Università degli Studi di Milano
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Specialistica in Tecnologie
dell'Informazione e Comunicazione

Uno strumento grafico per l'evoluzione degli schemi XML e l'adattamento dei documenti

Relatore:
Dott. Marco MESITI
Correlatore:
Prof.ssa Giovanna GUERRINI

Candidato:
Matteo Alberto SORRENTI
matricola 670789

Anno Accademico 2005–2006

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

ELENCO DEI LISTATI

ELENCO DEGLI ALGORITMI

Introduzione

Negli ultimi anni l'eXtensible Markup Language (XML) si è affermato come formato standard per l'interscambio dei dati. XML è un linguaggio di marcatura con il quale l'utente può definire un insieme di tag. Questi non hanno la funzione di specificare il modo in cui il documento verrà visualizzato, come avviene per altri linguaggi, ma servono per definire una significativa struttura semantica del documento. Per definizione, XML è quindi un linguaggio dotato di un insieme di regole, di linee guida e convenzioni utili a produrre file di testo non ambigui che, per loro natura, evitano la mancanza di estensibilità, l'assenza di un supporto per la localizzazione e la dipendenza da una determinata piattaforma.

La popolarità e le svariate soluzioni per l'implementazione di XML aumentano di giorno in giorno. Data l'importanza che ha assunto con il passare degli anni questo linguaggio cresce costantemente anche la necessità di avere a disposizione sempre un maggior numero di strumenti per poterne sfruttare appieno le innumerevoli potenzialità.

Originariamente la struttura di un documento XML veniva specificata attraverso le Document Type Definition (DTD), ma successivamento sono stati introdotti gli XML Schema che, essendo essi stessi dei documenti XML, ne ereditano tutte le caratteristiche. Inoltre gli schemi consentono una tipizzazione forte e più raffinata, fornendo peraltro un insieme di tipi di dato compatibile con quelli usati nelle basi di dati tradizionali. In più, permette di esprimere sequenze di sottoelementi in cui non conta l'ordine (`all`), consente di definire nei documenti XML più elementi con lo stesso nome ma con contenuto diverso, permette la definizione di elementi con contenuto vuoto (`nil`) ed elementi sostituibili, in più è possibile indicare la cardinalità ammessa per le occorrenze dei vari sottoelementi.

Risulta inevitabile che sia i dati che le strutture dei documenti tendano continuamente a cambiare. Questo è dovuto ad una moltitudine di fattori, tra cui: la necessità di adattare il sistema a nuove situazioni introducendo nuove funzionalità e permettendo di gestire nuovi tipi di dato. È naturale che le alleanze commerciali cambino e si espandano o che cambi il dominio applicativo a cui si riferisce un documento: è il mondo reale che evolve nel tempo, pertanto la sua rappresentazione deve potersi adeguare. Un altro aspetto che incide sulla necessità di modificare la struttura dei dati è legato al fatto che in molti contesti

si sceglie di usare XML come formato per la rappresentazione dei dati e occorre specificare schemi adeguati a tale contesti. Spesso, però, prima che una proposta venga ufficialmente adottata come unico standard, ne vengono formulate differenti versioni. Generalmente è indispensabile mantenere la compatibilità sia con le diverse varianti esistenti, sia con le vecchie versioni che non son state ancora identificate come obsolete.

Si rende quindi necessario affrontare l'esigenza di evolvere gli standard così come è importante permettere a standard simili di interoperare tra loro. Ed il nostro lavoro si è concentrato proprio sulle modifiche agli schemi e le rispettive ripercussioni sui documenti validi per tali schemi. Gli schemi XML possono essere modificati nelle loro componenti base: le dichiarazioni di elementi, le definizioni di tipi semplici e di tipi complessi. Quando un insieme di documenti è associato ad uno schema, la modifica dello schema può compromettere la validità dei documenti e, di conseguenza, delle applicazioni che operano sui documenti validi per tale schema.

Utilizzare i vari prodotti presenti sul mercato per determinare se una modifica allo schema ha alterato la validità dei documenti è un lavoro molto costoso, in termini di tempo, ed in alcuni casi inutile, dal momento che ci sono operazioni di modifica dello schema che non influenzano la validità dei documenti. I tool esistenti invece si limitano sempre ad effettuare una verifica sull'intero contenuto dei documenti.

Occorrono quindi tecniche che permettano la rivalidazione dei documenti tenendo in considerazione il fatto che i documenti sono validi per lo schema originario e molte operazioni di modifica dello schema possono alterare solo piccole parti dello schema e dei relativi documenti. Un altro problema da affrontare è identificare un approccio per modificare consistentemente i documenti in modo da renderli validi per il nuovo schema. Le modifiche da apportare devono essere minimali in modo da limitare i danni dovuti alla modifica del patrimonio informativo contenuto nei documenti. L'esecuzioni manuale di tali modifiche sui documenti sono di difficile applicazione e possono facilmente introdurre errori e inconsistenze. Per tale motivo occorrono approcci semi-automatici per adattare i documenti al nuovo schema.

