequest;
}union
{ attribute CopyRequest CopyRequest;
}union
{ attribute CatalogUploadRequest CatalogUploadRequest;
}
```

Interface OrderRequest

(Source semistructured OrderRequest.xml)

{attribute OrderRequestHeader OrderRequestHeader;

attribute ItemOut ItemOut;

attribute set<ItemOut> ItemOut_1;}

…..

- Per poter tradurre elementi come Request è necessario disporre di una funzione che, incontrando il simbolo % dell'entità parametrica, vada a ricercare la definizione corrispondente confrontando il nome ad essa assegnato. Una volta individuata, l'entità Request viene tradotta come fosse il contenuto dell'elemento Request.
- Il sottoelemento ItemOut di orderRequest ha cardinalità (1,n) nella DTD. Poiché in Odli3 non esite un modo diretto per esprimere questo tipo di cardinalità, l'elemento è stato tradotto creando due attribute, uno con cardinalità (1,1) per indicarne l'obbligatorietà, l'altro con cardinalità (0,n).
- La choice che costituisce l'entità Request viene tradotta tramite il costrutto UNION.

- Elemento **TelephoneNumber**:

<!ELEMENT TelephoneNumber (CountryCode, AreaOrCityCode, Number, Extension

<!ELEMENT Number (#PCDATA)>

……..

Interface TelephoneNumber

(Source semistructured OrderRequest.xml)

{attribute CountryCode CountryCode

attribute string AreaCityCode;

attribute int Number;

attribute string Exstension?;

}

- L'elemento Number è stato definito come PCDATA nella DTD, ma in $ODLi_3$ è possibile raggiungere un maggior livello di dettaglio nella definizione dei tipi. Per questo si prevede la possibilità che l'utenete intervenga a specificare il tipo di dato più appropriato per l'elemento o attributo in questione. In questo caso è stato scelto il tipo int.

- Elemento **Money**:

```
<!ELEMENT Money (#PCDATA)>
<!ATTLIST Money
   currency          %isoCurrencyCode;  #REQUIRED
   alternateAmount   %number;           #IMPLIED
   alternateCurrency %isoCurrencyCode;  #IMPLIED
>
……
```

```
Interface Money
(Source semistructured OrderRequest.xml)
{attribute string Money_Currency;
 attribute int Money_Number?;
 attribute string Money_AlternateCurrency?;
 attribute string PCDATA_Node
  }
```

- Money era un elemento di tipo semplice, corrispondente ad una stringa, ma è stato tradotto come interfaccia poiché contiene degli attributi, e quindi, in $ODLi_3$, equivale ad un tipo complesso. Per memorizzare l'informazione contenuta nel campo PCDATA, relativo all'elemento Money, è stato introdotto il nodo fittizio PCDATA_node.

7.3 xCBL:Common Business Library

La Common Business Library, o xCBL [], è un set di blocchi XML utilizzati per la creazione di documenti per l'e-commerce, robusti e riusabili.
xCBL è il risultato di una collaborazione fra Commerce One® , la e-commerce enterprise, e venditori di prodotti hardware e software, basata sull'analisi degli esistenti standard per l'e-commerce, come EDI (Electronic Data Interchange) e Rosetta Net.
Lo scopo è quello di fornire uno schema della struttura dei documenti per facilitare gli scambi commerciali via web.
In particolare con l'ultima versione, xCBL è in grado di supportare tutti i documenti essenziali e le transazioni nell'ambito dell'e-commerce. Questo significa che se si vogliono sviluppare soluzioni in xCBL è possibile scegliere di utilizzare diversi linguaggi, e, indipendentemente dal validatore di sintassi utilizzato, il documento rimane lo stesso. Quindi xCBL permette di superare l'ostacolo maggiore per il raggiungimento dell'interoperabilità, facilitando lo scambio di documenti commerciali.
Fra i formati consentiti si trova anche XSDL, Xml Schema Definition Language; in particolare si analizzerà in seguito il file OrderstatusRequest.xsd, disponibile nel sito di xCBL, e appartenenete al set di documenti che costituiscono l'ultima versione di xCbl, la 3.5. Questo file è stato internamente tradotto in $ODLi_3$ utilizzando le regole esposte nel quarto capitolo, dimostrandone l'effettiva validità e completezza. L'intera traduzione e la sorgente vengono riportati in appendice, di seguito vengono analizzati alcuni aspetti rilevanti del lavoro svolto.

7.3.1 Traduzione di OrderStatusrequest.xsd

La traduzione ha inizio dall'elemento radice OrderStatusRequest, che è il primo dello schema, e procede traducendo uno ad uno i suoi sottoelementi. Anche in questo caso la struttura gerarchica innestata della DTD viene perfettamente mantenuta nella traduzione in $Odli_3$.

Di seguito vengono analizzate alcune fasi significative della traduzione.

- Elemento **OrderStatusRequestheader**, figlio diretto dell'elemento radice:

```
<xsd:complexType name="OrderStatusRequest">
  <xsd:sequence>
    <xsd:element ref="OrderStatusRequestHeader"/>
    <xsd:element minOccurs="0" ref="OrderStatusRequestDetail"/>
    <xsd:element minOccurs="0" ref="OrderStatusRequestSummary"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="OrderStatusRequest" type="OrderStatusRequest"/>
<xsd:complexType name="OrderStatusRequestHeader">
  <xsd:sequence>
  ……
      <xsd:element name="OrderStatusIssueDate" type="xcblDatetime"/>
  ……
      <xsd:element minOccurs="0" name="OrderStatusListOfAttachment">
       <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="ListOfAttachment"/>
  ……
</complexType>
```

Interface OrderStatusRequest

```
(source semistructured OrderStatusRequest.xml)
{attribute OrderStatusRequestHeader OrderStatusRequestHeader;
 attribute OrderStatusRequestDetail OrderStatusRequestDetail*;
 attribute OrderStatusRequestSummary  OrderStatusRequestsummary*;
 attribute annotation_1 annotation_1;
}
```

Interface OrderStatusRequestHeader

(source semistructured OrderStatusRequest.xml)

{attribute OrderStatusRequestID OrderStatusRequestID;
attribute string OrderStatusIssueDate;
attributeOrderStatusParty OrderStatusParty;
attribute OrderStatusLanguage OrderStatusLanguage*;
attribute string OrderStatusrequestNode*;
attribute list<Attachemnt> OrderStatusListOfAttachemnt*;
}

OrderStatusIssuedate è stato tradotto come stringa in quanto il simpletype corrispondente faceva riferimento ad una path expression, non esprimibile in $ODLi_3$, quindi è stato indicato il suo tipo base.

Si osservi che nello schema OrderstatusListOfAttachement è un complexType contenente solo un riferimento a PartyCoded con cardinalità (1,n). Ciò significa che il primo elemento non è altro che un insieme di n ripetizioni del secondo, quindi è possibile tradurlo come una list.

Definizione del tipo semplice **Boolean**

```
<xsd:simpleType name="xcblBoolean">
   <xsd:restriction base="xsd:boolean">
         <xsd:pattern value="true|false"/>
   </xsd:restriction>
</xsd:simpleType>
```

Interface Boolean

(source semistructured OrderStatusRequest.xml)

{const string ‘true’;
}union
{const string ‘false’;}

Si è detto che non è possibile tradurre le pattern expression in ODLi$_3$, in questo caso però viene indicata una scelta tra due valori, quindi è traducibile come se fosse una choice

7.4 Osservazioni

La traduzione è stata effettuata in modo da attenersi il più possibile alla struttura dello schema XML, e quindi del documento relativo. Questo, in alcuni casi, ha comportato l'utilizzo di metodi di conversione laboriosi, che si sarebbero potuti evitare, a discapito, però, della coerenza con la scelta fatta di produrre un documento il più possibile simile all'originale.
Si osservi, per esempio, come è stato tradotto l'elemento PartyID:

Interface Party

```
(source semistructured OrderStatusRequest.xml)
{attribute PartyID PartyID;
  attribute set<Identifier> ListOfIdentifier*;
  attribute boolean MDFBusinnes*
  attribute NameAddress Nameaddress*;
  attribute OrderContact OrderContact*;
  attribute RecivingContact RecivingContact* ;
  attribute ShippingContact ShippingContact*;
  attribute list<Contact> OtherContacts*;
  attribute CorrespondenceLanguage CorrespondenceLanguage*;
}
```

Interface PartyID

```
(source semistructured OrderStatusRequest.xml)
{attribute Identifier identifier;
}
```

Interface Identifier

