Parole chiave:

Wrapper

MOMIS

XML Schema

ODL_{I^3}

Re-engineering

A mio nonno Athos

che non ha mai smesso

e mai smetterà

di credere in me.

Indice

Capitolo 1

Introduzione

Negli ultimi anni, di pari passo con l'innovazione tecnologica, è diventata una necessità, nel campo informatico, integrare informazioni provenienti da sorgenti di tipo differente. Basti pensare alla crescita esponenziale del numero di fonti di informazione nelle aziende o su Internet che rende possibile l'accesso ad un vastissimo insieme di dati distribuiti su macchine diverse. A questo si aggiunge il fatto che i dati possono essere eterogenei tra loro, il che fa assumere al problema un'importanza rilevante. Un chiaro esempio lo si può trovare nel momento in cui un'azienda deve poter trattare ed interagire con diversi tipi di dati in formato elettronico. In questo caso l'utilizzo di un sistema di integrazione di informazioni risulta molto utile per rendere omogenei i dati ed arrivare al risultato desiderato attraverso una sola query rendendo in tal modo confrontabili i dati.

Le maggiori difficoltà che si incontrano nell'integrazione tra sorgenti diverse consistono principalmente in eterogeneità strutturali ed implementative e alla mancanza di una ontologia comune che porta ad una eterogeneità semantica. Tale eterogeneità semantica è dovuta sostanzialmente a:

- *Differenti tecnologie*: nomi differenti per rappresentare gli stessi concetti in sorgenti diverse;
- *Differenti rappresentazioni strutturali*: utilizzo di modelli diversi per rappresentare informazioni simili.

Il problema dell'integrazione intelligente di informazioni venne individuato in primo luogo dall'agenzia americana ARPA (***A**dvanced **R**esearch **P**rojects **A**gency*) che creò un apposito gruppo di ricerca, I^3(***I**ntelligent **I**nformation **I**ntegration*), allo scopo di condurre ricerche, studiare e proporre linguaggi, protocolli e stru-

menti per risolverlo. Successivamente altri gruppi di ricerca hanno affrontato il problema e al momento sono disponibili diversi strumenti di integrazione.

Il sistema sviluppato dal gruppo di ricerca di Basi di Dati (DBgroup) del dipartimento di Ingegneria dell'Informazione dell'Università di Modena e Reggio Emilia prende il nome di MOMIS[1] (***M****ediator* ***E****nvironment for* ***M****ultiple* ***I****nformation* ***S****ources*)[8]. Si tratta di un sistema a mediatore per l'integrazione intelligente di informazioni provenienti da sorgenti eterogenee basato su un approccio semantico, ossia cercando di effettuare l'integrazione basandosi sul significato delle informazioni stesse. In particolare questo tipo di approccio in MOMIS [8]prevede sia l'annotazione delle informazioni rispetto all'ontologia lessicale di Wordnet sia l'utilizzo della logica descrittiva OLCD; tecniche che consentono di arrivare ad un'integrazione più precisa e ricca permettendo di estrarre e ricavare relazioni semantiche tra le informazioni. Questa caratteristica lo differenzia dagli altri sistemi proposti che si basano invece su un approccio sintattico, che considera invece solo l'aspetto formale dell'informazione e non il significato ad essa associato.

MOMIS[8] realizza l'integrazione di informazioni utilizzando il nuovo linguaggio ODL_{I^3}, estensione del linguaggio standard ODL (***O****bject* ***D****efinition* ***L****anguage*). Quest'ultimo è stato definito dal gruppo di standardizzazione ODMG (***O****bject* ***D****ata* ***M****anagement* ***G****roup*) per descrivere la conoscenza relativa ad uno schema ad oggetti in modo conforme all'ODMG Object Model ed è stato esteso in MOMIS[8] in accordo con le indicazioni del programma I^3dell'agenzia ARPA per creare un'architettura di riferimento per l'integrazione di sorgenti dati eterogenee in maniera automatica. Il linguaggio ODL esteso viene quindi definito con il nome di ODL_{I^3}, acronimo che unisce i due precedenti.

I componenti principali dell'architettura del sistema MOMIS[8] sono il *mediatore* e il *wrapper*.

Un wrapper è un componente che estrae l'informazione contenuta in una sorgente e la traduce nel linguaggio comune ODL_{I^3}: in questo modo tutte le sorgenti sono rappresentate con lo stesso formalismo ed il mediatore risulta svincolato dagli specifici formati con cui le informazioni sono rappresentate. I wrapper sono responsabili dell'accesso alle sorgenti, ed è quindi necessario avere un wrapper specifico per ogni tipo di sorgente che sia in grado di riconoscere ed interpretare il formato dei dati memorizzati in essa. Un wrapper incapsula la sorgente e fornisce al mediatore un'interfaccia standard, nascondendo i dettagli sul formato della sorgente stessa. La funzione di un wrapper è duplice:

- nella fase di integrazione, si occupano di descrivere gli schemi delle sorgenti

[1] http://www.dbgroup.unimo.it/Momis/

nel linguaggio ODL_{I^3} ;

- nella fase di query processing, devono tradurre le query ricevute dal mediatore in interrogazioni comprensibili alla sorgente con cui ognuno di essi opera e presentare al mediatore, attraverso il modello comune di dati utilizzato dal sistema, i dati ricevuti in risposta alla query presentata.

Il mediatore rappresenta il nucleo del sistema e ha il compito di costruire la visione globale a partire dalle descrizioni locali delle singole sorgenti.

Altri componenti rilevanti del sistema MOMIS[8] sono il modulo *Designer Intergrated Interface*, il *Global Schema Builder* ed il *Query Manager*, sottomoduli del mediatore. Il primo è uno strumento di supporto all'integrazione semiautomatica di schemi estratti da sorgenti eterogenee per il progettista dell'integrazione. Il secondo invece è un modulo di integrazione degli schemi locali, in grado di generare un unico schema globale a partire dalle descrizioni delle sorgenti. Il Query Manager è responsabile dell'elaborazione ed ottimizzazione delle interrogazioni sullo schema globale. A partire da una query dell'utente sullo schema globale, esso genera le query in linguaggio OQL (***O**bject **Q**uery **L**anguage*) che viene trasmessa ai singoli wrapper. I wrapper eseguono la query corrispondente alla sorgente che incapsulano e restituiscono il risultato al Query Manager che si occupa infine di fondere le singole risposte e di sintetizzare un risultato globale ed unificato da presentare all'utente[8].

La figura 1.1 illustra i componenti che caratterizzano il sistema MOMIS[8].

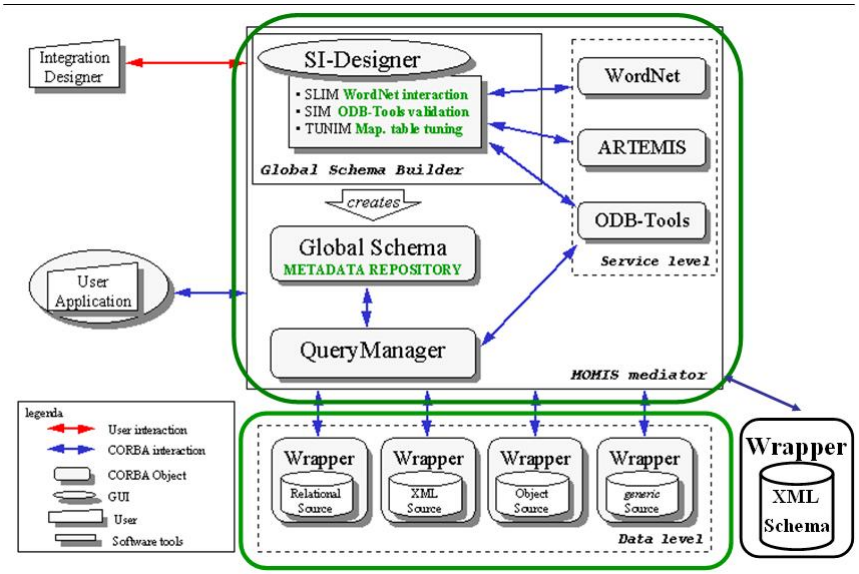

Figura 1.1: Architettura del sistema MOMIS[8]

,Come si può vedere dalla figura 1.2, il processo di acquisizione ed integrazione delle sorgenti prevede diverse fasi:

1. fase di Wrapping: le sorgenti, tramite gli appositi wrapper, sono acquisite e tradotte in schemi ODL_{I^3} locali, internamente al sistema;

2. fase di annotazione manuale delle sorgenti: ognuno degli identificatori presenti negli schemi locali viene annotato rispetto all'ontologia lessicale WordNet. Questa operazione consiste nell'associare un significato al nome di un elemento dello schema locale permettendo in tal modo di sfruttare la conoscenza semantica per ottenere un'integrazione più ricca e precisa.

3. fase di generazione del Common Thesaurus: esso consiste in un insieme di relazioni tra i termini usati negli schemi delle sorgenti. Queste relazioni sono di tipo estensionale ed intensionale (sinonimia, ipernimia, iponimia e relazioni tra i nomi) e descrivono la conoscenza intra ed inter-schema riguardante gli attributi e le classi delle sorgenti. In questa fase il database WordNet è usato per l'estrazione delle relazioni lessicali derivate sulla base delle annotazioni effettuate in precedenza.

4. fase di generazione della vista virtuale globale, GVV (***G****lobal* ***V****irtual* ***V****iew*): il sistema genera uno schema globale e un insieme di mapping tra lo schema globale e le sorgenti locali basato sulle relazioni del Common Thesaurus. In primo luogo viene calcolato un valore di affinità tra ogni coppia di classi appartenenti alla stessa sorgente o a due sorgenti differenti sulla base dei loro nomi, della loro struttura e delle relazioni definite nel Common Thesaurus. Quindi si raggruppano le classi più simili sulla base dei valori di affinità calcolati in precedenza con l'obiettivo di individuare le classi che devono essere integrate poiché esprimono lo stesso concetto. Infine per ogni gruppo di affinità viene definita una classe rappresentativa di tutte le classi dell'insieme e costituita dall'unione dei loro attributi. L'insieme delle classi rappresentative e delle loro relazioni è la vista globale.

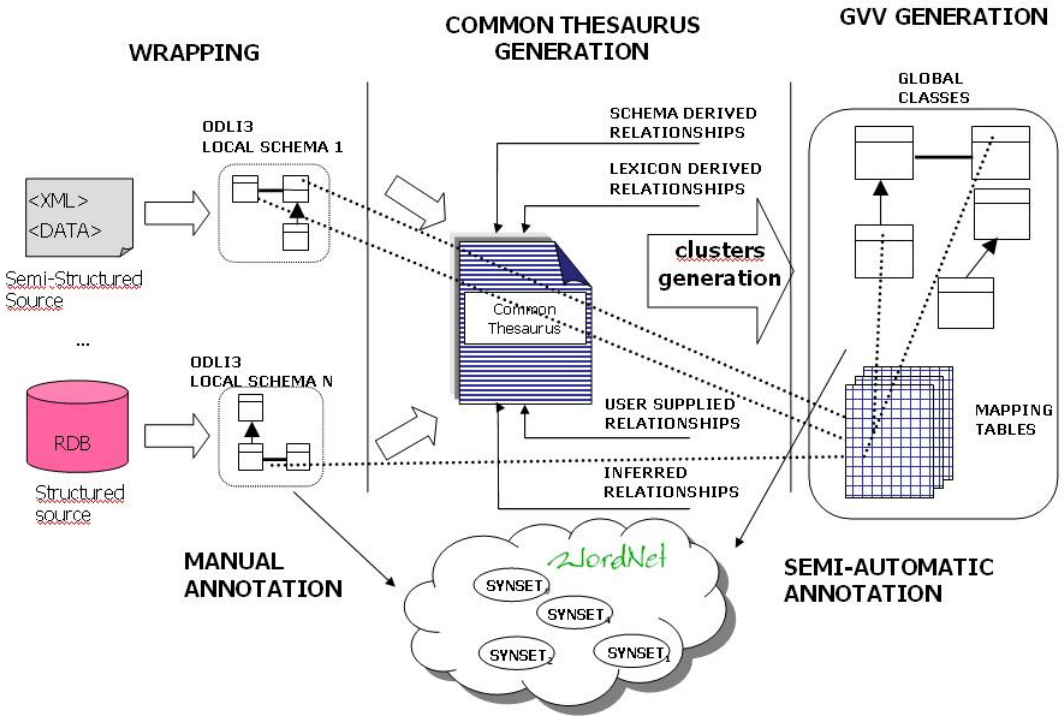

Figura 1.2: Processo I³ in MOMIS[8]

Particolare attenzione quindi deve essere rivolta all'implementazione dei wrapper per le diverse sorgenti. In questa tesi si farà riferimento alla sorgente di tipo XML Schema.

L'XML Schema è un linguaggio di descrizione del contenuto di un file XML (*e**X**tensible **M**arkup **L**anguage*) approvato e validato dal W3C (*W**orld **W**idw **W**eb **C**onsortium*). Il suo scopo è quello di delineare quali elementi sono permessi, quali tipi di dati sono ad essi associati e quale relazione gerarchica hanno fra loro gli elementi contenuti in un file XML. Questo permette principalmente la validazione dei file XML, ovvero laverifica che i suoi componenti siano in accordo con la descrizione in linguaggio XML Schema. Esso si propone inoltre come sostituto al DTD (*D**ocument **T**ype **D**efinition*) di cui ne implementa ed arricchisce la semantica.

