Università degli studi di Modena e Reggio Emilia

Facoltà di Ingegneria - Sede di Modena
Corso di Laurea Specialistica in Ingegneria Informatica

Progetto e realizzazione di un wrapper XML Schema per il sistema MOMIS

Relatore
Chiar.ma Prof. **Sonia Bergamaschi**

Tesi di Laurea di
Roberto Rasi

Correlatore
Chiar.mo Prof. **Maurizio Vincini**

Anno Accademico 2004/2005

Keywords:
XML Schema
ODL_{I^3}
MOMIS
Wrapper
Java

Ringraziamenti

Il primo ringraziamento va ai miei genitori, per il loro sostegno, economico e morale, che mi ha permesso di raggiungere questo importante traguardo con la serenità necessaria.

Il secondo ringraziamento, non meno importante, è per Alisia, che mi è stata sempre vicina, senza farmi mai mancare tutta la sua fiducia, e con la sua costante presenza ha saputo darmi, soprattutto in questi ultimi mesi, l'incoraggiamento e l'aiuto, fisico e morale, senza il quale non ce l'avrei mai fatta.

Desidero inoltre ringraziare la Prof. Sonia Bergamaschi, il Prof. Maurizio Vincini, l'Ing.Mirko Orsini, l'Ing. Francesco Guerra per gli aiuti, i consigli e la costante disponibilità dimostrata durante la realizzazione di questa tesi.

Indice

Introduzione

Il problema dell'integrazione è molto sentito in questi anni nel campo dell'informatica; nasce dalla necessità di avere un accesso integrato ad informazioni provenienti da sorgenti eterogenee, che renda cioè possibile l'esecuzione di query e la ricerca di informazioni simili ma provenienti da siti diversi. Si pensi ad esempio allo scenario in cui diverse aziende, operanti nello stesso settore, pubblichino i loro cataloghi su web con formati diversi, e che un cliente abbia la necessità di accedere ai vari cataloghi per effettuare un acquisto: avendo a disposizione uno strumento di integrazione potrebbe eseguire una sola query per tutti i cataloghi e confrontare i risultati in maniera semplice e rapida in quanto gli stessi sono già rappresentati in maniera integrata. Un altro esempio potrebbe essere il caso in cui un'azienda, in seguito a una fusione o una acquisizione, si trovi di fronte alla presenza di due sistemi informatici: in questo caso uno strumento di integrazione permetterebbe di evitare il processo di migrazione dei dati, potenzialmente problematico.

Come ambito di ricerca, quello dell'integrazione intelligente di informazioni è stato individuato inizialmente dalla agenzia americana ARPA (***A**dvanced **R**esearch **P**rojects **A**gency*), che ha creato un apposito gruppo di ricerca, denominato I^3 [1] (acronimo per ***I**ntelligent **I**nformation **I**ntegration*), allo scopo di formalizzare il problema e di studiare e proporre linguaggi, protocolli e strumenti per risolverlo. In seguito, altri gruppi di ricerca hanno affrontato il problema, e attualmente sono disponibili diversi strumenti.

Il sistema MOMIS[1] [2, 3] (***M**ediator envir**O**nment for **M**ultiple **I**nformation **S**ources*) è un sistema a mediatore per l'integrazione intelligente di informazioni estratte da sorgenti eterogenee, frutto di una collaborazione tra l'Università di Modena e l'Università di Milano. MOMIS si distingue dagli altri sistemi che sono stati proposti in quanto si basa su un approccio semantico anzichè sintattico all'integrazione. L'approccio sintattico, sicuramente più semplice a livello concettuale e soprattutto implementativo, considera però solo l'aspetto formale dell'informazione e non il significato ad essa associato, per cui ha un grosso limite applicativo nel non riconoscere informazioni simili rappresentate con nomi diversi tra loro; è inoltre difficilmente applicabile in contesti multilingua, in cui è richiesta l'integrazione di sorgenti scritte in lingue diverse oltre che in formati diversi. L'approccio semantico, invece, cerca di effettuare l'integrazione delle informazioni in maniera intelligente basandosi sul significato delle informazioni stesse. In particolare l'approccio semantico implementato in MOMIS prevede, da una parte, l'annotazione delle informazioni rispetto all'ontologia lessicale Wordnet, dall'altra, l'utilizzo della logica descrittiva OLCD: entrambe queste tecniche permettono di estrarre e ricavare relazioni semantiche tra le informazioni, permettendo così una integrazione più precisa e ricca, anche se più complessa dal punto di vista progettuale e realizzativo.

MOMIS è definito come un sistema a mediatore per l'integrazione intelligente di informazioni estratte da sorgenti eterogenee. In figura 1 è riportato un grafico dell'architettura di MOMIS. I componenti principali dell'architettura del sistema sono il *mediatore* ed i *wrapper*. Il mediatore rappresenta il nucleo del sistema ed ha il compito di costruire la visione globale a partire dalle descrizioni locali delle singole sorgenti. 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 informazio-

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

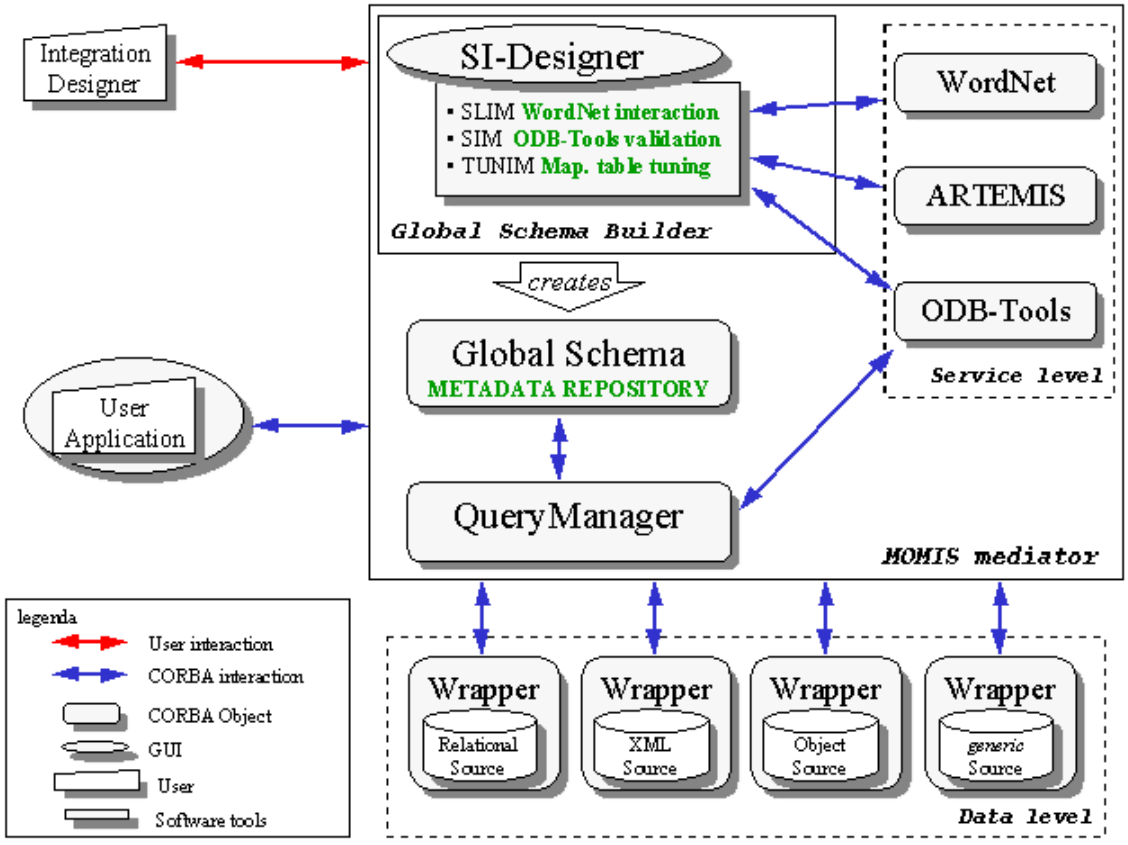

Figura 1: Architettura del sistema MOMIS

ni sono rappresentate. I wrapper sono responsabili dell'accesso alle sorgenti, ed è quindi necessario avere, per ogni tipologia di sorgente, un wrapper specifico che sia in grado di riconoscere ed interpretare il formato dei dati memorizzati in essa; un wrapper, di fatto, incapsula una sorgente, e fornisce al mediatore una interfaccia standard, nascondendo i dettagli sul formato della sorgente stessa. Altri due componenti notevoli del sistema sono il modulo *SI-Designer* ed il *Query Manager*, entrambi sottomoduli del mediatore. Il primo è uno strumento di supporto all'integrazione semiautomatica di schemi estratti da sorgenti eterogenee: possiamo considerarlo come l'interfaccia utente del sistema MOMIS, in quanto è con questo modulo che interagisce il progettista nel processo di integrazione

delle sorgenti. Il secondo invece è responsabile della elaborazione ed ottimizzazione delle interrogazioni. A partire da una query dell'utente sullo schema globale, esso genera le query in linguaggio OQL (**O**bject **Q**uery **L**anguage) da trasmettere ai singoli wrapper. I wrapper eseguono la loro query sulla sorgente che incapsulano e restituiscono il risultato al Query Manager, il quale si occupa infine di fondere le singole risposte e di sintetizzare un risultato globale unificato da presentare all'utente.

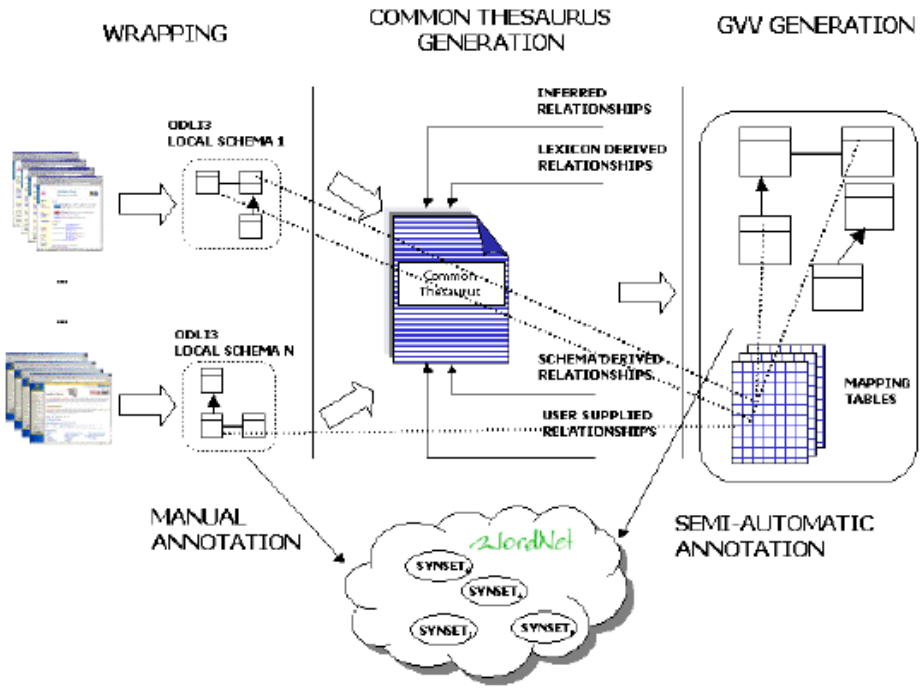

Figura 2: Sistema MOMIS: il processo di Integrazione

La figura 2 rappresenta schematicamente il processo di acquisizione e integrazione delle sorgenti, che spieghiamo brevemente.

1. Innanzitutto le sorgenti, tramite i *wrapper*, sono acquisite e tradotte in schemi ODL_{I^3} locali, internamente al sistema (fase di *Wrapping*).

2. Successivamente vi è la fase di *annotazione manuale* delle sorgenti, durante la quale ognuno degli identificatori presenti negli schemi locali viene *annotato* rispetto alla ontologia lessicale *WordNet*. L'operazione di annotazione consiste nell'associare al nome di un elemento dello schema locale, come ad esempio un attributo o una classe, un significato, permettendo così di sfruttare la conoscenza semantica per ottenere una integrazione migliore (più ricca e più precisa).

3. La terza fase è la generazione del *Common Thesaurus*. Il Common Thesaurus è un insieme di relazioni terminologiche di tipo estensionale ed intensionale, che descrivono la conoscenza intra e inter-schema riguardante gli attributi e le classi delle sorgenti. La conoscenza intra e inter-schema viene espressa tramite le relazioni terminologiche di tipo estensionale ed intensionale sinonimia, ipernimia, iponimia e relazione tra i nomi delle classi e degli attributi. In questa fase il database WordNet è usato per l'estrazione delle relazioni lessicali derivate, sulla base delle annotazioni effettuate in precedenza.

4. La quarta fase è la generazione della vista virtuale globale, la GVV (**G**lobal **V**irtual **V**iew). Dapprima si calcola un valore di affinità tra ogni coppia di classi appartenenti alla stessa sorgente o a due sorgenti diverse, sulla base dei loro nomi, della loro struttura e delle relazioni nel *Common Thesaurus*. In seguito si raggruppano le classi più simili sulla base dei valori di affinità precedentemente calcolati: l'obiettivo è quello di individuare le classi che devono essere integrate perchè esprimono lo stesso concetto. Infine, per ognuno dei gruppi di affinità si definisce 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.

Il lavoro di ricerca svolto in questa tesi si inserisce nell'ambito del progetto CROSS[2], promosso dalla regione Emilia-Romagna con la partecipazione dei principali centri di ricerca della regione, per il quale veniva richiesto l'uso del sistema MOMIS per l'integrazione di sorgenti di dati fornite sotto forma di documenti XML Schema.

Il linguaggio XML Schema [4, 5, 6] è stato sviluppato dal W3C (***W**orld **W**ide **W**eb **C**onsortium*) come standard per definire la struttura e la sintassi di una classe di documenti XML (*e**X**tensible **M**arkup **L**anguage*), detti istanze dello schema, e si propone come sostituto di DTD (***D**ocument **T**ype **D**efinition*). Questo nuovo linguaggio, oltre ad implementare ed arricchire la semantica di DTD, fa uso dei namespace XML, è interamente rappresentato in XML 1.0, e fornisce quel supporto completo alla validazione che le DTD garantiscono solo parzialmente. Si chiama validazione la procedura con la quale si controlla, nell'istanza, il rispetto dei vincoli e delle strutture definite nello schema. XML Schema è una raccomandazione W3C dal 28 ottobre 2004.

Lo scopo della presente tesi è appunto la progettazione e la realizzazione di un wrapper per il sistema MOMIS in grado di estrarre informazioni da una sorgente di tipo XML Schema e di tradurle nel linguaggio comune ODL_{I^3}, estendendo così le funzionalità del sistema e permettendo l'integrazione di sorgenti anche di tipo XSD (***X**ml **S**chema **D**efinition*).

[2] http://www.cross-lab.it

Struttura della Tesi

La presente tesi è costituita da cinque capitoli ed una appendice, oltre al presente capitolo introduttivo.

Nel **capitolo 1** saranno descritti i costrutti del linguaggio XML Schema, riportando per ciascuno le proprietà e la sintassi. Si parlerà dell'utilizzo dei namespace, di come viene definito uno schema e di come può essere specificata in uno schema la struttura degli elementi dell'istanza.

Nel **capitolo 2** si parlerà del linguaggio ODL$_{I^3}$, che come già detto è il linguaggio che il sistema MOMIS utilizza per rappresentare internamente le sorgenti di dati. Esso deriva dal linguaggio ODL (***O**bject **D**efinition **L**anguage*), definito dalla ODMG che sarà descritto nella prima parte del capitolo. Nella seconda parte invece si parlerà della estensione I^3, che da origine, appunto, al linguaggio ODL$_{I^3}$.

Nel **capitolo 3** saranno messi a confronto i due linguaggi descritti nei capitoli precedenti ed in particolare si descriveranno le regole di traduzione da XML Schema a ODL$_{I^3}$ utilizzate per la realizzazione del wrapper XML Schema.

Nel **capitolo 4** si descriveranno le caratteristiche del componente software realizzato.

Nel **capitolo 5** si trarranno le conclusioni del lavoro svolto e saranno presentate alcune proposte di sviluppo futuro.

Nella **appendice A** viene riportata la sintassi del linguaggio ODL$_{I^3}$ in formato BNF.

Capitolo 1

Il linguaggio XML Schema

Introduzione

Il linguaggio XML Schema [4, 5, 6] (*XSD*) è stato sviluppato dal World Wide Web Consortium (*W3C*) come standard per descrivere e specificare la struttura di classi di documenti XML. A questo scopo era inizialmente utilizzato il linguaggio XML 1.0 document type definition(*DTD*). XML Schema implementa tutte le funzionalità dello standard DTD, aggiungendone di nuove; inoltre è rappresentato interamente in XML 1.0 e fa uso dei namespace.

Lo standard XML Schema consente di specificare in un modello la struttura di una classe di documenti XML. 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. La validità di una istanza rispetto ad uno schema rappresenta il secondo livello di correttezza per un documento XML, mentre il primo livello consiste

nella correttezza formale rispetto la sintassi XML. 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 definizioni e dichiarazioni.

Tramite l'uso degli schemi è possibile controllare la creazione di documenti XML, mantenendo un certo grado di omogeneità e compatibilità strutturale. Allo stesso tempo si raggiunge l'obiettivo di mantenere fisicamente separati struttura e contenuto del documento: ad esempio, è possibile analizzare la struttura di un insieme di documenti senza dover processare tutti i dati in essi contenuti.

Nel seguito del capitolo saranno descritte le proprietà e la sintassi degli elementi del linguaggio XML Schema, a partire dallo schema nel complesso.

1.1 Struttura di un documento XML Schema

Un documento *XML Schema* è un insieme di *componenti*. I componenti sono i blocchi generici con i quali si costruiscono gli schemi.

Ci sono 13 tipi diversi di componenti di un XML Schema, divisi in tre gruppi:

- i componenti **primari**, che possono (le definizioni di tipi) o devono (le dichiarazioni di elementi o attributi) avere un nome:
  - *dichiarazioni di Attribute*
  - *dichiarazioni di Element*
  - *definizioni di Complex Type*
  - *definizioni di Simple Type*

- i componenti **secondari**, che devono necessariamente avere un nome:
  - *definizioni di Attribute Group*
  - *definizioni di Model Group*
  - *definizioni di Identity Constraint*
  - *dichiarazioni di Notation*
- i componenti **helper**, che rappresentano piccole parti di altri componenti, non sono indipendenti dal contesto e non possono avere un nome:
  - *Annotation*
  - *Model Group*
  - *Particle*
  - *Wildcard*
  - *Attribute Use*

Un modello XML Schema è a tutti gli effetti un documento XML e generalmente viene memorizzato in un file di testo con estensione del nome *".xsd"*. La radice di questo documento è data dall'elemento *<schema>*, all'interno del quale sono contenuti tutti i componenti.

Utilizzando la stessa notazione che sarà adottata nel seguito per descrivere i singoli componenti, possiamo dire che una dichiarazione di schema contiene le seguenti proprietà:

---

type definitions: un insieme di definizioni di Simple e Complex Type

attribute declarations: un insieme di dichiarazioni di Attribute globali

element declarations: un insieme di dichiarazioni di Element globali

attribute group definitions: un insieme di definizioni di Attribute Group

model group definitions: un insieme di definizioni di Model Group

notation declarations: un insieme di dichiarazioni di Notation

annotations: un insieme di Annotation

La sintassi della dichiarazione dello schema è la seguente:

```
<schema
  attributeFormDefault = (qualified | unqualified) :  unqualified
  elementFormDefault = (qualified | unqualified) :  unqualified
  blockDefault = (#all | List of (extension | restriction |
substitution)) :  ''
  finalDefault = (#all | List of (extension | restriction |
list | union)) :  ''
  version = token
  xml:lang = language
  targetNamespace = anyURI
  (xmlns[:prefix] = anyURI)* >
```

```
  Content:  ( ((include | import | redefine | annotation)*,
(((simpleType | complexType | group | attributeGroup)
| element | attribute | notation), annotation*)*))
</schema>
```

Gli attributi *attributeFormDefault* e *elementFormDefault* servono a specificare se, nell'istanza, i nomi rispettivamente degli attributi e degli elementi devono essere qualificati o meno. Queste dichiarazioni hanno entrambe valore di default *unqualified* e si applicano a tutte gli attributi o elementi dello schema. E' possibile anche specificare queste proprietà singolarmente per ogni Element o Attribute Declaration, tramite l'attributo *form*, come si vedrà nel seguito. Nell'esempio seguente viene definito uno schema in cui si richiede che gli elementi siano qualificati.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:po="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

  <element name="purchaseOrder" type="po:PurchaseOrderType"/>
  <element name="comment"       type="string"/>

  <complexType name="PurchaseOrderType">
    <!-- etc. -->
  </complexType>
</schema>
```

Un'istanza valida di questo schema potrebbe essere il seguente documento XML, nel quale si vede l'elemento *purchaseOrder*, ad esempio, istanziato con nome qualificato dal prefisso "apo".

```
<?xml version="1.0"?>
<apo:purchaseOrder xmlns:apo="http://www.example.com/PO1"
                   orderDate="1999-10-20">
  <apo:shipTo country="US">
    <apo:name>Alice Smith</apo:name>
    <apo:street>123 Maple Street</apo:street>
    <!-- etc. -->
  </apo:shipTo>
  <apo:billTo country="US">
    <apo:name>Robert Smith</apo:name>
    <apo:street>8 Oak Avenue</apo:street>
    <!-- etc. -->
  </apo:billTo>
  <apo:comment>Hurry, my lawn is going wild<!/apo:comment>
  <!-- etc. -->
</apo:purchaseOrder>
```

Gli attributi *blockDefault* e *finalDefault* servono a specificare i valori di default per lo schema per tutti i componenti cui sono applicabili, rispettivamente, gli attributi *block* e *final*, i quali saranno descritti nel seguito per i singoli componenti. Se non viene specificato niente, questi due attributi sono liste vuote ed indicano nessun vincolo di block o final di default.

Per quanto riguarda gli attributi *targetNamespace* e *xmlns* si veda il paragrafo 1.1.2.

Nel *content* della definizione di schema sono presenti tutti i componenti precedentemente indicati nelle proprietà. Inoltre si possono trovare altri tre tipi di elementi, *include*, *import* e *redefine*, che saranno illustrati nel paragrafo 1.1.3.

1.1.1 tipi predefiniti in XML Schema

Il linguaggio XML Schema definisce un certo numero di tipi di dato predefiniti che possono essere usati come valore per l'attributo *type* in tutte le dichiarazioni di elementi o attributi, oppure come tipo base da cui derivare nuovi tipi. I tipi predefiniti, detti anche *built-in Simple Type*, sono considerati dei *Simple Type*. Nelle tabelle 1.1 e 1.2 è riportato l'elenco dei tipi predefiniti assieme ad alcuni esempi. Questi tipi saranno utilizzati nel seguito per descrivere la sintassi dei costrutti di XML Schema.

1.1.2 Namespace in XML Schema

Una caratteristica fondamentale del linguaggio XML è la possibilità di definire tag personalizzati. Questo meccanismo offre molta libertà al programmatore; tuttavia al crescere della complessità e del numero di applicazioni sorge il problema dei conflitti nei nomi dei tag. Per assicurare l'associazione univoca tra nome e definizione, sono stati introdotti i namespace XML. Questo meccanismo, già previsto in altri linguaggi di programmazione, permette di evitare le collisioni tra i nomi delle definizioni racchiudendo le stesse in domini, detti appunto *namespace*, ciascuno identificato da un URI(Uniform Resource Identifier) univoco. La tecnica dei namespace consente inoltre di poter riutilizzare le definizioni presenti in un dominio all'interno di diversi documenti XML.

Tipo XML Schema	Esempio
byte	-128,...,-1,0,1,...,127
unsigedByte	0,1,...,255
anyType	
anySimpleType	
boolean	true,false
base64Binary	GpM7
hexBinary	0FB7
string	Questa è una stringa
normalizedString	Questa è una stringa
token	Questa è una stringa
Name	shipTo
QName	po:USAddress
NCName	USAddress
anyURI	http://www.w3.org/2001/XMLSchema
language	en-GB, en-US, fr
NMTOKEN	US, Brasil
NMTOKENS	"US UK", "Brasil Canada Mexico"
duration	P1Y2M3DT10H30M12.3S
dateTime	1999-05-31T13:20:00.000-05:00
date	1999-05-31
time	13:20:00.000-05:00
gYear	2005
gYearMonth	2005-01
gMonth	01
gMonthDay	01-16
gDay	16

Tabella 1.1: Tipi di dato predefiniti di XML Schema

Tipo XML Schema	Esempio
int	-2147483648,... ,2147483648
integer	...,-1,0,1,...
positiveInteger	1,2,...
nonNegativeInteger	0,1,2,...
negativeInteger	...-2,-1
nonPositiveInteger	...-2,-1,0
long	-922337203685477,...,-1,0 1,922337203685477
short	-32768,...,-1,0,1,...,32768
unsigedInt	0,1,...4294967295
unsigedLong	0,...18446744073709551615
unsigedShort	0,1,...65535
decimal	-1.23, 0, 123.4, 1000.00
float	-1E4, 12.78e-2, 12
double	1267.43233E12, -1234.46e-14

Tabella 1.2: Tipi di dato predefiniti in XML Schema

Il linguaggio XML Schema fa riferimento ai namespace XML attraverso due attributi dell'elemento schema, che sono denominati *targetNamespace* e *xmlns*. Uno schema può contenere una sola dichiarazione di *targetNamespace*; di seguito se ne riporta un esempio:

```
<schema targetNamespace = "http://www.dbgroup.unimo.it/provaXsd">
  <!-- ... -->
</schema>
```

Questo attributo dell'elemento <schema> deve essere di tipo anyURI, il tipo predefinito di XML Schema che rappresenta un generico URI (Uniform Resource Identifier); i valori che può assumere sono stringhe del tipo "http://www.dbgroup.unimo.it/provaXsd".

Il valore di *targetNamespace* indica il namespace che deve essere popolato con le definizioni contenute nello schema stesso. Nel seguito, per molti componenti sarà indicata una proprietà *target namespace*: di fatto il valore di questa proprietà viene definita a livello di schema tramite il suddetto attributo targetNamespace ed è quindi uguale per tutti gli elementi dichiarati al suo interno. Queste definizioni possono poi essere importate in altri schemi XSD secondo le modalità descritte al paragrafo 1.1.3, oppure utilizzati in un documento XML istanza dello schema stesso. Il W3C stablisce che L'associazione tra schema e istanza avvenga appunto tramite i namespace: da un lato, nello schema viene dichiarato un namespace target in cui sono inseriti i nomi dei tipi definiti e degli elementi e attributi dichiarati; dall'altro lato, per utilizzare i tag personalizzati in un documento XML, è necessario dichiarare l'URI del namespace utilizzato, come nel seguente esempio:

```
<?xml version"1.0" ?>
<transaction borrowDate="2006-01-18"
    xmlns = "http://www.dbgroup.unimo.it/provaXsd" />
  <!-- ... -->
</transaction>
```

Ovviamente lo schema corrispondente al namespace "http://www.dbgroup.unimo.it/provaXsd" dovrà dichiarare che la radice del documento istanza deve essere un elemento di nome "transaction" e dovrà spcificarne la struttura. E' compito del parser XML verificare che l'istanza rispetti tutti i vincoli, le dichiarazioni e le definizioni dello schema.

Si noti come non sia necessario che l'URI identifichi una risorsa fisica esistente su qualche server; al contrario, nel caso dei namespace XML, quasi sempre tale risorsa non esiste. E' sempre demandata al parser xml la risoluzione dell'URI del namespace usato nel documento XML e nello schema corrispondente.

L'elemento <schema> può contenere un numero qualsiasi di attributi *xmlns*, la cui sintassi è la seguente:

```
xmlns[:prefix] = anyURI
```

Il valore a destra delle dichiarazione deve identificare un namespace XML, ed è perciò un URI. A sinistra, oltre la parola chiave xmlns, si può trovare un prefisso, che se presente, viene associato all'URI e viene usato nel corpo dello schema per indicare a quale namespace apparetengono gli elementi usati. Ad esempio, dichiarando:

```
<schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:ipo="http://www.example.com/IPO"
        xmlns="http://www.dbgroup.unimo.it/provaXsd">
    <!-- ... -->
</schema>
```

si associa al namespace "http://www.example.com/IPO" il prefisso *ipo*: ogni volta che,nello schema ci si vuole riferire ad un elemento definito in questo namespace sarà sufficiente premettere al nome dell'elemento il prefisso. I nomi dotati del prefisso, come *ipo:Address*, sono detti *qualificati* e sono rappresentati dal tipo predefinito QName. I nomi senza prefisso sono detti invece *non qualificati* e sono rappresentati dal tipo predefinito NCName. Se si usa l'attributo xmlns senza alcun prefisso, allora si definisce un namespace di *default*; come è facile intuire, è possibile avere, per ogni schema, al massimo un namespace di default. Al namespace di default sono automaticamente associati tutti gli elementi presenti all'interno del documento i cui nomi non sono qualificati.

Un namespace particolare è associato all'URI "http://www.w3.org/2001/XMLSchema" ed è il target namespace dello schema che contiene le specifiche dei componenti di XML Schema. Questo schema è contenuto nella documentazione di XSD (vedi [5]) ed è chiamato anche "schema for schema". I parser e i tool XML Schema contengono una copia di questo schema che automaticamente associano all'URI suddetto. Solitamente, tutti gli schemi contengono un attributo *xmlns* per questo namespace; ad esso viene di norma associato il prefisso *xs* o *xsd*.

1.1.3 Schemi composti da più documenti

Abbiamo visto nei paragrafi precedenti la struttura di un documento XML Schema. In realtà è possibile creare un modello XML Schema per un certo namespace unendo tra loro documenti XSD diversi, ciascuno dei quali è costituito da un elemento <schema>, riutilizzando così insiemi di definizioni già create per altri schemi. A livello pratico, questo viene realizzato tramite le direttive *include*, *import* e *redefine* che devono essere specificate all'interno dell'elemento <schema>, come ad esempio:

```
<schema targetNamespace="http://www.example.com/IPO"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:ipo="http://www.example.com/IPO">
  ...
  <!-- include address constructs -->
  <include schemaLocation="address.xsd"/>
  ...
</schema>
```

La sintassi di queste direttive è la seguente:

```
<include
   schemaLocation = anyURI >
   Content:  (annotation?)
</include>

<redefine
   schemaLocation = anyURI >
   Content:  ((annotation | (simpleType |
complexType | group | attributeGroup))*)
</redefine>

<import
   schemaLocation = anyURI
   namespace = anyURI >
   Content:  (annotation?)
</import>
```

La direttiva *include* consente di includere, nello schema in cui è dichiarata,uno schema esterno. L'attributo obbligatorio *schemaLocation* indica al parser il percorso della risorsa che contiene lo schema incluso. L'inclusione è valida solo se lo schema incluso dichiara lo stesso target namespace dello schema che lo include: se questo è rispettato, tutte

le dichiarazioni e le definizioni dello schema incluso sono inserite in quello principale, e possono essere considerate come se fossero effettivamente specificate dentro quest'ultimo.

La direttiva *redefine* viene trattata alla stessa maniera della include ed ha, perciò, l'effetto di includere uno schema all'interno di un altro. Anch'essa è valida solo se i due schemi dichiarano lo stesso target namespace; a differenza della include, consente anche di ridefinire alcune delle definizioni e dichiarazioni incluse, specificandole nel corpo della direttiva stessa. Più precisamente si possono ridefinire tipi (semplici e complessi), gruppi di attributi e gruppi di modelli. Per farlo è necessario che la nuova definizione abbia lo stesso nome di quella ridefinita e che sia specificata nel corpo della redefine. Nell'esempio seguente si ridefinisce il tipo Address, che è già specificato nello schema address.xsd. Il nuovo tipo Address viene poi inserito nel target namespace dello schema e può essere utilizzato per definire elementi o derivare altri tipi. La redefine ha anche l'effetto di inserire nello schema in cui è dichiarata tutte le definizioni e le dichiarazioni dello schema incluso che non sono ridefinite,ed in questo si comporta esattamente come la include.

```
<schema targetNamespace="http://www.example.com/IPO"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:ipo="http://www.example.com/IPO">

  <!-- redefine -->
  <!-- ridefinizione del tipo Address contenuto
       nello schema address.xsd -->
  <xs:redefine schemaLocation="address.xsd">
    <xs:complexType name="Address">
      <xs:complexContent>
        <xs:extension base="Address">
```

```
          <!-- ...  -->
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:redefine>
</schema>
```

Si noti che solo all'interno dell'elemento <redefine> è possibile definire una struttura con lo stesso nome e lo stesso target namespace di una già esistente, in tutti gli altri casi questo è un errore. A livello pratico, la ridefinizione ha lo stesso meccanismo della derivazione (che sarà illustrato nel seguito per i componenti cui è applicabile): la differenza sta nel fatto che l'oggetto derivato di fatto sostituisce nello schema quello base in quanto i due componenti hanno lo stesso nome e appartengono allo stesso namespace.

La direttiva *import*, infine, serve per identificare namespace diversi dal target namespace dello schema in cui si trova, che sono poi utilizzati per effettuare riferimenti esterni,ad esempio mediante i nomi qualificati. Sono previsti due attributi,*namespace* e *schemaLocation*, i cui valori sono, rispettivamente, l'URI del namespace importato e l'URI del documento XML Schema corrispondente. Entrambi sono facoltativi, per cui è lasciato all'applicazione il compito di risolverli nel caso non siano specificati.

1.2 Descrizione dei componenti primari

In questo paragrafo verranno descritti i componenti primari di XML Schema elencati in precedenza. I componenti primari sono gli elementi di base del linguaggio e come tali forniscono le funzionalità essenziali senza le quali non sarebbe possibile costruire nessuno schema. Tra i componenti primari troviamo:

- Attribute Declaration
- Element Declaration
- Simple Type Definition
- Complex Type Definition

I primi due devono essere identificati da un nome; i rimanenti due invece possono anche essere privi di nome, cioè dichiarati in maniera anonima.

1.2.1 Attribute Declaration

La dichiarazione di un componente *Attribute* è caratterizzata dalle seguenti proprietà:

---

name: è il nome dell'attributo, inteso come un nome XML non qualificato

target namespace: è una proprietà facoltativa; se è presente, assieme al nome deve identificare univocamente l'attributo e consente la validazione di documenti XML con attributi qualificati

type definition: il tipo dell'attributo, può essere solo una definizione di Simple Type

scope: può essere globale o una definizione di Complex Type: un attribute infatti può essere dichiarato dentro una definizione di Complex Type, in questo caso risulta visibile solo localmente, nell'elemento in cui è dichiarato, oppure direttamente all'interno dell'elemento <schema> (cioè la radice del modello XML Schema): in questo caso si parla di attribute globale ed è possibile riferirsi ad esso in tutti i Complex Type che appartengono allo stesso schema

value constraint: facoltativo, è una coppia formata da un valore per l'attributo ed un identificatore scelto tra fixed e default

annotation: è un componente facoltativo di tipo Annotation

La sintassi della dichiarazione del componente Attribute è la seguente:

```
<attribute
  name = NCName
  ref = QName
  type = QName
  form = (qualified | unqualified)
  use = (optional | prohibited | required) :  optional
  fixed = string
  default = string>
```

```
  Content:  ((annotation?  | simpleType?))
</attribute>
```

L'attributo *ref* viene utilizzato all'interno di una definizione di Complex Type per fare riferimento ad un Attribute definito globalmente con un *name* uguale al valore assegnato a ref; in questo modo è possibile riutilizzare la stessa dichiarazione includendola in diversi tipi complessi. Gli attributi ref e name non possono essere presenti contemporaneamente, ma almeno uno dei due deve essere specificato; gli attribute definiti globalmente, inoltre, possono solo avere l'attributo *name*, non possono perciò fare riferimento ad altri attribute globali.

L'attributo *type* è il nome (qualificato) di un Simple Type che deve essere definito e visibile nello schema in cui è dichiarato l'Attribute. Questo attributo non è presente nel caso in cui ci sia una dichiarazione di tipo anonimo, cioè quando la definizione del Simple Type è nel contenuto dell'Attribute (e quindi visibile solo ad esso).

L'attributo *use* indica se l'uso dell'Attribute dichiarato è facoltativo (optional, è il default), obbligatorio (required), o vietato (prohibited).

L'attributo *form* indica se nell'istanza il nome di questo attributo deve essere qualificato o non qualificato. Questo attributo è opzionale, se non è specificato si applica il valore dell'attributo *attributeFormDefault* specificato nell'elemento schema (vedi pag. 13).

Gli attributi *fixed* e *default*, indicano come già detto, un vincolo, facoltativo, sul valore dell'Attribute dichiarato. Non possono essere presenti entrambi; inoltre se è presente l'attributo default, deve essere *use* = *optional*. L'attributo default specifica il valore che viene applicato all'Attribute quando quest'ultimo non è presente; l'attributo fixed

specifica il valore che l'Attribute deve obbligatoriamente avere se presente, e si comporta come il defualt se l'Attribute non è presente.

Per quanto riguarda il *content* del componente, all'interno di una dichiarazione di Attribute possiamo trovare un componente Annotation, facoltativo, o un componente Simple Type, anch'esso facoltativo, nel caso in cui sia presente una dichiarazione di tipo anonimo.

Segue uno schema di esempio con tre dichiarazioni di Attribute: uno globale, uno locale e uno riferito.

```
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.dbgroup.unimo.it/prova"
    targetNamespace="http://www.dbgroup.unimo.it/prova">

<!-- esempio di global attribute con definizione di
     tipo anonimo -->
    <xs:attribute name="partNum">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:pattern value="\d{3}-[A-Z]{2}"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>

    <xs:complexType name="PurchaseOrderType">
        <!-- ... -->
```

```
<!-- esempio di attribute dichiarato in maniera locale -->
        <xs:attribute name="orderDate" type="xs:date"/>
<!-- esempio di attribute dichiarato come riferimento ad
     un attributo globale e con utilizzo required -->
        <xs:attribute ref="partNum" use="required"/>
    </xs:complexType>
</xs:schema>
```

1.2.2 Element Declaration

La dichiarazione del componente Element è caratterizzata dalle seguenti proprietà:

---

name: è il nome dell'elemento, inteso come un nome XML non qualificato

target namespace: è una proprietà facoltativa; se è presente, assieme al nome, identifica univocamente l'elemento e consente la validazione di documenti XML con elementi qualificati

type definition: il tipo dell'elemento, può essere una definizione di Simple Type o di Complex Type

scope: può essere globale oppure una definizione di Complex Type: un Element può essere infatti dichiarato dentro una definizione di Complex Type o di Model Group (in questo caso risulta visibile solo localmente, nell'elemento in cui è inserito) oppure può essere inserito direttamente nell'elemento <schema> (la radice del modello XML Schema); in quest'ultimo caso si parla di elemento globale ed è possibile riferirsi ad esso in tutti i componenti appartenenti allo stesso schema

value constraint: facoltativo, come nel caso di Attribute consiste in una coppia formata da un valore ed una tra le parole chiave *fixed* e *default*

nillable: facoltativo, un valore booleano che indica se l'elemento può assumere il valore *nil* oppure no

identity-constraint definitions: facoltativa, un insieme di oggetti *Identity-constraint Definition*

substitution group affiliation: facoltativo,è una definizione di element globale

substitution group exclusions: un sottoinsieme di (*substitution*, *extension*); se è vuoto, indica che l'elemento dichiarato può appartenere al gruppo di sostituzione di un qualsiasi elemento dello stesso tipo o di un tipo derivato da esso; altrimenti specifica quale tipo di derivazione è esclusa

disallowed substitution: un sottoinsieme di (*substitution*, *extension*, *restriction*); similmente alla precedente, specifica se l'elemento può o meno essere sostituito nell'istanza da un altro elemento di tipo derivato per restrizione o estensione, o da un elemento appartenente al gruppo di sostituzione

abstract: un valore booleano, verrà spiegato nel seguito a proposito dei gruppi di sostituzione (pag. 33)

annotation: è un componente facoltativo di tipo Annotation

La sintassi della dichiarazione del componente Element è la seguente:

```
<element
   name = NCName
   ref = QName
   type = QName
   fixed = string
   default = string
   minOccurs = nonNegativeInteger :  1
   maxOccurs = (nonNegativeInteger | unbounded) :  1
   nillable = boolean :  false
   form = (qualified | unqualified)
   block = (#all | (List of (extension | restriction |
substitution)))
   final = (#all | (List of (extension | restriction)))
   abstract = boolean :  false
   substitutionGroup = QName >
   Content:  ((annotation?, ((simpleType | complexType)?,
(unique | key | keyref)*))
</element>
```

Come si vede, una Element Declaration ha in comune alcuni campi con il componente Attribute Declaration; vediamo ora il significato di questi e degli altri campi.

L'attributo *ref* viene utilizzato all'interno di una definizione di Complex Type per fare riferimento ad un Element definito globalmente con un *name* uguale al valore assegnato a ref; in questo modo è possibile riutilizzare la stessa dichiarazione includendola in diversi tipi complessi. Gli attributi ref e name non possono essere presenti contemporaneamente, ma almeno uno dei due deve essere specificato; gli elementi definiti globalmente, inoltre, possono solo avere l'attributo *name*, non possono perciò fare riferimento ad altri element globali.

L'attributo *type* è un nome (qualificato) che può identificare una definizione di Simple Type o di Complex Type, e deve essere risolvibile nello schema in cui è dichiarato l'elemento. Anche in questo caso è possibile definire il tipo (Complex o Simple) in modo anonimo, nel contenuto della dichiarazione stessa; in questo caso l'attributo type non viene specificato.

Gli attributi *fixed* e *default*, indicano come già detto, un vincolo, facoltativo, sul valore dell'elemento dichiarato. Il loro significato è uguale a quello descritto per la dichiarazione di Attribute. L'unica differenza consiste nel fatto che il valore di default si applica solo nel caso in cui il valore non sia specificato; se invece l'elemento è omesso, non viene inserito.

Gli attributi *minOccurs* e *maxOccurs* servono per controllare la molteplicità della dichiarazione. A questo proposito si possono verificare diverse situazioni:

- quando minOccurs è impostato a 0, l'elemento è opzionale;
- quando maxOccurs è impostato a unbounded, l'elemento può comparire un numero arbitrario di volte;

- in generale, l'elemento può essere ripetuto un numero di volte compreso tra minOccurs e maxOccurs, estremi inclusi.

Come impostazione di default, la molteplicità è impostata a (1,1), cioè l'elemento deve comparire una ed una sola volta.

L'attribute booleano *nillable* indica se, nell'istanza, l'elemento può avere l'attributo *nil* settato a true oppure no. Questo attributo viene definito come parte del namespace XML Schema per le istanze (rappresentato dall'URI http://www.w3.org/2001/XMLSchema-istances e, convenzionalmente, dal prefisso xsi); se è settato a true indica che l'elemento ha contenuto nullo, cioè non può contenere un valore o dei sottoelementi (può però avere degli attributi). L'attributo nil è applicabile solo agli elementi, non agli attributi.

L'attributo *form* indica se nell'istanza il nome di questo elemento deve essere qualificato o non qualificato. Questo attributo è opzionale; se non è specificato si applica il valore dell'attributo *elementFormDefault* specificato nell'elemento schema (vedi pag. 13).

Gli attributi *block* e *final* determinano i valorei delle suddetta proprietà "substitution group exclusions" e "disallowed substitution" rispettivamente. Se questi due attributi non sono specificati, vengono utilizzati gli attributi *blockDefault* e *finalDefault* dell'elemento schema.

Gli attributi *abstract* e *substitutionGroup* si riferiscono ad un costrutto avanzato di XML Schema chiamato *gruppo di sostituzione*. Tramite un gruppo di sostituzione è possibile indicare che un certo insieme di elementi globali possano essere utilizzati, nell'istanza, al posto di uno specifico elemento globale (che viene detto head element del gruppo) senza che la validazione fallisca. In particolare, l'attributo *substitutionGroup* indica a quale gruppo di sostituzione l'elemento dichiarato appartiene; esso deve avere come valore il nome qualificato dell'elemento di testa del gruppo. L'attributo *abstract*,

invece, può essere specificato in un head element per impedire che questo venga utilizzato, forzandone perciò la sostituzione.

Ecco un esempio di definizione di un gruppo di sostituzione:

```
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.dbgroup.unimo.it/prova"
    targetNamespace="http://www.dbgroup.unimo.it/prova">
    <!-- questo è un elemento globale dichiarato abstract
            non può essere utilizzato nel documento xml-->
    <xs:element name="comment" type="xs:string" abstract="true"/>

    <!-- questi sono due elementi globali appartenenti al gruppo
         di sostituzione di comment -->
    <xs:element name="myComment" type="xs:string"
                substitutionGroup="comment"/>
    <xs:element name="anotherComment" type="xs:string"
                substitutionGroup="comment"/>

    <xs:complexType name="ordine">
        <xs:sequence>
     <!-- all'interno di un elemento di tipo "ordine" devo trovare
          un elemento di nome comment e tipo string: potrò trovare
          al suo posto un elemento di nome myComment o di nome
          anotherComment, ma non comment perchè è stato dichiarato
          abstract -->
```

```
            <xs:element ref="comment"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Per quanto riguarda il *content*, all'iterno di una dichiarazione di Element possiamo trovare:

- un componente Annotation, facoltativo;
- una definizione di Simple Type o Complex Type, nel caso in cui sia presente una dichiarazione di tipo anonimo;
- zero, uno o più definizioni di Identity-constraint, di tipo *key*, *keyref* o *unique*.

1.2.3 Complex Type Definition

I componenti di tipo *Complex Type Definition* permettono di definire la struttura di un elemento complesso, stabilendo quali elementi e attributi deve avere nell'istanza per poter esser validato. Sicuramente questo è il componente più complesso del linguaggio XML Schema, soprattutto perchè la sua sintassi, come si vedrà, prevede delle opzionalità e quindi contempla di fatto diverse categorie di Complex Type.

Un componente Complex Type Definition è caratterizzato dalle seguenti proprietà:

---

name: è il nome del tipo definito, inteso come un nome XML non qualificato; è opzionale, e non è presente nel caso di definizione di un tipo anonimo all'interno di un oggetto *Element Declaration* (vedi paragrafo 1.2.2)

target namespace: è una proprietà facoltativa; se è presente, assieme al nome, identifica univocamente il Complex Type e consente la validazione di documenti XML con elementi qualificati

base type definition: un oggetto *Simple Type Definition* o *Complex Type Definition*, che rappresenta il tipo base da cui è derivato il nuovo Complex Type

derivation method: può essere *extension* o *restriction*

final: un sottoinsieme di (*extension*,*restriction*), indica i tipi di derivazione rispetto ai quali il Complex Type è finale, cioè non può essere usato come tipo base per un altro Complex Type

abstract: un valore booleano; se è true, il Complex Type non può essere usato come tipo per la validazione di un elemento, ma solo come tipo base per definire nuovi elementi

attribute uses: un insieme di oggetti *Attribute Use*, rappresentano gli attributi dell'elemento di cui il tipo complesso definisce la struttura

attribute wildcard: un oggetto Wildcard di tipo anyAttribute

content type: può essere *empty*, una *Simple Type Definition*, oppure una coppia formata da un oggetto *Particle*, che in questo caso viene detto anche *content model*, ed uno tra *mixed* ed *element-only*; questa proprietà determina la struttura interna dell'elemento di cui il tipo complesso definisce la struttura, e quindi la validazione degli elementi figli

prohibited substitution: un sottoinsieme di (*extension*,*restriction*); se specificato indica i tipi di derivazione per cui è proibita la sostituzione, in fase di validazione, con un tipo derivato dal presente Complex Type

annotations: un insieme di oggetti *Annotation*

La sintassi di un Complex Type è piuttosto complessa, soprattutto per quanto riguarda il *content*, cioè gli elementi interni alla definizione stessa. Nel seguito sarà riportata dapprima la sintassi generale, come viene fatto per gli altri componenti, e successivamente si distingueranno i casi particolari.

Un componente *Complex Type Definition* deve avere la seguente struttura:

```
<complexType
   name = NCName
   abstract = boolean :  false
   block = (#all | (List of (extension | restriction)))
   final = (#all | (List of (extension | restriction)))
   mixed = boolean :  false >
   Content:  ((annotation?, (simpleContent | complexContent |
   ((group | all | choice | sequence)?,
   ((attribute | attributeGroup)*,anyAttribute?)))))
</complexType>
```

L'attributo *name* indica il nome, non qualificato, del tipo. Questo nome può essere usato per specificare il tipo nella dichiarazione di un elemento. L'attributo *name* è specificato solo nei Complex Type definiti come top-level (cioè all'interno dell'elemento

<schema>), mentre non viene utilizzato quando la definizione si trova all'interno di un oggetto Element Declaration: in questo caso la definizione si dice *anonima*.

L'attributo *abstract* rappresenta la suddetta proprietà "abstract", ed il suo valore di default è *false*.

Gli attributi *block* e *final*, rispettivamente, rappresentano le proprietà "prohibited substitutions" e "final". Se non sono specificati, si applicano i valori degli attributi *blockDefault* e *finalDefault* dell'elemento <schema> (vedi pag. 14).

L'attributo *mixed* indica se il contenuto del Complex Type è di tipo *mixed* (valore true) o *element-only* (valore false, è il default). Questo attributo ha significato solo se il contenuto è di tipo *complexContent*. Se il contenuto è element-only, allora sarà validato un elemento che contenga al suo interno solo altri elementi. Se, al contrario, è specificato un contenuto mixed, allora sarà validato anche un istanza che contenga i sottoelementi inseriti in un testo, come ad esempio:

```
<letterBody>
<salutation>Dear Mr.<name>Robert Smith</name>.</salutation>
Your order of <quantity>1</quantity> <productName>Baby
Monitor</productName> shipped from our warehouse on
<shipDate>1999-05-21</shipDate>.
....
</letterBody>
```

Per quanto riguarda il *content* di una definizione di Complex Type dobbiamo distinguere i casi in cui il contenuto è di tipo complexContent o simpleContent. In entrambi i casi possiamo trovare una Annotation facoltativa. E' possibile anche definire il contenuto senza specificare nessuno dei costruttori complexContent o simpleContent: in

questo caso tutto il contenuto è trattato come se fosse definito all'interno di un elemento <complexContent>.