```
(source semistructured OrderStatusRequest.xml)
{attribute Agency Agency;
 attribute string Ident;}
```

Lo stesso concetto può essere espresso traducendo PartyID come un attribute di tipo identifier, evitando di generare un'interfaccia specifica:

Interface Party
(source semistructured OrderStatusRequest.xml)
{attribute identifier PartyID;
……}

Tuttavia, nell'istanza dello schema, PartyID è un elemento contenente un altro elemento, istanza di identifier. Traducendo in questo modo si perderebbe tale informazione.

CONCLUSIONI

Questa tesi si colloca nell'ambito del progetto MOMIS ed ha come obbiettivo quello di approfondire le conoscenze già disponibili in merito alla traduzione fra DTD e $ODLi_3$, e di ricavare delle regole che permettano l'esportazione di XML Schema in $ODLi_3$. Gli Schemi e DTD possono essere associati a file XML che si ottengono da pagine HTML mediante Wrapper, generati in modo automatico e semiautomatico. In particolare ci si è soffermati su XWRAP e XWRAP elite, il primo utilizzato per la generazione semiautomatica di Wrapper, il secondo per la generazione automatica. Sono emerse alcune differenze rilevanti fra i due sistemi.

Si è visto che XWRAP elite è efficiente solo lavorando su pagine ricche di data object, cioè pagine che presentano oggetti aventi struttura simile e in numero non troppo ridotto. Tali pagine sono solitamente il risultato di ricerche effettuate via HTML, appartenenti per esempio a Web sites che forniscono cataloghi di prodotti. XWRAP Elite non è stato progettato per lavorare con pagine Web personali, per questi tipi più complessi può essere utilizzato XWRAP Original. Tuttavia la versione Elite presenta dei notevoli vantaggi :richiede un intervento da parte dell'utente estremamente ridotto rispetto al modello Original, produce Wrapper molto più robusti in relazione ai cambiamenti di presentazione delle corrispondenti Web pages, infatti le modifiche al layout, o gli annunci introdotti, non hanno alcun effetto sulla sua performance; al contrario, la robustezza dei Wrapper generati da XWRAP original dipende fortemente dall'esperienza dell'utilizzatore del toolkit e dalla qualità di interazione tra lui e lo stesso toolkit.

Confrontando fra loro i tre linguaggi analizzati, XML Schema, DTD, e $ODLi_3$ si è giunti a delle importanti conclusioni.

XML e $ODLi_3$ sono nati con obbiettivi diversi, infatti mentre ODLi3 è stato generato come linguaggio in grado di descrivere una base di dati archiviata in sorgenti dal formato differente, XML nasce con l'intento di descrivere dati in modo che siano facilmente veicolati, in particolare tramite la rete Internet. Per questo motivo esistono alcune differenze fra i due, tuttavia lo scopo che risiede dietro alla definizione di $ODLi_3$, non può non far pensare che esistano punti in comune tra i due linguaggi. Infatti, sempre in $ODLi_3$, è prevista la descrizione di dati semistrutturati, ed un file XML è appunto un formato che risiede in tale categoria.

Quindi, malgrado la diversa filosofia abbia portato alla creazione di sintassi differenti per perseguire ognuna il proprio scopo, i punti in comune fra le due strutture che le loro sintassi descrivono sono numerosi.

Si può affermare che, mentre la DTD permette una migliore rappresentazione della struttura del dato, $ODLi_3$ consente una tipizzazione più precisa ed una referenziazione fra gli oggetti più accurata.Tali limitazioni dell' XML sono state superate con le innovazioni introdotte da XML Schema, e si può concludere che la semantica rappresentabile in $ODLi_3$ è un sottoinsieme di quella rappresentabile con XML Schema. Le differenze riguardano soprattutto la definizione precisa della struttura del dato che non trova analogo potere di espressione in $ODLi_3$.

Le regole di traduzione ricavate sono finalizzate alla generazione di Wrapper che permettano l'esportazione semiautomatica della descrizione della struttura di file XML in $ODLi_3$, in modo di procedere alla fase successiva di integrazione fra le risorse. Si fa riferimento ad una generazione semiautomatica in quanto, durante la traduzione, è previsto l'intervento del progettista, in modo tale da ottenere una descrizione che non si discosti di molto da quella originaria, ma che racchiuda tutte le informazioni necessarie all'integrazione. Nel caso in cui la sorgente sia una DTD tale interazione è necessaria per sopperire a delle carenze rappresentative del linguaggio XML rispetto alle potenzialità di $ODLi_3$. Sarà necessario prevedere un'apposita interfaccia tramite la quale l'utente inserisca informazioni aggiuntive, come il tipo di dato con cui può essere tradotto più nel dettaglio un PCDATA, la struttura che un elemento dichiarato come misto deve assumere, oppure confermi le relazioni esistenti fra gli oggetti. Nel caso in cui la sorgente sia un XML Schema si richiede un'interazione con il progettista molto ridotta, limitata ad una scelta fra più possibilità di traduzione per alcuni concetti, ma non volta all' inserimento di informazioni aggiuntive. Questo è spiegabile considerando che XML Schema fornisce un approccio object oriented, quindi si presenta molto più vicino al linguaggio $ODLi_3$ di quanto non lo sia DTD.

L'applicazione delle regole agli standard xCBL e cXML ha permesso di verificarne l'effettiva validità e completezza, e costituisce un esempio di utilizzo del lavoro svolto ad un impiego molto attuale della rete, quello del commercio elettronico. Fornendo un integrazione di questo tipo di risorse si potrebbe facilitare di molto l'interoperabilità fra i partecipanti agli scambi commerciali sul Web.

APPENDICE A

Il linguaggio descrittivo ODLi$_3$

Si riporta la descrizione in BNF del linguaggio descrittivo ODLi$_3$. Essendo questo una estensione del linguaggio standard ODL, si riportano in questo appendice solo le parti che differiscono dall'ODL originale, rimandando invece a quest'ultimo per le parti in comune.

```
<interface_dcl>::= <interface header>{[interface_body]};
<interface_header>::=interface <identifier>
                    [<inheritance_spec>]
                    [<type_property_list>]
<inheritance_spec>::=:<scoped_name> [,<inheritance_spec>]
<type_property_list>::=([<source_spec>][<extent_spec>]
                          [<key_spec>] [<f_key_spec>])
<source_spec>::=source <source_type><source_name>
<source type>::=relational | nfrelational | object | file
<source name>::=<identifier>
<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>
<attr_dcl>::=[readonly]attribute
               <domain_type><attribute_name>
               [<fixed_array_size>][<mapping_rule_dcl>]
<mapping_rule_dcl>::=mapping_rule<rule_list>
<rule_list>::=<rule> | <rule> ,<rule_list>
<rule>::=<local_attr_name> | '<identifier>'
          <and_expression> | <or_expression>
<and_expression>::=(<local_attr-name> and <and_list>)
<and_list>::=<local_attr_name> | <local_attr_name> and <and_list>
<or_expression>::=(<local_attr_name> or <or_list>)
<or_list>::=<local_attr_name> | <local_attr_name> or <or_list>
<local_attr_name>::=<source_name>.<class_name>.<attribute_name>
<relationships_list>::=<relationship_dcl>;|<relationship_dcl>;<relationship
s_list>
<relationships_dcl>::=<local_attr_name><relationship_type><local_attr_name>
<relationship_type>::=syn | bt |nt |rt
```

```
<rule_list>::=<rule_dcl>; | <rule_dcl>;<rule_list>
<rule_dcl>::=rule<identifier> <rule_pre> then <rule post>
<rule_pre>::=<forall><identifier> in <identifier> : <rule_body_list>
<rule_post>::=<rule_body_list>
<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>:<rul_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 B

Esempio di Traduzione da XML Schema a ODLi$_3$: OrderStatusRequest.xsd

Il file sorgente è reperibile sul sito www.xCBL.org.

La traduzione ha inizio dall'elemento radice OrderStatusRequest, e procede traducendo uno ad uno i suoi sottoelementi:

Interface OrderStatusRequest

(source semistructured OrderStatusRequest.xml)

{attribute OrderStatusRequestHeader OrderStatusRequestHeader;
 attribute OrderStatusRequestDetail OrderStatusRequestDetail*;
 attribute OrderStatusRequestSummary OrderStatusRequestsummary*;
 attribute annotation_1 annotation_1;
}

Interface OrderStatusRequestHeader

(source semistructured OrderStatusRequest.xml)

{attribute OrderStatusRequestID OrderStatusRequestID;
 attribute string OrderStatusIssueDate;
 attributeOrderStatusParty OrderStatusParty;
 attribute OrderStatusLanguage OrderStatusLanguage*;
 attribute string OrderStatusrequestNode*;
 attribute list<Attachemnt> OrderStatusListOfAttachemnt*;
 }

/* OrderStatusIssuedate è stato tradotto come stringa in quanto il simpletype corrispondente faceva riferimento ad una path expression, non esprimibile in ODLi$_3$,

quindi è stato indicato il suo tipo base. OrderstatusListOfAttachement si può vedere come una lista, infatti,anche se nello schema non è stato usato l'elemento listType, il concetto espresso è lo stesso.*/

Interface OrderstatusRequestID

(source semistructured OrderStatusRequest.xml)

{attribute Reference Reference;

}

Interface Reference

(source semistructured OrderStatusRequest.xml)

{attribute string RefNum;

attribute string RefDate*;}

Interface OrderStatusParty

(source semistructured OrderStatusRequest.xml)