Obiettivo della presente tesi è quello di analizzare e descrivere i cambiamenti apportati al wrapper relativo alla traduzione da sorgenti di tipo XML Schema, il cui primo prototipo è stato realizzato da R. Rasi[7]. Nel secondo capitolo verranno presentati i linguaggi di riferimento, XML Schema e ODL_{I^3}, mentre nel terzo verranno analizzate le regole di traduzione ideate da R. Rasi [7] che caratterizzano il wrapper XSD; si procede dunque alla fase di re-engineering in cui vengono individuati e risolti i problemi del wrapper stesso. I problemi trattati

riguardano la gestione di element di tipo globale, che non venivano analizzati e tradotti se non nel caso ristretto di "substitution group", il troncamento dei livelli di nesting, per ottenere un'ottimizzazione della traduzione. Si è inoltre sviluppata la gestione del Model Group choice, che verrà analizzata in maniera dettagliata; in particolare verrà ideata una nuova regola di traduzione per questo componente.

La tesi si conclude con il quarto capitolo in cui viene presentato un caso pratico di studio relativo al benchmark **THALIA**[5] attraverso il quale ci si propone di avere una validazione del lavoro svolto.

Infine nel quinto capitolo vengono introdotte alcune note conclusive.

Capitolo 2

I linguaggi di riferimento

In questo capitolo vengono analizzati i due linguaggi di riferimento su cui si basa il wrapper XML Schema: il linguaggio di partenza, che è appunto l'XML Schema [2, 4, 6] e il linguaggio a cui si perviene attraverso la traduzione, ovvero l'ODL_{I^3} .

2.1 Il linguaggio di partenza: XML Schema

Un XML Schema è una descrizione formale di una grammatica per un linguaggio di markup basato su XML. E' stato sviluppato dal World Wide Web Consortium (*W3C*) [1]come standard per descrivere e specificare la struttura di classi di documenti XML. Inizialmente era utilizzato a tale scopo un approccio basato sui DTD (***D**ocument **T**ype **D**efinition*) che consente di specificare la struttura del documento XML e di ciascun tag utilizzabile al suo interno con una precisione a prima vista accettabile.

Tuttavia, per disporre di un maggiore controllo sugli elementi che possono trovarsi all'interno di uno specifico tipo di documenti XML, i DTD non risultano più sufficienti. Ad esempio, i DTD non mettono a disposizione un meccanismo immediato per indicare che un elemento può contenere al massimo un numero predefinito di sottoelementi, né è possibile specificare che un attributo può assumere valori di un certo tipo di dato, ad esempio valori numerici.

A differenza di un DTD, che utilizza una propria sintassi specifica, un XML Schema utilizza la stessa sintassi XML per definire la grammatica di un linguaggio di markup, cosa che è indice dell'estrema flessibilità di XML.

Quindi uno XML Schema è un documento XML che descrive la grammatica di un linguaggio XML utilizzando un linguaggio di markup specifico; è

[1] http://www.w3.org/

rappresentato interamente in XML 1.0 e fa uso dei namespace.

In particolare un XML Schema ha la seguente struttura generale:

```
<?xml version"1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...  Definizione della grammatica ...
</xs:schema>
```

Un XML Schema è quindi un set di componenti come ad esempio le definizioni di tipi e le dichiarazioni di elementi. Questi possono essere usati per verificare la validità degli oggetti che compongono un documento XML. Si dice pertanto che un documento XML che istanzia uno schema deve essere validato rispetto allo schema stesso: questa procedura viene svolta da un apposito parser e consiste nel verificare che il documento rispetti tutte le strutture e le dichiarazioni dello schema stesso.

Vi sono due livelli di correttezza per un documento XML:

1. correttezza formale rispetto alla sintassi XML.
2. validità di una istanza rispetto a uno schema.

L'associazione tra schema e istanza avviene tramite il meccanismo dei namespace XML: da un lato, con la definizione di uno schema, si costruisce un namespace popolandolo con nuovi tipi di dati e dichiarazioni di elementi ed attributi; dall'altro, un documento XML dichiara di utilizzare uno o più namespace, dai quali importa tutte le dichiarazioni e definizioni.

Facendo uso degli schemi si raggiunge l'obiettivo di mantenere fisicamente separati struttura e contenuto del documento e allo stesso tempo è possibile controllare la creazione di documenti XML, mantenendo un certo grado di omogeneità e compatibilità strutturale.

Un modello XML è a tutti gli effetti un documento XML e in generale viene memorizzato in un file di testo con estensione del nome *".xsd"*.

2.1.1 Formato di un documento XML Schema

Un documento di XML Schema è racchiuso in un elemento <schema>, e può contenere, in varia forma ed ordine, i seguenti elementi:

- <**import**> ed <**include**> per inserire, in varia forma, altri frammenti di schema da altri documenti

- <**simpleType**> e <**complexType**> per la definizione di tipi denominati usabili in seguito
- <**element**> ed <**attribute**> per la definizione di elementi ed attributi globali del documento.
- <**attributeGroup**> e <**group**> per definire serie di attributi e gruppi di content model complessi e denominati.
- <**notation**> per definire notazioni non XML all'interno di un documento XML
- <**annotation**> per esprimere commenti per esseri umani o per applicazioni diverse dal parser di XML Schema.

2.1.2 I tipi in XML Schema

XML Schema usa i tipi per esprimere vincoli sul contenuto di elementi ed attributi.

- Un tipo *semplice* è un tipo di dati che non può contenere markup e non può avere attributi. In pratica è una sequenza di caratteri.
- Un tipo *complesso* è un tipo di dati che può contenere markup e avere attributi. E' l'equivalente di un tipo strutturato o misto.

XML predefinisce un grande numero di tipi semplici: string, decimal, float, boolean, uriReference, date, time, ecc. Ogni tipo semplice è caratterizzato da alcune proprietà, dette facets, che ne descrivono vincoli e formati (permessi ed obblighi). E' data possibilità di derivare nuovi tipi, sia per restrizione che per estensione di permessi ed obblighi.

2.1.2.1 Tipi Semplici

XML Schema introduce il concetto di tipo di dato semplice per definire gli elementi che non possono contenere altri elementi e non prevedono attributi.

Gli elementi e gli attributi sono istanze di un tipo. I tipi semplici sono tipi stringa non ulteriormente strutturati, e possono essere usati per entrambi.

XML Schema non fa nessuna distinzione tra attributi ed elementi con content model testo.

Ad esempio, la seguente dichiarazione:

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

permette l'utilizzo dell'elemento quantita in un documento XML consentendo soltanto un contenuto di tipo intero.

Sono predefiniti molti tipi semplici, che possono essere usati liberamente nelle definizioni. Il nome di un tipo semplice predefinito appartiene allo stesso namespace di XML Schema.

Si riporta di seguito una lista parziale dei tipi semplici più significativi:

- *string*: una stringa di caratteri.
- *boolean*: i valori 'true ' e 'false'.
- *decimal*: una stringa di numeri (con segno e punto): '-34.15 '
- *float*: un reale in notazione scientifica: '12.78E-12 '.
- *duration* : una stringa per una durata temporale nel formato PnYnMnDTnHnMnS. Ad esempio 'P1Y2M3DT10H30M ' è la durata di 1 anno, 2 mesi, 3 giorni, 10 ore, e 30 minuti.
- *date* : una data nel formato CCYY-MM-DD: '2001-04-25 '.
- *time* : un valore di orario nel formato hh:mm:ss con una appendice opzionale per l'indicazione del fuso orario. Es.: '13:20:00+01:00 ' significa 1:20 PM in Middle European Time (+01:00).
- *hexBinary*: dati binari arbitrari in formato esadecimale: '0FB7'.
- *anyURI*: la stringa di un URI, come "http://www.w3.org/ ". Accetta sia URI relativi che assoluti.
- *ID* , *IDREF*: Una stringa senza whitespace con le stesse proprietà e vincoli di ID e IDREF nei DTD.

E' possibile derivare i tipi semplici per restrizione, unione o lista.

Nella **derivazione per restrizione** si parte da un tipo già definito e ne si restringe il set di valori leciti attraverso l'uso di *facet*:

```
<xsd:element name="editore" type="Teditore"/>
<xsd:simpleType name="Teditore">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Addison Wesley"/>
        <xsd:enumeration value="Academic Press"/>
        <xsd:enumeration value="Morgan Kaufmann"/>
 </xsd:restriction>
```

```
</xsd:simpleType>
   <xsd:element name="data" type="Tdatarecente"/>
   <xsd:simpleType name=" Tdatarecente">
       <xsd:restriction base="xsd:date">
          <xsd:minInclusive value="2002-01-01"/>
    </xsd:restriction>
</xsd:simpleType>
```

Per ogni tipo si può quindi precisare dei **facets** ossia delle caratteristiche indipendenti tra di loro che specificano aspetti del tipo:

- *length, minLength, maxLength*: numero richiesto, minimo e massimo di caratteri
- *minExclusive, minInclusive, maxExclusive, maxInclusive*: valore massimo e minimo, inclusivo ed esclusivo
- *precision, scale*: numero di cifre significative e di decimali significativi
- *pattern*: espressione regolare che il valore deve soddisfare
- *enumeration*: lista all'interno dei quali scegliere il valore (simile alla lista di valori leciti degli attributi nei DTD).

Per quanto riguarda la **derivazione per unione** l'insieme dei valori leciti è data dall'unione dei valori leciti di due tipi semplici.

Si riporta di seguito un esempio:

```
<xsd:element name="prezzo" type="Tprezzo">
<xsd:simpleType name="Tprezzo">
    <xsd:union>
       <xsd:simpleType>
          <xsd:restriction base="xsd:decimal">
            <xsd:minExclusive value="0.0"/>
        </xsd:restriction>
     </xsd:simpleType>
       <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="gratis"/>
        </xsd:restriction>
     </xsd:simpleType>
```

```
        </xsd:union>
    </xsd:simpleType>
```

Infine per quanto riguarda la **derivazione per lista** sono valori leciti una lista separata da virgole di valori del tipo semplice specificato.

```
    <xsd:simpleType name="TListaDiNumeri">
        <xsd:list itemType='xsd:decimal'/>
    </xsd:simpleType>
    <xsd:attribute name="coord" type="TListaDiNumeri"/>
    <area coord="25,30,75,90"/>
```

In XML Schema i tipi possono essere predefiniti (solo nel caso di tipi semplici), denominati (con una definizione esplicita, come nei casi precedenti) o anonimi (interni alla definizione di un elemento e contrassegnati dal tag "/"). Ad esempio:

```
    <xsd:element name="editore">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="Addison Wesley"/>
                <xsd:enumeration value="Academic
            Press"/>
                <xsd:enumeration value="Morgan Kaufmann"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>
```

2.1.2.2 Tipi Complessi

I tipi di dato complessi si riferiscono ad elementi che possono contenere altri elementi e possono avere attributi. Definire un elemento di tipo complesso corrisponde a definire la relativa struttura.

Lo schema generale per la definizione di un elemento di tipo complesso è il seguente:

```
    <xs:element name="NOME_ELEMENTO">
        <xs:complexType>
            ...  Definizione del tipo complesso ...
            ...  Definizione degli attributi ...
        </xs:complexType>
```

```
</xs:element>
```

Se l'elemento può contenere altri elementi possiamo definire la sequenza di elementi che possono stare al suo interno utilizzando uno dei costruttori di tipi complessi previsti:

- *<xs:sequence>* Consente di definire una sequenza ordinata di sottoelementi. In particolare la sequenza (A, B, C) diventa:

```
<xsd:sequence>
    <xsd:element name=''A'' type="xsd:string"/>
    <xsd:element name=''B'' type="xsd:string"/>
    <xsd:element name=''C'' type="xsd:string"/>
</xsd:sequence>
```

- *<xs:choice>* Consente di definire un elenco di sottoelementi alternativi. La scelta (A | B | C) diventa:

```
<xsd:choice>
    <xsd:element name=''A'' type="xsd:string"/>
    <xsd:element name=''B'' type="xsd:string"/>
    <xsd:element name=''C'' type="xsd:string"/>
</xsd:choice>
```

- *<xs:all>* Consente di definire una sequenza non ordinata di sottoelementi. Tutti gli elementi debbono essere presenti, ma in qualunque ordine. (A & B & C) diventa:

```
<xsd:all>
    <xsd:element name=''A'' type="xsd:string"/>
    <xsd:element name=''B'' type="xsd:string"/>
    <xsd:element name=''C'' type="xsd:string"/>
</xsd:all>
```

Per ciascuno di questi costruttori e per ciascun elemento è possibile definire il numero di occorrenze previste utilizzando gli attributi *minOccurs* e *maxOccurs*. XML Schema permette non solo i valori 0, 1 e infinito, ma qualunque numero intero. Infinito è "unbounded", e può essere usato solo per maxOccurs. Per default entrambi valgono 1. Inoltre, minOccurs $\leq$ maxOccurs.

La definizione della struttura di un elemento è ricorsiva, cioè contiene la definizione di ciascun elemento che può stare all'interno della struttura stessa.

Per gli elementi vuoti è prevista una definizione basata sul seguente schema:

```
<xs:element name="NOME_ELEMENTO">
    <xs:complexType>
        <xs:complexContent>
            <xs:extension base="xs:anyType" />
              ...  Definizione degli attributi
            ...
        </xs:complexContent>
    </xs:complexType>
</xs:element>
```

In altri termini, un elemento vuoto è considerato un elemento di tipo complesso il cui contenuto non si basa su nessun tipo predefinito.

La derivazione può avvenire per restrizione o estensione.