1.2.3.1 Complex Type con contenuto semplice

Un Complex Type con contenuto semplice è utilizzato quando si vuole specificare che un elemento dell'istanza deve avere valore semplice e può avere degli attributi, come ad esempio:

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

Come si vede, l'elemento `<internationalPrice>` deve avere un valore di tipo decimal, ed un attributo di nome `currency` di tipo string. Se nello schema ci fosse la dichiarazione:

```
<xs:element name="internationalPrice" type="xs:decimal"/>
```

non si potrebbe avere l'attributo `currency` perchè, come si vedrà nel paragrafo 1.2.4, i Simple Type, non possono avere nè sottoelementi nè attributi. Per descrivere la struttura dell'elemento `<internationalPrice>` si dovrà usare la seguente espressione:

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

nella quale si dichiara un Complex Type a contenuto semplice derivante per estensione dal tipo semplice decimal, al quale aggiunge l'attributo currency di tipo string.

In generale la sintassi del *simpleContent* è la seguente:

```
<simpleContent>
  Content:  ((annotation?, (restriction | extension)))
</simpleContent>
```

All'interno di esso si può trovare una Annotation facoltativa e un elemento *restriction* o un elemento *extension*.

La sintassi di *extension* è:

```
<extension
  base = QName >
  Content:  ((annotation?, ((attribute | attributeGroup)*, anyAttribute?)))
</extension>
```

L'unico attributo è *base*, un nome qualificato che identifica un Simple Type o un Complex Type a contenuto semplice. Nel *content* dell'elemento <extension> si possono trovare una Annotation opzionale, un insieme di oggetti di tipo Attribute o Attribute Group (riferimenti ad Attribute Group globali), ed eventualmente una Wildcard di tipo anyAttribute. Per la descrizione di questi componenti si vedano i paragrafi successivi.

La sintassi di *restriction* è la seguente:

```
<restriction
  base = QName >
  Content: ((annotation?, (simpleType?, (minExclusive | minInclusive |
maxExclusive | maxInclusive | totalDigits | fractionDigits | length |
minLength | maxLength | enumeration | whiteSpace | pattern)*)?, ((attribute
| attributeGroup)*, anyAttribute?)))
</restriction>
```

L'unico attributo è *base*, un nome qualificato che identifica un Complex Type con contenuto semplice. Il *content* della dichiarazione aggiunge, ai componenti già visti nel caso dell'elemento <extension>, una definizione di Simple Type anonimo ed un insieme di *facet*, entrambi opzionali. Le facet, o sfaccettature, permettono di restringere i valori possibili per il contenuto, con il meccanismo descritto in seguito per i Simple Type. La dichiarazione anonima di Simple Type viene utilizzata per definire il tipo del contenuto (e ad essa si applicano le facet), sostituendosi di fatto al tipo di base.

1.2.3.2 Complex Type con contenuto complesso

Quando il contenuto è di tipo *complexContent*, si ha una definizione di Complex Type propriamente detta. La sintassi è la seguente:

```
<complexContent
  mixed = boolean >
  Content:  ((annotation?, (restriction | extension)))
</complexContent>
```

L'attributo *mixed* booleano indica se il contenuto è di tipo *mixed* o *element-only*. Il *content*, oltre ad una Annotation facoltativa, prevede un elemento *restriction* o *extension*.

La sintassi di *restriction* è:

```
<restriction
  base = QName >
  Content:  ((annotation?, (group | all | choice | sequence)?, ((attribute
| attributeGroup)*, anyAttribute?)))
</restriction>
```

La sintassi di *extension* è invece:

```
<extension
  base = QName >
  Content:  ((annotation?, ((group | all | choice | sequence)?, ((attribute
| attributeGroup)*, anyAttribute?))))
</extension>
```

In entrambi i casi si ha che:

- l'attributo *base* identifica un Complex Type dal quale il tipo che si sta definendo è derivato, rispettivamente per estensione o restrizione;
- nel *content* si possono trovare:
  - un numero qualsiasi di oggetti Attribute e Attribute Group, oltre ad una eventuale Wildcard di tipo anyAttribute;
  - un elemento <group>, che faccia riferimento ad una Model Group Definition globale, oppure un oggetto di tipo Model Group (quindi un elemento scelto tra <all>, <choice> o <sequence>)

1.2.3.3 Complex Type con contenuto nullo

Un caso particolare del linguaggio XML Schema sono gli elementi a contenuto nullo, come ad esempio:

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

Questi elementi non hanno nessun contenuto ma possono avere degli attributi. Per specificare la struttura di un elemento senza contenuto si definisce un Complex Type con contenuto complesso facendolo derivare per restrizione da un tipo che non ha elementi nel suo contenuto, come anyType; nel corpo dell'elemento <restriction> poi si specificano solo gli attributi desiderati come nell'esempio seguente:

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

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

1.2.4 Simple Type Definition

Gli oggetti *Simple Type Definition* permettono di creare dei tipi semplici da utilizzare nelle dichiarazioni di elementi o attributi. Questo tipo di componente è caratterizzato dalle seguenti proprietà:

---

name: è un nome non qualificato che identificato il tipo semplice che viene definito; è una proprietà opzionale in quanto è possibile avere dei Simple Type definiti in maniera anonima all'interno di un elemento <attribute> o <element>

target namespace: il nome del namespace XML in cui è inserito il tipo definito; è una proprietà facoltativa, se è presente, assieme al nome deve identificare univocamente il Simple Type

base type definition: è una definizione di Simple Type, dalla quale la presente definizione è derivata

facets: un insieme di *facet* (o sfaccettature) che restringono i valori ammessi per il tipo definito

final: un sottoinsieme di (*extension*,*restriction*,*list*,*union*), indica i tipi di derivazione (restriction o extension) rispetto ai quali il Simple Type è finale, cioè non può essere usato come tipo base, o i tipi di composizione (list o union) nei quali il Simple Type non può essere usato

variety: un valore tra (*atomic*,*list*,*union*); in base al valore scelto vengono definite ulteriori prorietà, in particolare:

atomic in questo caso il tipo definito è atomico, cioè con un singolo valore. Viene definita la proprietà:

primitive type definition: una definizione di Simple Type built-in; ad essa si applicano le sfaccettature.

list in questo caso il tipo definito è una lista, e consente di validare come contenuto una lista di valori separati da spazi. Le sfaccettature si applicano alla lista. Viene definita la proprietà:

item type definition: una definizione di Simple Type che stabilisce il tipo dei singoli elementi della lista

union in questo caso viene definito un tipo che valida una stringa valida per almeno uno dei tipi specificati nella seguente proprietà che viene definita:

member type definitions: è una lista non vuota di definizioni di Simple Type; rappresenta i tipi ammessi per questo tipo semplice.

Come nel caso del tipo list, le facet specificate si applicano al costrutto union (quindi sono ammesse solo quelle possibili per questo costrutto)

annotation: un oggeto Annotation facoltativo

La sintassi di un Simple Type Definition è la seguente:

```
<simpleType
  final = (#all | List of (list | union | restriction))
  name = NCName >
  Content:  ((annotation?, (restriction | list | union)))
</simpleType>
```

L'attributo *final* determina il valore della suddetta proprietà final. L'attributo *name* rappresenta il nome associato al Simple Type definito: non è presente se si sta definendo un tipo semplice anonimo. Il *content* di questo elemento è composto da una Annotation facoltativa e da un elemento scelto tra *restriction*, *list* o *union*.

La sintassi dell'elemento *list* è:

```
<list
  itemType = QName >
  Content:  ((annotation?, simpleType?))
</list>
```

In questo caso viene definito un nuovo Simple Type di tipo lista. L'attributo *itemType*, se presente, è il nome qualificato di un Simple Type e costituisce la proprietà "item type definition" suddetta; se non è presente, la stessa proprietà viene ricavata dalla definizione anonima di tipo semplice presente all'interno dell'elemento <list>. Nel contenuto può

essere presente anche un oggetto Annotation opzionale. Un esempio di tipo lista è il seguente:

```
<xsd:simpleType name="listOfMyIntType">
  <xsd:list itemType="myInteger"/>
</xsd:simpleType>
```

Un elemento valido in una istanza potrebbe essere:

```
<listOfMyInt>20003 15037 95977 95945</listOfMyInt>
```

La sintassi dell'elemento *union* è:

```
<union
  memberTypes = List of QName >
  Content:  ((annotation?, simpleType*))
</union>
```

Questo elemento serve per definire un nuovo Simple Type in cui la proprietà variety vale *union*. La suddetta proprietà "member type definitions" è definita come unione tra l'insieme di tipi individuato dall'attributo *memberTypes* e i l'insieme di Simple Type definiti in maniera anonima all'interno dell'elemento union stesso. Un esempio di questo componente è il seguente:

```
<xsd:simpleType name="zipUnion">
  <xsd:union memberTypes="USState listOfMyIntType"/>
</xsd:simpleType>
```

La sintassi dell'elemento *restriction* è la seguente:

```
<restriction
  base = QName >
  Content: ((annotation?, (simpleType?, (minExclusive | minInclusive
| maxExclusive | maxInclusive | totalDigits | fractionDigits | length |
minLength | maxLength | enumeration | whiteSpace | pattern)*)))
</restriction>
```

Il costrutto *restriction* serve per derivare un Simple Type da un altro Simple Type restringendone l'insieme dei valori tramite un insieme di *facet*. In questo caso la proprietà variety è determinata dalla corrispondente proprietà del tipo base. L'attributo *base* è il nome qualificato del Simple Type dal quale questo tipo è derivato. Se non è presente, il tipo base è dichiarato in maniera anonima nel contenuto della dichiarazione. Il *content* della dichiarazione prevede, oltre alla dichiarazione anonima di Simple Type e ad una Annotation, entrambe facoltative, un certo numero di *facet* tramite le quali viene ristretto l'insieme dei valori ammissibili per questo tipo.

Esistono 12 tipi diversi di sfaccettuature, 6 di esse sono applicabili solo ai tipi ordinati, mentre le altre 6 sono applicabili a tutti i Simple Type. Per una descrizione più completa di questi costrutti si veda [6].

1.2.4.1 Facet applicabili a tipi ordinati

Le sfaccettature applicabili ai soli tipi ordinati, cioè quelli per cui si può definire un ordinamento tra i valori ammissibili, sono le seguenti:

- minExclusive
- minInclusive
- maxExclusive
- maxInclusive
- totalDigits
- fractionDigits

Le prime quattro servono per limitare il range di valori ammessi per il tipo, in particolare:

minExclusive rappresenta il limite inferiore esclusivo (sono validi tutti i valori maggiori di quello specificato);

minInclusive rappresenta il limite inferiore inclusivo (sono validi tutti i valori maggiori o uguali a quello specificato);

maxExclusive rappresenta il limite superiore esclusivo (sono validi tutti i valori minori di quello specificato);

maxInclusive rappresenta il limite superiore inclusivo (sono validi tutti i valori minori o uguali a quello specificato);

Le altre due sfaccettature invece inseriscono un vincolo sul numero di cifre ammesse per rappresentare un valore numerico, per cui si applicano solo ad alcuni tipi ordinati; in particolare totalDigits si riferisce al numero totale di cifre totali, mentre fractionDigits impone un vincolo sul numero di cifre della parte frazionaria.

Un esempio di utilizzo di questi elementi è il seguente codice:

```
<simpleType name='celsiusBodyTemp'>
  <restriction base='decimal'>
    <totalDigits value='4'/>
    <fractionDigits value='1'/>
    <minInclusive value='36.4'/>
    <maxInclusive value='40.5'/>
  </restriction>
</simpleType>
```

1.2.4.2 Facet applicabili a tipi generici

Le seguenti sfaccettature sono applicabili a tipi generici:

- length
- minLength
- maxLength
- enumeration
- whiteSpace
- pattern

La sfaccettature length serve per indicare la lunghezza in caratteri di una stringa. Può essere applicata anche ad un Simple Type di tipo list, ed in questo caso indica il numero di elementi della lista (praticamente trasforma la lista in un array). Le sfaccettature minLength e maxLength sono molto simili a length, ma invece di indicare una lunghezza specifica, permettono di definire un vincolo inferiore o superiore, rispettivamente. (i limiti si intendono in senso inclusivo). Seguono alcuni esempi di utilizzo.

```
<simpleType name='productCode'>
   <restriction base='string'>
     <length value='8' fixed='true'/>
   </restriction>
</simpleType>

<simpleType name='non-empty-string'>
  <restriction base='string'>
    <minLength value='1'/>
  </restriction>
</simpleType>

<simpleType name='form-input'>
  <restriction base='string'>
    <maxLength value='50'/>
  </restriction>
</simpleType>
```

La sfaccettatura pattern permette di definire una espressione regolare che deve essere soddisfatta dalle stringhe del tipo definito.

La sfaccettatura whiteSpace definisce come devono essere trattati gli spazi bianchi, le tabulature, i carriage return ed i line feed, nelle stringhe che rappresentano il valore del tipo.

La sfaccettatura enumeration serve per definire per enumerazione uno spazio di valori per il tipo definito, come nell'esempio seguente:

```
<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>
```

1.3 Descrizione dei componenti secondari

In questa sezione vengono descritti i componenti secondari del linguaggio XML Schema. Questi componenti sono tutti dotati di nome. Essi aumentano la capacità espressiva del linguaggio, aggiungendo nuovi costrutti a quelli primari presentati in precedenza. I componenti secondari sono:

- Attribute Group Declaration
- Model Group Declaration
- Identity-constraint Definition
- Notation Declaration

1.3.1 Attribute Group Definition

Il linguaggio XML Schema dà la possibilità di raggruppare un insieme di *Attribute Declaration* e di associare a questo gruppo un nome. Il gruppo viene inserito nel target namespace dello schema in cui è definito, e può essere utilizzato all'interno di altri componenti i quali possono riferirsi ad esso tramite il suo nome qualificato. Questo meccanismo permette di poter incorporare lo stesso gruppo di attributi in più oggetti dello schema senza doverli ripetere, ma semplicemente indicando il gruppo, come nell'esempio seguente.

```
<xs:attributeGroup name="myAttrGroup">
  <xs:attribute name="attr1" ... />
  <xs:attribute name="attr2" ... />
  ...
</xs:attributeGroup>
```

```
<xs:complexType name="myCompType">
  ...
  <xs:attribute ref="myAttrGroup"/>
</xs:complexType>

<xs:attributeGroup name="myAttrGroup2">
  <xs:attribute ref="myAttrGroup"/>
</xs:attributeGroup>
```

Una definizione di *Attribute Group* è caratterizzata dalle seguenti proprietà:

---

name: il nome che viene associato al gruppo

target namespace: facoltativo, è il nome del namespace in cui il gruppo definito deve essere incluso

attribute uses: un insieme di oggetti Attribute Use

attribute wildcard: una Wildcard di tipo anyAttribute facoltativa

annotation: una Annotation facoltativa

Di seguito viene riportata la sintassi di questo componente:

```
<attributeGroup
  name = NCName
  ref = QName >
  Content: ((annotation?, ((attribute | attributeGroup)*,
anyAttribute?)))
</attributeGroup>
```

L'attributo *name* indica il nome non qualificato del gruppo di attributi che si sta definendo.

L'attributo *ref* indica il nome qualificato del gruppo di attributi a cui fa riferimento la dichiarazione.

I due attributi name e ref non possono comparire assieme, ma uno dei due deve sempre essere specificato. Essi vengono utilizzati in due specifiche occasioni:

- quando un Attribute Group è definito come figlio dell'elemento <schema> o di una direttiva <redefine>, deve essere specificato l'attributo *name*: questo utilizzo corrisponde alla vera e propria definizione di un gruppo;
- quando invece si trova nel corpo di un elemento <complexType> o <attributeGroup>, deve essere specificato l'attributo *ref* con il nome di un gruppo esistente; in questo modo gli attributi dichiarati vengono importati nella definizione dell'elemento padre.

Nel *content* di un Attribute Group possiamo trovare i seguenti elementi:

- un insieme di dichiarazioni di Attribute e di riferimenti ad altri Attribute Group: l'insieme delle dichiarazioni di attributi locali e di quelle importate dagli altri gruppi definisce la suddetta proprietà *attribute uses*;
- una Wildcard di tipo anyAttribute (vedi paragrafo 1.4.4);
- una Annotation facoltativa.

1.3.2 Model Group Definition

Il costrutto Model Group Definition permette di associare ad un gruppo di modello (vedi il paragrafo 1.4.2 per la descrizione dei Model Group) un nome, ed eventualmente una nota, permettendo così di riutilizzare il gruppo in diversi componenti facendo riferimento al nome.

Una definizione di Model Group è caratterizzata dalle seguenti proprietà:

---

name: un nome non qualificato che viene associato al gruppo

target namespace: il target namespace in cui la definizione è inserita; di fatto è il target namespace dello schema in cui il gruppo è definito (se dichiarato)

model group: un componente Model Group, come quello descritto nel paragrafo 1.4.2

annotation: una Annotation facoltativa

La sintassi di questo componente è la seguente:

```
<group
  maxOccurs = (nonNegativeInteger | unbounded) :  1
  minOccurs = nonNegativeInteger :  1
  name = NCName
  ref = QName >
  Content:  (annotation?, (all | choice | sequence)?)
</group>
```

Come nei casi precedenti, gli attributi *name* e *ref* sono esclusivi, per cui uno dei due è sempre presente, ma non compaiono mai insieme. Anche in questo caso, quindi, è necessario distinguere tra i casi in cui è specificato il primo e quelli in cui è presente il secondo.

Se è specificato l'attributo *name*, allora si ha una vera e propria definizione di Model Group. Nel documento XML Schema possiamo trovare questo tipo di costrutto come figlio dell'elemento <schema> o di un elemento <redefine>. In questo caso gli attributi *minOccurs* e *maxOccurs* non sono mai specificati, mentre il contenuto della definizione deve essere un elemento tra <all>, <choice> o <sequence> (cioè un Model Group), più un eventuale oggetto Annotation.

L'attributo *ref* viene usato invece quando si vuole includere un gruppo definito all'interno di un altro componente, come ad esempio un Complex Type o un altro Model Group. In questo caso il contenuto del componente deve essere nullo, mentre si possono

utilizzare gli attributi minOccurs e maxOccurs per specificare la cardinalità del gruppo incluso. In questo caso si ha una Model Group Definition impropria, poichè di fatto è un componente di tipo *Particle* (vedi il paragrafo 1.4.3).

Riportiamo di seguito un esempio di come può essere utilizzato questo componente.

```
<xs:schema ... >
  <xs:group name="myModelGroup">
    <xs:sequence>
      <xs:element ref="someThing"/>
        ...
    </xs:sequence>
  </xs:group>

  <xs:complexType name="trivial">
    <xs:group ref="myModelGroup"/>
    ...
  </xs:complexType>

  <xs:complexType name="moreSo">
    <xs:choice>
      <xs:element ref="anotherThing"/>
      <xs:group ref="myModelGroup"/>
    </xs:choice>
    ...
  </xs:complexType>
</xs:schema>
```

1.3.3 Identity-constraint Definition

Questo tipo di definizione consente di specificare dei vincoli di unicità, chiave e chiave riferita all'interno di uno schema. I seguenti sono alcuni esempi di Identity-constraint Definition.

```
<xs:key name="fullName">
  <xs:selector xpath=".//person"/>
  <xs:field xpath="forename"/>
  <xs:field xpath="surname"/>
</xs:key>

<xs:keyref name="personRef" refer="fullName">
  <xs:selector xpath=".//personPointer"/>
  <xs:field xpath="@first"/>
  <xs:field xpath="@last"/>
</xs:keyref>

<xs:unique name="nearlyID">
  <xs:selector xpath=".//*"/>
  <xs:field xpath="@id"/>
</xs:unique>
```

Una oggetto Identity-constraint Definition è caratterizzato dalle seguenti proprietà:

---

name: un nome non qualificato che identifica l'elemento definito

target namespace: il namespace in cui viene inserita la definizione; è il target namespace dello schema in cui è definito il vincolo

identity-constraint category: il tipo di vincolo definito, cioè *key*, *keyref* o *unique*

selector: una espressione XPath che individua nello schema il set di elementi ai quali il vincolo si applica

fields: una lista non vuota di espressioni XPath che individuano i campi coperti dal vincolo (ad esempio il campo che deve essere unico in un vincolo di unicità)

referenced key: è obbligatorio se il vincolo è di tipo *keyref*, proibito altrimenti; se presente, individua una Identity-constraint Definition di tipo *key* o *unique*

annotation: una Annotation opzionale

Come già detti, gli oggetti Identity-constraint Definition si suddividono in tre categorie: vincoli di unicità (*unique*), chiavi (*key*), e chiavi riferite.

1.3.3.1 Vincoli di unicità

Un vincolo di unicità assicura che, in un insieme di elementi (o di tuple), un certo campo assuma valori unici o nulli.

In XML Schema è possibile esprimere dei vincoli di unicità tramite l'elemento <unique>, la cui sintassi è la seguente:

```
<unique
   name = NCName >
   Content:  ((annotation?, (selector, field+)))
</unique>
```

Il componente ha una struttura molto semplice: presenta infatti solo l'attributo *name*, tramite il quale è possibile specificare un nome da associare al vincolo definito, mentre il contenuto del nodo prevede una Annotation opzionale, un elemento <selector> e uno o più elementi <field>.

L'elemento <selector> ha la seguente struttura:

```
<selector
   xpath = an XPath expression >
   Content:  ((annotation?))
</selector>
```

Questo elemento comprende, oltre ad un eventuale sottoelemento di tipo Annotation, anche un attributo di nome *xpath* il cui valore deve essere una espressione XPath. Il valore dell'attributo xpath di <selector> individua gli elementi dello schema a cui è applicato il vincolo di unicità.

Un elemento <field> ha la stessa sintassi di quello precedente, e cioè:

```
<field
   xpath = an XPath expression >
   Content:  ((annotation?))
</field>
```

Valgono le stesse spiegazioni date per l'elemento <selector>, ma questa volta il valore dell'attributo *xpath* individua, nell'insieme di elementi selezionato in precedenza, qual è il campo coinvolto dal vincolo di unicità. E' possibile anche specificare un vincolo di unicità che coinvolga più di un campo, in quanto l'elemento <unique> può contenere uno o più oggetti field.

1.3.3.2 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 (cioè un valore è sempre specificato). In XML Schema questo è reso possibile dall'elemento <key>, la cui sintassi è uguale a quella di <unique>, e quindi è:

```
<key
   name = NCName >
   Content:  ((annotation?, (selector, field+)))
</key>
```

Dal momento che ad ogni dichiarazione di *key* viene associato un nome, è possibile riferirsi ad essa nello schema e quindi anche creare delle chiavi riferite, in modo simile a quanto viene fatto nel modello relazionale con i vincoli di foreign key. Questo viene fatto tramite il costrutto *keyref*, la cui sintassi è:

```
<keyref
   name = NCName
   refer = QName >
   Content:  ((annotation?, (selector, field+)))
</keyref>
```

L'attributo *refer* deve appunto far riferimento al nome di un oggetto *key* definito nello stesso schema. Per quanto riguarda gli altri campi, invece, valgono i discorsi fatti in precedenza.

1.3.4 Notation Declaration

Un oggetto di questo tipo è caratterizzato dalle seguenti proprietà:

---

name: un nome non qualificato che viene associato alla dichiarazione

target namespace: il namespace in cui è inserita la definizione; di fatto è il target namespace dello schema in cui l'oggetto è dichiarato

system identifier: un URI (ad esempio il persorso di un eseguibile), opzionale se è specificato il public identifier

public identifier: un identificatore pubblico (come image/jpeg), opzionale se è specificato il system identifier

annotation: una Annotation opzionale

La sintassi di questa dichiarazione è la seguente:

```
<notation
   name = NCName
   public = token
   system = anyURI >
   Content:  ((annotation?))
</notation>
```

Un oggetto di tipo Notation Declaration permette di associare un nome ad un identificatore (pubblico o di sistema). Questo tipo di dichiarazione è legata all'utilizzo del tipo predefinito NOTATION e non interviene nel processo di validazione delle istanze. Per una trattazione più completa si rimanda ai documenti [5] e [4].

Di seguito si riporta un esempio di utilizzo.

```
<xs:notation name="jpeg" public="image/jpeg" system="viewer.exe">
```

1.4 Descrizione dei componenti helper

I componenti helper rappresentano parti di altri componenti. Non hanno un nome e non sono indipendenti dal contesto in cui si trovano; vengono utilizzati solamente nella costruzione di altri elementi complessi. In questa categoria si trovano 5 componenti:

- Annotation
- Model Group
- Particle
- Wildcard
- Attribute Use

1.4.1 Annotation

Questo componente viene utilizzato per inserire nello schema delle note come documentazione per lo schema o per le applicazioni che utilizzano lo schema stesso. Il componente Annotation non partecipa al processo di validazione e quindi non influisce sulla validità di una istanza.