{attribute BuyerParty BuyerParty;

attribute BuyerTaxInformation BuyerTaxINformation*;

attribute SellerParty SellerParty ;

attribute SellerTaxInformation SellerTaxInformation*;

attribute ShipToParty ShipToParty* ;

attribute BillToParty BillToParty*;

attribute RemitToParty RemitToParty*;

attribute ShipFromparty ShipFromParty*;

attribute WareHouseParty WareHouseParty*;

attribute SoldToParty SoldToParty*;

attribute ManufacturingToParty ManufacturingToParty*;

attribute MaterialIssuer MaterialIssuer*;

attribute List<PartyCoded> ListOfPartyCoded*;

}

/*Si osservi che nello schema ListOfPartyCoded è un complexType contenente solo un riferimento a PartyCoded con cardinalità (1,n). Ciò significa che il primo elemento non è altro che un insieme di n ripetizioni del secondo, quindi è possibile tradurlo come una list,*/

Interface BuyerParty

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;

}

Interface BuyerTaxInformation

(source semistructured OrderStatusRequest.xml)

{attribute PartytaxInformation PartyTaxInformation;

}

Interface SellerParty

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;

}

Interface SellerTaxInformation

(source semistructured OrderStatusRequest.xml)

{attribute PartytaxInformation PartyTaxInformation;

}

Interface ShipToParty

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;}

Interface BillToParty

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;

}

Interface RemitToParty

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;

}

Interface ShipFromparty

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;

}

Interface WareHouseParty

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;

}

Interface SoldToParty

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;

}

Interface ManufacturingToParty

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;

}

Interface MaterialIsuer

(source semistructured OrderStatusRequest.xml)

{attribute Party Party;

}

Interface PartyCoded: Party

(source semistructured OrderStatusRequest.xml)

{attribute PartyRoleCoded PartyRoleCoded

attribute string PartyRoleCodedOther*;
}

Interface PartyRoleCoded

(source semistructured OrderStatusRequest.xml)
{attribute enum {'Other';
'AcceptanceLocation';
'AcceptingParty';
'Accountant';
…….
}PartyRoleCode;
}

Interface Party

(source semistructured OrderStatusRequest.xml)
{attribute PartyID PartyID;
attribute set<Identifier> ListOfIdentifier*;
attribute boolean MDFBusinnes*
attribute NameAddress Nameaddress*;
attribute OrderContact OrderContact*;
attribute RecivingContact RecivingContact* ;
attribute ShippingContact ShippingContact*;
attribute list<Contact> OtherContacts*;
attribute CorrespondenceLanguage CorrespondenceLanguage*;
}

Interface PartyID

(source semistructured OrderStatusRequest.xml)
{attribute Identifier identifier;
}

Interface Identifier

(source semistructured OrderStatusRequest.xml)
{attribute Agency Agency;

attribute string Ident;}

Interface Agency

(source semistructured OrderStatusRequest.xml)

{attribute enum{'other',

'AAMVA';

……..;

}AgencyCode;

attribute string AgencyCodeOther*;

attribute string Agencydescription*;

attribute enum{'other';

'AcceptanceSiteCode";

……;

}CodeListIdentifierCoded*;

attribute string CodeListIdentifierCodedOther*;

}

Interface Boolean

(source semistructured OrderStatusRequest.xml)

{const string 'true';

}union

{const string 'false';

}

/*Si è detto che non è possibile tradurre le pattern expression in ODLi$_3$, in questo caso però viene indicata una scelta tra due valori, quindi è traducibile come se fosse una choice*/

Interface NameAddress

(source semistructured OrderStatusRequest.xml)

{attribute string ExternalAddressID*;

attribute string Name1;

attribute string Name2*;

attribute string Name3*;

attribute Identifier Identifier*;
attribute PoBox PoBox* ;
attribute string street*;
attribute string HouseNumber*;
attribute string streetSupplement1*;
attribute string streetSupplement2*;
attribute string Building*;
attribute string Floor*;
attribute string RoomNumber*;
attribute string InHouseMail*;
……;
attribute Region Region*;
attribute Country Country*;
attribute TimeZone TimeZone*;
attribute NameAddress_AddressTypeCode NameAddress_AddressTypeCode*;
attribute string NameAddress_ AddressTypeCodeOther*;
}

Interface Pobox

(source semistructured OrderStatusRequest.xml)
{attribute string Pobox_PoboxPostalCode*;
attribute string Pobox_node;
}

/*un'istanza possibile di questo elemento è la seguente: <Pobox_PoboxPostalCode=234er>tr</Pobox>*/

Interface Region

(source semistructured OrderStatusRequest.xml)
{attribute enum {'other';
'AUACT';
'AUNSW';
……;
}RegionCode;

```
 attribute string RegionCodeOther*;
}
```

Interface Country

```
(source semistructured OrderStatusRequest.xml)
{attribute enum {'other';
                 'AE';
                 'AF';
                  ……;
                }CountryCode;
 attribute string CountryCodeOther*;
}
```

Interface TimeZone

```
(source semistructured OrderStatusRequest.xml)
{attribute enum {'other';
                 '12.00';
                 '11.45';
                  ……;
                }TimeZoneCode;
 attribute string TimeZoneCodeOther*;
}
```

Interface NameAddress_AddressTypeCode

```
(source semistructured OrderStatusRequest.xml)
{attribute enum {'other';
                'AcceptanceLocation';
                'AccountsPayableOffice';
                'AcknowledgementRecipient';
                 ……;
                }AddressTypeCode;
 }
```

Interface OrderContact

(source semistructured OrderStatusRequest.xml)

{attribute Contact Contact;

}

Interface RecivingContact

(source semistructured OrderStatusRequest.xml)

{attribute Contact Contact;

}

Interface ShippingContact

(source semistructured OrderStatusRequest.xml)

{attribute Contact Contact;

}

Interface Contact

(source semistructured OrderStatusRequest.xml)

{attribute ContactID ContactID*;

attribute string ContactName;

attribute ContactFunction ContactFunction*;

attribute string ContactDescription*;

attribute list<ContactNumber> ListOfContactNumber;

}

Interface ContactID

(source semistructured OrderStatusRequest.xml)

{attribute Identifier Identifier;

}

Interface ContactFunction

(source semistructured OrderStatusRequest.xml)
{attribute enum {'other';
'AcceptingContact';
'AcceptingOfficial';
'Accountingcontact';
'AccountsPayableContact';
'AccountsReceivableContact';
……
}ContactFunctionCode;
attribute string contactFunctioncodeOther*;
}

Interface ContactNumber

(source semistructured OrderStatusRequest.xml)
{attribute string ContactNumberValue;
attribute enum {'other';
'telephoneNumber';
'faxNumber';
……;
} ContactNumberTypeCode;
attribute string ContactNumberTypeCodeOther*;
}

Interface Correspondencelanguage

(source semistructured OrderStatusRequest.xml)
{attribute Language Language;
}

Interface Language

(source semistructured OrderStatusRequest.xml)
{attribute enum {'other';
'aa';
'ab';

‘af’;
……;
} LanguageCode;
attribute string LanguageCodeOther*;
attribute enum {’other’;
‘ae’;
‘af’;
‘ag’;
……;
} LocalCode;
attribute string LocalCodeOther*;

}

Interface PartyTaxInformation

(source semistructured OrderStatusRequest.xml)
{attributeTaxIdentifier TaxIdentifier*;
attribute string Registeredname*;
attribute string registeredOffice*;
atttribute TaxLocation TaxLocation* ;
}

Interface TaxIdentifier

(source semistructured OrderStatusRequest.xml)
{attribute Identifier Identifier;
}

Interface TaxLocation

(source semistructured OrderStatusRequest.xml)
{attribute Location Location;
}

Interface Location

(source semistructured OrderStatusRequest.xml)

{attribute enum {'other';

'JurisdictionToReceiveCreditForUniformCommercialCodeFiling';

'TransmittingUtility';

'Consignor';

'CensusScheduleD';

……;

}LocationQualifierCode;

attribute string LocationQualifierCodeOther*;

attribute Location_union_3 Location_union_3;

attribute GPSCoordinates GPSCoordinates*;

}

Interface Location_union_3

(source semistructured OrderStatusRequest.xml)

{attribute LocationIdentifier LocationIdentifier;

}union

{attribute string ExternalAddressID;

}union

{attribute NameAddress NameAddress;

}

Interface LocationIdentifier

(source semistructured OrderStatusRequest.xml)

{attribute LocID LocID;

attribute string LocationDescription*;

}

Interface LocID

(source semistructured OrderStatusRequest.xml)

{attribute Identifier Identifier;

}

Interface GPSCoordinates

(source semistructured OrderStatusRequest.xml)

{attribute string GPSSYstem;

attribute string Latitude;

attribute string Longitude;

}

Interface OrderStatusLanguage

(source semistructured OrderStatusRequest.xml)

{attribute Language Language;

}

Interface Attachemnt

(source semistructured OrderStatusRequest.xml)