- per **restrizione**: si limitano ulteriormente i vincoli espressi: modificando minOccurs e maxOccurs, fissando dei valori per certi elementi o attributi, o imponendo ad un elemento un sottotipo del tipo originario.

```
<xsd:complexType name=''nomecognome''>
    <xsd:sequence>
        <xsd:element name=''nome''type=''xsd:string''
     minOccurs=''0'' maxOccurs=''unbounded''/>
        <xsd:element name=''cognome'' type=''xsd:string''/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name=''cognome'' >
    <xsd:restriction base=''nomecognome''>
        <xsd:element name=''cognome'' type=''xsd:string''/>
    </xsd:restriction>
</xsd:complexType>
```

- per **estensione**: aggiungendo al content model nuovi elementi o nuovi attributi. I nuovi elementi devono essere posti necessariamente alla fine degli altri.

```
<xsd:complexType name=''nomecognome''>
    <xsd:sequence>
```

```
            <xsd:element name=''nome''type=''xsd:string''
          minOccurs=''0'' maxOccurs=''unbounded''/>
            <xsd:element name=''cognome'' type=''xsd:string''/>
      </xsd:sequence>
  </xsd:complexType>
    <xsd:complexType name=''nomecontitolo'' >
        <xsd:extension base=''nomecognome''>
            <xsd:sequence>
                <xsd:element ref=''title'' minOccurs=''0''/>
          </xsd:sequence>
      </xsd:extension>
  </xsd:complexType>
```

2.1.3 Elementi ed Attributi

Per definire elementi ed attributi si usano gli elementi <**element**> e <**attribute**>.

Questi hanno vari attributi importanti:

- *Name*: il nome del tag o dell'attributo (definizione locale)
- *Ref*: il nome di un elemento o attributo definito altrove (definizione globale)
- *Type*: il nome del tipo, se non esplicitato come content
- *maxOccurs, minOccurs*: il numero minimo e massimo di occorrenze
- *Fixed, default, nullable, ecc.*: specificano valori fissi, di default e determinano la possibilità di elementi nulli.

Gli elementi e gli attributi possono essere definiti in maniera locale o globale.

Una definizione si dice **globale** se è posta all'interno del tag <*schema*>. In questo caso l'elemento o l'attributo può essere riutilizzato in ogni punto del documento. Si dice **locale** invece se è inserita all'interno di un tag <**complexType**>. In questo caso l'elemento o l'attributo esiste solo se esiste un'istanza di quel tipo, e non può essere riutilizzato fuori della dichiarazione del tipo complesso.

E' possibile inoltre raccogliere gli elementi e gli attributi in gruppi attraverso i costrutti *gruop* ed *attributeGroup*, ad esempio:

```
    <xsd:element name=''A''>
```

```
        <xsd:complexType> <xsd:group ref=''elemA'' />
            <xsd:attributeGroup ref=''attrsA'' />
    </xsd:complexType>
</xsd:element>
        <xsd:group name=''elemA'' />
            <xsd:sequence>
                <xsd:element name=''B'' type="xsd:string"/>

                <xsd:element name=''C'' type="xsd:string"/>
         </xsd:sequence>
    </xsd:group>
<xsd:attributeGroup name=''attrsA">
        <xsd:attribute name=''p'' type=?''sd:string''/>
        <xsd:attribute name=''q'' type=''xsd:string''/>
</xsd:attributeGroup>
```

2.1.4 Annotazioni

In XML Schema esiste un posto specifico dove mettere note ed istruzioni: l'elemento <*annotation*>.

L'elemento <annotation> può contenere elementi <*documentation*>, pensati per essere letti da esseri umani oppure elementi <*appInfo*>, pensati per essere digeriti da applicazioni specifiche.

```
   <xsd:element name=?pippo'>
        <annotation>
            <documentation>elemento pippo</documentation>
     </annotation>
        ...  Il resto della definizione
</xsd:element>
```

2.1.5 I Namespace

Un **namespace** è un insieme di nomi di elementi e nomi di attributi identificati univocamente da un identificatore.

L'identificatore univoco individua l'insieme dei nomi distinguendoli da eventuali omonimie in altri namespace. Per fare un esempio, se nell'ambito di una

grammatica per descrivere dei dati anagrafici è stato definito un elemento indirizzo, questo nome potrebbe essere confuso con l'elemento indirizzo definito nell'ambito di una grammatica che descrive messaggi di posta elettronica. L'identificatore del relativo namespace consente di distinguere i due elementi omonimi.

Nell'ambito delle tecnologie XML, un XML Schema definisce implicitamente un namespace degli elementi e degli attributi che possono essere usati in un documento XML.

In un documento XML si fa riferimento ad un namespace utilizzando un attributo speciale (*xmlns*) associato al root element, come nel seguente esempio:

```
<articolo xmlns="http://www.dominio.it/xml/articolo">
```

Questo indica che l'elemento articolo ed i suoi sottoelementi utilizzano i nomi definiti nel namespace identificato dall'identificatore http://www.dominio.it/xml/articolo.

L'identificatore di un namespace può essere rappresentato da una qualsiasi stringa, purché sia univoca. Proprio per garantirne l'univocità, è prassi ormai consolidata utilizzare un URI (Uniform Resource Identifier) come identificatore.

Non è necessario che l'indirizzo specificato come identificatore di namespace corrisponda ad un file pubblicato sul Web. Esso è utilizzato semplicemente come identificatore ed il parser non accederà al Web per verificare l'esistenza dell'URL.

Per mettere in relazione un namespace con il relativo XML Schema occorre dichiararlo nel root element :

```
<articolo
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.dominio.it/xml/articolo"
    xmlns="http://www.dominio.it/xml/bibliografia"
    xsi:schemaLocation="http://www.dominio.it/xml/articolo

    articolo.xsd"
    xsi:schemaLocation="http://www.dominio.it/xml/bibliografia
    bibliografia.xsd"
>
```

L'attributo `xmlns:xsi` specifica la modalità con cui viene indicato il riferimento allo schema, mentre l'attributo `xsi:schemaLocation` indica il namespace ed il file in cui è definito il relativo XML Schema separati da uno spazio.

E' possibile combinare più namespace facendo in modo che ciascun elemento utilizzato faccia riferimento al proprio namespace.

Occorre tener presente che quando si fa riferimento ad un namespace, questo riferimento vale per l'elemento corrente e per tutti gli elementi contenuti, a meno che non venga specificato un diverso namespace.

Nell'ambito dei namespace per definire il namespace del documento da validare si usa la dichiarazione di *targetNamespace.*

Tramite gli attributi *elementFormDefault* e *attributeFormDefault* viene consentito di controllare se l'uso del prefisso è necessario per i tipi non globali. Queste dichiarazioni hanno entrambe valore di defaut *unqualified* e si applicano a tutti gli attributi o elementi dello schema.

```
<schema xmlns="http://www.w3.org/2000/08/XMLSchema"
    xmlns:po="http://www.example.com/PO1"
    targetNamespace="http://www.example.com/PO1"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">
    <element name=''A'' type="po:prova"/>
    <element name=''C'' type="string"/>
    <complexType name=''po:prova">
        <sequence>
            <element name=''B'' type=''string'' >
            <element ref=''C'' />
     </sequence>
  </complexType>
</schema>
```

In XML Schema, esistono meccanismi per dividere lo schema in più file o per importare definizioni appartenenti ad altri namespace:

- *Include*: Le nuove definizioni appartengono allo stesso namespace, ed è come se venissero inserite direttamente nel documento.
- *Redefine*: come include, le definizioni appartengono allo stesso namespace, ma possono venire ridefiniti tipi, elementi, gruppi, ecc.
- *Import*: le nuove definizioni appartengono ad un altro namespace, ed è l'unico modo per fare schemi che riguardino namespace multipli.

Queste direttive devono essere specificate all'interno dell'elemento <schema>. Ad esempio:

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

```
        targetNamespace="http://www.w1.org"
        xmlns="http://www.w1.org"
        xmlns:cd="http://www.w2.org">
<import namespace="http://www.w2.org" schemaLocation="http://www.w2.org/s1.xsd"/>
   ...
   </xsd:schema>
```

2.1.6 Unicità e Chiavi

In XML Schema, è possibile richiedere che certi valori siano unici, o che certi valori siano chiavi o chiavi di riferimenti:

- **Vincoli di unicità**: assicurano che, in un insieme di elementi (o di tuple), un certo campo assuma valori unici o nulli. In XML Schema questo viene espresso tramite l'elemento *<unique>*.

```
<xsd:unique name="code">
    <xsd:selector xpath="/books/book"/>
    <xsd:field xpath="author"/>
    <xsd:field xpath="title"/>
</xsd:unique>
```

- **Chiavi e chiavi riferite**: definire una chiave su un certo campo per un insieme di elementi equivale a definire un vincolo di unicità sullo stesso campo e, in aggiunta, a stabilire che quel campo non può assumere valori nulli, ossia un valore è sempre specificato.Questo è reso possibile dall'elemento *<key>*. Poiché ad ogni dichiarazione di key viene associato un nome, è possibile riferirsi ad essa nello schema e quindi anche creare delle chiavi riferite. Questo viene fatto tramite in costrutto *<keyref>*. Ad esempio:

```
<xsd:key name="code">
    <xsd:selector xpath="/books/book"/>
    <xsd:field xpath="@code"/>
</xsd:key>
<xsd:keyref name="list" refer="code">
    <xsd:selector xpath="/list/objects"/>
    <xsd:field xpath="@id"/>
</xsd:keyref>
```

2.2 Il linguaggio di arrivo: ODL_{I^3}

Per descrivere la conoscenza relativa ad uno schema ad oggetti in modo conforme all'ODMG Object Model, è stato definito dal gruppo di standardizzazione ODMG il linguaggio standard ODL (***O**bject **D**efinition **L**anguage*). Il linguaggio ODL_{I^3} rappresenta un'estensione di esso, in accordo con le indicazioni del programma I^3(***I**ntelligent **I**nformation **I**ntegration*), volta a rispondere alle problematiche di integrazione di informazioni da fonti eterogenee. Attraverso questo nuovo linguaggio, il sistema MOMIS[8] realizza l'integrazione di informazioni provenienti da fonti diverse, comprendenti anche sorgenti di dati semistrutturati.

Nei prossimi paragrafi verranno illustrati brevemente il linguaggio standard ODL e la sua estensione ODL_{I^3} .

2.2.1 ODL

Di seguito verranno analizzati in sintesi i costrutti del linguaggio ODL, tramite i quali possono essere rappresentati schemi di dati ad oggetti ma non schemi di dati semistrutturati, ossia dati dotati di una struttura irregolare soggetta a cambiamenti.

Il linguaggio ODL distingue i suoi tipi di dato in due categorie:

- **Tipi Classe** (*Interface*): o tipi complessi, sono identificati univocamente, infatti tutte le istanza possiedono un proprio OID (Object IDentifier) che deve essere unico. Sono impiegati per la descrizione di oggetti complessi.
- **Tipi Valore** (*Value Type*): sono identificati direttamente dal proprio valore. Vengono utilizzati per la dichiarazione di attributi semplici e di variabili.

Esiste un tipo semplice particolare che è il tipo Any e rappresenta il supertipo da cui derivano tutti gli altri tipi, sia valore che classe, presenti nel linguaggio ODL_{I^3} . Il suo contenuto può essere quindi sia un attributo semplice che un oggetto complesso.

2.2.1.1 I tipi Valore

I tipi valore (*Value Type*) si dividono in due categorie: i tipi semplici e i tipi costrutti.

I **tipi semplici** (*SimpleType*) sono costituiti da:

- tipi atomici di base (*BaseType*): come i tipi boolean, char, string, date, float, int, unsigned long, etc.

- tipi collezione (*TemplateType*): utilizzati per rappresentare collezioni di dati semplici, sono di quattro tipi diversi:
  - bag (per una collezione di elementi non ordinati).

    ```
    bag <int> IntBag;
    ```

  - set (per un insieme non ordinato di elementi senza duplicati).

    ```
    set <string> StringSet;
    ```

  - list (per una collezione ordinata di elementi non duplicati).

    ```
    list <string> StringList;
    ```

  - array (per elenchi ordinati con un numero fisso di elementi).

    ```
    array <int,5> IntArray;
    ```

I **tipi costrutti** (*ConstrType*) sono divisi in tre sottocategorie:

- tipi enumerati (*EnumType*): restringono il dominio si un SimpleType ad un elenco di valori possibili.

  ```
  enum val {1,5,6,4}
  ```

- tipi struttura (*StructType*): utilizzati per definire strutture di tipi valore.

  ```
  typedef struct Nome {
      string a;
      int b;
      date c;
  } tipostruct;
  ```

- tipo unione (*UnionType*): consente di scegliere il tipo di appartenenza di un attributo sulla base del valore di una variabile in una clausola switch.

  ```
  union NumberType switch(val) {
      case 1:  string num;
      case 2:  int num;
  };
  ```

2.2.1.2 Il tipo Classe

Nel linguaggio ODL le classi vengono chiamate Interfacce (*Interface*). La sua definizione è costituita da due parti fondamentali:

- ***Interface Header***: vengono specificate le caratteristiche dell'interfaccia come il nome della classe (identificatore unico dell'oggetto, *ObjectID*), se presenti le superclassi da cui eredita ed una lista di proprietà (*PropertyList*). Nella PropertyList possono essere specificate proprietà come l'*extent*, che definisce l'insieme delle istanze di una classe all'interno di un Database, ed è possibile definire una chiave (*key*) per l'interfaccia composta da un singolo attributo oppure da un insieme di attributi. In un Interface Header sono consentite una sola definizione di extent e di key mentre è ammessa l'ereditarietà multipla.
- ***Interface Body***: viene descritta la struttura interna dell'interfaccia stessa e definiti l'insieme di attributi e di metodi che ne fanno parte. Un Interface Body può essere composto da:
  - *Attributi*: possono essere di tipo semplice (*ValueType*) oppure di tipo complesso (*Interface*). Hanno cardinalità di default 1.
  - *Relazioni*: sono attributi complessi in quanto possono avere come oggetto solo interfacce e non tipi valore. E' possibile aggiungere all'interno di una relazione informazioni sulla relazione inversa.
  - *Operazioni*: definiscono i metodi della classe e sono utilizzate per descrivere il comportamento della classe stessa.
  - *Costanti*.

  Ciascuno di questi componenti deve avere un nome unico all'interno dell'Interface Body.