La sintassi di questo componente è la seguente:

```
<annotation>
  Content:  ((appinfo | documentation)*)
</annotation>
```

```
<appinfo source = anyURI >
   Content:  ((any)*)
</appinfo>
<documentation
   source = anyURI
   xml:lang = language >
   Content:  ((any)*)
</documentation>
```

Un elemento <annotation> può contenere al suo interno un numero qualsiasi di sottoelementi <documentation> o <appinfo>: i primi rappresentano delle note fornite come documentazione dello schema per un lettore umano; i secondi invece sono destinati alle applicazioni che elaborano ed utilizzano lo schema. Entrambi questi costrutti possono avere come contenuto un qualsiasi corpo XML ben formato ed hanno un attributo opzionale *source* in cui è possibile specificare un URI di riferimento per aggiungere significato alla dichiarazione. L'elemento documentation, essendo diretto ad un lettore umano, contiene anche un attributo *xml:lang*, opzionale ma fortemente consigliato dal W3C, in cui si può specificare la lingua in cui la nota è scritta (ad esempio en-US o en-UK).

Di seguito viene riportato un esempio di Annotation.

```
<xs:annotation>
  <xs:documentation>
    A type for experts only
  </xs:documentation>
```

```
  <xs:appinfo>
    <fn:specialHandling>checkForPrimes</fn:specialHandling>
  </xs:appinfo>
</xs:annotation>
```

1.4.2 Model group

Un componente di tipo Model Group viene utilizzato per specificare il contenuto di elementi che devono contenere altri sottoelementi. Di seguito sono riportati 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>
```

Le proprietà di questo componente sono:

---

compositor: può essere *all*, *choice* o *sequence*

particles: una lista di oggetti di tipo Particle

annotation: una Annotation opzionale

Un model group è formato da un insieme di oggetti Particle, cioè di elementi, wildcard e altri gruppi. E' possibile specificare la semantica della composizione in tre modi diversi:

- gruppo *all*: significa che il gruppo può contenere solo dichirazioni di elementi (locali o globali) e la cardinalità dei sottoelementi può essere solamente (0,1) o (1,1); in questo caso, nell'istanza gli elementi possono comparire in qualsiasi ordine;
- gruppo *sequence*: in questo caso gli elementi nell'istanza devono comparire nello stesso ordine in cui sono specificati nello schema;
- gruppo *choice*: questo gruppo rappresenta una scelta tra i componenti del gruppo; ciò significa che nell'istanza potrà comparire soltanto uno degli elementi specificati.

Di seguito viene riportata la sintassi dei tre tipi di Model Group.

```
<all
  minOccurs = 1 :  1
  maxOccurs = (0 | 1) :  1 >
  Content:  ((annotation?, element*))
</all>

<choice
  maxOccurs = (nonNegativeInteger | unbounded) :  1
  minOccurs = nonNegativeInteger :  1 >
  Content:  ((annotation?, (element | group | choice | sequence
| any)*))
</choice>

<sequence
  maxOccurs = (nonNegativeInteger | unbounded) :  1
  minOccurs = nonNegativeInteger :  1 >
  Content:  ((annotation?, (element | group | choice | sequence
| any)*))
</sequence>
```

1.4.3 Particle

Il componente Particle contribuisce alla definizione di un modello di contenuto. Le proprietà di questo componente sono:

---

min occurs: un valore intero non negativo

max occurs: un valore intero non negativo oppure la stringa *unbounded*

term: un componente di tipo Model Group, Wildcard o Element Declaration

---

Di seguito si riportano alcuni esempi di utilizzo di questo componente.

```
<xs:element ref="egg" minOccurs="12" maxOccurs="12"/>
```

```
<xs:group ref="omelette" minOccurs="0"/>
```

```
<xs:any maxOccurs="unbounded"/>
```

1.4.4 Wildcard

Un Wildcard è un componente che, se dichiarato all'interno di un gruppo o di un Complex Type, consente di includere un generico oggetto preso da uno specifico namespace, o da un insieme di namespace.

Esistono due tipi di Wildcard: la prima è rappresentata da un elemento <any> e consente di inserire in un componente un qualsiasi elemento XML ben formato; la seconda è rappresentato dall'attributo <anyAttribute> e permette di inserire in un elemento un qualsiasi attributo XML.

La sintassi di questo componente è la seguente:

```
<any
  namespace = ((##any | ## other) | List of (anyURI |
(##targetNamespace | ##local))) :  ##any >
  Content:  ((annotation?))
</any>

<anyAttribute
  namespace = ((##any | ## other) | List of (anyURI |
(##targetNamespace | ##local))) :  ##any >
  Content:  ((annotation?))
</anyAttribute>
```

L'attributo *namespace* specifica l'insieme di appartenenza della Wildcard e può assumere i seguenti valori:

- ##any: indica che la wildcard può essere un qualsiasi elemento XML well-formed;
- ##local: indica che la wildcard può essere un qualsiasi elemento XML well-formed non qualificato;

- ##other: la wildcard può essere un qualsiasi elemento XML well-formed che non appartiene al target namespace dello schema in cui è definita;
- una lista di URI: la wildcard può essere un qualsiasi elemento XML appartenente ad uno dei namespace indentificati dagli URI.

Di seguito sono riportati alcuni esempi di Wildcard:

```
<xs:any />
```

```
<xs:any namespace="##other"/>
```

```
<xs:any namespace="http://www.w3.org/1999/XSL/Transform"/>
```

```
<xs:any namespace="##targetNamespace"/>
```

```
<xs:anyAttribute namespace="http://www.w3.org/XML/1998/namespace"/>
```

1.4.5 Attribute Use

Un componente Attribute Use è utilizzato per contollare l'occorrenza ed i vincoli di valore di una Attribute Declaration. Rappresenta per un oggetto Attribute Declaration l'equivalente del componente Particle per un Element Declaration.

Un Attribute Use è caratterizzato dalle seguenti proprietà:

---

attribute declaration: un oggetto di tipo Attribute Declaration

required: un valore booleano che indica se l'attributo è obbligatorio o opzionale

value constraint: una coppia, opzionale, di un valore ed uno tra *default* e *fixed*

A livello pratico un componente Attribute Use equivale ad una Attribute Declaration per cui è possibile specificare l'attributo *use*. Per questo motivo si rimanda al paragrafo 1.2.1 per la descrizione e la sintassi.

In conclusione si riporta un esempio di un Attribute Use.

```
<xs:attribute ref="globalAttr1" use="required"/>

<xs:attribute ref="po:globalAttr2" default="defvalue"/>
```

1.5 Diagramma di un documento XML Schema

Nella pagina seguente viene riportato un diagramma rappresentante la struttura di un generico documento XMLSchema.

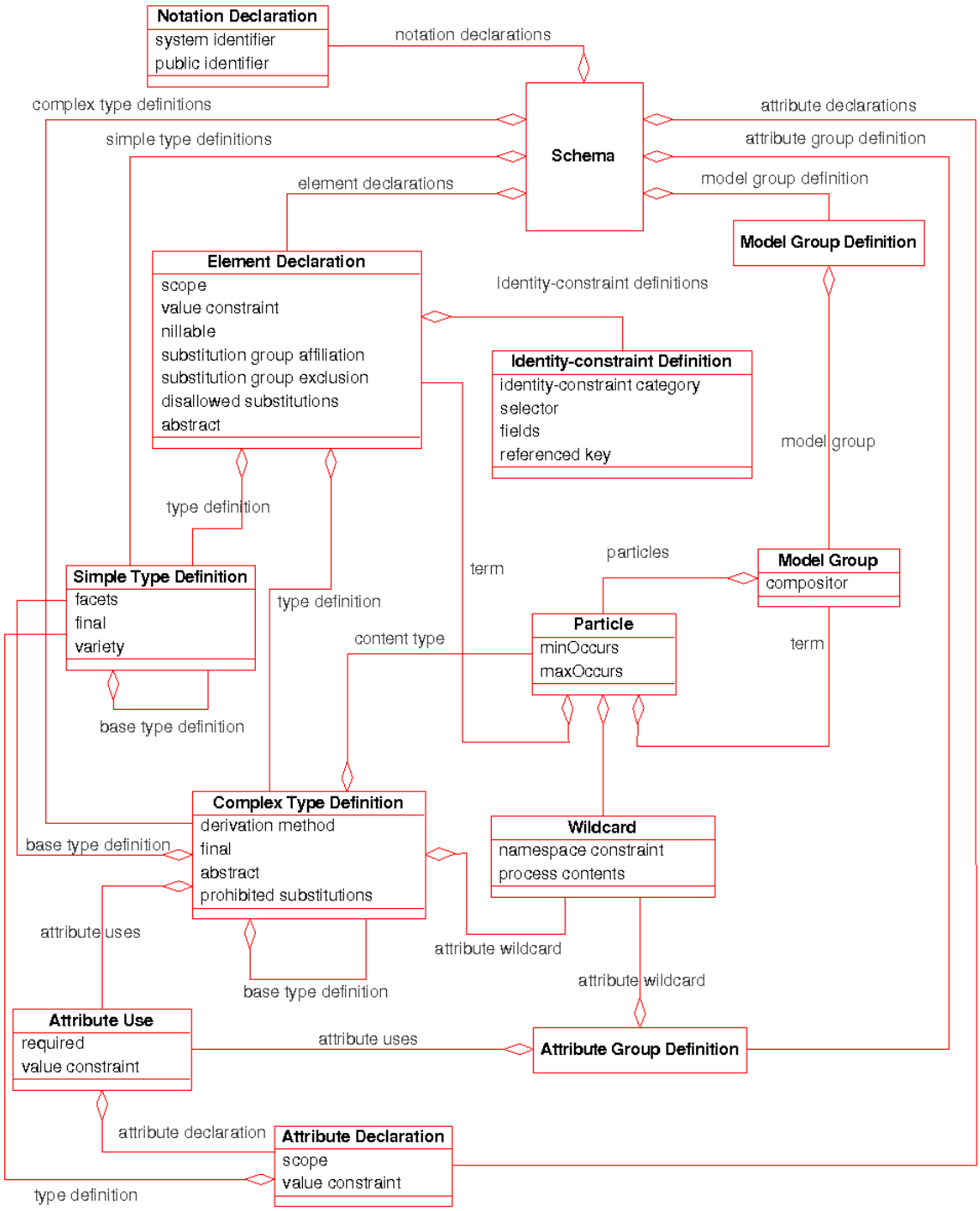

Figura 1.1: Struttura di un documento XML Schema

Capitolo 2

Il linguaggio ODL$_{I^3}$

Introduzione

Il linguaggio ODL$_{I^3}$ è un'estensione del linguaggio standard ODL (***O**bject **D**efinition **L**anguage*), definito dal gruppo di standardizzazione ODMG [7, 8, 9] per descrivere la conoscenza relativa ad uno schema ad oggetti in modo conforme all'ODMG Object Model. Per rispondere alle problematiche di integrazione di informazioni da fonti eterogenee il linguaggio ODL è stato esteso in accordo con le indicazioni del programma I^3 (***I**ntelligent **I**nformation **I**ntegration*) dell'agenzia ARPA, che si propone di creare una architettura di riferimento per l'integrazione di sorgenti dati eterogenee in maniera automatica. Il linguaggio ODL esteso viene quindi definito con il nome ODL$_{I^3}$, acronimo che unisce i due precedenti. Utilizzando questo nuovo linguaggio, il sistema MOMIS realizza l'integrazione di informazioni provenienti da fonti eterogenee, comprendenti anche sorgenti di dati semistrutturati. Nei prossimi paragrafi saranno presentati brevemente il linguaggio standard ODL ed il linguaggio esteso ODL$_{I^3}$.

2.1 Il linguaggio ODL

In questo paragrafo saranno brevemente analizzati i costrutti del linguaggio ODL, tramite i quali possono essere rappresentati schemi di dati ad oggetti ma non schemi di dati semistrutturati.

2.1.1 Tipi classe e tipi valore

Il linguaggio ODL distingue i suoi tipi di dati un due macrocategorie:

- *Tipi Classe (Interface)*
- *Tipi Valore (ValueType)*

I tipi classe (o tipi complessi) sono identificati univocamente, infatti tutte le istanze possiedono un proprio OID (Object IDentifier) che deve essere unico. In pratica l'identità di un oggetto complesso è data dal proprio OID. La principale distinzione tra i tipi classe ed i tipi valore è che per i secondi l' identità è definita direttamente dal proprio valore. I tipi classe vengono impiegati per la descrizione di oggetti complessi, i tipi valore sono invece 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 presenti nel linguaggio ODL_{I^3}, sia i tipi valore che i tipi classe. Il contenuto di un oggetto any può essere quindi sia un attributo semplice che un oggetto complesso.

2.1.2 I tipi valore

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

- *SimpleType*
- *ConstrType*

2.1.3 I tipi semplici

La categoria dei *SimpleType* (o tipi semplici) è costituita dai tipi atomici di base (*BaseType*) e dai *TemplateType*. Vi sono diversi tipi semplici di base predefiniti nel linguaggio ODL; eccone un elenco:

- *boolean*
- *char*
- *string*
- *date*
- *octet*
- *float*
- *double*
- *int*
- *short*
- *long*

- *unsigned int*
- *unsigned short*
- *unsigned long*

Oltre a quelli sopra elencati, anche il tipo *Any* può appartenere a questa categoria. I tipi Int, Short e Long che di default sono *signed*, possono essere anche *unsigned*.

2.1.4 I tipi collezione

I tipi collezione (*TemplateType*) vengono utilizzati per rappresentare collezioni di dati semplici. Le collezioni che possono essere sono di quattro tipi diversi:

- *bag*
- *set*
- *list*
- *array*

Per creare una collezione di elementi non ordinati si utilizza il costruttore *bag*:

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

Il tipo collezione *set* rappresenta un insieme non ordinato di elementi che non contiene duplicati; se si desidera una collezione di questo tipo la dichiarazione sarà la seguente:

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

Se si desidera invece una collezione ordinata di elementi non duplicati si utilizzerà il tipo collezione *list*:

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

Il tipo collezione *array*, infine, viene utilizzato per creare elenchi ordinati con un numero fisso di elementi, ad esempio:

```
array <int,8> IntArray ;
```

2.1.5 I ConstrType

L'altra categoria di tipi valore è quella dei *ConstrType* (tipi costrutti), la quale a sua volta è suddivisa nelle seguenti tre sottocategorie:

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

I tipi enumerati (*EnumType*) restringono il dominio di un *SimpleType* ad un elenco di valori possibili. Consideriamo il seguente esempio esplicativo:

```
enum odd {1 , 3 , 5 , 7 , 9};
```

Una variabile di tipo `odd` potrà assumere solamente uno dei valori (di tipo *integer*) contenuti all'interno dell'elenco fra le parentesi graffe.

Il tipo struttura (*StructType*) è utilizzato per definire strutture di tipi valore, ad esempio:

```
typedef struct Nomestruct {
  string a;
  boolean b;
  unsignedlong c;
} tipostruct ;
```

Un tipo struttura può contenere solo elementi di tipo valore. Una variabile strutturata come nell'esempio precedente può essere dichiarata nel seguente modo:

```
tipostruct var1 ;
```

Oppure, utilizzando il termine opzionale `Nomestruct` che deve sempre essere preceduto dalla parola chiave `struct` è possibile dichiararla come segue:

```
struct Nomestruct var2;
```

L'ultima categoria dei *ConstrType* è il tipo unione, utilizzato per scegliere, in base al valore di una variabile in una clausola switch, il tipo di appartenenza di un attibuto. Ad esempio scrivendo:

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

si dichiara una variabile di nome `NumberType` che può essere una stringa di caratteri oppure un intero. Come per il tipo struttura, le variabili contenute in un tipo unione possono essere solo di tipo valore.

2.1.6 Il tipo classe

Le classi nel linguaggio ODL vengono chiamate interfacce (*Interface*), la definizione di un interfaccia è costituita da due parti fondamentali:

- *Interface Header*
- *Interface Body*

All'interno dell'*Interface Header* vengono specificate le caratteristiche dell'interfaccia come il nome della classe, le superclassi da cui eredita (se presenti) ed una lista di proprietà chiamata *PropertyList*. Il nome di un'interfaccia è l'identificatore unico dell'oggetto (*ObjectID*) all'interno di uno schema ODL. Nella PropertyList possono essere specificate proprietà come l'*extent* che definisce l'insieme delle istanze di una classe all'interno di un Database. Sempre all'interno di questa lista di proprietà è possibile definire una chiave (*key*) per l'interfaccia: una chiave può essere formata da un singolo attributo presente nel corpo dell'interfaccia (*chiave semplice*) oppure da un insieme di più attributi (chiave composita). Nella descrizione di un *Interface Header* sono consentite una sola definizione di extent e una sola definizione di key. Ogni classe nel linguaggio ODL può avere una o più *superclassi*, è cioè ammessa l'ereditarietà multipla: da ogni superclasse vengono ereditati tutti gli attributi e tutti i metodi specificati nel corpo della stessa.

La seconda parte della definizione di una *Interface* è il corpo (*Interface Body*). Il corpo dell'interfaccia descrive la struttura interna della stessa, qui è definito l'insieme di attributi e di metodi che ne fanno parte. Un Interface Body può essere composto dai seguenti elementi:

- *Attributi*
- *Relazioni*
- *Operazioni*
- *Costanti*

Gli attributi che fanno parte di una interfaccia ODL possono essere di tipo semplice o complesso: gli attributi semplici sono di tipo valore (*ValueType*), gli attributi complessi sono invece di tipo *Interface*. La cardinalità degli attributi è di default 1, una cardinalità maggiore può essere specificata indicandola tra parentesi quadre a destra del nome dell'attributo cui si riferisce. Le relazioni (*Relationship*) sono in pratica attributi complessi, infatti possono avere come oggetto solo interfacce e non tipi valore. All'interno di una relazione è possibile inoltre aggiungere informazioni sulla relazione inversa. Le operazioni definiscono i metodi della classe, sono quindi utilizzate per descrivere il comportamento della classe stessa. All'interno del corpo di un'interfaccia possono poi essere definite delle costanti indicando il nome della costante, il tipo semplice della stessa (*SimpleType*) ed il suo valore. Attributi, relazioni, operazioni e costanti devono avere nomi unici all'interno dell'*Interface Body*. Un esempio di definizione di interfaccia ODL è il seguente:

```
interface Professor:Person
( extent Professors
  keys faculty_id, sec_no )
```

```
{
  attribute string name;
  attribute unsigned short faculty_id[6];
  attribute long sec_no[10] ;
  attribute Address address;
  attribute Department department;
  relationship set<Course> teach inverse Course::taught_by;
};
```

L'interfaccia Professor, che ha come superclasse l'interfaccia Person, rappresenta le istanze di tipo Professors all'interno del database e ha una chiave composita formata da due attributi. Il corpo dell'interfaccia è formato da tre attributi semplici, da due attributi complessi (i cui oggetti sono interfacce) e da una relazione, per la quale è definita anche l'inversa.

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

Il linguaggio ODL, ben progettato per la rappresentazione della conoscenza relativa ad un singolo schema ad oggetti, risulta insufficiente per la descrizione e l'integrazione di un insieme di sorgenti eterogenee. Nell'ambito del progetto MOMIS, si sono quindi rese necessarie una serie di modifiche ed estensioni per rappresentare la conoscenza relativa al processo di integrazione intelligente delle informazioni: seguendo le indicazioni dell'I^3 workgroup, si è arrivati a specificare un nuovo linguaggio denominato ODL$_{I^3}$. Di seguito saranno descritte le estensioni e le modifiche apportate al linguaggio ODL che hanno portato alla definizione del linguaggio ODL$_{I^3}$.

2.2.1 Estensioni ai tipi valore

Ai tipi semplici di base (*BaseType*) presenti in ODL è stato aggiunto un nuovo tipo, chiamato *Range* che può essere utilizzato per rappresentare un valore intero compreso tra un estremo inferiore ed uno superiore. Eccone un esempio di utilizzo:

```
range 10,100 number;
```

In questo caso viene definito un attributo number che può assumere tutti i valori interi compresi tra l'estremo inferiore (10) e l'estremo superiore (100). E' inoltre possibile avere come valore degli estremi il valore infinito, per rappresentare questo concetto si utilizzano i termini *+infinite* e *-infinite*. Nel seguente esempio la variabile numero può assumere tutti i valori compresi tra 100 ed infinito:

```
range 100,+infinite number;
```

2.2.2 Estensioni al tipo classe

Nella definizione del linguaggio ODL$_{I^3}$ sono state apportate diverse estensioni al tipo classe (*Interface*) di ODL:

- All'interno dell'*Interface Header* è possibile indicare il tipo della sorgente dati a cui appartiene l'interfaccia: la sorgente può essere di tipo relazionale (*relational* o *nfrelational*), semistrutturato (*semistructured*), ad oggetti (*object*), o file (*file*). Oltre al tipo, è inoltre necessario indicare il nome della sorgente che deve essere unico nello schema ODL$_{I^3}$. Questa estensione è resa necessaria dal fatto che durante il trattamento di diverse sorgenti di dati esiste la possibilità che due classi abbiano lo stesso nome: la presenza del nome della sorgente rende unica una classe all'interno

di uno schema ODL$_{I^3}$. Il nome ed il tipo della sorgente, che rappresentano una caratteristica della classe, vengono dichiarate all'interno della *PropertyList* assieme alle dichiarazioni previste da ODL.

- Per descrivere schemi relazionali, nel linguaggio ODL$_{I^3}$ è stata aggiunta la possibilità di definire delle *foreign key*. Le dichiarazioni di foreign key è sempre contenuta nella *PropertyList* dell'interfaccia. Per dichiarare una chiave estera occorre indicare i nomi degli attributi che compongono la chiave e il nome della classe cui si riferisce la foreign key.

- Poichè il linguaggio ODL$_{I^3}$ deve essere in grado di descrivere anche schemi di dati semistrutturati, è stato aggiunto il costrutto *union* tramite il quale si possono definire più *Interface Body* per una stessa interfaccia. In schemi di dati semistrutturati, infatti, un elemento può comparire nello stesso documento con rappresentazioni differenti, quindi in ODL$_{I^3}$ un'interfaccia potrà essere rappresentata da diverse strutture dati (corrispondenti a differenti *Interface Body*). Sempre per rappresentare sorgenti semistrutturate è stato aggiunto per gli attributi il costrutto *optional*. Nella definizione del corpo di un'interfaccia, postponendo un asterisco (*) alla dichiarazione di un attributo, è possibile indicare che l'attributo è opzionale, definendo quindi una cardinalità minima uguale a zero e una cardinalità massima pari a uno.

- Per l'operazione di integrazione delle sorgenti di dati in ODL$_{I^3}$ è possibile dichiarare attributi globali oltre ai normali attributi locali. Un attributo globale (*globalAttribute*) ha le stesse caratteristiche di un normale attributo ODL ma, in aggiunta, ha una funzione di collegamento ad un oggetto di tipo *MappingRule*.

2.2.3 Gli oggetti MappingRule

Il software del progetto MOMIS, 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. Si è resa quindi necessaria la definizione di regole di mediazione (*MappingRule*) per meglio specificare l'accoppiamento tra gli attributi globali e gli attributi locali originali. Se all'interno di un corpo di un interfaccia, un attributo presenta un riferimento ad una regola di mapping, esso è considerato automaticamente un attributo globale. Nella mediazione tra attributi globali e locali si possono verificare i seguenti casi:

- Corrispondenza fra un attributo globale ed un solo attributo locale (di una sola interfaccia);
- Corrispondenza fra un attributo globale ed un insieme di attributi locali in *and* tra loro: una serie di attributi appartenenti a classi differenti, ritenuti simili tra loro, vengono fusi in un attributo globale;
- Corrispondenza fra un attributo globale ed una serie di attributi locali in *union* tra loro: in questo caso l'attributo globale corrisponde solamente ad un attributo locale per volta;
- Corrispondenza di un attributo globale con un valore di *default*: può servire per esprimere un concetto nel caso in cui non ci sia corrispondenza tra un attributo globale e gli attributi locali.

2.2.4 Relazioni terminologiche

Il linguaggio ODL$_{I^3}$ offre inoltre la possibilità di definire relazioni terminologiche tra nomi di classi e di attributi. Le relazioni terminologiche vengono definite, durante il processo di integrazione, nella fase di generazione del *Thesaurus*. Le relazioni possono essere definite tra classi, attributi o miste tra classi ed attributi: le relazioni tra classi vengono memorizzate in un oggetto *InterfaceRel*, quelle tra attributi in un oggetto *AttributeRel* e quelle miste in un oggetto *AttrIntRel*. Possono essere definite relazioni terminologiche di quattro tipi:

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

Durante la fase di traduzione dal linguaggio ODL$_{I^3}$ alla logica descrittiva OLCD le equivalenze terminologiche vengono poi rese effettive. Una relazione di ipernimia o iponimia tra i nomi di due classi comporta la definizione di un vincolo di ereditarietà tra le due classi. Una relazione di sinonimia tra nomi di classi o nomi di attributi comporta un vincolo di uguaglianza tra le classi o gli attributi. Una relazione di associazione tra due classi comporta l'introduzione di un attributo con dominio la seconda classe nella prima delle due.

2.2.5 Regole d'integrità

ODL$_{I^3}$ rende possibile la dichiarazione di vincoli di integrità sotto forma di regole di tipo *if-then* (*rule*), in grado di definire vincoli non specificabili altrimenti. Le condizioni citate in queste regole devono essere rispettate dalle istanze delle classi ODL$_{I^3}$ specificate appartenenti ad uno schema. La sintassi di una *if-then rule* è:

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

oppure può essere la seguente:

```
rule nomerule [{case of indentifier : caselist }]
```

Il termine `nomerule` è il nome della regola d'integrità; l'`iteratore` rappresenta una istanza di un elemento tra quelli appartenenti a `collezione`; `collezione` è un insieme di istanze rappresentata da una classe o parte di essa; la parte antecedente definisce le condizioni della regola, formata da una serie di predicati booleani in AND fra loro; la parte conseguente descrive gli effetti della regola.

2.2.6 Relazioni intensionali

Una ulteriore importante funzionalità nel processo di integrazione delle sorgenti è la possibilità di esprimere, sotto forma di proposizioni, relazioni estensionali tra gli oggetti di classi distinte. Questi assiomi esprimono le relazioni insiemistiche tra le estensioni di classi definite in sorgenti autonome (indipendentemente dalla loro intensione). Ogni classe dello schema integrato può avere associati gli assiomi estensionali che predicano le relazioni di inclusione, equivalenza e disgiunzione. Gli assiomi estensionali sono espressi come *rule* nel linguaggio ODL$_{I^3}$ (e quindi tradotte in logica OLCD), senza ulteriori

estensioni alla sintassi. La ragione che consente di ricorrere alla sintassi delle rule risiede nel fatto che assiomi estensionali e rule sono semanticamente equivalenti. Le relazioni di disgiunzione, inclusione ed equivalenza tra le estensioni, vengono espresse sottintendendo l'intersezione in quanto si presume che classi tra loro affini abbiano almeno estensioni la cui intersezione non è nulla.

La dichiarazione delle relazioni estensionali è la seguente:

- se A e B sono estensioni disgiunte: $A \bigcap B = \oslash$

```
rule RE1 forall x in (A and B) then x in bottom
```

- se B è inclusa in A: $B \subseteq A$

```
rule RE2 forall x in B then x in A
```

- se A e B sono equivalenti: $A = B$

```
rule RE3 forall x in A then x in B
```

```
rule RE4 forall x in B then x in A
```

2.2.7 Annotazioni rispetto a WordNet

Una fase molto importante del processo di integrazione delle sorgenti del sistema MOMIS è la fase di *annotazione*, durante la quale un utente può selezionare manualmente uno o più significati per ogni elemento dello schema della sorgente locale utilizzando l'ontologia lessicale *WordNet*. Successivamente si avrà la fase di generazione di un dizionario comune (*Common Thesaurus Generation*). Per poter rappresentare le annotazioni rispetto a

WordNet il linguaggio ODL$_{I^3}$ aggiunge il costrutto *WNAnnotation*, eccone un esempio di utilizzo:

```
wnAnnotation ComputerScience.Professor
  lemmaValue="professor",
  lemmaSyntacticCategory=1,
  lemmaSenseNumber=1;
```

Questa annotazione assegna al nome dell'interfaccia `Professor` appartenente alla sorgente `ComputerScience` il valore `"professor"` indicando una categoria sintattica uguale ad `1`(corrispondente alla categoria sintattica *"nome"*) ed un numero di significato uguale ad `1`(corrispondente al significato: *"someone who is a member of the faculty at a college or university"*). Se per un termine sono previste più annotazioni diverse vengono ripetute diverse dichiarazioni di *WNAnnotation*.

Capitolo 3

Specifiche del traduttore XML Schema/ODL$_{I^3}$

Introduzione

In questo capitolo vengono analizzati e confrontati gli elementi dei linguaggio XML Schema e ODL$_{I^3}$, e vengono ricavate delle regole per tradurre i costrutti XML Schema in oggetti ODL$_{I^3}$. Le specifiche di traduzione descritte nel seguito sono state utilizzate per la realizzazione del wrapper XML Schema per il sistema MOMIS. La traduzione che viene proposta è basata sulla versione del linguaggio ODL$_{I^3}$ attuale, implementata nel sistema MOMIS stesso; nel capitolo 5 saranno invece discusse eventuali proposte di estensione del linguaggio ODL$_{I^3}$ per un migliore supporto alla traduzione degli elementi di XML Schema. Nel seguito saranno indicati i punti in cui non è possibile, per disomogeneità dei costruttori dei due linguaggi, effettuare una traduzione esatta e completa, e le soluzioni adottate di conseguenza.

3.1 Concetti preliminari

Prima di affrontare la traduzione dei componenti del linguaggio XML Schema, è necessario fare alcune premesse, in particolare riguardo:

- il mapping tra i tipi di dato predefiniti dei due linguaggi;
- la gestione dei namespace in ODL_{I^3}.

In particolare, come vedremo meglio nel seguito, il concetto di namespace non è direttamente traducibile in ODL_{I^3} se non estendendo il linguaggio stesso. Di questo si parlerà nel capitolo 5.

3.1.1 Mapping tra i tipi di dato predefiniti

Per prima cosa, è necessario definire le regole per la conversione dai tipi di dato predefiniti di XML Schema(XSD) ai *BaseType* di ODL_{I^3}. Si osservi come non sia possibile una traduzione uno a uno, essendo i tipi predefiniti per XSD in numero maggiore rispetto ai suddetti BaseType. Le tabelle 3.1 e 3.2 mostrano le scelte di conversione che sono state adottate nell'ambito di questo progetto. Si noti inoltre che, in base alla documentazione, i tipi predefiniti di XML Schema sono considerati dei Simple Type, per cui queste regole di mapping sono complementari a quelle espresse nel paragrafo 3.2.4.

Tipo XML Schema	Esempio	Tipo ODL$_{I^3}$
int	-2147483648,... ,2147483648	int
integer	...,-1,0,1,...	int
positiveInteger	1,2,...	int
nonNegativeInteger	0,1,2,...	int
negativeInteger	...-2,-1	int
nonPositiveInteger	...-2,-1,0	int
long	-922337203685477,...,-1,0 1,922337203685477	long
short	-32768,...,-1,0,1,...,32768	short
unsigedInt	0,1,...4294967295	unsigned int
unsigedLong	0,...18446744073709551615	unsigned long
unsigedShort	0,1,...65535	unsigned short
decimal	-1.23, 0, 123.4, 1000.00	float
float	-1E4, 12.78e-2, 12	float
double	1267.43233E12, -1234.46e-14	double
byte	-128,...,-1,0,1,...,127	char
unsigedByte	0,1,...,255	char
anyType		any
anySimpleType		any
boolean	true,false	boolean

Tabella 3.1: Conversione dei tipi di dato predefiniti di XML Schema

Tipo XML Schema	Esempio	Tipo ODL$_{I^3}$
base64Binary	GpM7	octet
hexBinary	0FB7	octet
string	Questa è una stringa	string
normalizedString	Questa è una stringa	string
token	Questa è una stringa	string
Name	shipTo	string
QName	po:USAddress	string
NCName	USAddress	string
anyURI	http://www.w3.org/2001/XMLSchema	string
language	en-GB, en-US, fr	string
NMTOKEN	US, Brasil	date
NMTOKENS	“US UK”, “Brasil Canada Mexico”	string
duration	P1Y2M3DT10H30M12.3S	string
dateTime	1999-05-31T13:20:00.000-05:00	date
date	1999-05-31	date
time	13:20:00.000-05:00	date
gYear	2005	date
gYearMonth	2005-01	date
gMonth	01	date
gMonthDay	01-16	date
gDay	16	date

Tabella 3.2: Conversione dei tipi di dato predefiniti di XML Schema

3.1.2 Namespace e schemi composti

Come descritto al paragrafo 1.1.2, XML Schema fa uso dei namespace XML. Nel linguaggio ODL_{I^3} il concetto di Namespace non è presente, per cui questo tipo di informazione non può essere mantenuta. Tuttavia è necessario fare attenzione al caso in cui un documento XML Schema faccia riferimento a oggetti (definizioni di tipi, dichiarazioni di elementi o attributi) appartenenti a namespace diversi sia dal target namespace dello stesso schema, sia dal namespace base del linguaggio (http://www.w3.org/2001/XMLSchema). In questo caso infatti si potrebbero trovare nello stesso schema oggetti con lo stesso nome non qualificato, cioè privo del prefisso indicante il namespace, ma appartenenti a namespace diversi. Ad esempio si possono avere due Complex Type di nome *po:Address* e *ipo:Address* corrispondenti a definizioni diverse; nella traduzione si avrebbero così due *interface* con lo stesso nome *Address*, e si verificherebbe perciò una collisione nei nomi delle classi. Per ovviare a questo problema si utilizzerà il prefisso che in XML Schema identifica il namespace, anche per il nome delle Interface in ODL_{I^3}. Pertanto, l'esempio precedente porterebbe ad avere due interfacce di nome rispettivamente *po_Address* e *ipo_Address*; in questo modo non si creano collisioni in quanto XML Schema assicura l'unicità del nome all'interno del namespace. Un altro problema da affrontare nella traduzione riguarda la gestione degli schemi composti da più documenti, ed in particolare delle direttive *include*, *redefine* e *import*. Nel primo caso sarà necessario recuperare il file in cui è memorizzato lo schema che viene incluso ed estrarre da esso tutte le dichiarazioni e definizioni per poterle tradurre ed includere nel documento ODL_{I^3} di destinazione la loro rappresentazione. Nel secondo caso si procederà come nel primo per tutti gli elementi che non sono ridefiniti, mentre per quelli ridefiniti sarà necessario sostituire la dichiarazione importata con quella ridefinita. Nel terzo caso, infine, si prederanno dal namespace importato le definizioni degli elementi che sono effettivamente referenziate

nel corpo dello schema, seguendo le regole sui prefissi dei nomi descritte in precedenza. Quindi, ogni volta che, nella traduzione di un costrutto, si incontrerà un riferimento ad un oggetto del namespace importato, si dovrà recuperare la definizione di tale oggetto, tradurla con le regole che andremo ad illustrare, ed inserirla nel documento ODL_{I^3} finale.

3.2 Traduzione dei componenti primari

In questa sezione vengono presentate le regole di traduzione per i componenti primari di XML Schema. Per ciascun componente viene riportata la sintassi della dichiarazione. Si rimanda al capitolo 1 per la descrizione dei componenti e la semantica dei campi.

3.2.1 Dichiarazioni di Attribute

La sintassi della dichirazione di Attribute è la seguente:

```
<attribute
   name = NCName
   ref = QName
   type = QName
   form = (qualified | unqualified)
   use = (optional | prohibited | required) :  optional
   fixed = string
   default = string >
```

```
  Content:  ((annotation?  | simpleType?))
</attribute>
```

Consideriamo per ora solo gli attributi dichiarati localmente, affronteremo quelli globali nel paragrafo 3.5. Abbiamo quindi due alternative:

- l'attributo è dichiarato localmente;
- la dichiarazione referenzia un Attribute globale.

Nel secondo caso sarà necessario recuperare la definizione globale e tradurla come se fosse dichiarata localmente. Così la dichiarazione:

```
<xs:attribute name="prova" type="string"/>
<xs:complexType name="transactionType">
  <xs:attribute ref="prova"/>
</xs:complexType>
```

Sarà tradotta come se fosse:

```
<xs:complexType name="transactionType">
  <xs:attribute name="prova" type="string"/>
</xs:complexType>
```

Le dichiarazioni di Attribute presenti all'interno di un componente saranno tradotte aggiungendo un nuovo *attribute* all'interfaccia ODL_{I^3} corrispondente al contenitore. Per evitare collisioni con la traduzione degli Element, illustrata nel paragrafo seguente. il nome dell'attribute sarà *Cname_ Aname*, dove *Cname* è il nome dell'interfaccia e *Aname* è il nome dell'attributo.

Per quanto riguarda il tipo esso sarà preso dal valore dell'attributo *type* o dal Simple Type definito in maniera anonima al suo interno. Nel secondo caso sarà necessario tradurre il nuovo tipo secondo le regole di traduzione dei Simple Type (vedi 3.2.4).

L'attributo *form* riguarda la qualificazione dei nomi nelle istanze, perciò non sarà tradotto in ODL_{I^3}.

Se è specificato *use* = *prohibited*, allora semplicemente non sarà tradotto, oppure, nel caso di Complex Type ridefiniti o derivati per restrizioni, saranno eliminati dall'interface (lo vedremo in seguito nel paragrafo 3.2.3); se *use* = *required* o *use* = *optional*, allora l'attribute ODL_{I^3} avrà rispettivamente cardinalità (1,1) o (0,1); se *use* non è specificato, allora si assumerà il valore di default che è *optional*.

Se non sono specificati gli attributi *fixed* e *default*, allora l'attributo sarà tradotto come nell'esempio seguente:

```
<xs:complexType name="transactionType">
  <xs:attribute name="counter"  type="xs:int"/>
</xs:complexType>
```

diventa:

```
interface transactionType
{
  attribute integer transactionType_counter?;
  // l'attribute è opzionale
}
```

Se l'attributo *fixed* è specificato, allora tradurremo in ODL_{I^3} con una costante anzichè con un attributo; il valore della costante sarà quello specificato per fixed. Il tipo della

costante deve essere un BaseType, per cui estrarremo dal tipo della dichiarazione il tipo XSD predefinito da cui deriva, e lo tradurremo secondo la tabella 3.1 o 3.2. Ad esempio:

```
<xs:complexType name="transactionType">
  <xs:attribute name="counter"  type="xs:int" fixed="400"/>
</xs:complexType>
```

Sarà tradotto con:

```
interface transactionType
{
  const integer transactionType_counter = 400;
}
```

Se è specificato invece l'attributo *default*, allora non è possibile la traduzione esatta in quanto il linguaggio ODL_{I^3} non consente l'utilizzo dei valori di default. In questo caso, la documentazione di XML Schema dice che l'Attribute dichiarato deve essere obbligatoriamente opzionale, per cui sarà tradotto con un attributo opzionale, ignorando di fatto la presenza dell'attributo *default*. Ad esempio:

```
<xs:complexType name="transactionType">
  <xs:attribute name="counter" type="xs:int" default="400"/>
</xs:complexType>
```

Sarà tradotto con:

```
interface transactionType
{
  attribute integer transactionType_counter?;
}
```

3.2.2 Dichiarazioni di Element

La sintassi della dichiarazione del componente Element è la seguente:

```
<element
   name = NCName
   ref = QName
   type = QName
   fixed = string
   default = string
   minOccurs = nonNegativeInteger :  1
   maxOccurs = (nonNegativeInteger | unbounded) :  1
   nillable = boolean :  false
   form = (qualified | unqualified)
   block = (#all | (List of (extension | restriction |
substitution)))
   final = (#all | (List of (extension | restriction)))
   abstract = boolean :  false
   substitutionGroup = QName >
Content:  ((annotation?, ((simpleType|complexType)?,(unique|key|keyref)*))
</element>
```

Poichè nel linguaggio ODL_{I^3} non esiste la differenza tra elementi e attributi come invece in XML Schema, anche le dichiarazioni di Element saranno tradotte in ODL_{I^3} come attribute; in questo caso il nome del nuovo attribute sarà *Ename*, uguale al nome dell'elemento dichiarato.

Come per gli Attribute, consideriamo ora solo gli Element dichiarati all'interno di altri componenti, tratteremo quelli globali nel paragrafo 3.5.

Come prima si presentano i due casi:

- l'elemento è dichiarato localmente;
- la dichiarazione referenzia un Element globale.

Nel secondo caso sarà necessario recuperare la definizione globale e tradurla come se fosse dichiarata localmente, così come avviene per gli Attribute. Della dichiarazione locale con attributo *ref*, saranno però considerati gli attributi *minOccurs* e *maxOccurs*.

Per quanto riguarda il tipo dell'elemento, abbiamo anche qui due possibilità:

- il valore dell'attributo *type* identifica un tipo, semplice o complesso, definito nello schema: in questo caso utilizzaremo il nome del tipo corrispondente nello schema tradotto;
- la dichiarazione contiene una definizione di tipo anonimo: in questo caso il tipo sarà tradotto con le regole che illustreremo in seguito ed avrà nome *Cname_ Ename_ type*, dove *Ename* è il nome dell'elemento e *Cname* il nome dell'interfaccia nello schema ODL_{I^3} che lo deve contenere.

Ad esempio, nel secondo caso:

```
<xs:complexType name="Persona">
  <xs:element name="dataNascita">
    <xs:complexType>
      <xs:element name="giorno" type="xs:integer">
        ....
    </xs:complexType>
  </xs:element>
</xs:complexType>
```

sarà tradotto come:

```
interface Persona
{
  attribute Persona_dataNascita_type dataNascita;
}

interface Persona_dataNascita_type
{
  attribute integer giorno;
}
```

Gli attributi *maxOccurs* e *minOccurs* determinano la cardinalità dell'elemento nel componente in cui è dichiarato. Se non sono specificati, assumiamo che valgano entrambi 1. Li tradurremo secondo le seguenti regole:

- se *minOccurs* e *maxOccurs* valgono 0, l'elemento non viene tradotto (è un elemento proibito);

- se *minOccurs* vale 0, l'attribute corrispondente sarà dichiarato opzionale;
- se *maxOccurs* vale unbounded, il tipo dell'attribute sarà *set<Etype>* anzichè semplicemente *Etype*;
- se *maxOccurs* e *minOccurs* sono entrambi pari a n, con n > 1 e finito, allora tradurremo l'attributo come un array di dimensione n;
- se *maxOccurs* è finito e maggiore di minOccurs tradurremo con un set, rilassando il vincolo di cardinalità; ad esempio, se la cardinalità è (2,5), diventerà (1,n), se è (0,4) diventerà (0,n)

Per quanto riguarda l'attributo *nillable*, in ODL_{I^3} non è possibile distinguere tra elementi opzionali ed elementi a contenuto nullo, per cui se è *nillable* = *true* tradurremo come un attribute opzionale.

Gli attributi *form*, *block* e *final* riguardano l'uso dei nomi qualificati nell'istanza, e le sostituzioni con elementi di tipi derivati, perciò non saranno tradotti.

Come per gli oggetti Attribute Declaration, anche in questo caso l'attributo *default* non sarà tradotto, in quanto in ODL_{I^3} non è possibile specificare valori di default per gli attributi delle classi.

Per quanto riguarda l'attributo *fixed*, esso specifica di fatto una costante. A differenza di ODL_{I^3} in XML Schema è possibile anche specificare delle costanti complesse, perciò l'attributo fixed sarà tradotto solo nel caso in cui l'elemento è di tipo semplice e con cardinalità unitaria: in questo caso l'elemento di nome *Ename* e tipo *Etype* sarà tradotto con una costante di nome *Ename*, di tipo corrispondente al tipo predefinito de cui deriva *Etype* e valore pari al valore dell'elemento. In tutti gli altri casi l'elemento sarà tradotto ignorando l'attributo *fixed*.

Il contenuto della dichiarazione di Element può contenere oltre ad una definizione di tipo anonimo anche un numero arbitrario di definizioni di Identity-constraint, per la cui traduzione si rimanda al paragrafo 3.3.3.

Traduzione dei gruppi di sostituzione

Tratteremo in questa sezione la traduzione degli attributi *abstract* e *substitutionGroup* che, come già detto, sono propri solo degli elementi dichiarati globalmente.

Tradurremo un gruppo di sostituzione con un'interfaccia alla quale aggiungeremo un interface body per ogni elemento del gruppo, più un interface body per l'elemento di testa del gruppo (se questo non è dichiarato abstract). Il nome del gruppo sarà dato da *Sname_union*, dove *Sname* è il nome del substitution group definito.

Ad esempio, lo schema seguente:

```
<xs:schema ... >
  <xs:element name="comment" type="xs:string"/>
  <xs:element name="authorComment" type="xs:string"
          substitutionGroup="comment"/>
  <xs:element name="readerComment" type="xs:string"
          substitutionGroup="comment"/>
  <xs:complexType name="book">
    ...
    <xs:element ref="comment">
    ...
  </xs:complexType>
</xs:schema>
```

sarà tradotto con:

```
interface comment_union
{  attribute string comment; }
union
{  attribute string authorComment; }
union
{  attribute string readerComment; }

interface book
{  attribute comment_union comment; }
```

Come si vede, il riferimento all'elemento testa del substitution group viene tradotto utilizzando il nuovo tipo-classe creato. Per questo sarà necessario tenere traccia degli elementi globali per cui è definito un gruppo di sostituzione.

Se l'elemento *comment* fosse stato dichiarato come *abstract*, cioè:

```
<xs:element name="comment" type="xs:string" abstract="true"/>
```

la traduzione sarebbe stata:

```
interface comment_union
{  attribute string authorComment; }
union
{  attribute string readerComment; }

interface book
{  attribute comment_union comment; }
```

3.2.3 Definizioni di Complex Type

La sintassi di questa definizione è la seguente:

```
<complexType
   name = NCName
   abstract = boolean :  false
   block = (#all | (List of (extension | restriction)))
   final = (#all | (List of (extension | restriction)))
   mixed = boolean :  false >
   Content:  ((annotation?, (simpleContent | complexContent |
   ((group | all | choice | sequence)?,
   ((attribute | attributeGroup)*,anyAttribute?)))))
</complexType>
```

In generale tradurremo un Complex Type di nome *Cname* aggiungendo allo schema ODL_{I^3} una 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* questa volta è il nome, in ODL_{I^3}, dell'elemento che nello schema XSD racchiude la dichiarazione anonima. Ad esempio:

```
<xs:complexType name="Persona">
  <xs:element name="dataNascita">
    <xs:complexType>
```

```
      <xs:element name="giorno" type="xs:integer">
       ....
    </xs:complexType>
  </xs:element>
</xs:complexType>
```

Sarà tradotto come:

```
interface Persona
{
  attribute Persona_dataNascita_type dataNascita;
}

interface Persona_dataNascita_type
{
  attribute integer giorno;
}
```

Gli attributi *block* e *abstract* riguardano il controllo sulle sostituzioni del tipo con i suoi sottotipo nelle istanze, per cui non saranno tradotti. L'attributo *final* non sarà tradotto in quanto non esiste in ODL_{I^3} un meccanismo per specificare che una classe è terminale rispetto alla derivazione ed inoltre non esiste distinzione tra la derivazione per restrizione e quella per estensione.

Per quanto riguarda il contenuto, distingueremo i casi in cui il contenuto è semplice, complesso o nullo. Nel caso in cui non sia specificato nè l'elemento <complexContent> nè l'elemento <complexContent>, allora si tradurrà il contenuto della dichiarazione come come se fosse complesso (essendo questo il caso di defaulto per XML Schema).

3.2.3.1 Complex Type con contenuto semplice

In questo caso il contenuto è formato da un elemento <simpleContent>, il quale a sua volta contiene un oggetto di tipo <restriction> o <extension>. Si riporta per chiarezza la sintassi dei tre componenti.

```
<simpleContent>
  Content:  ((annotation?, (restriction | extension)))
</simpleContent>
```

La sintassi dell'elemento <extension> è:

```
<extension
  base = QName >
  Content:  ((annotation?, ((attribute | attributeGroup)*, anyAttribute?)))
</extension>
```

L'unico attributo è *base*, un nome qualificato che identifica un SimpleType o un Complex Type a contenuto semplice. Nel secondo caso la nuova interfaccia sarà specificata come classe derivata da quella relativa al Complex Type identificato dall'attributo *base*. Nel secondo caso invece alla nuova interfaccia sarà aggiunto un attribute di nome *Cname_node*, dove *Cname* è il nome dell'interfaccia; il tipo del nuovo attribute sarà ottenuto traducendo il Simple Type indicato dall'attributo *base* secondo le regole descritte al paragrafo 3.2.4. In entrambi i casi poi:

- l'elemento Annotation sarà tradotto secondo le regole esposte al paragrafo 3.4.1;
- per ogni Attribute e Attribute Group si aggiungerà un nuovo attributo all'interfaccia secondo le regole descritte, rispettivamente, nel paragrafo 3.2.1 e 3.3.1.

Ad esempio, il seguente costrutto:

```
<xs:complexType name="InternationalPriceType">
  <xs:simpleContent>
    <xs:extension base="xsd:decimal">
      <xs:attribute name="currency" type="xsd:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

sarà tradotto come:

```
interface InternationalPriceType
{
  attribute float InternationalPriceType_node;
  attribute string InternationalPriceType_currency?;
}
```

La sintassi di restriction è la seguente:

```
<restriction
   base = QName >
   Content: ((annotation?, (simpleType?, (minExclusive | minInclusive |
maxExclusive | maxInclusive | totalDigits | fractionDigits | length |
minLength | maxLength | enumeration | whiteSpace | pattern)*)?, ((attribute
| attributeGroup)*, anyAttribute?)))
</restriction>
```

In questo caso l'attributo *base* identifica un Complex Type con contenuto semplice. Non è possibile esprimere in ODL_{I^3} il concetto di derivazione per restrizione. In questo caso dovremo recuperare l'interface, di nome *CbaseName*, che traduce il Complex Type indicato nell'attributo *base* e copiarne il corpo nella nuova interfaccia, di nome *Cname*. Elimineremo l'attributo *CbaseName_ node* e lo sostituiremo con un nuovo attribute di nome *Cname_ node*; il tipo di questo attributo sarà derivato secondo le regole di derivazione per restrizione dei Simple Type (vedi paragrafo 3.2.4) partendo

- dal Simple Type anonimo contenuto nella definizione, se presente;
- altrimenti, dal Simple Type che rappresenta il contenuto del Complex Type di base.

ed applicando le sfaccettature definite nel corpo dell'elemento <restriction>. Quindi, passeremo a tradurre gli Attribute e gli Attribute Group secondo le regole esposte nei paragrafi 3.2.1 e 3.3.1. Gli attribute così ottenuti saranno aggiunti alla nuova interface o, se già presenti, sostituiranno quelli omonimi. In particolare dovremo eliminare dall'interface la traduzione di Attribute per i quali viene specificato *use=prohibited*.