{attribute string AttachemntPurpose ;

attribute string FileName*;

attribute string AttachemntTitle*;

attribute string AttachemntDescription*;

attribute Language Language* ;

attribute string MIMEType*;

attribute Boolean ReplacementFile*;

attribute string AttachemntLocation;

}

InterfaceOrderStatusRequestDetail

(source semistructured OrderStatusRequest.xml)

{attribute set<OrderStatusDetailRequest> ListOfOrderStatusRequestDetail;

}

Interface OrderStatusDetailRequest

(source semistructured OrderStatusRequest.xml)

{attribute OrderStatusReference OrderStatusReference ;

attribute string GeneralLineItemNote*;

```
 attribute LineItemAttachemnt LineItemAttachemnt*;
 }

Interface OrderStatusReference
(source semistructured OrderStatusRequest.xml)
{attribute AccountCode AccountCode*;
 attribute BuyerReferenceNumber BuyerReferenceNumber;
 attribute SellerReferenceNumber SellerReferenceNumber;
 attribute OtherReference OtherReference*;
 attribute string OrderDate;
 attribute list<OrderStatusItem> ListOfOrderStatusItem*;
 }
```

Interface AccountCode

```
(source semistructured OrderStatusRequest.xml)
{attribute Reference Reference;
 }
```

Interface BuyerReferenceNumber

```
(source semistructured OrderStatusRequest.xml)
{ attribute Reference Reference;
 }
```

Interface SellerReferenceNumber

```
(source semistructured OrderStatusRequest.xml)
{ attribute Reference Reference;
 }
```

Interface OtherReference

```
(source semistructured OrderStatusRequest.xml)
{ attribute list<ReferenceCoded> ListOfReferenceCoded;
 }
```

Interface ReferenceCoded

(source semistructured OrderStatusRequest.xml)

{attribute enum {'other';

'AbbreviatedNewDrugApplicationNumber';

'AcceptableSourceDUNSNumber';

'AcceptableSourcePurchaserID';

'AcceptableSourceSupplierID';

……;

} ReferenceTypeCode;

attribute string ReferenceTypeCodeOther*;

attribute PrimaryReference PrimaryReference;

attribute SupportingReference SupportingReference*;

attribute SupportingSubReference SupportingSubReference*;

attribute string ReferenceDescription*;

}

Interface PrimaryReference

(source semistructured OrderStatusRequest.xml)

{ attribute Reference Reference;

}

Interface SupportingReference

(source semistructured OrderStatusRequest.xml)

{ attribute Reference Reference;

}

Interface SupportingSubReference

(source semistructured OrderStatusRequest.xml)

{ attribute Reference Reference;

}

Interface OrderStatusItem:BaseItemDetail

(source semistructured OrderStatusRequest.xml)

{ attribute OrderStatusItemTransport OrderStatusItemTransport*;

```
  attribute VarianceQty VarianceQty;
 }
```

Interface OrderStatusItemTransport

```
(source semistructured OrderStatusRequest.xml)
{ attribute Transport Transport;
 }
```

Interface Transport

```
(source semistructured OrderStatusRequest.xml)
{ attribute int TransportID;
   attribute TransportMode TransportMode* ;
  attribute TransportMeans TransportMeans*;
  attribute carrierID carrierID*;
  attribute string CustShippingContractNum*;
    ……
 }
```

Interface TransportMode

```
(source semistructured OrderStatusRequest.xml)
{attribute enum {'other';
                    'Air';
                    'AirCharter';
                      ……;
                    } transportModeCode;
 attribute string TransportModeCodeOther*;
}
```

Interface TransportMeans

```
(source semistructured OrderStatusRequest.xml)
{attribute enum {'other';
                    '20FtILContainer-ClosedTop';
                    '20FtILContainer-OpenTop';
                      ……;
```

} TransportMeansCode;
attribute string TransportMeansCodeOther*;
}

Interface carrierID

(source semistructured OrderStatusRequest.xml)
{attribute Identifier Identifier;
}

interface VarianceQTY

(source semistructured OrderStatusRequest.xml)
{attribute Quantity Quantity;
}

Interface Quantity

(source semistructured OrderStatusRequest.xml)
{attribute Quantity_union_1 Quantity_union_1;
attribute UnitOfMeasurement UnitOfMeasurement;
}

Interface Quantity_union_1

(source semistructured OrderStatusRequest.xml)
{attribute QuantityValue QuantityValue;
}union
{attributeQuantityRange QuantityRange;
}

Interface QuantityValue

(source semistructured OrderStatusRequest.xml)
{attribute string QuantityValue_SignificanceCode*;
attribute string QuantityValue_SignificanceCodeOther*;
attribute string QuantityValue_ConditionCode* ;
attribute string QuantityValue_ConditionCodeOther* ;

attribute string QuantityValue_node;
}

Interface QuantityRange

(source semistructured OrderStatusRequest.xml)

{attribute MinimumValue MinimumValue ;
attribute MaximumValue MaximumValue ;
}

Interface MinimumValue

(source semistructured OrderStatusRequest.xml)

{attribute QuantityValue_SignificanceCode QuantityValue_SignificanceCode*;
attribute string QuantityValue_SignificanceCodeOther*;
attribute QuantityValue_ConditionCode QuantityValue_ConditionCode* ;
attribute string QuantityValue_ConditionCodeOther* ;
attribute string QuantityValue_node;
}

Interface MaximumValue

(source semistructured OrderStatusRequest.xml)

{attribute QuantityValue_SignificanceCode QuantityValue_SignificanceCode*;
attribute string QuantityValue_SignificanceCodeOther*;
attribute QuantityValue_ConditionCode QuantityValue_ConditionCode* ;
attribute string QuantityValue_ConditionCodeOther* ;
attribute string QuantityValue_node;
}

enum {'Other';
'Approximately';
'EqualTo';
…….
} **QuantityValue_SignificanceCode**

enum {'Other';
'WhereAirEquals1';
'WhereButylAcetateEquals1';
…….
} **QuantityValue_ConditionsCode**

interface UnitOFMeasurement
(source semistructured OrderStatusRequest.xml)
{attribute enum {'other';
'1';
'4';
……;
} UOMCode;
attribute string UOMCodeOther*;
}
Interface BaseItemDetail
(source semistructured OrderStatusRequest.xml)
{attribute LineItemNum LineItemNum;
attribute LineItemType LineItemType*;
attribute ParentItemNumber ParentItemNumber* ;
attribute ItemIdentifier ItemIdentifier* ;
attribute list<Dimension> ListOfDimension;
attribute TotalQuantity TotalQuantity ;
attribute MaxBackOrderQuantity MaxBackOrderQuantity;
attribute Boolean OffCatalogFlag;
attribute ListOfItemReference ListOfItemReference;
attribute CountryOfOrigin CountryOfOrigin*;
attribute CountryOfDestination CountryOfDestination*;
attribute FinalRecipient FinalRecipient*;
attribute ConditionsOfSale ConditionsOfSale* ;
attribute HazardousMaterials HazardousMaterials*;
}

Interface LineItemNum

(source semistructured OrderStatusRequest.xml)

{attribute int BuyerLineItemNum;

atrribute int SellerLineItemNum*}

Interface LineItemType

(source semistructured OrderStatusRequest.xml)

{attribute enum {'Item';

'ComponentGroup';

'other'……;

} LineItemTypeCode;

attribute string LineItemTypeCodeOther*;

}

Interface ParentItemNumber

(source semistructured OrderStatusRequest.xml)

{attribute LineItemNumberReference LineItemNumberReference;

}

Interface LineItemNumberReference

(source semistructured OrderStatusRequest.xml)

{attribute enum {'Buyer';

'Seller';

} LineItemNumtypeCode*;

attribute string LineItemNumberReference_node;

}

/*Il valore di default è uno dei due valori specificati con il costrutto enumeration, perciò in questo caso non è importante tradurlo*/

Interface ItemIdentifier

(source semistructured OrderStatusRequest.xml)

{attribute PartNumbers PartNumbers*;
attribute string ItemDescription*;
attribute list<ItemCharacteristic> ListOfItemCharacteristic*;
attribute CommodityCode CommodityCode*;
}

Interface PartNumbers
(source semistructured OrderStatusRequest.xml)
{attribute SellerPartNumbers SellerPartNumbers*;
attribute BuyerPartNumber BuyerPartNumber*;
attribute ManufacturerPartNumber ManufacturerPartNumber*;
attribute StandardPartNumber StandardPartNumber*;
attribute SubstitutePartNumbers SubstitutePartNumbers*;
attribute OtherItemIdentifiers OtherItemIdentifiers*;
}

Interface SellerPartNumbers
(source semistructured OrderStatusRequest.xml)
{attribute PartNum PartNum;
}

Interface BuyerPartNumbers
(source semistructured OrderStatusRequest.xml)
{attribute PartNum PartNum;
}

interface PartNum
(source semistructured OrderStatusRequest.xml)
{attribute string PartID;
attribute string PartIDext*;
}