Si riporta di seguito un esempio di interfaccia ODL:

```
interface Student:Person
( extent Students
keys matr_id, sec_no )
{
    attribute string name;
    attribute unsigned short matr_id [4];
    attribute long set_no [10];
    attribute General_Information information;
    relationship set <Lesson> follow inverse Lesson::followed_by;
};
```

2.2.2 ODL$_{I^3}$

Per quanto riguarda la rappresentazione della conoscenza relativa ad un singolo schema ad oggetti il linguaggio ODL risulta ben progettato, ma risulta insufficiente per la descrizione e l'integrazione di un insieme di sorgenti eterogenee. Per ovviare a ciò sono state apportate, nell'ambito del progetto MOMIS[8], una serie di modifiche ed estensioni per rappresentare la conoscenza relativa al processo di integrazione intelligente delle informazioni e si è arrivati a specificare un nuovo linguaggio denominato ODL$_{I^3}$.

In questa sezione verranno riportate e descritte brevemente le estensioni e le modifiche apportate al linguaggio ODL che hanno consentito di arrivare alla definizione del linguaggio ODL$_{I^3}$.

2.2.2.1 Estensioni del linguaggio ODL

Le estensioni apportate al linguaggio ODL riguardano i tipi valore e i tipi classe.

Per quanto riguarda i **tipi valore**, è stato aggiunto ai BaseType un nuovo tipo, denominato *Range*, che può essere usato per rappresentare un valore compreso tra un estremo superiore e uno inferiore. Come valori degli estremi è possibile disporre del valore infinito, attraverso i termini *+infinite* e *-infinite*.

```
range 1,10 number;
range 1,+infinite number;
```

Al **tipo classe** sono state effettuate diverse modifiche.

All'interno dell'Interface Header, nella PropertyList, è possibile indicare il tipo della sorgente dati a cui appartiene l'interfaccia ed il nome, che deve essere unico. Il tipo può essere relazionale (*relational* o *nfrealtional*), semistrutturato (*semistructured*), ad oggetti(*object*) o file (*file*).

Sempre nella PropertyList, è stata aggiunta la possibilità di definire delle *foreign key* per descrivere schemi relazionali.

Per descrivere schemi di dati semistrutturati, è stato inserito il costrutto *union*, attraverso il quale di possono definire più Interface Body per una stessa interfaccia, e per gli attributi il costrutto *optional*, indicato da un asterisco (*) postposto alla dichiarazione dell'attributo stesso definendo quindi una cardinalità minima uguale a zero e una cardinalità massima uguale a uno.

E' stato reso inoltre possibile, per l'operazione di integrazione delle sorgenti dati, la dichiarazione di attributi globali (*globalAttribute*). Essi hanno le stesse caratteristiche di un attributo normale ODL, ma in aggiunta hanno una funzione di collegamento ad un oggetto di tipo *MappingRule*.

2.2.2.2 Ulteriori modifiche al linguaggio ODL

Il linguaggio ODL_{I^3} presenta ulteriori innovazioni rispetto al linguaggio di origine ODL che vengono riportate di seguito.

- Introduzione di **regole di mediazione** (*MappingRule*) per meglio specificare l'accoppiamento tra gli attributi globali e gli attributi locali originali. Queste MappingRule sono state ideate in quanto il software del progetto MOMIS[8], durante la fase di integrazione delle sorgenti di dati, effettua un raggruppamento delle classi ritenute tra loro simili, generando nuove classi che contengono attributi globali. Se all'interno di Interface Body, un attributo presenta un riferimento ad una regola di mapping, esso è considerato automaticamente un attributo globale.

- Possibilità di definire **relazioni terminologiche** tra nomi e classi di attributi. Durante il processo di integrazione, queste relazioni vengono definite nella fase di generazione del Thesaurus e possono essere definite tra classi (*InterfaceRel*), attributi (*AttributeRel*) o miste tra classi e attributi (*AttrIntRel*). Le relazioni terminologiche sono di quattro tipi: Ipernimia (BT), Iponimia (NT), Sinonimia (SYN), Associazione (RT).

- Possibilità di dichiarare **vincoli di integrità** sotto forma di regole *if-then* (*rule*), in grado di definire vincoli non specificabili altrimenti.

- Possibilità di esprimere **relazioni estensionali** tra gli oggetti di classi distinte sotto forma di preposizioni. Questi assiomi esprimono le relazioni insiemistiche tra le estensioni di classi definite in sorgenti autonome. Le relazioni estensionali sono disgiunzione ($A \cap B = \emptyset$), inclusione ($B \subseteq A$) ed equivalenza ($A = B$).

- Possibilità di **rappresentare le annotazioni rispetto a WordNet** attraverso il costrutto *WNAnnotation*.

Capitolo 3

Il wrapper XSD

Nella seguente parte del documento verranno brevemente illustrate le regole di traduzione elaborate da R. Rasi [7] e che governano il comportamento del wrapper XSD.

Saranno quindi messi in evidenza i problemi individuati nel wrapper stesso e proposte delle soluzioni ai problemi stessi.

3.1 Regole di traduzione

Per rendere possibile l'implementazione del wrapper, sono state individuate delle regole per consentire la traduzione da XML Schema a ODL_{I^3} .

Non sempre però è possibile effettuare una traduzione esatta e completa a causa della disomogeneità dei costruttori dei due linguaggi; sono state quindi adottate soluzioni alternative. I punti critici riguardano:

- il **mapping tra i tipi di dato predefiniti** dai due linguaggi. In questo caso si può notare che non è possibile una traduzione uno a uno, essendo i tipi predefiniti per XSD in numero maggiore rispetto ai BaseType di ODL_{I^3} .

- la gestione dei **namespace** in ODL_{I^3} . Nel linguaggio ODL_{I^3} il concetto di Namespace non è presente, per cui questo tipo di informazione non può essere mantenuta. A causa di ciò si potrebbe incorrere in casi in cui un documento XML Schema faccia riferimento a oggetti appartenenti a namespace diversi sia dal target namespace stesso, sia dal namespace base del linguaggio (http://www.w3.org/2001/XMLSchema) rendendo così possibile che nello stesso schema si possano trovare oggetti con lo stesso nome non qualificato, ma appartenenti a namespace diversi. Per ovviare

a questo problema di utilizzerà il prefisso che in XML Schema identifica il namespace anche per il nome delle Interface in ODL_{I^3} .

Nella parte seguente saranno trattare le regole di traduzione dei componenti primari del linguaggio XML Schema. Sarà riportata prima una dichiarazione generale del componente in XML Schema e di seguito la corrispondente traduzione in ODL_{I^3} .

3.1.1 Attribute

- Dichiarazione di Attribute all'interno di un contenitore dichiarato localmente:

```
<xs:complexType name=''[Cname]''>
<xs:attribute name=''[Aname]'' type=''[Atype]''/>
<xs:complexType/>

interface [Cname]
{attribute [Atype] [Cname]_[Aname]?;}
```

  Il nome dell'attribute sarà Cname_Aname, dove Cname è il nome dell'interfaccia e Aname quello dell'attributo. Il tag "?" indica che l'attributo è opzionale ed è impostato di default, se non viene specificato in *use*.

- Attribute *required*

```
<xs:complexType name=''[Cname]''>
<xs:attribute name=''[Aname]'' type=''[Atype]'' use=''required''/>
<xs:complexType/>

interface [Cname]
{attribute [Atype] [Cname]_[Aname];}
```

- Attribute *fixed*

```
<xs:complexType name=''[Cname]''>
<xs:attribute name=''[Aname]'' type=''[Atype]'' fixed=''400''/>
<xs:complexType/>

interface [Cname]
{const [Atype] [Cname]_[Aname]=400;}
```

In questa situazione si tradurrà l'attributo con una costante invece che con un attribute; il valore della costante sarà quello specificato per fixed e il tipo dovrà essere un BaseType, per cui si estrarrà dal tipo della dichiarazione il tipo XSD predefinito da cui deriva.

- Attribute *reference*. In questo caso la dichiarazione referenzia un Attribute globale e viene tradotta come segue:

```
<xs:attribute name=''[Aname]'' type=''Atype''/>

<xs:complexType name=''[Cname]''>
<xs:attribute ref=''[Aname]''/>
<xs:complexType/>

interface [Cname]
{const [Atype] [Cname]_[Aname];}
```

3.1.2 Element

- Dichiarazione locale di Element

```
<xs:complexType name=''[Cname]''>
<xs:element name=''[Ename]'' type=''[Etype]''/>
</xs:complexType>

interface [Cname]
{attribute [Etype] [Ename];}
```

- Element *reference*

```
<xs:element name=''[Ename]'' type=''[Etype]''/>

<xs:complexType name=''Cname''>
<xs:attribute ref=''[Ename]'' minOccurs=''0'' maxOccurs=''unbounded''/>
</xs:complexType>

interface [Cname]
{attribute set <[Etype]> [Ename]?;}
```

Gli attributi *maxOccours* e *minOccours* determinano la cardinalità dell'elemento nel componente in cui è dichiarato. Se non specificati si assume che valgano 1. Quando minOccurs="0", l'attributo è opzionale e si indica in ODL_{I^3} con un *"?"* postposto

al nome dell'attributo. Nel caso di limite superiore non definito (maxOccours="unbounded") il tipo dell'attribute sarà *set* <*Etype*> anziché *Etype*. Queste considerazioni si applicano ad ogni tipo di dichiarazione di Element.

- Element con dichiarazioni di tipo anonimo

```
<xs:complexType name=''[Cname]''>
<xs:element name=''[Ename]''/>
<xs:complexType>
<!- ...  ->
</xs:complexType>
</xs:element>
<xs:complexType>

interface [Cname]
{attribute [Cname_Ename_type] [Ename];}

interface [Cname_Ename_type]
{...}
```

- Substitution Group. Caratteristica degli elementi dichiarati globalmente, viene tradotto con un'interfaccia alla quale verrà aggiunto un interface body per ogni elemento del gruppo, più uno per l'elemento di testa del gruppo. Nel caso in cui questo venga dichiarato come abstract non viene tradotto.

```
<xs:element name=''[Sname]'' type=''[Stype]''/>
<xs:element name=''[SubName1]'' type=''[Stype]'' substitutionGroup=''Sname''/>
<xs:element name=''[SubName2]'' type=''[Stype]'' substitutionGroup=''Sname''/>

<xs:complexType name=''[Cname]''>
...
<xs:element ref =''[Sname]''/>
...
</xs:complexType>

interface [Sname]_union
{attribute [Stype] [Sname];}
union
{attribute [Stype] [SubName1];}
```

```
union
{attribute [Stype] [SubName2];}

interface [Cname]
{ ...
attribute [Sname]_union [Sname];
...  }
```

3.1.3 Tipi semplici e complessi

3.1.3.1 Complex Type

- Complex Type generico

```
<xs:complexType name=''[Cname]''>
<xs:sequence>
<xs:element name=''UNO'' type=''xs:string''/>
<xs:element name=''DUE'' type=''xs:string''/>
<xs:element name=''TRE'' type=''xs:string''/>
...
</xs:sequence>
</xs:complexType>

interface [Cname]
{ attribute string UNO;
attribute string DUE;
attribute string TRE;}
```

  In generale si tradurrà un Complex Type di nome *Cname* aggiungendo allo schema ODL_{I^3} un nuova interface di nome *Cname*. I Complex Type dichiarati in maniera anonima saranno tradotti come un normale Complex Type di nome *Cname_ type*, dove *Cname* è questa volta il nome in ODL_{I^3} dell'elemento che nello schema XSD racchiude la dichiarazione anonima.

- Complex Type a contenuto semplice derivato da un Simple Type. In questo caso il contenuto è formato da un elemento <simpleContent>, il quale a sua volta contiene un oggetto di tipo <restriction> o <extension>.

```
<xs:complexType name=''[Cname]''>
<xs:simpleContent>
<xs:extension base=''[SimpleTypeName]''>
```

```
<xs:attribute name=''[Aname]'' type=''[Atype]''/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>

interface [Cname]
{ attribute [SimpleType Name] [Cname]_node;
attribute [Atype] [Cname]_[Aname]?;}
```

L'attributo *base* è un nome qualificato che identifica un tipo a contenuto semplice. In questo caso viene aggiunto alla nuova interfaccia un attributo di nome *Cname_ node*, dove *Cname* è il nome dell'interfaccia.

- Complex Type a contenuto complesso derivato da un altro Complex Type

```
<xs:complexType name=''[BaseCname]''>
<xs:sequence>
...
</xs:sequence>
</xs:complexType>

<xs:complexType name=''[DerivedCname]''>
<xs:complexContent>
<xs:extension base=''[BaseCname]''>
...
</xs:extension>
</xs:complexContent>
</xs:complexType>

interface [BaseCname]
{ ... }

interface [DerivedCname] : [BaseCname]
{ ... }
```

Può essere presente l'elemento <restriction> o <extension>. Nel primo caso posso mantenere l'informazione sulla superclasse impostando una superclasse per la nuova interface recuperando l'interfaccia relativa al Complex Type indicato dall'attributo *base*. Se è

presente l'elemento <restriction> invece non si può mantenere l'informazione sulla superclasse. Si dovrà quindi recuperare l'interface relativa al Complex Type indicato dall'attributo base, copiarla nella nuova interfaccia e poi modificare gli oggetti che sono dichiarati all'interno dell'elemento <restriction>.