3.2.3.2 Complex Type con contenuto complesso

Quando il contenuto è di tipo *complexContent*, la sintassi del contenuto è la seguente:

```
<complexContent
  mixed = boolean >
  Content:  ((annotation?, (restriction | extension)))
</complexContent>
```

L'attributo *mixed* booleano indica se il contenuto è di tipo mixed o element-only. Nel caso sia *mixed=false*, tradurremo normalmente i sottoelementi del Complex Type in attribute della corrispondente interfaccia. Nel caso il contenuto sia invece di tipo mixed (cioè quando *mixed=true*), per mantenere l'informazione contenuta nel nodo, aggiungeremo un nuovo attribute facoltativo di nome *Cname_ mixedContent*, dove Cname è il nome del Complex Type, e di tipo *string*.

La sintassi di extension è:

```
<extension
  base = QName >
  Content:  ((annotation?, ((group | all | choice | sequence)?, ((attribute
| attributeGroup)*, anyAttribute?))))
</extension>
```

La sintassi di restriction è:

```
<restriction
  base = QName >
  Content: ((annotation?, (group | all | choice | sequence)?, ((attribute
| attributeGroup)*, anyAttribute?)))
</restriction>
```

Nel linguaggio ODL_{I^3} esiste il concetto di ereditarietà multipla tra classi, ma la semantica della derivazione è solo di tipo additivo.

Nel caso sia presente l'elemento <extension> possiamo mantenere l'informazione sulla superclasse, in particolare imposteremo una superclasse per la nuova interface recuperando l'interfaccia relativa al Complex Type indicato dall'attributo *base*. In seguito si tradurranno tutti gli elementi interni all'elemento *extension*, in particolare:

- l'elemento Annotation sarà tradotto secondo le regole esposte al paragrafo 3.4.1.
- per ogni Attribute e Attribute Group si aggiungerà un nuovo attributo all'interfaccia secondo le regole descritte, rispettivamente, nel paragrafo 3.2.1 o 3.3.1;
- l'elemento a scelta tra <all>, <choice> o <sequence> è un oggetto di tipo Particle, per la cui traduzione si rimanda al paragrafo 3.4.3;
- l'elemento <group> è un riferimento ad un oggetto di tipo Model Group Definition globale, per la sua traduzione si veda il paragrafo 3.3.2.

Nel caso sia presente l'elemento <restriction>, invece, non possiamo mantenere l'informazione sulla superclasse. In questo caso dovremo recuperare l'interface relativa al

Complex Type indicato dall'attributo *base*, copiarla nella nuova interfaccia e poi modificare gli oggetti che sono dichirati all'interno dell'elemento <restriction>. In particolare andremo a sostituire tutti gli oggetti che sono ridefiniti e ad eliminare gli attributi con *use=prohibited*. Tali oggetti saranno tradotti secondo le stesse regole usate per l'elemento <extension>.

Ad esempio la seguente descrizione XML Schema:

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

<xs:complexType name="USAddress">
  <xs:complexContent>
    <xs:extension base="address">
      <xs:sequence>
        <xs:element name="state" type="xs:string"></xs:element>
        <xs:element name="zip" type="xs:positiveInteger"></xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Sarà tradotta in ODL$_{I^3}$ come:

```
interface address
{
  attribute string name;
  attribute string street;
  attribute string city;
}

interface USAddress : address
{
  attribute string state;
  attribute int zip;
}
```

3.2.3.3 Complex Type con contenuto nullo

Per quanto riguarda gli elementi a contenuto nullo, essi sono considerati di default come elementi complexContent, element-only e derivati per restrizione dal tipo anyType (che non ha nè sottoelementi nè attributi). Nel caso siano definiti attributi o altro, il Complex Type sarà trattato come descritto nel paragrafo precedente per l'elemento <restriction>. Quindi il seguente tipo:

```
<xsd:complexType name="InternationalPriceTypeEmpty">
  <xsd:complexContent>
    <xsd:restriction base="xsd:anyType">
      <xsd:attribute name="currency" type="xsd:string"/>
```

```
      <xsd:attribute name="value"    type="xsd:decimal"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Sarà tradotto come:

```
interface InternationalPriceTypeEmpty
{
  attribute float InternationalPriceTypeEmpty_value ?;
  attribute string InternationalPriceTypeEmpty_currency ?;
}
```

3.2.4 Definizioni di Simple Type

In questo paragrafo si esporranno le regole di traduzione adottate per i Simple Type. Di fatto anche i tipi predefiniti di XML Schema appartengono alla categoria dei Simple Type, ma questi devono essere tradotti secondo le regole di mapping tra i tipi di dato discusse al paragrafo 3.1.1. Nel seguito si darà per sottointeso che i tipi predefiniti devono essere tradotti in questo modo.

La sintassi di un oggetto Simple Type Definition è la seguente:

```
<simpleType
  final = (#all | List of (list | union | restriction))
  name = NCName >
  Content:  ((annotation?, (restriction | list | union)))
</simpleType>
```

L'attrbituo *final* non verrà tradotto poichè riguarda l'utilizzo del tipo nei documenti XML Schema e non esistono costrutti simili nel linguaggio ODL$_{I^3}$.

Distingueremo ora i diversi tipi di definizione di Simple Type, che saranno tradotti con regole diverse.

Un tipo semplice di nome *Tname* sarà tradotto definendo con il costrutto typedef un nuovo tipo di nome *Tname*; se il Simple Type è definito in maniera anonima, allora il nuovo tipo ODL$_{I^3}$ avrà nome *Cname_type* dove *Cname* è il nome che nello schema ODL$_{I^3}$ corrisponde alla definizione del contenitore del tipo anonimo.

3.2.4.1 Traduzione dei Simple Type di tipo List

In questo caso l'elemento <simpleType> contiene al suo interno un elemento <list> la cui sintassi è:

```
<list
  itemType = QName >
  Content:  ((annotation?, simpleType?))
</list>
```

I Simple Type di tipo lista saranno tradotti in ODL$_{I^3}$ usando il costrutto *list*<*type*>; il tipo *type* sarà ottenuto traducendo il Simple Type indicato dall'attributo *itemType* oppure il Simple Type definito in maniera anonima all'interno dell'elemento <list> stesso. Ad esempio il costrutto seguente:

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

Sarà tradotto come:

```
typedef list<int> listOfIntType;
```

3.2.4.2 Traduzione dei Simple Type di tipo Union

In questo caso l'elemento <simpleType> contiene al suo interno un elemento <union> la cui sintassi è:

```
<union
  memberTypes = List of QName >
  Content:  ((annotation?, simpleType*))
</union>
```

In questo caso il Simple Type sarà tradotto in ODL_{I^3} definendo un nuovo tipo union di nome Tname. Per ciascuno dei Simple Type in union (cioè quelli indicati dall'attributo *memberTypes* più quelli definiti in maniera anonima) si aggiungerà alla struttura un nuovo elemento. Ad esempio la seguente definizione:

```
<xsd:simpleType name="myUnion">
  <xsd:union memberTypes="integer string"/>
</xsd:simpleType>
```

sarà tradotta come:

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

3.2.4.3 Traduzione dei Simple Type derivati per restrizione

La sintassi di questa definizione è la seguente:

```
<restriction
   base = QName >
   Content: ((annotation?, (simpleType?, (minExclusive | minInclusive
| maxExclusive | maxInclusive | totalDigits | fractionDigits | length |
minLength | maxLength | enumeration | whiteSpace | pattern)*)))
</restriction>
```

In questo caso il tipo che viene definito può essere un tipo atomico, list o union, a seconda del tipo base. Per prima cosa sarà necessario recuperare il tipo base (indicato dall'attributo *base*) o tradurre il simpleType anonimo dichiarato all'interno della dichiarazione, a seconda dei casi.

Nel caso il tipo base sia un tipo lista, allora considereremo le sfaccettature *lentgh* o *maxLength*: nel caso siano impostate, allora la traduzione sarà un array di lunghezza pari al valore di length (o maxlength) e di tipo dato dal valore di itemType della base.

Ad esempio:

```
<xs:simpleType name="simpleList">
    <xs:list itemType="xs:string"/>
</xs:simpleType>

<xs:simpleType name="simpleRest">
    <xs:restriction base="simpleList">
        <xs:length value="10"/>
    </xs:restriction>
</xs:simpleType>
```

Sarà tradotto come:

```
typedef list<string> simpleList ;
typedef array<string,10> simpleRest ;
```

Nel caso il tipo base sia invece un tipo atomico, allora considereremo le sfaccettature minLength, maxLength, minExclusive, minInclusive ed enumeration.

Nel caso sia specificata la sfaccettatura *enumeration*, allora la traduzione sarà fatta con un nuovo tipo enumerato, i cui valori sono i valori di tutte le clausole enumeration presenti. Ad esempio:

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

```
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>
```

Sarà tradotto come:

```
enum USState { AK, AL, AR ... }
```

Nel caso sia specificata almeno una delle sfaccettature minLength, maxLength, minExclusive e minInclusive, ed il tipo base è traducibile con un tipo intero, allora, potremo tradurre il costrutto con un tipo range, in cui:

- il limite inferiore è dato dal valore di *minInclusive*, o dal valore di *minExclusive* aumentato di 1, o da *-infinite* se nessuno dei due è specificato;
- il limite superiore è dato dal valore di *maxInclusive*, o dal valore di *maxExclusive* diminuito di 1, o da *+infinite* se nessuno dei due è specificato.

Ad esempio il seguente tipo:

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

Sarà tradotto con:

```
typedef range 1,31 dayOfMonth ;
```

In tutti i casi diversi da quelli considerati la traduzione sarà fatta con il tipo base dell'elemento restriction.

3.3 Traduzione dei componenti secondari

In questa sezione vengono presentate le regole di traduzione per i componenti secondari di XML Schema. Per ciascun componente viene riportata la sintassi della dichiarazione. Si rimanda al capitolo 1 per la descrizione dei componenti e la semantica dei campi.

3.3.1 Definizioni di Attribute Group

Riportiamo ora la sintassi della definizione di Attribute Group.

```
<attributeGroup
   name = NCName
   ref = QName >
   Content:  ((annotation?, ((attribute | attributeGroup)*,
anyAttribute?)))
</attributeGroup>
```

Un Attribute Group di nome *AGname* sarà tradotto in ODL$_{I^3}$ creando una nuova *interface* di nome *AGname_group*. Tutti gli oggetti che si trovano nel corpo del gruppo saranno tradotti secondo le regole opportune e poi inseriti nella nuova interfaccia. Ogni dichirazione di Attribute Group nello schema che faccia riferimento ad un gruppo di nome

AGname, sarà tradotta come un attribute di nome *AGname* e tipo *AGname_group*.

Per esempio, il seguente frammento di codice XML Schema:

```
<xs:attributeGroup name="myAttrGroup">
    ...
</xs:attributeGroup>

<xs:complexType name="myCompType">
  ...
  <xs:attribute ref="myAttrGroup"/>
</xs:complexType>

<xs:attributeGroup name="myAttrGroup2">
  ...
  <xs:attribute ref="myAttrGroup"/>
</xs:attributeGroup>
```

sarà tradotto in ODL_{I^3} come:

```
interface myAttrGroup_group
{
  attribute ...
}

interface myCompType
{ ...
  attribute myAttrGroup_group myAttrGroup }
```

```
interface myAttrGroup2_group
{ ...
  attribute myAttrGroup_group myAttrGroup
}
```

3.3.2 Definizioni di Model Group

Riportiamo di seguito la sintassi di questo componente:

```
<group
   maxOccurs = (nonNegativeInteger | unbounded) :  1
   minOccurs = nonNegativeInteger :  1
   name = NCName
   ref = QName >
   Content:  (annotation?, (all | choice | sequence)?)
</group>
```

La definizione di un Model Group di nome *MGname* sarà tradotto creando una nuova interface di nome *MGname_ mgroup*. Il corpo di questo interfaccia sarà riempito con la traduzione dell'elemento interno alla definizione (uno tra <all>, <choice> e <sequence>) secondo le regole descritte al paragrafo 3.4.2.

I riferimenti ad un Model Group *MGname* definito in un certo namespace (cioè gli oggetti Model Group Definition per cui è l'attributo *ref* specificato) saranno tradotti invece con un attribute di nome *MGname* e tipo *MGname_ mgroup*. In questo caso sarà

necessario interpretare anche gli attributi *minOccurs* e *maxOccurs*: si procederà come specificato per gli oggetti Particle, in particolare:

- se *minOccurs* e *maxOccurs* valgono 0, l'elemento non verrà tradotto;
- se *minOccurs* vale 0, l'attribute corrispondente sarà dichiarato opzionale;
- se *maxOccurs* vale unbounded, il tipo dell'attribute sarà *set*<*MGname_ mgroup*> anzichè semplicemente *MGname_ mgroup*;
- se *maxOccurs* e *minOccurs* sono entrambi pari a n, con n $>$ 1 e finito, allora tradurremo l'attributo come un array di dimensione n;
- se *maxOccurs* è finito e maggiore di minOccurs tradurremo con un set, rilassando il vincolo di cardinalità; ad esempio, se la cardinalità è (2,5), diventerà (1,n), se è (0,4) diventerà (0,n)

Ad esempio, lo schema:

```
<xs:schema ... >
  <xs:group name="myModelGroup">
    ...
  </xs:group>

  <xs:complexType name="trivial">
    <xs:group ref="myModelGroup"/>
    ...
  </xs:complexType>
</xs:schema>
```

verrà tradotto in ODL_{I^3} con le seguenti classi:

```
interface myModelGroup_mgroup
{
  ...
}

interface trivial
{
  attribute myModelGroup_mgroup myModelGroup;
  ...
}
```

3.3.3 Definizioni di Identity-constraint

Gli oggetti Identity-constraint Definition si suddividono in tre categorie: vincoli di unicità (*unique*), chiavi (*key*), e chiavi riferite. Tratteremo i tre costrutti separatamente.

3.3.3.1 Vincoli di Unicità

E' rappresentato da un elemento *unique*, la cui sintassi è la seguente:

```
<unique
   name = NCName >
   Content:  ((annotation?, (selector, field+)))
</unique>
```

In ODL_{I^3} esiste la possibilità di esprimere un vincolo di unicità tramite il costrutto *key*, il quale permette di definire una chiave semplice (se comprende un solo attribute) o composta (se comprende più attributi). Tradurremo quindi il vincolo di unicità aggiungendo una proprietà *key* nel *header* dell'interface in cui il vincolo è definito. L'elemento selector sarà utilizzato per ricavare il contesto del vincolo, cioè l'interface in cui è definito; l'elemento field sarà utilizzato per ricavare gli attributi della suddetta classe da includere nella chiave. Per entrambi sarà necessario poter valutare le espressioni *XPath* per tradurre il vincolo. Ad esempio, la dichiarazione:

```
<unique name="myUnique1">
  <selector xpath="r:regions/r:zip" />
  <field xpath="@code" />
</unique>
```

Sarà tradotto aggiungendo all'interfaccia *zip* una *key* relativa all'attributo *code*:

```
interface zip
(key (code))
{
 attribute int code;
  ...
}
```

3.3.3.2 Chiavi

La sintassi in questo caso è la seguente:

```
<key
   name = NCName >
   Content:  ((annotation?, (selector, field+)))
</key>
```

Il costrutto *key* di XML Schema sarà tradotto con il costrutto *key* di ODL$_{I^3}$. Anche in questo caso sarà necessario valutare le espressioni *XPath* contenute nell'elemento *selector*, per ricavare l'interface in cui inserire la chiave, e nell'elemento *field*, per determinare gli attributi da inserire nel vincolo dichiarato.

3.3.3.3 Chiavi Riferite

L'elemento *keyref* di XML Schema ha la seguente sintassi:

```
<keyref
   name = NCName
   refer = QName >
   Content:  ((annotation?, (selector, field+)))
</keyref>
```

Le chiavi riferite di XML Schema saranno tradotte in ODL$_{I^3}$ utilizzando il costrutto *foreign_ key*, che permette di definire delle chiavi estere. La traduzione degli elementi *selector* e *field* avviene con le stesse modalità con cui si traducono le chiavi viste al paragrafo precedente. L'attributo *refer* sarà poi utilizzato per individuare la *key* a cui la *foreign_ key* si riferisce. Ad esempio, i seguenti elementi:

```
<key name="pNumKey">
  <selector xpath="r:parts/r:part" />
  <field xpath="@number" />
</key>

<keyref name="dummy2" refer="p:NumKey" >
  <selector xpath="r:zip/r:part" />
  <field xpath="@number" />
</keyref>
```

Saranno tradotti in ODL$_{I^3}$ come:

```
interface parts_part
( key (number) )
{
  attribute int number;
  ...
}
```

```
interface zip_part
( foreign_key (number) references parts_part )
{
  attribute int number;
  ...
}
```

3.3.4 Dichiarazioni di Notation

Gli oggetti di tipo notation non saranno tradotti in quanto non portano valore informativo.

3.4 Traduzione dei componenti helper

3.4.1 Annotation

La sintassi di un elemento di tipo Annotation è la seguente:

```
<annotation>
  Content:  ((appinfo | documentation)*)
</annotation>
```

Gli elementi *appinfo* non saranno tradotti poichè non hanno contenuto informativo utile alla comprensione dello schema, ma servono alle applicazioni che accedono allo schema e che lo utilizzano. Tradurremo perciò solo gli elementi di tipo *documentation*. La traduzione è elementare in quanto, per ogni elemento <documentation> presente, andremo a creare una nuova costante di tipo *string*, valore pari al contenuto dell'oggetto *documentation* e nome *annotation_ i*, dove i è un valore intero progressivo che parte da 1 e aumenta ad ogni Annotation (questo è necessario perchè potrebbero esserci più elementi di tipo documentation). La costante ODL_{I^3} così creata sarà aggiunta all'interfaccia corrispondente all'oggetto che nello schema contiene il componente Annotation. Ad esempio la dichiarazione:

```
<xs:complexType name="myCompType">
  <xs:annotation>
    <xs:documentation>
      A type for experts only
    </xs:documentation>
```

```
    <xs:appinfo>
      <fn:specialHandling>checkForPrimes</fn:specialHandling>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:complexType>
```

Sarà tradotta come:

```
interface myCompType
{
  const string annotation_1 "A type for expert only";
  ...
}
```

3.4.2 Model Group

Un oggetto di tipo Model Group può essere rappresentato da un elemento <all>, <sequence> o <choice>.

La dichiarazioni *all* ha la seguente sintassi:

```
<all
  minOccurs = 1 :  1
  maxOccurs = (0 | 1) :  1 >
  Content:  ((annotation?, element*))
</all>
```

Se l'elemento <all> è figlio diretto di un elemento <complexType>, <complexContent> o <group> e la coppia (*minOccurs*,*maxOccurs*) vale (1,1), allora la traduzione sarà eseguita inserendo nell'interface corrispondente all'elemento contenitore la traduzione di tutti gli Element all'interno della dichiarazione *all* e dell'eventuale Annotation, secondo le regole descritte nei paragrafi 3.2.2 e 3.4.1.

Negli altri casi:

- creeremo una nuova interface di nome *Cname_ all*, dove Cname è il nome dell'interface corrispondente al contenitore dell'oggetto *all*.
- tradurremo il contenuto dell'elemento <all> nella nuova interface *Cname_ all*, con la stesse modalità di prima;
- aggiungeremo all'interface *Cname* un attribute di tipo *Cname_ all* e nome *Cname_ all*; se è *minOccurs=0* allora imposteremo il nuovo attributo come opzionale.

Ad esempio il Complex Type:

```
<xs:complexType name="person">
  <xs:all>
    <xs:element name="Nome" type="xs:string"/>
    <xs:element name="Cognome" type="xs:string"/>
  </xs:all>
</xs:complexType>
```

Sarà tradotto come:

```
interface person
{
  attribute string Nome;
  attribute string Cognome;
}
```

Le dichiarazioni di *sequence* hanno la seguente struttura:

```
<sequence
   maxOccurs = (nonNegativeInteger | unbounded) :  1
   minOccurs = nonNegativeInteger :  1 >
   Content:  ((annotation?, (element | group | choice | sequence
| any)*))
</sequence>
```

Dato che in ODL_{I^3} non esiste un modo per specificare che un insieme di attributi debba comparire in una certa sequenza, la traduzione dell'elemento <sequence> seguirà le stesse regole dell'elemento <all> visto in precedenza, quindi:

- se l'elemento <sequence> è figlio diretto di un elemento <complexType>, <complexContent> o <group> e la coppia (*minOccurs*,*maxOccurs*) vale (1,1), allora la traduzione sarà fatta inserendo nell'interface corrispondente all'elemento contenitore la traduzione di tutti gli Element, i Model Group, e l'eventuale Annotation contenuti all'interno della dichiarazione *sequence* secondo le regole descritte nei rispettivi paragrafi di questo capitolo;

- negli altri casi:
  - creeremo una nuova interface di nome *Cname_ sequence*, dove Cname è il nome dell'interface corrispondente al contenitore dell'oggetto *sequence*.
  - tradurremo il contenuto dell'elemento <sequence> nella nuova interface *Cname_ sequence*, con la stesse modalità di prima;
  - aggiungeremo all'interface *Cname* un attribute di tipo *Cname_ sequence* e nome *Cname_ sequence*, tenendo in considerazione le regole per gli attributi *minOccurs* e *maxOccurs* espresse nel paragrafo 3.4.3.

La sintassi dell'elemento *choice* è invece la seguente:

```
<choice
  maxOccurs = (nonNegativeInteger | unbounded) :  1
  minOccurs = nonNegativeInteger :  1 >
  Content:  ((annotation?, (element | group | choice | sequence
| any)*))
</choice>
```

L'elemento choice sarà sempre tradotto creando una nuova interface di nome *Cname_ choice*, dove *Cname* è il nome dell'oggetto che contiene la dichiarazione <choice>. Per ognuno degli elementi nel *content* della dichiarazione, aggiungeremo alla suddetta interface un nuovo IntBody nel quale inseriremo la traduzione dell'elemento fatta secondo le regole opportune. Infine aggiungeremo all'interface *Cname* un nuovo attributo di nome *Cname_ choice* e tipo *Cname_ choice*, tenendo in considerazione le regole per gli

attributi *minOccurs* e *maxOccurs* espresse nel paragrafo 3.4.3. Ad esempio, la seguente dichiarazione:

```
<xs:complexType name="myComplex">
 <xs:choice>
  <xs:element ref="left"/>
  <xs:element ref="right"/>
 </xs:choice>
</xs:complexType>
```

sarà tradotta come:

```
interface myComplex
{
  attribue myComplex_choice myComplex_choice;
}

interface myComplex_choice
{  attribute ... left;  }
 union
{  attribute ... right; }
```

3.4.3 Particle

Un elemento di tipo Particle è un Model Group, una Element Declaration o una Wildcard, perciò sarà tradotto seguendo le regole espresse al paragrafo 3.3.2 o 3.2.2 o 3.4.4. In tutti e tre i casi gli attributi *minOccurs* e *maxOccurs* devono essere trattati nel seguente modo:

- 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*> anzichè semplicemente *type*;
- se *maxOccurs* e *minOccurs* sono entrambi pari a n, con n $>$ 1 e finito, allora tradurremo l'attributo come un array di dimensione n;
- se *maxOccurs* è finito e maggiore di minOccurs tradurremo con un set, rilassando il vincolo di cardinalità; ad esempio, se la cardinalità è (2,5), diventerà (1,n), se è (0,4) diventerà (0,n).

3.4.4 Wildcard

Questo tipo di elementi non verrà tradotto in quanto non esiste in ODL_{I^3} un costrutto simile alla wildcard, non essendo presente, in questo linguaggio, il concetto di namespace.

3.4.5 Attribute Use

La traduzione di questo tipo di elementi è già stata contempleta al paragrafo 3.2.1 a proposito della traduzione degli oggetti Attribute Declaration.

3.5 Traduzione di un documento XML Schema nel complesso

A livello globale, la traduzione di un documento XML Schema verrà effettuata nel seguente modo:

1. per prima cosa sarà necessario includere/importare tutte le definizioni provenienti da namespace esterni nell'insieme delle strutture da tradurre;

2. in secondo luogo andremo a recuperare tutte le definizioni dei Simple Type globali (cioè non dichiarati in modo anonimo) ed a tradurli secondo le regole illustrate al paragrafo 3.2.4;

3. quindi passeremo ad analizzare gli oggetti Element Declaration globali alla ricerca di eventuali gruppi di sostituzione, e li tradurremo come descritto al paragrafo 3.2.2;

4. a questo punto sarà affrontata la traduzione degli oggetti Attribute Group Definition e Model Group Definition, secondo le regole ricavate al paragrafo 3.3.1 e 3.3.2 rispettivamente.

5. infine andremo a tradurre tutti i Complex Type definiti globalmente, in base alle regole del paragrafo 3.2.3;

6. come ultima cosa, sarà necessario prevedere una fase di post-processing in cui vengano validati i tipi complessi ODL_{I^3} ed associate le definizioni agli identificatori usati nei tipi degli attributi; questa fase non è necessaria da un punto di vista logico, ma è richiesta dal punto di vista implementativo in quanto durante la costruzione dello schema ODL_{I^3} l'associazione tra l'identificatore del tipo-classe e

la sua definizione non è automatica, ma viene lasciata in sospeso, e deve quindi essere risolta al termine della creazione dello schema.

3.6 Tabella riassuntiva

Nelle pagine seguenti sono riportate le tabelle con le regole di traduzione illustrate in precedenza, a titolo riassuntivo di questo capitolo.

Nome	Costrutto XML Schema	Traduzione in ODL$_{I^3}$
Tipi predefiniti	Vedi tabelle 1.1 e 1.2	Vedi tabelle 3.1 e 3.2 a pag. 92 e 93
Nomi qualificati	ipo:Address	diventa: • Address (se ipo è il prefisso associato al targetNamespace dello schema) • ipo_Address (altrimenti)
Schemi su più documenti	<import schemaLocation=.../> <include schemaLocation=.../> <redefine schemaLocation=.../>	Si devono gestire le definizioni esterne come descritto nel paragrafo 3.1.2 a pag. 94
Attribute	<xs:complexType name="[Cname]"> <xs:attribute name="[Aname]" type="[Atype]"/> </xs:complexType>	interface [Cname] { attribute [Atype] [Cname]_[Aname]?; }
Attribute required	<xs:complexType name="[Cname]"> <xs:attribute name="[Aname]" type="[Atype]" use="required"/> </xs:complexType>	interface [Cname] { attribute [Atype] [Cname]_[Aname]; }
Attribute con valore fixed	<xs:complexType name="[Cname]"> <xs:attribute name="[Aname]" type="[Atype]" fixed="400"/> </xs:complexType>	interface [Cname] { const [Atype] [Cname]_[Aname] = 400; }
Attribute reference	<xs:attribute name="[Aname]" type="[Atype]"/> <xs:complexType name="[Cname]"> <xs:attribute ref="[Aname]"/> </xs:complexType>	interface [Cname] { attribute [Atype] [Cname]_[Aname]; }
Element	<xs:complexType name="[Cname]"> <xs:element name="[Aname]" 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]?; }

Tabella 3.3: Traduzione dei costrutti XML Schema in ODL$_{I^3}$

Nome	Costrutto XML Schema	Traduzione in ODL_{I^3}
Element con dichirazione 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] { ... }
Element opzionale	<xs:complexType name="[Cname]"> <xs:element name="[Aname]" type="[Etype]" minOccurs="0"/> </xs:complexType>	interface [Cname] { attribute [Etype] [Ename]?; }
Element con cardinalità (1,n)	<xs:complexType name="[Cname]"> <xs:element name="[Aname]" type="[Etype]" maxOccurs="unbounded"/> </xs:complexType>	interface [Cname] { attribute set<[Etype]> [Ename]; }
Element tradotto con array	<xs:complexType name="[Cname]"> <xs:element name="[Aname]" type="[Etype]" minOccurs="4" maxOccurs="4"/> </xs:complexType>	interface [Cname] { attribute [Etype] [Ename][4]; }
Element con cardinalità (0,5)	<xs:complexType name="[Cname]"> <xs:element name="[Aname]" type="[Etype]" minOccurs="2" maxOccurs="5"/> </xs:complexType>	interface [Cname] { attribute set<[Etype]> [Ename]?; }
nillable Element	<xs:complexType name="[Cname]"> <xs:element name="[Aname]" type="[Etype]" nillable="true"/> </xs:complexType>	interface [Cname] { attribute [Etype] [Ename]?; }

Substitution Group	<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];
		...
		}

Tabella 3.4: Traduzione dei costrutti XML Schema in ODL_{I^3}

Nome	Costrutto XML Schema	Traduzione in ODL$_{I^3}$

Substitution Group con Abstract Element	<xs:element name="[Sname]" type="[Stype]" abstract="true"/>
	<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] [SubName1]; }
		union
		{ attribute [Stype] [SubName2]; }
		
		interface [Cname]
		{
		...
		attribute [Sname]_union [Sname];
		...
		}

Complex Type Generico	<xs:complexType name="[Cname]">
	<xs:sequence>
	<xs:element name=name type=xs:string/>
	<xs:element name=street type=xs:string/>
	<xs:element name=city type=xs:string/>
	...
	</xs:sequence>
	</xs:complexType>
		interface [Cname]
		{
		attribute string name;
		attribute string street;
		attribute string city;
		}

Complex Type a contenuto semplice derivato da un Simple Type	<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 [SimpleTypeName] [Cname]_node;
		attribute [Atype] [Cname]_[Aname]?;
		}

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] { ... }

Tabella 3.5: Traduzione dei costrutti XML Schema in ODL$_{I^3}$

Nome	Costrutto XML Schema	Traduzione in ODL$_{I^3}$

Complex Type con contenuto nullo	<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] ?;
		}

Simple Type di tipo lista	<xs:simpleType name="[ListTypeName]"> <xs:list itemType="[itemTypeName]"/> </xs:simpleType>	typedef list<[itemTypeName]> [ListTypeName];

Simple Type di tipo unione	<xs:simpleType name="[UnionTypeName]">
	<xs:union memberTypes="[memberType_1] [memberType_2] ..."/>
	</xs:simpleType>
		union [UnionTypeName] switch(val) {
		case 1: [memberType_1];
		case 2: [memberType_2];
		case ...
		}

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] ;

Tabella 3.6: Traduzione dei costrutti XML Schema in ODL$_{I^3}$

Nome	Costrutto XML Schema	Traduzione in ODL_{I^3}

Attribute Group Definition	<xs:schema ...>
	<xs:attributeGroup name="[AGname]">
	...
	</xs:attributeGroup>
	
	<xs:complexType name="[Cname]">
	...
	<xs:attribute ref="[AGname]"/>
	</xs:complexType>
	</schema>
		interface [AGname]_group
		{ ... }
		
		interface [Cname]
		{
		...
		attribute [AGname]_group [AGname];
		}

Model Group Definition	<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]; }
Identity-constraint unique	<unique name = "[Uname]"> <selector xpath = "[XPathS]" /> <field xpath = "[XPathF] </unique>	Tradotto con un costrutto *key*. L'espressione $XPathS$ serve per determinare l'interface in cui inserire la chiave. L'espressione $XPathF$ determina gli attributi da includere nella chiave.
Identity-constraint key	<key name = "[Kname]"> <selector xpath = "[XPathS]" /> <field xpath = "[XPathF] </key>	Tradotto con un costrutto *key*. L'espressione $XPathS$ serve per determinare l'interface in cui inserire la chiave. L'espressione $XPathF$ determina gli attributi da includere nella chiave.
Identity-constraint keyref	<keyref name = "[Kname]" refer = "[Rname]"> <selector xpath = "[XPathS]" /> <field xpath = "[XPathF] </keyref>	Tradotto con un costrutto $foreign\_key$. L'espressione $XPathS$ serve per determinare l'interface in cui inserire la chiave. L'espressione $XPathF$ determina gli attributi da includere nella chiave. L'attributo *refer* di valore [Rname] serve a determinare la chiave cui riferire la nuova $foreig\_key$

Tabella 3.7: Traduzione dei costrutti XML Schema in ODL_{I^3}

Nome	Costrutto XML Schema	Traduzione in ODL$_{I^3}$

Annotation	<xs:complexType name="[Cname]">
	<xs:annotation>
	<xs:documentation>
	...documentation.....
	</xs:documentation>
	<xs:appinfo>
	...appinfo.....
	</xs:appinfo>
	...
	</xs:annotation>
		interface Cname
		{
		const string annotation_1 "...documentation ...";
		...
		}

Model Group <all>	<xs:complexType name="[Gname]">
	<xs:all>
	<xs:element name="[Ename_1]" type="[Etype_1]"/>
	<xs:element name="[Ename_2]" type="[Etype_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="[Etype_1]"/>
	<xs:element name="[Ename_2]" type="[Etype_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] { attribue [Gname]_choice [Gname]_choice; } interface [Gname]_choice { attribute ... [Union_1]; } union { attribute ... [Union_2]; } union { attribute ... }

Tabella 3.8: Traduzione dei costrutti XML Schema in ODL$_{I^3}$

Capitolo 4

Implementazione del wrapper per XML Schema

Come già detto, lo scopo di questa tesi è la progettazione e lo sviluppo di un wrapper per il sistema MOMIS per l'integrazione di sorgenti in formato XML Schema. Al generico componente wrapper vengono richieste due funzionalità principali: quella di fornire una descrizione in formato ODL_{I^3} delle informazioni contenute in un sorgente e quella di consentire l'esecuzione delle query provenienti dal query manager sulla specifica sorgente. La prima funzionalità verrà realizzata implementando nel wrapper le regole di traduzione ricavate al capitolo 3, le quali permettono di tradurre uno schema XSD in uno schema ODL_{I^3}. La seconda funzionalità, per motivi di tempo, non è stata considerata nell'ambito di questa tesi, in cui si è preferito invece dare più spazio e attenzione al problema della traduzione: dunque, si lascierà l'implementazione di questa funzionalità ad uno sviluppo futuro del wrapper. Il wrapper, come tutto il sistema MOMIS, è stato sviluppato utilizzando il linguaggio java, perciò risulta portabile su diverse architettu-

re (teoricamente tutte quelle per cui esiste una java virtual machine diponibile) senza bisogno di ricompilazione.

Nel seguito saranno presentati alcuni diagrammi delle classi del sistema MOMIS che sono stati usati come riferimento per la progettazione e per la realizzazione del software. In particolare saranno presentati i diagrammi relativi al package di definizione del linguaggio ODL$_{I^3}$ (figure da 4.1 a 4.5) ed al package di definizione dei componenti wrapper (figure 4.6 e 4.7).

4.1 Package ODL$_{I^3}$

La figura 4.1 rappresenta la classe *MomisObject* che è alla base della gerarchia delle classi definite nel package java `it.unimo.dbgroup.momis.odli3` relativo alla definizione del linguaggio ODL$_{I^3}$. Da essa derivano le classi *TypeContainer*, *Attribute*, *Rule*, *Constant*, *Type* e *Case* che rappresentano i costrutti base del linguaggio e dai quali derivano tutte le altre classi del package.

La figura 4.2 rappresenta il diagramma delle classi che derivano da *TypeContainer*. 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 una singola sorgente, *Interface*, che rappresenta un generico tipo-classe, e *IntBody*, che rappresenta il corpo di una interface.

La figura 4.3 rappresenta la classe *Attribute*, la quale deriva dalla classe *MomisObject*, implementa l'interfaccia *AttributeInterface* e si specializza nelle sottoclassi *SimpleAttribute*, che rappresenta un attributo semplice, e *Relationship*, che invece rappresenta un attributo di tipo relazione.

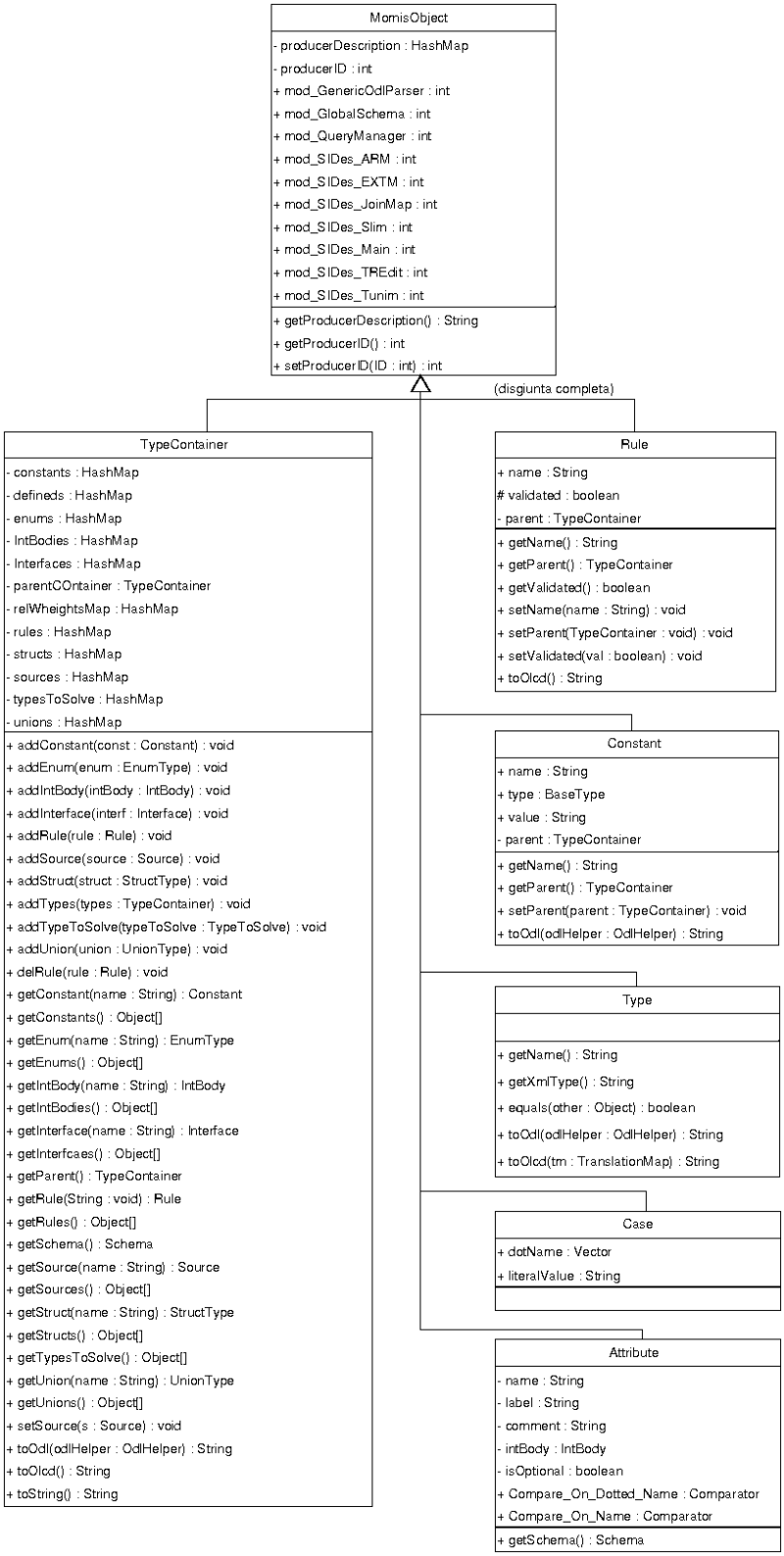

Figura 4.1: class diagram: MomisObject

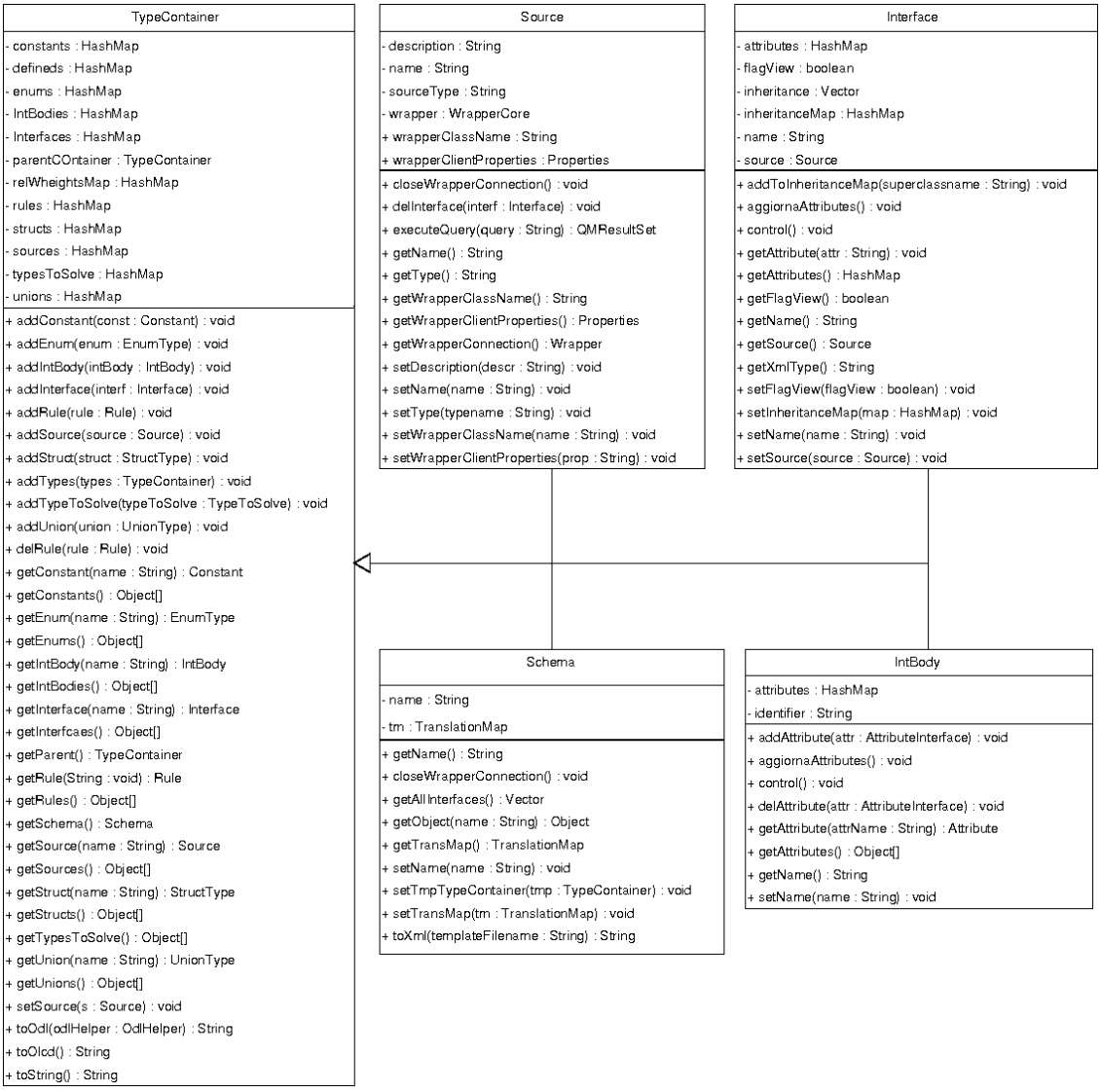

Figura 4.2: class diagram: TypeContainer

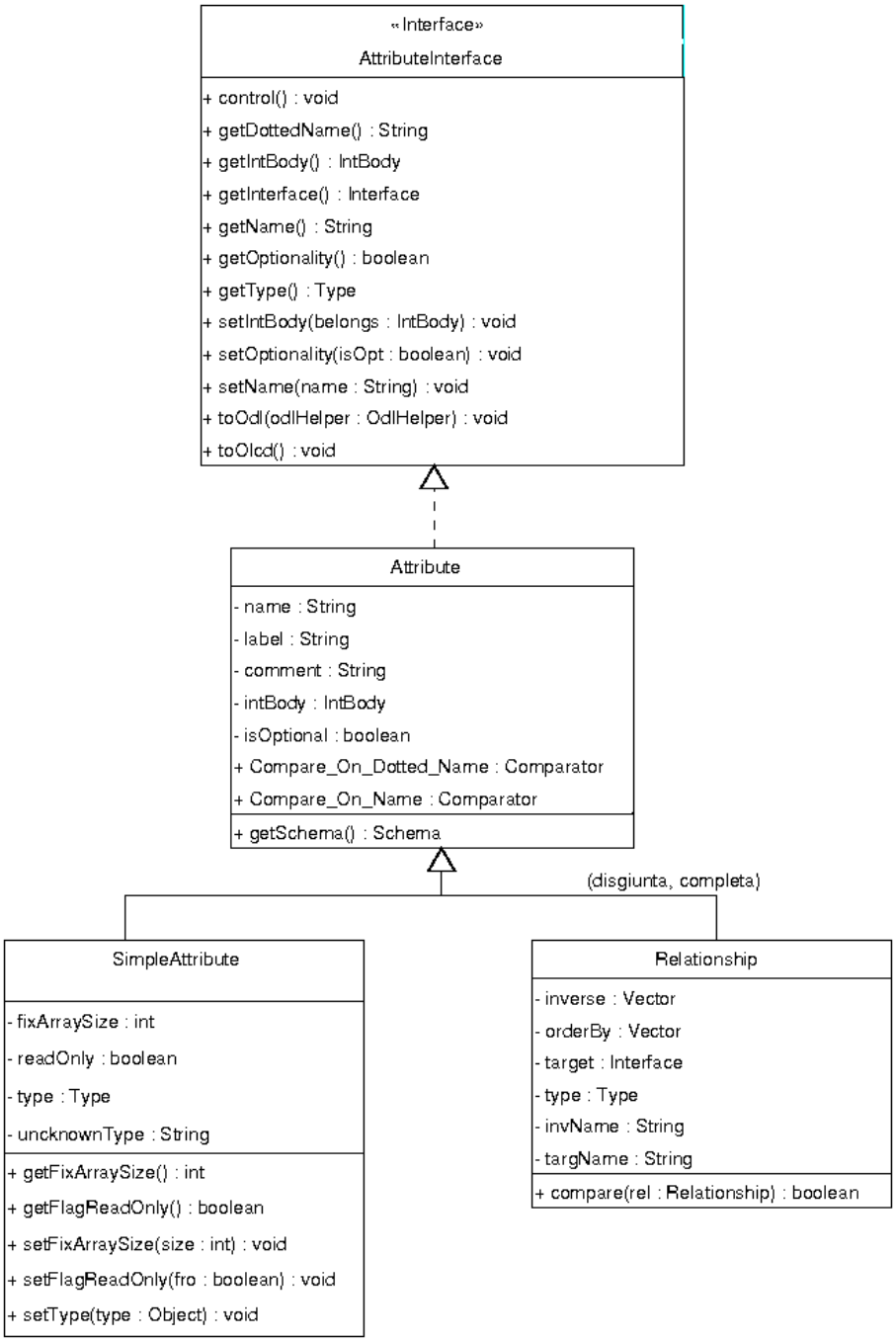

Figura 4.3: class diagram: Attribute

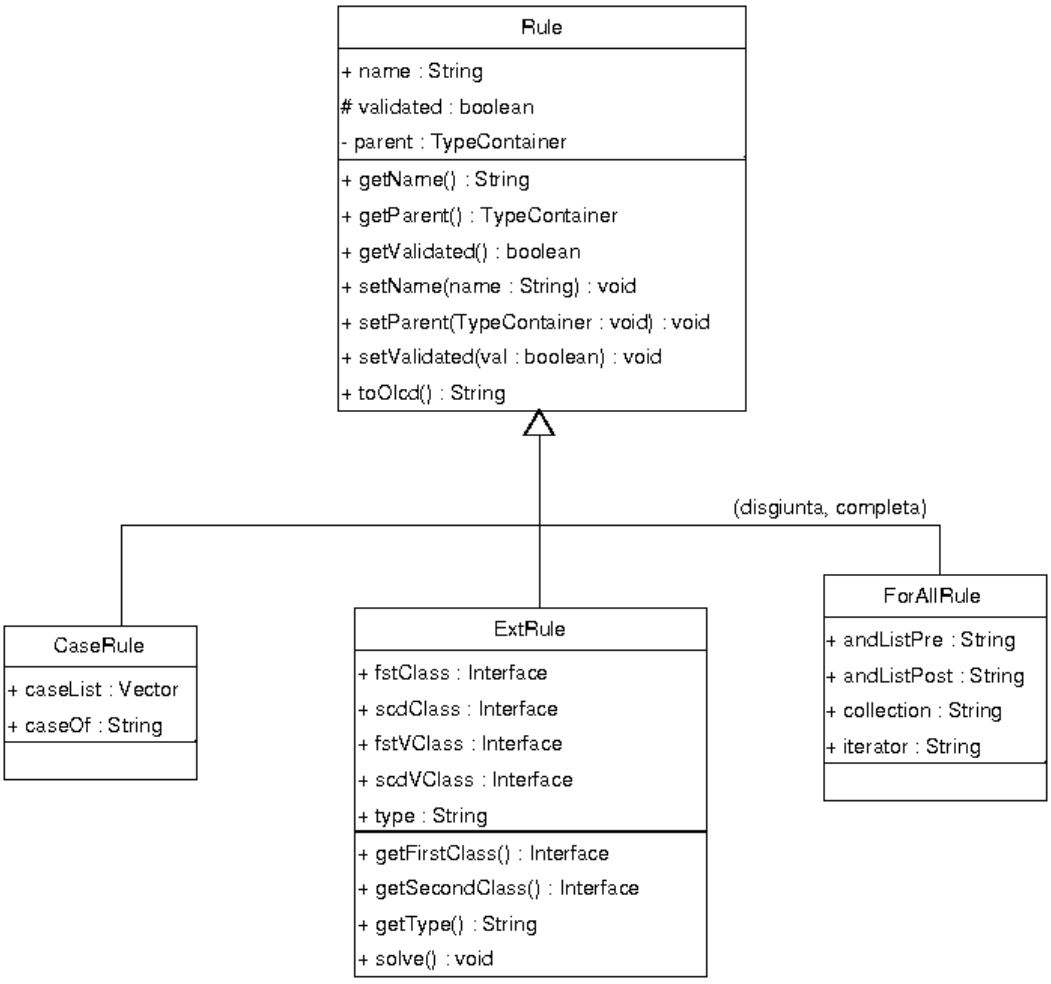

Figura 4.4: class diagram: Rule

La figura 4.4 rappresenta il diagramma della classe *Rule* che rappresenta una generica regola e si specializza nelle sottoclassi *CaseRule*, *ExtRule*, e *ForAllRule*. una regola di tipo ForAll.

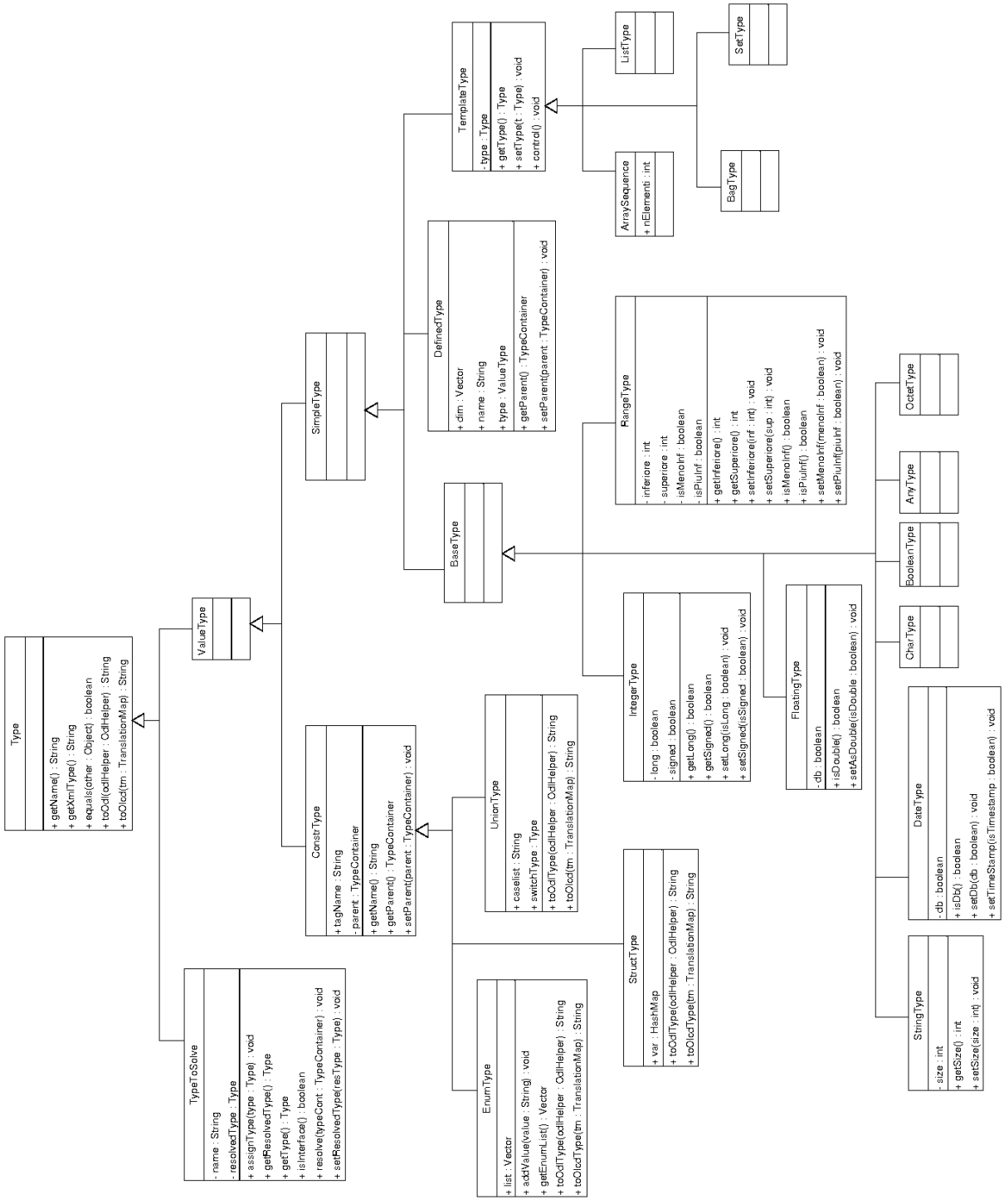

Figura 4.5: class diagram: Type

La figura 4.5 rappresenta tutti i tipi che si possono avere nel linguaggio ODL$_{I^3}$. La classe *Type* si specializza nelle sottoclassi *TypeToSolve* e *ValueType*: la prima rappresenta i tipi-classe, i quali, infatti, devono essere risolti in una fase di post-processing dopo l'acquisizione della sorgente; la seconda invece rappresenta in generale i tipi valore di ODL$_{I^3}$. Dalla classe *ValueType* derivano le classi *ConstrType* e *SimpleType*. La classe *ConstrType* rappresenta i tipi costrutti e si specializza nelle classi *EnumType*, *UnionType* e *StructType*, che rappresentano, rispettivamente, il tipo unione, enumerato e struttura. Dalla classe *SimpleType*, che rappresenta i tipi semplici, derivano le classi *BaseType*, *DefinedType* e *TemplateType*. La prima rappresenta i tipi base del linguaggio ODL$_{I^3}$ e si specializza nelle sottoclassi che rappresentano tutti i tipi predefiniti. La classe *DefinedType* rappresenta i tipi definiti in uno schema ODL$_{I^3}$ tramite il costrutto `typedef`. La classe *TemplateType*, infine, rappresenta i tipi collezione e si specializza nelle sottoclassi *ArraySequence*, *BagType*, *ListType* e *SetType* che corrispondono ai quattro tipi di collezione ammessi in ODL$_{I^3}$.

4.2 Package Wrapper

La figura 4.6 rappresenta l'interfaccia *WrapperCore*, che estende l'interfaccia *Wrapper*, e tutte la classi che implementano tale interfaccia e che sono i wrapper attualmente disponibili per il sistema MOMIS. Tra di esse è evidenziata la classe *WrapperXsdCore*, che corrisponde al wrapper per XML Schema di cui si è discusso in questa tesi.

La figura 4.7 rappresenta più in dettaglio la classe *WrapperXsdCore* con i suoi metodi e attributi. E' rappresentato il nuovo package creato, a cui tale classe appartiene, e la dipendenza dal package del linguaggio ODL_{I^3}, descritto in precedenza, e dal package `org.eclipse.xsd`[1], la libreria utilizzata per accedere alle strutture dati del documento XML Schema. Questa libreria, scritta in java, è stata creata nell'ambito del progetto EMF (**E**clipse **M**odeling **F**ramework), è distribuita come software open source e può essere utilizzata sia come per plug-in XSD per l'ambiente di sviluppo Eclipse, sia, in modalità *standalone*, come libreria per applicazioni esterne che abbiano necessità di accedere a documenti XML Schema. Essa implementa le specifiche "*XML Schema Infoset Api Requirements*" [10] e consente quindi di caricare documenti XSD e di accedere alla struttura dei componenti sulla base degli insiemi di proprietà descritti al capitolo 1 nascondendo i dettagli sul parsing del file.

[1] http://www.eclipse.org/xsd/

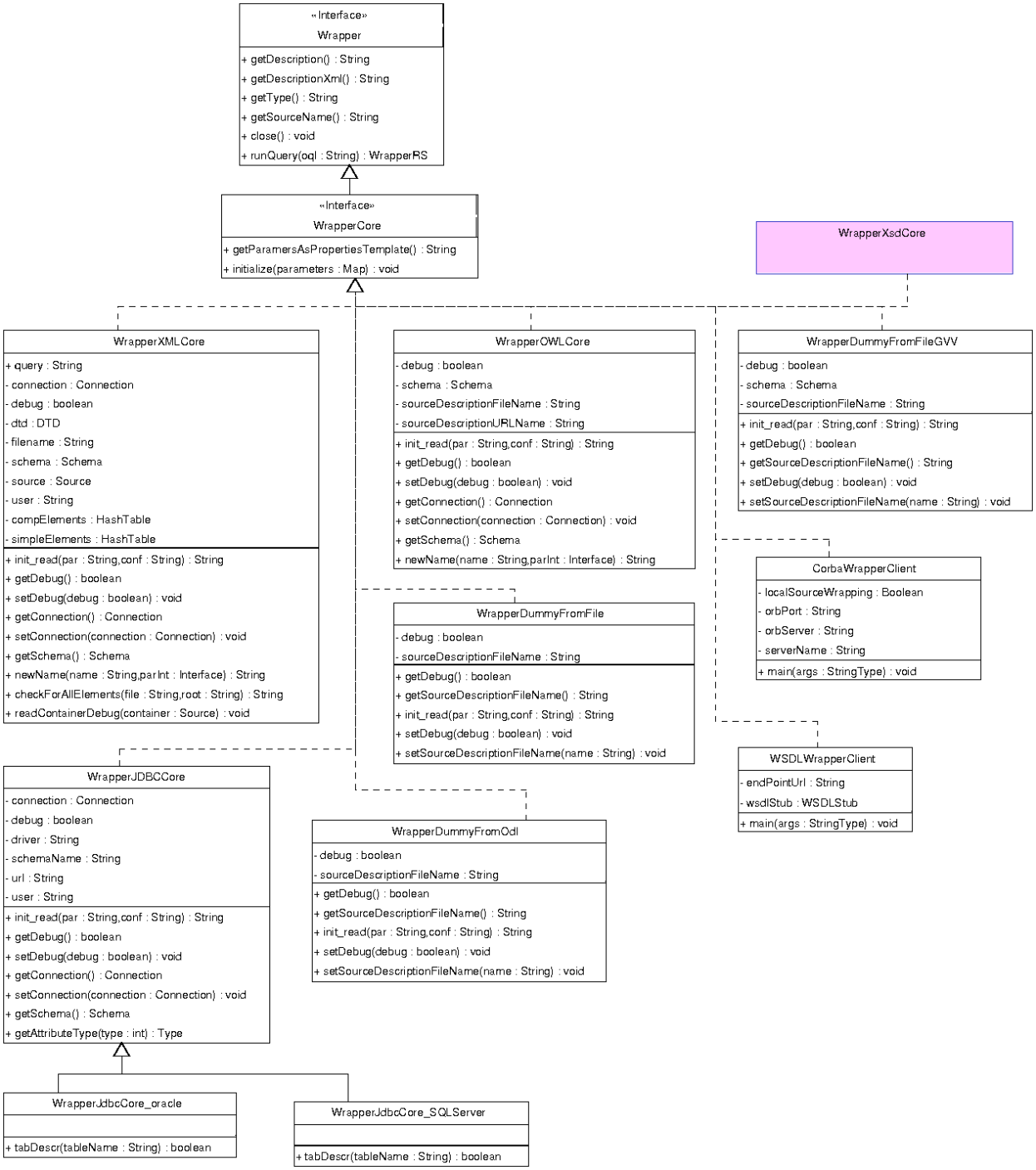

Figura 4.6: class diagram: Wrapper

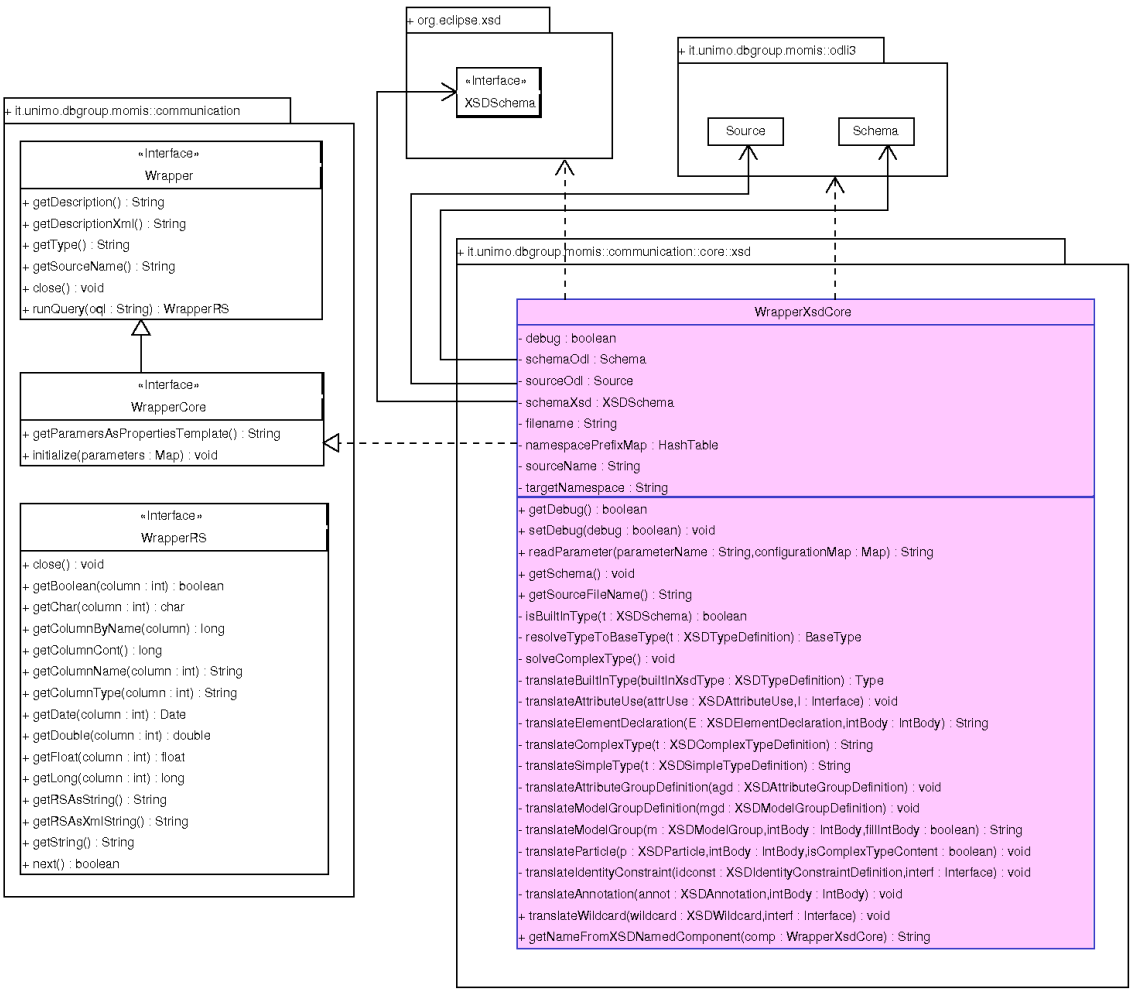

Figura 4.7: class diagram: Wrapper XML Schema

Capitolo 5

Conclusioni

In questa tesi è stato progettato e realizzato un wrapper per il sistema MOMIS per l'acquisizione automatica di sorgenti di dati in formato XML Schema. Il sistema MOMIS richiede che un wrapper sia in grado di prelevare ed interpretare le informazioni presenti in un certo tipo di sorgente (in questo caso di tipo XML Schema), e di tradurle nel linguaggio ODL_{I^3}, usato internamente al sistema per rappresentare tutte le sorgenti e per eseguire l'integrazione. Per raggiungere lo scopo prefissato dapprima si è studiata ed analizzata la sintassi e la semantica del linguaggio XML Schema. In seguito, tenendo presente i costrutti del linguaggio ODL_{I^3}, sono state ricavate un insieme di regole di traduzione per i diversi costrutti di XML Schema. Questo insieme di regole, che è riassunto nelle tabelle da 3.3 a 3.8 a partire da pag. 138, è finalizzato alla realizzazione di un wrapper per l'esportazione automatica di documenti XSD in schemi ODL_{I^3}. In seguito, infatti, è stato realizzato il modulo software che implementa sia le richieste del sistema per il componente wrapper che le specifiche di traduzione ricavate in precedenza. In questo modo sono state ampliate le funzionalità del sistema MOMIS, rendendo possibile l'integrazione anche di sorgenti in formato XML Schema .

Per ricavare le regole di traduzione è stato necessario confrontare la sintassi di XML Schema e di ODL$_{I^3}$: da questo esame sono emerse alcune differenze tra i due linguaggi. In particolare sono emerse due carenze principali del linguaggio ODL$_{I^3}$. La prima riguarda la mancanza del concetto di namespace in ODL$_{I^3}$. XML Schema fa uso dei namespace XML, uno strumento che consente di racchiudere un insieme di dichiarazioni all'interno di un contenitore, detto appunto namespace. L'uso dei namespace permette di definire oggetti diversi con lo stesso nome, purchè siano inseriti in contenitori diversi; principalmente i namespace vengono utilizzati per associare uno schema XSD ad una istanza, oppure per riutilizzare gruppi di dichiarazioni in documenti XML Schema diversi, in modo simile all'uso delle librerie nei linguaggi di programmazione. Il linguaggio ODL$_{I^3}$ non prevede l'utilizzo di namespace o di strumenti simili, ma le definizioni usate in una sorgente devono essere locali alla sorgente stessa. Possiamo dire che il linguaggio ODL$_{I^3}$ utilizza un modello di sorgente piatto, nel senso che tutti gli elementi di una sorgente devono essere inseriti nello stesso spazio dei nomi, ed i tipi degli attributi devono far riferimento a tipi classe o tipi valore definiti nella stessa sorgente. La seconda mancanza del linguaggio ODL$_{I^3}$ è l'impossibilità di associare agli attributi dei valori di default. Nel paragrafo 5.1 saranno proposte alcune significative estensioni al linguaggio ODL$_{I^3}$ relative a queste mancanze.

A livello generale si osserva una maggiore espressività del linguaggio XML Schema rispetto a ODL$_{I^3}$. Questa differenza può essere attribuita al fatto che il linguaggio XML Schema è stato progettato per descrivere la struttura di documenti XML, quindi semi-strutturati, per cui è necessario che consenta una certa flessibilità nella specifica delle strutture; si può dire che XML Schema cerca di applicare l'approccio object oriented ad un ambito semistrutturato. ODL$_{I^3}$ invece nasce dalla necessità di estendere il linguaggio ODL, a oggetti, per permettere l'integrazione di sorgenti di dati eterogenee. Queste sorgenti, in un primo momento, erano solo di tipo strutturato (a oggetti o relazionale).

Solo in un secondo momento si è affrontato il problema dell'integrazione di dati semi-strutturati (XML in particolare), per risolvere il quale si è reso necessario estendere il linguaggio con nuovi costrutti. Si può dire, quindi, che ODL$_{I^3}$ ha subito una evoluzione inversa a quella di XML Schema.

5.1 Proposte di estensione del linguaggio ODL$_{I^3}$

La prima proposta riguarda l'introduzione dei namespace in ODL$_{I^3}$. Questa estensione è molto complessa e coinvolge il linguaggio nella sua totalità. Quello che si vorrebbe poter fare è racchiudere le classi e i tipi definiti all'interno di una sorgente (intesa come sorgente ODL$_{I^3}$) in contenitori. A livello di sintassi si potrebbe aggiungere il costrutto *namespace*, come blocco all'interno del quale inserire gli altri costrutti. Si dovrebbe poi lasciare la possibilità di inserire una classe in un namespace o direttamente dentro la sorgente, realizzando così il concetto di default namespace. Da un lato si richiede quindi di poter associare ad una interfaccia, oltre al nome della sorgente, anche il nome del namespace; dall'altro lato, questo permetterebbe di poter definire classi omonime nella stessa sorgente, purchè appartenenti a namespace diversi. Si dovrebbe poi introdurre un meccanismo che consenta di dichirare nel nome del tipo di un attribute anche il path per recuperare quel particolare tipo all'interno di un certo namespace, come ad esempio;