Interface ManufacturerPartNumber:PartNum
(source semistructured OrderStatusRequest.xml)

{attribute ManufacturerID ManufacturerID;
}

Interface ManufacturerID
(source semistructured OrderStatusRequest.xml)
{attribute Identifier Identifier;
}

Interface StandardPartNum
(source semistructured OrderStatusRequest.xml)
{attribute ProductIdentifierCoded ProductIdentifierCoded;
}

Interface ProductIdentifierCoded
(source semistructured OrderStatusRequest.xml)
{attribute enum {'Other';
'AdditionalProductIdentificationAssignedByTheManufacturer';
'AdvertisingPackageIdentificationCode';
……..
} ProductIdentifierQualifierCode;
attribute string ProductIdentifierQualifierCodeOther*;
attribute string ProductIdentifier ;
attribute string ProductIdentifierExt* ;
}

Interface SubstitutePartNumbers
(source semistructured OrderStatusRequest.xml)
{attribute set<ProductIdentifierCoded> ListOfProductIdentifierCoded;
}

Interface OtherItemIdentifiers
(source semistructured OrderStatusRequest.xml)
{attribute list<ProductIdentifierCoded> ListOfProductIdentifierCoded;
}

Interface ItemCharacteristic

(source semistructured OrderStatusRequest.xml)

{attribute enum {'Other';

'AdditionalSectorialCharacteristics'

'Age';

…….

}ItemCharacteristicCode*;

attribute string ItemCharacteristicCodedOther*;

attribute enum {'Other';

'All'

'BackOfCab';

…….

} SurfaceLayerPositionCoded*;

attribute string SurfaceLayerPositionCodedOther* ;

attribute string ItemCharacteristicValue;

attribute UnitOfMeasurement UnitOfMeasurement*;

}

interface CommodityCode

(source semistructured OrderStatusRequest.xml)

{attribute Identifier Identifier;

}

Interface Dimension

(source semistructured OrderStatusRequest.xml)

{attribute Measurement Measurement;

attribute enum {'Other';

'ConsolidatedWeight'

UnitNetWeight';

…….

} DimensionCode*;

attribute string DimensionCodeOther*;

}

Interface Measurement

(source semistructured OrderStatusRequest.xml)

{attribute Measurement_union_1 Measurement_union_1;

attribute UnitOfMeasurement UnitOfMeasurement ;

}

Interface Measurement_union_1

(source semistructured OrderStatusRequest.xml)

{attribute MeasurementValue MeasurementValue ;

}union

{attribute MeasurementRange MeasurementRange;

}

Interface MeasurementValue

(source semistructured OrderStatusRequest.xml)

{attribute QuantityValue_SignificanceCode

MeasurementValue_SignificanceCode*;

attribute string MeasurementValue_SignificanceCodeOther*;

attribute; QuantityValue_ConditionCode

MeasurementValue_ConditionsCode* ;

attribute string MeasurementValue_ConditionsCodeOther* ;

attribute float MeasurementValue_node ;

}

Interface MeasurementRange

(source semistructured OrderStatusRequest.xml)

{attribute MinimumValue MinimumValue ;

attribute MaximumValue MaximumValue ;

}

Interface TotalQuantity

(source semistructured OrderStatusRequest.xml)

{attribute Quantity Quantity;

}

Interface MaxBackOrderQuantity

(source semistructured OrderStatusRequest.xml)

{attribute Quantity Quantity;

}

Interface ListOfItemReference

(source semistructured OrderStatusRequest.xml)

{attribute list<referenceCoded> ListOfReferenceCoded;

}

interface LineItemAttachemnt

(source semistructured OrderStatusRequest.xml)

{attribute list<Attachemnt > ListOfAttachement;

}

interface OrderStatusRequestSummary

(source semistructured OrderStatusRequest.xml)

{attribute int TotalNumberOfLineItem*;

}

Interface annotation_1

(source semistructured OrderStatusRequest.xml)

{const string annotation_1_lang “en”;

const string annotation_1_value “XML Common Business Library 3.0 Copyright 2000 Commerce One, Inc.Permission is granted to use, copy, modify and distribute the DTD's, schemas and modules in the Commerce One XML Common Business Library Version 3.0 subject to the terms and conditions specified at http://www.xcbl.org/license.html”}

APPENDICE C

Esempio Di Traduzione Di una DTD in $ODLi_3$: cXML.dtd

Il file sorgente è reperibile sul sito www.cXML.org

Interface Responce

(Source semistructured OrderRequest.xml)

{attribute Status Status;

attribute Response_union_1 Response_union_1?;

}

Interface Status

(Source semistructured OrderRequest.xml)

{attribute int Status_Code;

attribute string Staus_Text;

attribute string Staus_lang?;

}

Interface Response_union_1

(Source semistructured OrderRequest.xml)

{attribute ProfileResponse ProfileResponse;

}union

{attribute PunchOutSetupResponse PunchOutSetupResponse;

}union

{attribute ProviderSetupResponse ProviderSetupResponse;

}union

{attribute GetPendingResponse GetPendingResponse ;

```
 }union
{attribute  SubscriptionListResponse SubscriptionListResponse ;
 }union
{attribute  SubscriptionContentResponse SubscriptionContentResponse ;
 }union
{attribute SupplierListResponse SupplierListResponse;
 }union
{attribute SupplierDataResponse SupplierDataResponse ;
 }
```

Interface ProfileResponse

(Source semistructured OrderRequest.xml)

```
{attribute set<Option> Option;
 attribute Transaction Transaction;
 attribute set<Transaction> Transaction_1;
 attribute string ProfileResponse_ effectiveDate;
 attribute string ProfileResponse_ lastRefresh;
 const string ProfileResponse_a-dtype ='effectiveDate dateTime.tz lastRefresh
dateTime.tz';
}
```

Interface Option

(Source semistructured OrderRequest.xml)

```
{attribute string Option_name;
 const string Option_a-dtype='name string';
 attribute string PCDATA_node;
 }
```

Interface Transaction

(Source semistructured OrderRequest.xml)

```
{attribute URL URL;
```

```
 attribute Option Option;
 attribute string Transaction_ requestName;
 const string Transaction_a-dtype='requestName NMTOKEN'
 }
```

Interface URL

```
(Source semistructured OrderRequest.xml)
{attribute string URL_Name?;
  attribute string PCDATA_node;
 }
```

Interface PunchOutSetupResponse

```
(Source semistructured OrderRequest.xml)
{attribute StartPage StartPage;
 }
```

Interface StartPage

```
(Source semistructured OrderRequest.xml)
{attribute URL URL;
 }
```

ProviderSetupResponse

```
(Source semistructured OrderRequest.xml)
{attribute StartPage StartPage;
 }
```

GetPendingResponse

```
(Source semistructured OrderRequest.xml)
{attribute Cxml Cxml;
  attribute set<Cxml> Cxml; }
```

Interface Cxml

(Source semistructured OrderRequest.xml)

{attribute Cxml_union_1 Cxml_union_1 ;

attribute const string Cxml_version='1.2.007';

attribute string Cxml_Payload;

attribute string Cxml_Timestamp;

attribute string Cxml_lang?;}

Interface Cxml_union_1

{attribute Header Header;

attribute Cxml_union_1_union_2 Cxml_union_1_union_2 ;

}union

{attribute Response Response;

}

Interface Header

(Source semistructured OrderRequest.xml)

{attribute From From;

attribute To To;

attribute Sender Sender;

attribute Path Path?;

attribute OriginalDocument OriginalDocument?;

}

Interface From

(Source semistructured OrderRequest.xml)

{attribute Credential Credential;

attribute set<Credential> Credential_1

}

Interface Credential

(Source semistructured OrderRequest.xml)

{attribute Identity Identity;

```
  attribute Credential_union_2 Credential_union_2;
  attribute string  Credential_Domain;
  attribute enum {'marketplace'} Credential_Type?;
}
```

Interface Identity

```
(Source semistructured OrderRequest.xml)
{attribute string lastChangedTimestamp;
 attribute any Any_node;
}
```

Interface Credential_union_2

```
(Source semistructured OrderRequest.xml)
{ attribute any Shared_Secret;
}union
{ attribute DigitalSignature DigitalSignature;
}
```

Interface DigitalSignature

```
(Source semistructured OrderRequest.xml)
{attribute DigitalSignature_Type DigitalSignature_Type?;
  attribute DigitalSignature_Encoding DigitalSignature_Encoding?;
 attribute any ANY_node ;
}
```

Interface DigitalSignature_Type

```
(Source semistructured OrderRequest.xml)
{const string DigitalSignature_Type_value "PK7 self-contained";
}union
{attribute string Type_node;
 }
```

Interface DigitalSignature_Encoding

(Source semistructured OrderRequest.xml)

{const string DigitalSignature_Encoding_value “Base64”;

}union

{attribute string Encoding_node;

}

Interface To

(Source semistructured OrderRequest.xml)