- Complex Type con contenuto nullo. Sono considerati di default come elementi complexContent, element-only e derivati per restrizione dal tipo anyType, che non ha nè sottoelementi nè attributi.

```
<xs:complexType name=''EmptyCname''>
<xs:complexContent>
<xs:restriction base=''xs:anyType''>
<xs:attribute name=''[Aname]'' type=''[Atype]''/>
</xs:restriction>
</xs:complexContent>
</xs:complexType>

interface [EmptyCname]
{ attribute [Atype] [EmptyCname]_[Aname]?;}
```

3.1.3.2 Simple Type

- Simple Type di tipo lista

```
<xs:simpleType name=''[ListTypeName]''>
<xs:list itemType=''[itemTypeName]''/>
</xs:seimpleType>

typedef list <itemTypeName> [ListTypeName];
```

- Simple Type di tipo union

```
<xs:simpleType name=''[UnionTypeName]''>
<xs:union memberType=''[memberType_1] [memberType_2] ...
''>
</xs:simpleType>

union [UnionTypeName] switch (val)
{ case 1:  [memberType_1];
case 2:  [memberType_2];
case ...  }
```

- Simple Type derivati per restrizione
  - Simple Type derivato da un tipo lista con sfaccettatura length

```
<xs:simpleType name=”[ListName]”>
<xs:  list itemType=”[itemType]”/>
</xs:simpleType>

<xs:simpleType name =”DerivedName”>
<xs:restriction base=”ListName”>
<xs:length value=”[i]”/>
</xs:restriction>
</xs:simpleType>

typedef list <[itemType]> [ListName];
typedef array <[itemType],[i]> [DerivedName];
```

  - Simple Type enumerato

```
<xs:simpleType name=”[EnumType]”>
<xs:restriction base=”[BaseType]”>
<xs:enumeration value=”[Val_1]”/>
<xs:enumeration value=”[Val_2]”/>
<xs:enumeration value=”[Val_3]”/>
...
</xs:restriction>
</xs:simpleType>

enum [EnumType] { [Val_1], [Val_2], [Val_3], ...  }
```

  - Simple Type traducibile con un tipo range

```
<xs:simpleType name=”[Rname]”>
<xs:restriction base=”[IntegerType]”>
<xs:minInclusive value=”[min]”/>
<xs:maxInclusive value=”[max]”/>
</xs:restriction>
</xs:simpleType>

typedef range [min],[max] [Rname];
```

3.1.4 Attribute Group

```
<xs:  schema ...  >
<xs:attributeGroup name=''[AGname]''>
...
</xs:attributeGroup>

<xs:complexType name=''[Cname]''>
...
<xs:attribute ref=''[AGname]''/>
</xs:complexType>
</xs:schema>

interface [AGname]_group
{ ...  }

interface [Cname]
{ ...
attribute [AGname]_group [AGname];}
```

Un Attribute Group di nome *AGname* sarà tradotto in ODL_{I^3} creando una nuova interface di nome *AGname_group*. Tutti gli oggetti contenuti nel corpo del gruppo saranno tradotti con le opportune regole ed inseriti nella nuova interfaccia. Ogni dichiarazione di Attribute Group nello schema che faccia riferimento ad un gruppo di nome *AGname* sarà tradotta come un attributo di nome *AGname* e di tipo *AGname_group*.

3.1.5 Model Group

```
<xs:schema ...  >
<xs:group name=''MGname''>
...
</xs:group>

<xs:complexType name=''[Cname]''>
<xs:group ref=''MGname''/>
...
</xs:complexType>
</xs:schema>
```

```
interface [MGname]_mgroup
{ ...  }

interface Cname
{ ...
attribute [MGname]_mgroup [MGname];}
```

La definizione di un Model Group di nome MGname è tradotto creando una nuova interface di nome *MGname_mgroup*. Il corpo dell'interfaccia sarà riempito con la traduzione dell'elemento interno alla definizione: uno tra <all>, <sequence> e <choice>.

- Model Group <all>

```
<xs:complexType name=''[Gname]''>
<xs:all>
<xs:element name=''[Ename_1]'' type=''[Ename_1]''/>
<xs:element name=''[Ename_2]'' type=''[Ename_2]''/>
</xs:all>
</xs:complexType>

interface [Gname]
{ attribute [Etype_1] [Ename_1];
attribute [Etype_2] [Ename_2]; }
```

- Model Group <sequence>

```
<xs:complexType name=''[Gname]''>
<xs:sequence>
<xs:element name=''[Ename_1]'' type=''[Ename_1]''/>
<xs:element name=''[Ename_2]'' type=''[Ename_2]''/>
</xs:sequence>
</xs:complexType>

interface [Gname]
{ attribute [Etype_1] [Ename_1];
attribute [Etype_2] [Ename_2]; }
```

- Model Group <choice>

```
<xs:complexType name=''[Gname]''>
<xs:choice>
```

```
<xs:element ref=''[Union_1]''/>
<xs:element ref=''[Union_2]''/>
...
</xs:choice>
</xs:complexType>

interface [Gname]
{ attribute [Gname]_choice [Gname]_choice;}

interface [Gname]_choice
{attribute ...  [Union_1];}
union
{attribute ...  [Union_1];}
union
{attribute ...}
```

3.1.6 Identity-constraint

Gli oggetti Identity-constraint si dividono in tre categorie: vincoli di unicità (*unique*), chiavi (*key*) e chiavi riferite. Questi oggetti vengono tradotti con un costrutto *key*. L'espressione *XPathS* serve per determinare l'interface in cui inserire la chiave e gli attributi da includere.

- Identity constraint unique

```
<unique name=''[Uname]''>
<selector xpath=''[XPathS]''/>
<field xpath=''[XPathF]''>
</unique>
```

- Identity constraint key

```
<key name=''[Kname]''>
<selector xpath=''[XPathS]''/>
<field xpath=''[XPathF]''>
</key>
```

- Identity constraint keyref

```
<keyref name=''[Kname]'' refer=''[Rname]''>
<selector xpath=''[XPathS]''/>
<field xpath=''[XPathF]''>
```

```
</keyref>
```

L'attributo *refer* di valore *[Rname]* serve a determinare la chiave con cui riferire la nuova *foreign_ key*.

3.1.7 Annotation

Gli oggetti di tipo Annotation possono contenere elementi *appinfo*, che non vengono tradotti in quanto non hanno contenuto informativo utile alla comprensione, o elementi *documentation*. Questi ultimi verranno tradotti creando una nuova costante di tipo *string* con valore pari al contenuto dell'oggetto *documentation* e nome *annotation_ i*, dove *i* è un valore progressivo che parte da 1 e aumenta ad ogni Annotation.

```
<xs:complexType name=''[Cname]''>
<xs:annotation>
<xs:documentation>
...  documentation ...
</xs:documentation>
<xsappinfo>
...  appinfo ...
</xs:appinfo>
...
</xs:annotation>

interface Cname
{ const string annotation_1 "...  documentation ...";
...  }
```

3.2 Re-engineering del wrapper XSD

In questa sezione vengono messi in evidenza i problemi individuati analizzando il wrapper XSD implementato e verranno quindi proposte delle soluzioni ad essi.

3.2.1 Problema 1: gestione di element globali

Testando il wrapper con vari schemi XSD si è notato che esso analizzava e traduceva solo schemi con determinati componenti. Dallo studio del codice tramite operazioni di debug è risultato che il wrapper non riusciva ad interpretare gli element dichiarati in maniera globale.

Riporto il seguente schema XSD su cui ho eseguito le prove:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.provaprova.org/abc">
    <complexType name="UNO">
       <sequence>
          <element name="ELEMENT1" type="string"/>
          <element name="ELEMENT2" type="string"/>
     </sequence>
       <attribute name="ATTRIBUTE" type="string"
     use="optional"/>
  </complexType>
       <element name="PROVA_ELEMENT" type="string"/>
</schema>
```

Si tratta di uno schema composto da un complexType di nome *"UNO"*, che contiene una sequenza di elementi e un attributo, e da un elemento globale anonimo.

Analizzando l'output da console ho rilevato che il wrapper procede correttamente all'analisi e traduzione del complexType *"UNO"* e del suo contenuto, mentre non interpreta l'elemento globale:

```
   ------------------------------------------------
   [XSDWrapper-2007.05.30 14:17:12] Trying to solve unresolved
references
   ------------------------------------------------
   ------------------------------------------------
   [XSDWrapper-2007.05.30 14:17:12] Solve Complex.  Interface:
UNO
   ------------------------------------------------
   [XSDWrapper-2007.05.30 14:17:12] Solve Complex.  IntBody:

   [XSDWrapper-2007.05.30 14:17:12] Solve Complex.  Attribute:
ELEMENT1
   [XSDWrapper-2007.05.30 14:17:12] Solve Complex.  Attribute:
UNO_ATTRIBUTE
   [XSDWrapper-2007.05.30 14:17:12] Solve Complex.  Attribute:
ELEMENT2
   ------CompType Translated
   UNO
```

```
------SimpleType Translated
------Substitution Group Translated
```

Il wrapper esegue quindi le operazioni secondo il diagramma di sequenza [3] riportato in figura 3.1:

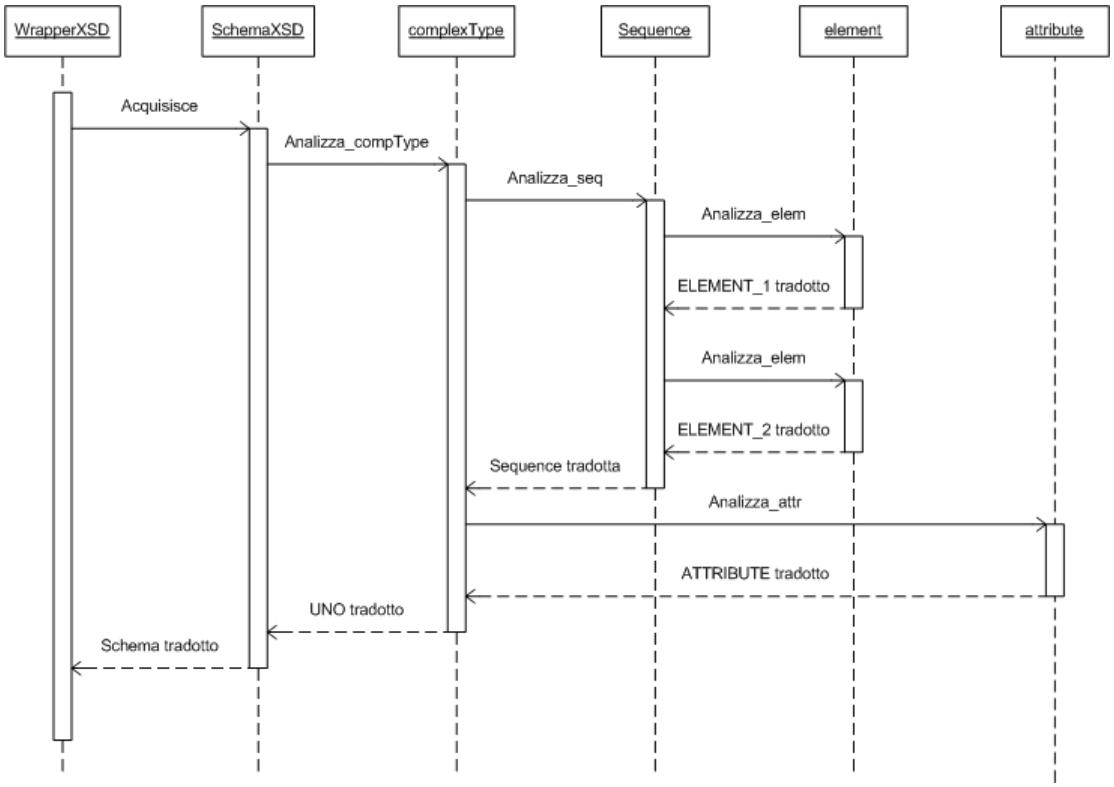

Figura 3.1: Sequence Diagram del comportamento del wrapper prima della modifica.

In questo caso ci si trova di fronte a un global element anonimo, ma se fosse presente un elemento con contenuto semplice o complesso le informazioni contenute andrebbero perse. E' questo il caso degli schemi, riportati in appendice, relativi alle università americane, che quindi non riuscivano ad essere nè analizzati nè tradotti.

Il comportamento del wrapper deve invece rispettare il diagramma di sequenza [3] della figura 3.2:

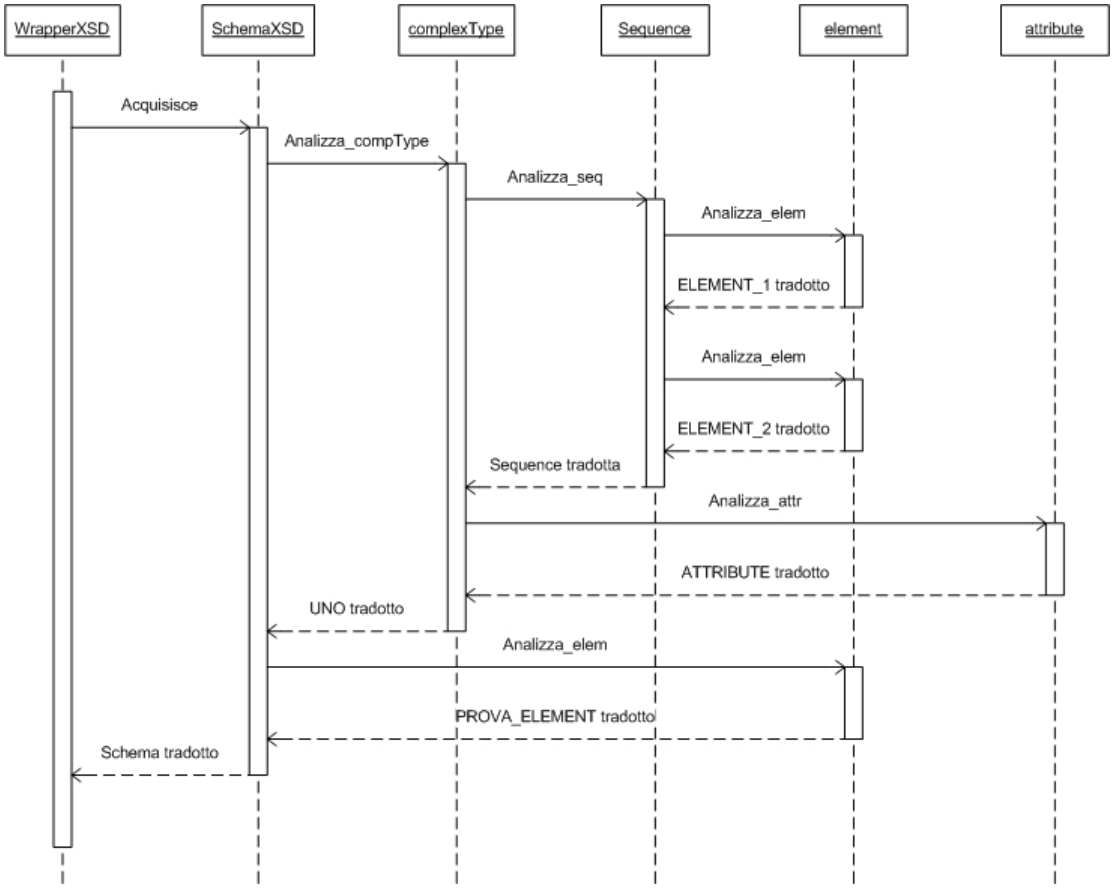

Figura 3.2: Sequence Diagram del comportamento del wrapper corretto.

Compiendo operazioni di debug ho rilevato un malfunzionamento nel metodo *getSchema()* della classe *WrapperXsdCore*. Questo metodo crea e popola un ODL_{I^3} Schema, leggendo e analizzando gli elementi dello schema XSD. In particolare nella sezione in cui vengono analizzati i global element e i substitution group, parte decisiva per la risoluzione del problema, ho notato che il wrapper si limitava a rilevare la presenza dei global element analizzandoli solo nel caso in cui essi facciano parte di un gruppo di sostituzione. Ho quindi provveduto ad aggiungere la parte di codice mancante che funge da anello di raccordo con il resto del codice e che consente quindi di tradurre lo schema in maniera corretta [1].

Il codice inserito attua le seguenti operazioni all'interno del ciclo while già presente per individuare ed analizzare ulteriori element:

1. crea una nuova interfaccia con il nome dell'elemento globale.

2. aggiunge l'interfaccia appena creata al contenitore ODL.

3. se l'elemento contiene al suo interno altri elementi:
   (a) crea un nuovo Interface Body
   (b) lo aggiunge all'interfaccia precedentemente creata
   (c) traduce le dichiarazioni di element presenti passando come parametri l'elemento che si sta analizzando e l'Interface Body appena creato.
4. se l'elemento è astratto non viene tradotto.

Si nota, consultando l'output da console che l'elemento viene ora interpretato e tradotto correttamente:

```
   ------------------------------------------------
   [XSDWrapper-2007.06.20 20:46:33] Trying to solve unresolved
references
   ------------------------------------------------
   ------------------------------------------------
   [XSDWrapper-2007.06.20 20:46:33] Solve Complex.  Interface:
PROVA_ELEMENT
   ------------------------------------------------
   [XSDWrapper-2007.06.20 20:46:33] Solve Complex.  IntBody:
0
   [XSDWrapper-2007.06.20 20:46:33] Solve Complex.  Attribute:
PROVA_ELEMENT
   ------------------------------------------------
   [XSDWrapper-2007.06.20 20:46:33] Solve Complex.  Interface:
UNO
   ------------------------------------------------
   [XSDWrapper-2007.06.20 20:46:33] Solve Complex.  IntBody:
   [XSDWrapper-2007.06.20 20:46:33] Solve Complex.  Attribute:
ELEMENT1
   [XSDWrapper-2007.06.20 20:46:33] Solve Complex.  Attribute:
ELEMENT2
   [XSDWrapper-2007.06.20 20:46:33] Solve Complex.  Attribute:
ATTRIBUTE
   ------CompType Translated
   UNO
   ------SimpleType Translated
   ------Substitution Group Translated
```

3.2.2 Problema 2: troncamento dei livelli di nesting

Analizzando i risultati delle traduzioni di alcuni schemi di riferimento, si è notato che vi era un numero eccessivo di interface. Esse corrispondono ad un numero analogo di tabelle, quindi come nel caso dello schema *asu.xsd* (vedi sezione 4.1) si vengono a creare cinque tabelle per due elementi globali. Questo diventa un problema rilevante se si ha a che fare con una grande quantità di dati: sorge quindi un problema di ottimizzazione della traduzione.

Vengono di seguito riportate le modifiche attuate:

- Prima: per ogni element globale con contenuto complesso vengono create due interface:

  1. la prima relativa all'element avente come attributo <*nameInterface*> di tipo <*nameInterface*>_*type*;
  2. la seconda di nome <*nameInterface*>_*type* avente come attributi gli oggetti contenuti dentro al complexType.

  Si è deciso in questo caso di eliminare l'interfaccia <nameInterface>_type e di mettere direttamente in collegamento la classe dell'elemento globale con il contenuto del complexType. Come risultato quindi ottengo una tabella in meno e vengono messe in maggior evidenza le relazioni tra gli elementi dello schema.

- La seconda modifica riguarda i tag assegnati ai nomi delle interfacce e degli attributi per rendere più chiara la comprensione della traduzione ODL_{I^3} . Per ogni schema che contenga un Model Group di tipo sequence o choice si è deciso di adottare i seguenti tag postposti al nome dell'elemento o dell'attributo:

  – nel caso di una sequence: <*nameInterface*>_*sequence*_*i*
  – nel caso di una choice: <*nameInterface*>_*choice*_*i*

  dove <*nameInterface*> è il nome dell'elemento dello schema XSD in cui è contenuta la sequenza o la choice, *i* è un numero intero progressivo con valore iniziale uguale a 1 usato per differenziare le interfacce e gli attributi nel caso sia presente più di una sequenza o choice.

Il risultato finale a cui si approda per quanto riguarda il file asu.xsd, vedi sezione 4.1, è il seguente:

```
interface ${asu} ( extent ${asu} ) transient {
readonly attribute asu__choice_1 ${asu};
```

```
    }
    ;
    interface ${asu__choice_1} ( extent ${asu_} ) transient
{
    readonly attribute set <Course> ${Course} ?;
    }
    ;
    interface ${Course} ( extent ${Course} ) transient {
    readonly attribute set <Course> ${Course_sequence_1} ?;
    readonly attribute string ${Title} ?;
    }
    ;
    interface ${Course_sequence_1} ( extent ${Course_sequence_1}
) transient {
    readonly attribute string ${Description} ?;
    readonly attribute string ${MoreInfo.URL} ?;
    }
    ;
```

Come viene messo in evidenza dalla traduzione ODL_{I^3} , ho un'interfaccia di nome asu corrispondente all'elemento globale "asu" e avente come attributo asu di tipo asu_choice_1, è quindi evidente che all'interno dell'elemento asu è presente un costrutto choice. Qust'ultima interfaccia ha come attributo Course di tipo Course e rappresenta un set di Course in cui posso avere da un minimo di zero a infiniti elementi di quel tipo. L'interfaccia Course a sua volta ha due attributi, uno, Title,è un attributo semplice di tipo stringa, è un attributo che identifica un set di una sequenza di corsi. Questa sequenza viene tradotto in un'interfaccia Course_sequence_1 che ha come attributi Description e MoreInfo.Url di tipo stringa.

3.2.3 Problema 3: gestione del Model Group choice

Analizzando i risultati della traduzione da XML Schema a ODL_{I^3} si è notato che, nel caso si dovesse tradurre un Model Group di tipo *choice*, la traduzione che veniva effettuata non corrispondeva alle regole di traduzione ideate da R. Rasi [7]. Si è quindi proceduto ad individuare una nuova regola di traduzione che riuscisse a rendere più esauriente il significato del Model Group choice.

In questa sezione verrà innanzitutto analizzato in maniera dettagliata in costrutto choice, quindi verrà illustrata la regola di traduzione originale e infine quella finale che è consente la traduzione di questo componente in maniera corretta.

3.2.3.1 Caratteristiche del Model Group di tipo <choice>

Un componente di tipo Model Group viene utilizzato per specificare il contenuto di elementi che devono contenere altri sottoelementi. Le proprietà di questo componente sono:

- **{annotations}**

  Una sequenza di oggetti di tipo Annotation opzionale.

- **{compositor}**

  Può essere uno tra *{all, choice, sequence}*. E' obbligatorio.

- **{particles}**

  Una lista di componenti Particle, ossia di un elemento che contribuisce alla definizione di un modello di contenuto.

Vengono riportati di seguito alcuni esempi di utilizzo:

```
<xs:all>
<xs:element ref="cats"/>
<xs:element ref="dogs"/>
</xs:all>
<xs:sequence>

<xs:choice>
<xs:element ref="left"/>
<xs:element ref="right"/>
</xs:choice>
<xs:element ref="landmark"/>
</xs:sequence>
```

Un Model Group è quindi formato da un insieme di oggetti Particle, cioè di elementi, wildcard (ossia un componente che permette di includere un generico oggetto preso da uno specifico o da un insieme di namespace, e altri gruppi. Vi sono tre tipi i composizione:

- *all*: significa che il gruppo può contenere solo dichiarazioni di elementi (locali o globali) e la cardinalità degli elementi può essere solamente (0,1) o (1,1). In questo caso nell'istanza gli elementi possono comparire in qualsiasi ordine.

- *sequence*: in questo caso gli elementi nell'istanza devono comparire nello stesso ordine in cui sono specificati nello schema.

- *choice*: questo gruppo rappresenta una scelta tra i componenti del gruppo; questo significa che nell'istanza potrà comparire soltanto uno degli elementi specificati.

Prendendo solamente in considerazione il gruppo di tipo **choice**, la sua sintassi è la seguente:

```
<choice
id = ID
maxOccurs = (nonNegativeInteger | unbounded) :  1
minOccurs = nonNegativeInteger :  1
{any attributes with non-schema namespace .  .  .}>
Content:  (annotation?, (element | group | choice | sequence
| any)*)
</choice>
```

dove:

{maxOccurs} rappresenta il numero massimo di occorrenze e può essere un numero finito non negativo {0,1,2,...} oppure non finito (unbounded). Se non specificato vale 1.

{maxOccurs} identifica il numero minimo di occorrenze. Può essere un numero finito non negativo {0,1,2,...} oppure se non specificato assume il valore di default 1.

Di seguito viene definita la terminologia riguardante il tipo choice ed usata per le regole di validazione.

Ogni Model Group M denota un linguaggio **L(M)**, i cui componenti sono le sequenze di oggetti accettati da M.

Quando una sequenza S di oggetti è controllata rispetto a un Model Group M, la sequenza di Particles P a cui corrispondono gli oggetti di S, nell'ordine, è un *path* di S in M. Dati S e P, il path di S in P non è necessariamente unico.

Se M è un gruppo di tipo choice vengono definiti **L(M)**, il set di path in M, e **V(M)**, il set delle sequenze valide rispetto a M.

- Quando il {compositor} di M è di tipo choice e i {particles} di M sono la sequenza P1, P2, ..., Pn, allora L(M) è L(P1)∪L(P2)∪ ... ∪L(Pn), e il set di path di S in P è il set Q=Q1∪Q2∪...∪Qn, dove Qi è il set di path di S in Pi, per $0<i\leq n$.

  Meno formalmente, quando M di tipo choice tra P1, P2, ..., Pn, allora L(M) contiene qualsiasi sequenza accettata da qualsiasi dei particles P1,

P2, ...,Pn e qualsiasi path di S in qualsiasi dei particles P1, P2, ..., Pn è un path di S in P.

- Il set V(M) è il set delle sequenze S che sono in L(M) e che hanno un path valido in M. Questo significa che se una delle choice presenti in M assegna un componente a una wildcard, e un'altra choice assegna lo stesso componente ad un particle, allora la seconda choice è usata per la validazione del Model Group. Per esempio, se M è

  ```
  <xsd:choice>
  <xsd:any/>
  <xsd:element name="a"/>
  </xsd:choice>
  ```

  allora il path valido per la sequenza (<a/>) contiene solo l'elemento particle ed è a questo particle che verrà attribuito l'elemento in input; il path alternativo, contenente solo la wildcard, non è rilevante per la validazione.

Risulta quindi che se il valore del {compositor} è choice, ci deve essere un particle tra tutti i {particles} tale per cui la sequenza di elementi sia valida.

3.2.3.2 Regola di traduzione precedente

In base ai concetti sopra riportati si era definita la seguente regola di traduzione da XML Schema a ODL_{I^3} .

Secondo la regola riportata in maniera generale al paragrafo 2.1.5, l'elemento choice viene tradotto creando una nuova interfaccia di nome *Gname_ choice*, dove *Gname* è il nome dell'oggetto che contiene la dichiarazione <choice>. Per ognuno degli elementi del content della dichiarazione, si aggiunge alla suddetta interfaccia un nuovo IntBody nel quale viene inserita la traduzione dell'elemento fatta secondo le regole opportune. Infine si aggiungerà all'interfaccia *Gname* un nuovo attributo di nome *Gname_ choice* e tipo *Gname_ choice*, tenendo in considerazione le seguenti regole per gli attributi *minOccurs* e *maxOccurs*:

- se *minOccurs* e *maxOccurs* valgono entrambi 0, l'oggetto non viene tradotto;
- se *minOccurs* vale 0, l'attribute corrispondente sarà dichiarato opzionale (*?*);
- se *maxOccurs* vale unbounded, il tipo dell'attribute sarà set<type> anziché semplicemente type;

- se *minOccurs* e *maxOccurs* sono entrambi pari a n, con n>1 e finito, allora si tradurrà l'attributo come un array di dimensione n;
- se *maxOccurs* è finito e maggiore di minOccurs si tradurrà con un set, rilassando il vincolo di cardinalità; ad esempio, se la cardinalità è (3,5) diventerà (1,n), se è (0,4) diventerà (0,n).

Ad esempio la seguente dichiarazione:

```
<xs:complexType name=''Gname''>
    <xs:choice>
        <xs:element ref=''Union_1''/>
        <xs:element ref=''Union_2''/>name_
 </xs:choice>
</xs:complexType>
```

viene tradotta come:

```
interface Gname
{ attribute Gname_choice Gname_choice; }

interface Gname_choice
{ attribute ...  Union_1; }
union
{attribute ...  Union_2; }
```

Si ha quindi la creazione di due interfacce, cosa che, se si ha a che fare con grandi volumi di dati, si può tradurre in un raddoppiamento delle tabelle.

Per ovviare a ciò, è stata introdotta una nuova regola che sostituisce quella precedente.

3.2.3.3 Nuova regola di traduzione

Secondo questa nuova regola, che rivede e ottimizza quella precedente, l'elemento choice verrà tradotto creando una nuova interfaccia di nome *Gname_choice_i*, dove *Gname* è il nome dell'oggetto che contiene la dichiarazione <choice> ed *i* è un numero intero progressivo con valore iniziale pari a 1: Esso viene usato per differenziare interface o attribute se è presente più di una choice all'interno dello stesso Model Group.

Ogni elemento presente nel content della dichiarazione viene tradotto secondo le regole opportune ed inserito in un nuovo IntBody aggiunto all'interfaccia *Gname_choice_i*. Gli elementi di questa interfaccia sono separati tra loro dalla

stringa UNION che esaurisce in pieno il significato attribuito al Model Group di tipo choice di cui sopra. Per gli attributi *minOccurs* e *maxOccurs* valgono le regole precedenti.

In maniera generale la regola si comporta quindi nella seguente maniera:

```
<xs:complexType name=''[Gname]''>
    <xs:choice>
        <xs:element name=''[Uname_1]'' type=''[Utype_1]''/>
        <xs:element name=''[Uname_2]'' type=''[Utype_2]''/>
        ...
  </xs:choice>
</xs:complexType>

interface [Gname]_choice_i {
attribute [Utype_1] [Uname_1];
union
attribute [Utype_2] [Uname_2];
union
attribute ...  ;
}
```

Se ad esempio ho la seguente dichiarazione:

```
<xs:complexType name="UNIONE">
    <xs:choice>
        <xs:element name="UNO" type="xs:string"
     />
        <xs:element name="DUE" type="xs:string"
     />
        <xs:element name="TRE" type="xs:string"
     />
  </xs:choice>
</xs:complexType>
```

la traduzione avverrà nella maniera schematizzata dal seguente diagramma di sequenza [3] (figura 3.3):

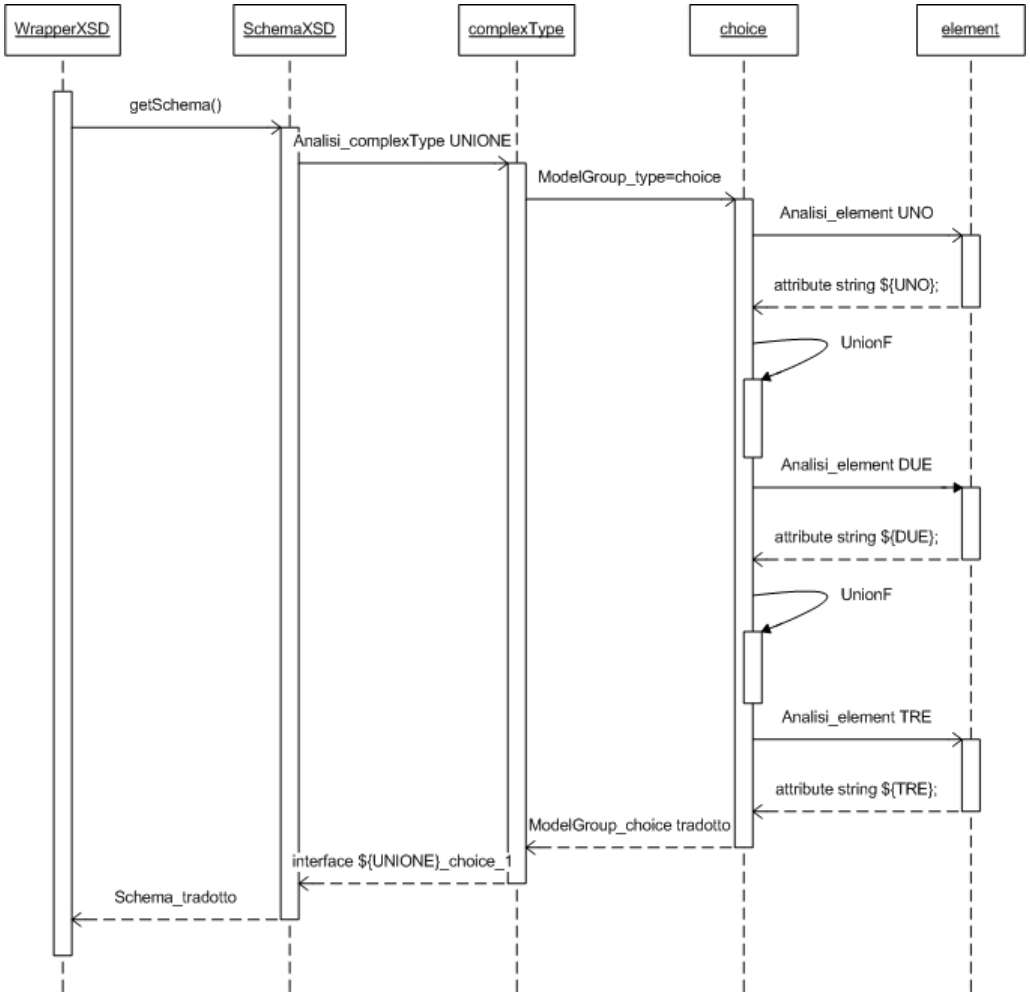

Figura 3.3: Sequence Diagram della traduzione dello schema precedente.

Si procede a creare un'interfaccia di nome *UNIONE_ choice_ 1* il cui contenuto è composto dai tre attributi di tipo stringa separati tra loro dalla stringa UNION. Si perviene quindi alla seguente traduzione:

```
interface ${UNIONE_choice_1}{
attribute string ${TRE};
union
attribute string ${DUE};
union
attribute string ${UNO};
}
;
```

Per arrivare a questo risultato si è dovuto interagire con l'intero package ODL_{I^3} .

Di seguito, figura 3.4, si rappresenta il diagramma delle classi [3] che derivano da *TypeContainer*, particolare della classe *MomisObject* che è alla base della gerarchia delle classi nel package java `it.unimo.dbgroup.momis.odli3` relativo alla definizione del linguaggio ODL_{I^3} .

La classe TypeContainer rappresenta un generico tipo di contenitore del linguaggio ODL_{I^3} e si specializza nelle sottoclassi Schema, che rappresenta uno schema ODL_{I^3} , Source, che rappresenta la singola sorgente, Interface, che rappresenta un generico tipo-classe, e IntBody, che rappresenta il corpo di una interface.

Sono state apportate modifiche alla classe *TypeContainer* ed è stato aggiunto un metodo (**UnionF**) alla classe *IntBody,* messo anche in risalto dal diagramma di sequenza precedente. Attraverso questo metodo, ogni componente del corpo della choice viene tradotto come attribute e ciascuno di essi viene separato dalla stringa UNION.

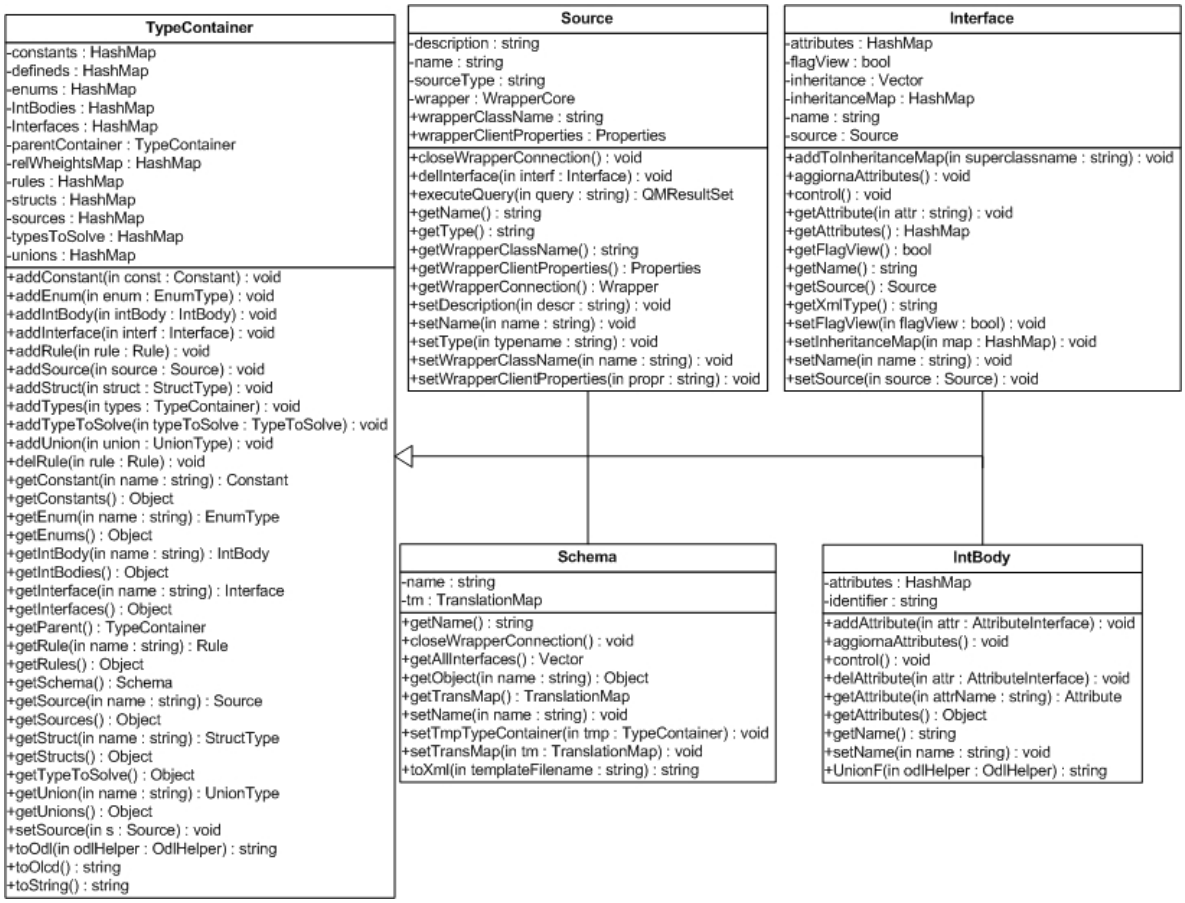

Figura 3.4: Class Diagram TypeContainer e classi derivate.

Capitolo 4

Validazione su caso di studio: il benchmark THALIA

Nel presente capitolo viene riportato il caso di studio relativo al benchmark **THALIA** [5] al quale vengono applicate le regole precedentemente trovate per ottenere una validazione pratica del lavoro svolto.

THALIA è un benchmark pubblico e disponibile per i sistemi di integrazione di infermazioni. Esso fornisce oltre 40 fonti scaricabili che rappresentano i cataloghi dei corsi delle Università di computer science di tutto il mondo. Lo scopo di questo benchmark è è una classificazione sistematica dei differenti tipi di eterogeneità sintattiche e semantiche che sono descritte da venti query che vengono fornite. Per ogni caso, una benchmark query è stata formulata e viene applicata ad un *target schema* così come ad un *challenge schema* che fornisce l'eterogeneità risolta dal sistema di integrazione.

Nel corso di questo studio si è lavorato sui dieci schemi forniti dal *Package.ver_ 1.6*; di seguito sono riportati cinque di questi target schema in formato XML Schema e i rispettivi challenge schema in ODL_{I^3} .

4.1 File asu.xsd

Target schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="asu">
    <xs:annotation>
```

```
<xs:documentation>Arizona State University</xs:documentation>
</xs:annotation>
<xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Course" minOccurs="0"
    maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence minOccurs="0"
          maxOccurs="unbounded">
            <xs:element name="MoreInfo.URL"
          type="xs:string" minOccurs="0"/>
            <xs:element name="Description"
          type="xs:string" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="Title"
          type="xs:string" use="optional"/>
            </xs:complexType>
        </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
  </xs:schema>
```

Challenge schema:

```
  interface ${asu} ( extent ${asu} ) transient {
  readonly attribute asu__choice_1 ${asu};
  }
  ;
  interface ${asu__choice_1} ( extent ${asu_} ) transient
{
  readonly attribute set <Course> ${Course} ?;
  }
  ;
  interface ${Course} ( extent ${Course} ) transient {
  readonly attribute set <Course> ${Course_sequence_1} ?;
  readonly attribute string ${Title} ?;
  }
  ;
```

```
   interface ${Course_sequence_1} ( extent ${Course_sequence_1}
) transient {
   readonly attribute string ${Description} ?;
   readonly attribute string ${MoreInfo.URL} ?;
   }
   ;
```

4.2 File brown.xsd

Target schema:

```
   <?xml version="1.0" encoding="UTF-8"?>
   <xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="brown">
      <xs:annotation>
      <xs:documentation>Brown University</xs:documentation>
      </xs:annotation>
         <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
              <xs:element name="Course" minOccurs="0"
             maxOccurs="unbounded">
                 <xs:complexType>
                 <xs:sequence minOccurs="0"
              maxOccurs="unbounded">
                 <xs:element name="Code" type="xs:string"
              minOccurs="0"/>
                 <xs:element name="Instructor"
              type="xs:string" minOccurs="0"/>
                 <xs:element name="Title"
              type="xs:string" minOccurs="0"/>
                 <xs:element name="Room" type="xs:string"
              minOccurs="0"/>
                 </xs:sequence>
                 </xs:complexType>
             </xs:element>
            </xs:choice>
         </xs:complexType>
      </xs:element>
```

```
</xs:schema>
```