```
attribute nome_sorgente.nome_namespace.nome_tipo nome_attribute;
```

L'altra estensione riguarda la possibilità di dichiarare un valore di default per un certo tipo di dato, da utilizzare ogni volta che ad un attributo non è assegnato nessun valore. Questo riguarda principalmente i tipi valore, per i quali ha sicuramente più significato il concetto di valore di default. E' necessario aggiungere una proprietà `default` al costrutto `typedef`, o al costrutto `attribute`, per poter dichiarare, ad esempio:

```
typedef integer num_elementi ( default = 1 );

attribute boolean myAttr (default = false);
```

o anche:

```
typedef integer num_elementi : 1;

attribute boolean myAttr : false;
```

5.2 Lavoro futuro

Oltre alle proposte di estensione del linguaggio ODL$_{I^3}$, si può identificare un possibile sviluppo futuro per il wrapper realizzato nell'implementazione delle funzionalità e del modulo che consentono di effettuare interrogazioni sulle sorgenti acquisite.

Un altro possibile lavoro futuro potrebbe essere l'implementazione del wrapper inverso, quello cioè che effettua la traduzione di uno schema ODL$_{I^3}$ in un documento XML Schema.

Ancora, tra i lavori futuri aggiungiamo quello di testare più approfonditamente il wrapper, magari in un caso concreto di utilizzo, considerando anche la sua validità ai fini dell'integrazione.

Appendice A

Sintassi ODL$_{I^3}$

In questa appendice si riporta la sintassi BNF del linguaggio ODL$_{I^3}$. Essendo ODL$_{I^3}$ un'estensione del linguaggio standard ODL, si riportano solo le parti che differiscono dall'ODL originale, rimandando invece a quest'ultimo per le parti in comune.

```
<interface_dcl>       ::=  <interface_header> { [<interface_body>] };
<interface_header>    ::=  interface <identifier>
                           [<inheritance_spec>]
                           [<type_property_list>]
<inheritance_spec>    ::=  : <scoped_name> [,<inheritance_spec>]
<type_property_list>  ::=  ( [<source_spec>] [<extent_spec>]
                           [<key_spec>] [<f_key_spec>])
<source_spec>         ::=  source <source_type> <source_name>
<source_type>         ::=  relational | nfrelational | object | file
<source_name>         ::=  <identifier>
```

```
<extent_spec>        ::=  extent <extent_list>

<extent_list>        ::=  <string> | <string> , <extent_list>

<key_spec>           ::=  key[s] <key_list>

<f_key_spec>         ::=  foreign_key <f_key_list>

<relationships_list>::=   <relationship_dcl>; |
                          <relationship_dcl> ; <relationships_list>

<relationships_dcl> ::=   <local_attr_name> <relationship_type> <local_attr_name>

<relationship_type> ::=   syn | bt | nt | rt

<rule_list>          ::=  <rule_dcl>; | <rule_dcl>; <rule_list>

<rule_dcl>           ::=  rule <identifier> <rule_pre> then <rule_post>

<rule_pre>           ::=  <forall> <identifier> in <rule_identifier> : <rule_body_list>

<rule_identifier>    ::=  <identifier> | ( <identifier> and <identifier> )

<rule_post>          ::=  <rule_body_list>

<rule_body_list>     ::=  ( <rule_body_list> ) |<rule body> |
                          <rule_body_list> and <rule_body> |
                          <rule_body_list> and ( <rule_body_list> )

<rule_body>          ::=  <dotted_name> <rule_const_op> <literal_value> |
                          <dotted_name> <rule_const_op> <rule_cast> <literal_value> |
                          <dotted_name> in <dotted_name> |
                          <forall> <identifier> in <dotted_name> : <rule_body_list> |
                          exists <identifier> in <dotted_name> : <rule_body_list>

<rule_const_op>      ::=  = | ≥ | ≤ | > | <

<rule_cast>          ::=  ( <simple_type_spec> )

<dotted_name>        ::=  <identifier> | <identifier>.<dotted_name>

<forall>             ::=  for all | forall
```

Bibliografia

[1] R. Hull and R. King et al. Arpa i^3 reference architecture, 1995. Available at http://www.isse.gmu.edu/I3_Arch/index.html.

[2] D. Beneventano, S. Bergamaschi, F. Guerra, and M. Vincini. The MOMIS approach to information integration. In *AAAI International Conference on Enterprise Information Systems (ICEIS 2001)*, 2001.

[3] A. Corni. *Intelligent Information Integration: The MOMIS Project.* Tesi di Dottorato di Ricerca, Università di Bologna, 2000.

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

[5] Henry Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML schema part 1: Structures second edition. Technical Report EDIINFRR0645, The University of Edinburgh, October 2004.

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

[7] R. G. G. Cattell, editor. *The Object Database Standard: ODMG93.* Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[8] R. Cattel. *The Object Data Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[9] R. G. G. Cattell and Douglas K. Barry. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[10] Bob Schloss. XML schema infoset API requirements. Technical report, IBM, February 2002.