{attribute Credential Credential;

attribute set<Credential> Credential_1

}

Interface Sender

(Source semistructured OrderRequest.xml)

{attribute Credential Credential;

attribute set<Credential> Credential_1

attribute string UserAgent;}

Interface Path

(Source semistructured OrderRequest.xml)

{attribute Node Node;

attribute set<Node> Node_1 ;

}

Interface Node

(Source semistructured OrderRequest.xml)

{attribute Credential Credential;

attribute set<Credential> Credential_1;

attribute enum{’copy’;

‘route’;

```
                    }Node_type;
 attribute enum {'yes'} ItemDetailsRequired?;
}
```

Interface OriginalDocument

```
(Source semistructured OrderRequest.xml)
{attribute string OriginalDocument_PayloadID;
 }
```

Cxml_union_1_union_2

```
(Source semistructured OrderRequest.xml)
{ attribute Message Message;
}union
{ attribute Request Request;
}
```

Interface Message

```
(Source semistructured OrderRequest.xml)
{attribute Status Status;
 attribute Message_union_2 Message_union_2;
 attribute enum { 'production';
                    'Test ' ;
                  } Message_deploymentMode?;
 attribute string Message_inReplyTo?;
 }
```

Interface Message_union_2

```
(Source semistructured OrderRequest.xml)
{ attribute PunchOutOrderMessage PunchOutOrderMessage ;
}union
```

```
{ attribute ProviderDoneMessage ProviderDoneMessage;
}
{ attribute SubscriptionChangeMessage SubscriptionChangeMessage;
}union
{ attribute DataAvailableMessage DataAvailableMessage;
}
}union
{ attribute SupplierChangeMessage SupplierChangeMessage ;
}
```

Interface PunchOutOrderMessage

```
(Source semistructured OrderRequest.xml)
{attribute any BuyerCookie;
 attribute PunchOutOrderMessageHeader PunchOutOrderMessageHeader;
 attribute set<ItemIn> ItemIn;
 }
```

Interface PunchOutOrderMessageHeader

```
(Source semistructured OrderRequest.xml)
{attribute SourcingStatus SourcingStatus;
 attribute Total Total;
 attribute ShipTo ShipTo?;
  attribute Shipping Shipping?;
  attribute Tax Tax?;
  attribute enum { ‘create’;
                   ‘inspect ‘;
                   ‘edit’;
                  } PunchOutOrderMessageHeader_ operationAllowed;
attribute enum { ‘pending’;
                   ‘final ‘;
                  } PunchOutOrderMessageHeader_ quoteStatus;
 }
```

Interface SourcingStatus

(Source semistructured OrderRequest.xml)

```
{attribute enum { ‘approve’;
                  ‘cancel‘;
                  ‘deny’;
                 } SourcingStatus_action;
 attribute string SourcingStatus_lang;
}
```

interface Total

(Source semistructured OrderRequest.xml)

```
{attribute Money Money; }
```

Interface Money

(Source semistructured OrderRequest.xml)

```
{attribute string Money_Currency;
 attribute int Money_Number?;
 attribute string Money_AlternateCurrency?;
 attribute string PCDATA_Node
   }
```

Interface ShipTo

(Source semistructured OrderRequest.xml)

```
{attribute Address Address;
   }
```

Interface Address

(Source semistructured OrderRequest.xml)

```
{attribute Name Name;
 attribute PostalAddress PostalAddress?;
```

attribute Email Email?;
attribute Phone phone?,
attribute Fax Fax ?;
attribute URL URL ?;
attribute string Address_IsoCountryCode?;
attribute string Address_AddressID?;
}

Interface Name

(Source semistructured OrderRequest.xml)
{attribute string Name_lang;
attribute string PCDATA_node;
}

Interface PostalAddress

(Source semistructured OrderRequest.xml)
{attribute set<string> DeliverTo;
attribute string Street;
attribute set<string> Street_1;
attribute string City;
attribute string State?;
attribute string PostalCode?;
attribute Country Country;
attribute string PostalAddress_Name?; }

interface Country

(Source semistructured OrderRequest.xml)
{attribute string country_IsoCountryCode;
attribute string PCDATA_node;
}

Interface Email

(Source semistructured OrderRequest.xml)

{attribute string Email_Name?;

attribute string PCDATA_node;

}

Interface Phone

(Source semistructured OrderRequest.xml)

{attribute TelephoneNumber TelephoneNumber;

attribute string Phone_Name?;

attribute string PCDATA_node;

}

Interface TelephoneNumber

(Source semistructured OrderRequest.xml)

{attribute CountryCode CountryCode

attribute string AreaCityCode;

attribute int Number;

attribute string Exstension?;

}

interface CountryCode

(Source semistructured OrderRequest.xml)

{attribute string CountryCode_IsoCoutryCode;

attribute string PCDATA_node;

}

Interface Fax

(Source semistructured OrderRequest.xml)

{attribute Fax_union_1 Fax_union_1;

attribute string Fax_Name?;

}

Interface Fax_union_1

{attribute TelephoneNumber TelephoneNumber;

}union

{attribute URL URL;

}union

{attribute Email Email;

}

Interface Shipping

(Source semistructured OrderRequest.xml)

{attribute Money Money;

attribute Description Description;

attribute string Shipping_TrackingDomain?;

attribute string Shipping_TrackingID?;

attribute string Shipping_Tracking?;

}

interface Description

(source semistructured….)

{attribute set<Description_union_1> Description_union_1;

attribute string Description_lang;

}

Interface Description_union_1

{attribute string ShortName;

}

union

{attribute string PCDATA_NODE;

}

Interface Tax

(Source semistructured OrderRequest.xml)

{attribute Money Money;

attribute Description Description;

attribute set <TaxDetail> TaxDetail;

}

Interface TaxDetail

(Source semistructured OrderRequest.xml)

{attribute TaxableAmount TaxableAmount?;

attribute TaxAmount TaxAmount;

attribute TaxLocation TaxLocation?;

attribute Description Description?;

attribute string TaxDetail_Purpose?;

attribute string TaxDetail_category;

attribute string TaxDetail_PercentageRate;

attribute enum {'yes'} IsVatRecoverable;

}

Interface TaxableAmount

(Source semistructured OrderRequest.xml)

{attribute Money Money;

}

Interface TaxAmount

(Source semistructured OrderRequest.xml)

{attribute Money Money;

}

interface TaxLocation

(Source semistructured OrderRequest.xml)

{attribute string TaxLocation_lang;

attribute string PCDATA_node;

}

```
Interface ItemIn
(Source semistructured OrderRequest.xml)
{attribute ItemID ItemID;
 attribute Path Path?;
 attribute ItemDetail ItemDetail;
 attribute SupplierID SupplierID;
 attribute ShipTo ShipTo?;
  attribute Shipping Shipping?;
  attribute Tax Tax?;
attribute int ItemID_Quantity;
attribute int ItemID_lineNumber?;
 }
```

Interface ItemID

```
(Source semistructured OrderRequest.xml)
{attribute string SupplierPartID;
 attribute any SupplierPartAuxiliaryID?
 }
```

Interface ItemDetail

```
(Source semistructured OrderRequest.xml)
{attribute UnitPrice UnitPrice;
  attribute Description Description;
  attribute set<Description> Description_1 ;
  attribute string  UnitOfMeasure;
  attribute Classification Classification ;
  attribute set<Classification> Classification_1 ;
  attribute string ManufacturerPartID?;
  attribute ManufacturerName ManufacturerName?;
  attribute URL URL;
  attribute any Exstrinsic;
}
```

interface UnitPrice

(Source semistructured OrderRequest.xml)

{attribute Money Money;

}

Interface Classification

(Source semistructured OrderRequest.xml)

{attribute string Classification_Domain;

attribute string PCDATA_node; }

Interface ManufacturerName

(Source semistructured OrderRequest.xml)

{attribute string ManufacturerName_lang;

attribute string PCDATA_node;

}

Interface SupplierID

(Source semistructured OrderRequest.xml)

{attribute string SupplierID_Domain;

attribute string PCDATA_node;

}

interface ProviderDoneMessage

(Source semistructured OrderRequest.xml)

{attribute string OriginatorCookie;

attribute set<ReturnData> ReturnData;

}

Interface ReturnData

(Source semistructured OrderRequest.xml)

{attribute string ReturnValue;

attribute Name Name;
attribute string ReturData_Name?;
}

Interface SubscriptionChangeMessage

(Source semistructured OrderRequest.xml)
{attribute Subscription Subscription;
attribute set<Subscription> Subscription;
attribute enum {'new';
'update';
'delete';
} SubscriptionChangeMessage_Type ;
}

Interface Subscription

(Source semistructured OrderRequest.xml)
{attribute InternalID InternalID;
attribute Name Name;
attribute string ChangeTime;
attribute SupplierID SupplierID?;
attribute Format Format?;
attribute Description Description?;
}

Interface InternalID

(Source semistructured OrderRequest.xml)
{attribute string InternalID_Domain;
attribute string PCDATA_node;
}