Challenge schema:

```
   interface ${brown} ( extent ${brown} ) transient {
   readonly attribute brown__choice_1 ${brown};
   }
   ;
   interface ${brown__choice_1} ( extent ${brown_} ) transient
{
   readonly attribute set <Course> ${Course} ?;
   }
   ;
   interface ${Course} ( extent ${Course} ) transient {
   readonly attribute set <Course> ${Course_sequence_1} ?;
   }
   ;
   interface ${Course_sequence_1} ( extent ${Course_sequence_1}
) transient {
   readonly attribute string ${Room} ?;
   readonly attribute string ${Instructor} ?;
   readonly attribute string ${Code} ?;
   readonly attribute string ${Title} ?;
   }
   ;
```

4.3 File cmu.xsd

Targer schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.provaprova.org/abc">
    <xs:element name="cmu">
    <xs:annotation>
    <xs:documentation>Carnegie Mellon University</xs:documentation>
    </xs:annotation>
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
```

```
                <xs:element name="Course" minOccurs="0"
              maxOccurs="unbounded">
                  <xs:complexType>
                  <xs:sequence minOccurs="0"
                maxOccurs="unbounded">
                  <xs:element name="Code" type="xs:string"
                minOccurs="0"/>
                  <xs:element name="Sec" type="xs:string"
                minOccurs="0"/>
                  <xs:element name="CourseX-Listed"
                type="xs:string" minOccurs="0"/>
                  <xs:element name="CourseTitle"
                type="xs:string" minOccurs="0"/>
                  <xs:element name="Lecturer"
                type="xs:string" minOccurs="0"/>
                  <xs:element name="Room" type="xs:string"
                minOccurs="0"/>
                  <xs:element name="Day" type="xs:string"
                minOccurs="0"/>
                  <xs:element name="Time" type="xs:string"
                minOccurs="0"/>
                  <xs:element name="Units"
                type="xs:string" minOccurs="0"/>
                  </xs:sequence>
                  </xs:complexType>
              </xs:element>
            </xs:choice>
          </xs:complexType>
        </xs:element>
    </schema>
```

Challenge schema:

```
    interface ${cmu} ( extent ${cmu} ) transient {
    readonly attribute cmu__choice_1 ${cmu};
    }
    ;
    interface ${cmu__choice_1} ( extent ${cmu_} ) transient
{
    readonly attribute set <Course> ${Course} ?;
```

```
   }
   ;
   interface ${Course} ( extent ${Course} ) transient {
   readonly attribute set <Course> ${Course_sequence_1} ?;
   }
   ;
   interface ${Course_sequence_1} ( extent ${Course_sequence_1}
) transient {
   readonly attribute string ${CourseX-Listed} ?;
   readonly attribute string ${Lecturer} ?;
   readonly attribute string ${Room} ?;
   readonly attribute string ${Time} ?;
   readonly attribute string ${Units} ?;
   readonly attribute string ${Day} ?;
   readonly attribute string ${Code} ?;
   readonly attribute string ${CourseTitle} ?;
   readonly attribute string ${Sec} ?;
   }
   ;
```

4.4 File toronto.xsd

Target schema:

```
   <?xml version="1.0" encoding="UTF-8"?>
   <xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
       <xs:element name="toronto">
       <xs:annotation>
       <xs:documentation>University of Toronto</xs:documentation>
       </xs:annotation>
           <xs:complexType>
               <xs:choice minOccurs="0" maxOccurs="unbounded">
                 <xs:element name="course" minOccurs="0"
              maxOccurs="unbounded">
                 <xs:complexType>
                   <xs:sequence minOccurs="0"
                maxOccurs="unbounded">
```

```
                    <xs:element name="title"
                  type="xs:string" minOccurs="0"/>
                    <xs:element name="instructor"
                  minOccurs="0">
                    <xs:complexType>
                    <xs:simpleContent>
                    <xs:extension base="xs:string">
                    <xs:attribute name="Email"
                  type="xs:string" use="optional"/>
                    <xs:attribute name="Name"
                  type="xs:string" use="optional"/>
                    </xs:extension>
                    </xs:simpleContent>
                    </xs:complexType>
                    </xs:element>
                    <xs:element name="location"
                  type="xs:string" minOccurs="0"/>
                    <xs:element name="coursewebsite"
                  type="xs:string" minOccurs="0"/>
                    <xs:element name="prereq"
                  type="xs:string" minOccurs="0"/>
                    <xs:element name="text" type="xs:string"
                  minOccurs="0"/>
                    </xs:sequence>
                <xs:attribute name="No" type="xs:string"
                use="optional"/>
                   <xs:attribute name="level" type="xs:string"
                use="optional"/>
                    <xs:attribute name="offeredTerm"
                  type="xs:string" use="optional"/>
                    </xs:complexType>
                    </xs:element>
              </xs:choice>
          </xs:complexType>
      </xs:element>
  </xs:schema>
```

Challenge schema:

```
   interface ${course} ( extent ${course} ) transient {
   readonly attribute string ${course_sequence_1} ?;
   readonly attribute string ${level} ?;
   readonly attribute string ${offeredTerm} ?;
   readonly attribute string ${No} ?;
   }
   ;
   interface ${course_sequence_1} ( extent ${course_sequence_1}
) transient {
   readonly attribute string ${text} ?;
   readonly attribute string ${title} ?;
   readonly attribute instructor ${instructor} ?;
   readonly attribute string ${location} ?;
   readonly attribute string ${prereq} ?;
   readonly attribute string ${coursewebsite} ?;
   }
   ;
   interface ${instructor} ( extent ${instructor} ) transient
{
   readonly attribute string ${Name} ?;
   readonly attribute string ${Email} ?;
   }
   ;
   interface ${toronto} ( extent ${toronto} ) transient {

   readonly attribute toronto__choice_1 ${toronto};
   }
   ;
   interface ${toronto__choice_1} ( extent ${toronto_} ) transient
{
   readonly attribute set <course> ${course} ?;
   }
   ;
```

4.5 File umich.xsd

Target schema:

```
   <?xml version="1.0" encoding="UTF-8"?>
```

```
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="umich">
    <xs:annotation>
    <xs:documentation>University of Michigan</xs:documentation>
    </xs:annotation>
        <xs:complexType>
            <xs:sequence>
              <xs:element name="course" minOccurs="0"
            maxOccurs="unbounded">
              <xs:complexType>
              <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="prerequisite"
            type="xs:string" minOccurs="0"/>
              <xs:element name="field" type="xs:string"
            minOccurs="0"/>
              <xs:element name="credits" type="xs:string"
            minOccurs="0"/>
              <xs:element name="description"
            type="xs:string" minOccurs="0"/>
              </xs:sequence>
              <xs:attribute name="subject"
            type="xs:string" use="required"/>
              <xs:attribute name="catalognumber"
            type="xs:string" use="required"/>
              </xs:complexType>
              </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Challenge schema:

```
interface ${course} ( extent ${course} ) transient {
readonly attribute string ${field} ?;
readonly attribute string ${prerequisite} ?;
readonly attribute string ${catalognumber};
readonly attribute string ${description} ?;
```

```
readonly attribute string ${subject};
readonly attribute string ${name};
readonly attribute string ${credits} ?;
}
;
interface ${umich} ( extent ${umich} ) transient {
readonly attribute umich_ ${umich};
}
;
interface ${umich_} ( extent ${umich_} ) transient {
readonly attribute set <course> ${course} ?;
}
;
```

Capitolo 5

Conclusioni

In questa tesi è stato presentato il lavoro di re-engineering svolto sul wrapper XML Schema progettato ed implementato da R. Rasi[7] che consente di effettuare una traduzione dal linguaggio XML Schema al linguaggio ODL$_{I^3}$. Gli interventi che sono stati apportati hanno consentito al wrapper di funzionare in maniera corretta ed ottimizzata rispetto alla versione precedente e di poter quindi essere utilizzato come wrapper nel sistema MOMIS[8].

Per raggiungere questo risultato si sono innanzitutto studiate ed analizzate la sintassi e la semantica del linguaggio di partenza, ovvero l'XML Schema, e del linguaggio di arrivo, ovvero l'ODL$_{I^3}$.

In primo luogo si è agito sulla gestione di element di tipo globali: notando che non venivano analizzati e tradotti (se non nel caso in cui essi facevano parte di un substitution group), si è operato implementando la parte di codice mancante che consente al wrapper di funzionare in maniera corretta e di fornire quindi la traduzione esatta secondo le regole ideate da R. Rasi[7].

Per ottenere un'ottimizzazione della traduzione, è stato in secondo luogo effettuato un troncamento dei livelli di nesting che hanno consentito di ridurre il numero di interfacce e quindi di tabelle create attraverso la traduzione stessa. Inoltre, nel caso in cui uno schema contenga un Model Group di tipo sequence o choice, sono stati adottati nuovi tag che vengono postposti al nome dell'elemento o dell'attributo e che consentono una più chiara comprensione della traduzione ODL$_{I^3}$.

Infine si è intervenuto sulla gestione del Model Group di tipo choice. Avendo notato che non vi era corrispondenza tra la traduzione e le regole di traduzione di R. Rasi[7] e dopo aver analizzato in maniera approfondita il costrutto di tipo choice, è stata ideata una nuova regola che rende più esauriente il significato di questo componente.

In conclusione gli interventi apportati al wrapper migliorano la traduzione

effettuata, rendendo più chiara ed esauriente la corrispondenza tra i due linguaggi pur tenendo presente che a livello generale il linguaggio XML Schema è caratterizzato da una maggiore espressività rispetto a ODL_{I^3}. Il linguaggio XML Schema è stato infatti progettato per descrivere la struttura di documenti XML semistrutturati, per cui è necessario che consenta una certa flessibilità nella specifica delle strutture. ODL_{I^3} nasce invece dalla necessità di estendere il linguaggio ODL a oggetti per permettere l'integrazione di sorgenti di dati eterogenee. Queste sorgenti inizialmente erano solo di tipo strutturato; in un secondo momento si è proceduto all'integrazione di sorgenti semistrutturate che hanno reso indispensabile l'aggiunta di nuovi costrutti che hanno esteso il linguaggio.

In particolare si è notato che nella nuova traduzione effettuata da XML Schema a ODL_{I^3} a seguito del troncamento di livelli di innestamento venivano perse alcune informazioni relative alle interfacce eliminate. In compenso però si è ottenuta una maggiore compattezza dei traduzione.

Il lavoro svolto in questa tesi è terminato attraverso l'applicazione del wrapper modificato al caso di studio THALIA[5] che ha messo in evidenza il corretto funzionamento del wrapper stesso ottenendo una validazione dei cambiamenti apportati.

Bibliografia

[1] http://europa-mirror1.eclipse.org/modeling/mdt/xsd/javadoc/2.3.0/.

[2] Priscilla Walmsley David C. Fallside. XML Schema part 0: Primer second edition. Recommendation REC-xmlschema-0-20041028, World Wide Web Consortium, October 2004.

[3] Martin Fowler. *UML Distilled: a brief guide to the standard object modeling language*. Addison Wesley.

[4] Murray Maloney Noah Mendelsohn Henry Thompson, David Beech. XML Schema part 1: Structures second edition. Technical Report EDIINFRR0645, The University of Edimburgh, October 2004.

[5] O. Topsakal J.Hammer, M.Stonebraker. THALIA: Test Harness for the Assessment of Legacy Information Integration Approaches. ICDE 2005: 485-486.

[6] Ashok Malhotra Paul V. Biron. XML Schema part 2: Datatypes second edition. Recommendation REC-xmlschema-0-20041028, World Wide Web Consortium, 2004.

[7] Roberto Rasi. Progettazione e realizzazione del wrapper XML Schema per il sistema MOMIS. Tesi di Laurea Specialistica, Università di Modena e Reggio Emilia, 2005.

[8] D. Beneventano M. Vincini S. Bergamaschi, S. Castano. Semantic Integration of Heterogeneous Information Sources. *Special Issue on Intelligent Information Integration, Data and Knowledge Engineering*, 36(1):215–249, 2001. Elsevier Science B.V.

Ringraziamenti

In primo luogo desidero ringraziare la Professoressa Sonia Bergamaschi per la grande disponibilità e cortesia dimostratemi durante il compimento della presente tesi.

Un ringraziamento particolare va all'Ingegner Maurizio Vincini per la costante disponibilità e l'aiuto determinante fornitomi durante il periodo di realizzazione del presente documento.

Esprimo la mia più grande gratitudine a mia cugina Carla che è stata il mio punto di riferimento durante questo periodo grazie al suo sostegno continuo e immancabile.

Un grazie di cuore va anche a Daniele per avermi supportata e sopportata ogni giorno durante la preparazione della tesi.

I miei più sentiti ringraziamenti vanno ai miei genitori, ai miei fratelli Marco e Luca, e a tutti i miei famigliari per avermi trasmesso la serenità necessaria per conseguire in tranquillità questo obiettivo.

Ringrazio inoltre Elisa, Tuan, Chicco, Chiara, tutti i miei amici e compagni di corso con i quali ho condiviso momenti belli e brutti sia di vita quotidiana che universitaria.

Un immancabile grazie va infine a mio nonno Athos. Se ho raggiunto questo importante traguardo lo devo soprattutto a lui e alla forza che riesce ad infondermi anche se non è più qui. So che ovunque sarà continuerà ad essere al mio fianco e a vegliare su di me.