Interface Format

(Source semistructured OrderRequest.xml)
{attribute string Format_Version;

attribute string PCDATA_node;
}

Interface DataAvailableMessage

(Source semistructured OrderRequest.xml)
{attribute InternalID InternalID;
}

Interface SupplierChangeMessage

(Source semistructured OrderRequest.xml)
{attribute Supplier Supplier ;
attribute set<Supplier> Supplier ;
}

interface Supplier

(Source semistructured OrderRequest.xml)
{attribute Name Name;
attribute Comments Comments?;
attribute SupplierID SupplierID;
attribute set<SupplierID> SupplierID_1;
attribute set<SupplierLOcation> SupplierLocation;
attribute string SupplierID_CorporateURL ?;
attribute string SupplierID_StoreFrontURL ?;
}

Interface Comments

(source semistructured OrderRequest.xml)
{attribute Comments_union_1 Comments_union_1;
attribute string Comments_lang ?;
}

Interface Comments_union_1

(source semistructured OrderRequest.xml)
{attribute string PCDATA_NODE;

}
union
{attribute Attachment Attachment;
}

Interface Attachment
(source semistructured OrderRequest.xml)
{attribute URL URL;
}

Interface SupplierLocation
Interface SupplierID
(Source semistructured OrderRequest.xml)
{attribute Address Address;
attribute OrderMethods OrderMethods;
}

interface OrderMethods
(Source semistructured OrderRequest.xml)
{ attribute OrderMethod OrderMethod;
attribute set<OrderMethod> OrderMethod_1;
attribute Contact Contact?;
}

Interface OrderMethod
(Source semistructured OrderRequest.xml)
{ attribute OrderTarget OrderTarget;
attribute string OrderProtocol?;
}

Interface OrderTarget
(Source semistructured OrderRequest.xml)
{ attribute Phone Phone;

```
  attribute Email Email;
  attribute Fax Fax ;
  attribute URL URL ;
  attribute OtherOrderTarget OtherOrderTarget;
 }
```

Interface OtherOrderTarget

```
(Source semistructured OrderRequest.xml)
{ attribute string OtherOrderTarget_name;
  attribute any ANY_node
 }
```

Interface Request

```
(Source semistructured OrderRequest.xml)
{ attribute any ProfileRequest;
}union
{ attribute OrderRequest OrderRequest;
}union
{ attribute MasterAgreementRequest MasterAgreementRequest;
}union
{ attribute PunchOutSetupRequest PunchOutSetupRequest;
}union
{ attribute ProviderSetupRequest ProviderSetupRequest;
}union
{ attribute StatusUpdateRequest StatusUpdateRequest;
}union
{ attribute GetPendingRequest GetPendingRequest;
}union
{attribute enum {' SubscriptionListRequest ' ;
                 'NO_ SubscriptionListRequest ;
                }Request_SubscriptionListRequest SupplierListRequest;
```

```
} union
{attribute enum {' SupplierListRequest ' ;
                 'NO_ SupplierListRequest ;
                }Request_SupplierListRequest;

}union
{ attribute SupplierDataRequest SupplierDataRequest;
}union
{ attribute CopyRequest CopyRequest;
}union
{ attribute CatalogUploadRequest CatalogUploadRequest;
}
```

Interface OrderRequest

```
(Source semistructured OrderRequest.xml)
{attribute OrderRequestHeader OrderRequestHeader;
 attribute ItemOut ItemOut;
 attribute set<ItemOut> ItemOut_1;}
```

Interface OrderRequestHeader

```
(Source semistructured OrderRequest.xml)
{attribute Total Total;
 attribute ShipTo ShipTo?;
 attribute BillTo BillTo;
 attribute Shipping Shipping?;
 attribute Tax Tax?;
 attribute Payment Payement?;
 attribute set<Contact> Contact;
 attribute Comments Comments ?;
 attribute FollowUp FollowUp?;
```

attribute DocumentReference DocumentReference?;
attribute any Exstrinsic;
}

/* La traduzione di questa interfaccia necessita di alcuni commenti:

- **Per poter tradurre le entità di questo tipo è necessario disporre di una funzione che, incontrando il simbolo % dell'entità parametrica, vada a ricercare la definizione corrispondente confrontando il nome ad essa assegnato. Consideriamo, per esempio, l'entità %isoCurrencyCode, corrispondente al tipo dell'attributo Currency: la funzione riconosce il riferimento l'entità dal simbolo %, quindi va a ricercare una definizione di entità con lo stesso nome per poi sostituirla al posto del suo riferimento.**
- **I tipi specificati per gli attributi non derivano da una traduzione immediata, vengono scelti dall'utente.**
- **Money era un elemento di tipo semplice, corrispondente ad una stringa, ma è stato tradotto come interfaccia poiché contiene degli attributi, e quindi, in ODLi$_3$, equivale ad un tipo complesso. Per memorizzare l'informazione contenuta nel campo PCDATA, relativo all'elemento Money, è stato introdotto il nodo fittizio PCDATA_node. */**

Interface BillTo

(Source semistructured OrderRequest.xml)
{attribute Address Address;
}

Interface Payment

(Source semistructured OrderRequest.xml)
{attribute Pcard Pcard;
}

Interface Pcard

(Source semistructured OrderRequest.xml)

{attribute PostalAddress PostalAddress?;
attribute int PostalAddress_Number;
attribute string Expiration;
attribute string Name?;
}

Interface Contact

(Source semistructured OrderRequest.xml)
{attribute Name Name;
attribute set<PostalAddress> PostalAddress;
attribute set<Email> Email;
attribute set<Phone> Phone;
attribute set<Fax> Fax;
attribute set<URL> URL;
attribute string Contact_Role?;
attribute string Contact_AddressID?;
}

Interface FollowUp

(Source semistructured OrderRequest.xml)
{attribute URL URL;
}

Interface DocumentReference

(Source semistructured OrderRequest.xml)
{attribute string DocumentReference_Payload;
}

Interface ItemOut

{attribute ItemID ItemID;
attribute Path Path?;

```
 attribute ItemDetail ItemDetail?;
 attribute ItemOut_union_4 ItemOut_union_4?;
 attribute  string ItemOut_Quantity;
 attribute int ItemOut_LineNumber?;
 attribute string ItemOut_RequisitionID?;
 attribute string ItemOut_ AgreementItemNumber?;
 attribute string ItemOut_ requestedDeliveryDate?;
}
```

Interface ItemOut_union_4

```
(Source semistructured OrderRequest.xml)
{attribute SupplierID SupplierID;
}union
{attribute SupplierList SupplierList;
 }
```

Interface SupplierList

```
(Source semistructured OrderRequest.xml)
{attribute Supplier Supplier;
 attribute set< Supplier> Supplier_1;
 }
```

Interface MasterAgreementRequest

```
(Source semistructured OrderRequest.xml)
{attribute MasterAgreementRequestHeader MasterAgreementRequestHeader;
 attribute set<AgreementItemOut> AgreementItemOut;
}
```

Interface MasterAgreementRequestHeader

(Source semistructured OrderRequest.xml)
{attribute MaxAmount MaxAmount?;
attribute MinAmount MinAmount?;
attribute MaxReleaseAmount MaxReleaseAmount?;
attribute MinReleaseAmount MinReleaseAmount?;
attribute Contact Contact ;
attribute Comments Comments?;
attribute DocumentReference DocumentReference?;
}

interface MaxAmount
(Source semistructured OrderRequest.xml)
{attribute Money Money;}

Interface MinAmount
(Source semistructured OrderRequest.xml)
{attribute Money Money;}

Interface MaxReleaseAmount
(Source semistructured OrderRequest.xml)
{attribute Money Money;}

Interface MinReleaseAmount
(Source semistructured OrderRequest.xml)
{attribute Money Money;}

Interface AgreementItemOut
(Source semistructured OrderRequest.xml)
{attribute MaxAmount MaxAmount?;
attribute MinAmount MinAmount?;
attribute MaxReleaseAmount MaxReleaseAmount?;
attribute MinReleaseAmount MinReleaseAmount?;
attribute ItemOut ItemOut;

```
 attribute int AgreementItemOut_MaxQuantity?;
 attribute int AgreementItemOut_MinQuantity ?;
 attribute int AgreementItemOut_MaxReleaseQuantity?;
 attribute int AgreementItemOut_MinReleaseQuantity?;
}
```

Interface PunchOutSetupRequest

```
(Source semistructured OrderRequest.xml)
{attribute any BuyerCookie;
 attribute any Extrinsic;
 attribute BrowserFormPost BrowserFormPost?;
 attribute set<Contact> Contact;
 attribute SupplierSetup SupplierSetup?;
 attribute ShipTo ShipTo?;
 attribute SelectedItem SelectedItem?;
 attribute set<ItemOut> ItemOut;
 attribute enum {'create';
                 'inespect';
                  'edit'
                  'source'} PunchOutSetupRequest_ operation;
 }
```

Interface BrowserFormPost

```
(Source semistructured OrderRequest.xml)
{attribute URL URL;}
```

Interface SupplierSetup

```
(Source semistructured OrderRequest.xml)
{attribute URL URL;}
```

Interface SelectedItem

```
(Source semistructured OrderRequest.xml)
```

{attribute ItemID ItemID;
}

Interface ProviderSetupRequest

(Source semistructured OrderRequest.xml)
{attribute string OriginatorCookie;
attribute BrowserFormPost BrowserFormPost?;
attribute FollowUp FollowUp;
attribute string SelectedService;
attribute any Extrinsic;
}

Interface StatusUpdateRequest

(Source semistructured OrderRequest.xml)
{attribute DocumentReference DocumentReference;
attribute status status;
attribute FollowUp FollowUp;
attribute enum {' PaymentStatus';
'SourcingStatus';
'InvoiceStatus'
}StatusEs;
}

Interface GetPendingRequest

(Source semistructured OrderRequest.xml)
{attribute string MessageType;
attribute set< string> MessageType;
attribute int GetPendingRequest_MaxMessage?;
attribute string GetPendingRequest_ lastReceivedTimestamp;
}

Interface SupplierDataRequest

(Source semistructured OrderRequest.xml)

{attribute SupplierID SupplierID;
attribute set<SupplierID>SupplierID;
}

Interface CopyRequest

(Source semistructured OrderRequest.xml)
{attribute Cxml Cxml;
}

Interface SubscriptionListResponse

(Source semistructured OrderRequest.xml)
{attribute Subscription Subscription;
attribute set<Subscription> Subscription;
}

Interface SubscriptionContentResponse

(Source semistructured OrderRequest.xml)
{attribute Subscription Subscription;
attribute set<Subscription> Subscription;
}

Interface SubscriptionContent

(Source semistructured OrderRequest.xml)
{ attribute string CifContent ;
attribute Index Index;
attribute Contract Contract ;
}

interface Index

(Source semistructured OrderRequest.xml)
{ attribute SupplierID SupplierID ;
attribute set <SupplierID> SupplierID;
attribute Comments Comments? ;
attribute set<SearchGroup> SearchGroup;
attribute IndexItem IndexItem;
}

Interface SearchGroup

(Source semistructured OrderRequest.xml)
{attribute Name Name;
attribute SearchAttribute SearchAttribute ;
attribute set< SearchAttribute > SearchAttribute;
}

Interface SearchAttribute

(Source semistructured OrderRequest.xml)
{attribute String SearchAttribute_Name;
attribute string SearchAttribute_Type ;
}

Interface IndexItem

(Source semistructured OrderRequest.xml)
{attribute IndexItemAdd IndexItemAdd;
}union
{attribute IndexItemDelete IndexItemDelete ;
} union
{attribute IndexItemPunchout IndexItemPunchout ;
}

Interface IndexItemAdd

(Source semistructured OrderRequest.xml)
{attribute ItemID ItemID;
attribute ItemDetail ItemDetail;
attribute IndexItemDetail IndexItemDetail;
}

Interface IndexItemDetail

(Source semistructured OrderRequest.xml)
{attribute string leadTime;
attribute string ExpirationDate?;
attribute string EffectiveDate? ;
attribute SearchGroupData SearchGroupData
}

Interface SearchGroupData

(Source semistructured OrderRequest.xml)
{attributeName Name;
attribute SearchDataElement SearchDataElement ;
attribute set< SearchDataElement> SearchDataElement;
}

Interface SearchDataElement

(Source semistructured OrderRequest.xml)
{attribute Name Name;
attribute string SearchDataElement_Name ;
attribute string SearchDataElement_Value;
}

Interface IndexItemDelete

(Source semistructured OrderRequest.xml)
{attribute ItemID ItemID;
}

Interface IndexItemPunchout

(Source semistructured OrderRequest.xml)
{attribute ItemID ItemID;
attribute PunchoutDetail PunchoutDetail;
}

Interface PunchoutDetail
(Source semistructured OrderRequest.xml)
{attribute Description Description;
attribute set<Description> Description_1
attribute URL URL;
attribute Classification Classification ;
attribute set<Classification> Classification_1 ;
attribute ManufacturerName ManufacturerName?;
attribute string ManufacturerPartID?;
attribute string ExpirationDate?;
attribute string EffectiveDate?;
attribute set<SearchGroupData> SearchGroupData;
attribute set<string> TerritoryAvailable
attribute any Exstrinsic;
}

Interface SupplierListResponse
(Source semistructured OrderRequest.xml)
{attribute Subscription Subscription;
attribute SubscriptionContent SubscriptionContent ;
attribute set<SubscriptionContent> SubscriptionContent;
}

Interface SupplierDataResponse
(Source semistructured OrderRequest.xml)
{attribute Subscription Subscription; }

BIBLIOGRAFIA

[1] Ling Liu, Calton Pu, Wei Han. 2000. Georgia Institute of Technology.
"XWRAP: AN XML-enabled Wrapper Construction System for Web Information Sources"

[2] XWRAP Home Page. http://www.cse.ogi.edu/sysl/projects/XWRAP/xwrap.html

[3] XWRAP Elite Home Page. http:/www.cc.gatech.edu/projects/disl/XWRAPElite/

[4] Wei Han, David Buttler, Calton Pu. 2001. Georgia institute of Technology.
"Wrapping Data into XML"

[5] David Buttler, Ling Liu, Calton Pu. Aprile 2001.
"A fully automated object extraction system for the word wide web"

[6] Manlio Marchica. *La Conoscenza di MOMIS: il ruolo di XML-Schema e RDF.* Università degli studi di Modena e Reggio Emilia, facoltà di Ingegneria, corso di laurea in Ingegneria informatica, anno accademico 2001/2002.

[7] N.Guarino. *Semantic matching: Formal ontological distinctions for information organization, extraction, and integration.* Technical report, Summer
School on Information Extraction, Frascati, Italy, July 1997.

[8] N.Guarino. *Understanding, building, and using ontologies*. A commentary to 'Using Explicit Ontologies in KBS Development', by van Heijst, Schreiber, and Wielinga.

[9] H.Garcia-Molina et al. *The TSIMMIS approach to mediation: Data models and languages.* In NGITS workshop, 1995. ftp://db.stanford.edu/pub/garcia/1995/tisimmis-models-languages.ps.

[10] 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 Databases, Athens, Greece, 1997.

[11] Y. Arens, C. A. Knoblock, and C. Hsu. *Query processing in the sims information mediator*. Advanced Planning Technology, 1996.

[12] S.Bergamaschi, S. Castano, S. De Capitani di Vimercati, S. Montanari, and M. Vincini. *An intelligent approach to information integration.* In Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS'98), Trento, Italy, june 1998.

[13] S.Bergamaschi, S. Castano, S. De Capitani di Vimercati, S. Montanari, and M. Vincini. *Exploiting schema knowledge for the integration of heterogeneous sources*. In Sesto Convegno Nazionale su Sistemi Evoluti per Basi di Dati - SEBD98, Ancona, pages 103–122, 1998.

[14] R. Hull and R. King et al. Arpa I^3 reference architecture, 1995. Available at http://www.isse.gmu.edu/I3 Arch/index.html.

[15] Franceso Guerra. *MOMIS: il Wrapper per Sorgenti XML.* Università degli Studi di Modena e Reggio Emilia, facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, Anno Accademico 1999/2000.

[16] Andrea cataldo. *Da linguaggio standard per WWW a linguaggio standard per OODB: il traduttore XML/ODLi3.* Università degli Studi di Modena e Reggio Emilia, facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, Anno Accademico 1999/2000.

[17] R.G.G Cattel editor. *The Object database Standard: ODMG93*. Morgan Kaufmann publishers. San Marco, CA, 1994

[18] R.G.G Cattel and others, editors. *The Object database Standard: ODMG2.0*. Morgan Kaufmann publishers. San Francisco, CA,1997

[19] *XML Tutto&Oltre.* Steven Holzner, Apogeo. 2001.

[20] *DEV. Developing Software Solution.* Pubblicazione mensile del gruppo editoriale Infomedia. Novembre 2001.

[21] World Wide Web Consortium. *W3C XML Specification DTD ("XL Spec").* Version 2.1. http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm.

[22] World Wide Web Consortium. *XML Schema Part 0: Primer. W3C Recommendation, 2 May 2001.* http://www.w3.org/TR/xmlschema-0/

[23] World Wide Web Consortium. *XML Schema Part 1: Structures. W3C Recommendation, 2 May 2001.* http://www.w3.org/TR/xmlschema-1/

[24] World Wide Web Consortium. *XML Schema Part : Datattypes. W3C Recommendation, 2 May 2001.* http://www.w3.org/TR/xmlschema-2/

[25] cXML.org-Commerce XML Resources. http://www.cxml.org/

[26] *cXML Users' Guide.* Version 1.2.1007. novembre, 2001.
http://xml.cxml.org/current/cXMLUsersGuide.pdf

[27] xCBL.org-XML Common Businnes Library. http://www.xcbl.org/