In precedenti tesi da una parte sono state studiate le operazioni di modifica degli schemi e gli effetti che possono comportare sulla validità dei documenti, mentre dall'altra è stata sviluppata un'interfaccia grafica per visualizzare schemi e documenti XML. A partire da questi lavori, in questa tesi, innanzitutto si sono analizzati e verificati gli operatori studiati oltre a correggere errori presenti nell'interfaccia grafica in modo renderla perfettamente funzionante. Dopodiché si sono affrontati i seguenti punti:

- Implementazione ed integrazione degli operatori di evoluzione nell'interfaccia grafica per la visualizzazione di schemi e documenti XML.
- Sviluppo di un algoritmo per la validazione incrementale dei documenti XML a seguito di modifiche allo schema.
- Sviluppo di un algoritmo per l'adattamento semi-automatico di documenti validi per lo schema originario al nuovo schema.
- Valutazione sperimentale dell'algoritmo di validazione incrementale e di adattamento semi-automatico di documenti.

Si sono sviluppati ed integrati, all'interno dell'interfaccia grafica, gli operatori di evoluzione degli schemi che in precedenza erano stati studiati solo dal punto di vista teorico. Dopo l'esecuzione di una primitiva di evoluzione viene verificata la validità ed effettuata la ristrutturazione dei documenti solo all'interno dei sottoalberi che sono interessati alla modifica e non in tutto il documento XML. Per la ristrutturazione dei documenti si è studiato e sviluppato un algoritmo apposito che, in base ai nodi presenti nel documento ed alla struttura prevista nello schema, effettua le opportune modifiche aggiungendo o rimuovendo il minor numero possibile di elementi. Il fatto di operare una verifica solo su una parte dell'albero XML ci ha permesso di avere prestazioni migliori rispetto agli altri approcci correntemente proposti dalla comunità scientifica e industriale perché essi effettuano una verifica sull'intero documento e non tengono in considerazione il fatto che il documento era valido per lo schema orginario e se solo una parte dello schema è oggetto di una modifica allora solo una parte del documento dovrà essere analizzata.

Una demo del software sviluppato (denominato XEvolution) è stata presentata alla conferenza europea di basi di dati EDBT 2006 (International Conference on Extending Database Technology) lo scorso marzo ricevendo commenti più che positivi. I risultati di questa tesi sono stati raccolti in un rapporto tecnico [5] e sottomessi a conferenza.

Il lavoro svolto può aprire le porte verso nuove ed interessanti direzioni di ricerca. Si può sviluppare un linguaggio di interrogazione che permetta di popolare i nuovi elementi introdotti nello schema con valori già presenti nei documenti validi per lo schema originario (attualmente sono stati predefiniti dei valori di default). Inoltre si vuole considerare la possibilità di spostare dichiarazioni di elementi da una parte all'altra dello schema e spostare di conseguenza gli elementi nei documenti. In più c'è l'intenzione di integrare, nell'interfaccia sviluppata, meccanismi di controllo dell'accesso per documenti XML e verificare gli effetti dell'evoluzione dello schema sulle politiche di controllo dell'accesso definite su schemi e documenti.

Organizzazione della tesi Nel Capitolo 1 sono presentati i documenti XML con gli XML Schema (utilizzati per validare i documenti) e vengono descritte, in modo generale, le caratteristiche principali di questi linguaggi. Si parte dalla nascita di dei documenti XML per passare poi ai documenti ben formattati ed al il problema della validazione. Riguardo agli schema vengono discussi i concetti di tipo nativo e di tipo seemplice, illustrando i meccanismi che consentono di definire un tipo derivato. Inoltre vengono presentati gli elementi (sia locali che globali) ed i tipi complessi, per finire con problema del conflitto dei nomi.

Il Capitolo 2 descrive gli alberi etichettati, la nostra rappresentazione formale dei documenti XML e degli schemi includendo, per questi ultimi, la rappresentazione dei tipi semplici e dei tipi complessi. Sono stati introdotti anche simboli e notazioni che verrano utilizzati nei capitoli successivi per la descrizione delle primitive di evoluzione e degli algoritmi ideati.

Nel Capitolo 3 sono presentate le primitive di evoluzione proposte per apportare modifiche ad uno schema e ne viene specificata la semantica. Queste primitive permettono di apportare una qualsiasi modifica allo schema e, se lo schema originale è ben formato, si garantisce la correttezza dello schema modificato.

Nel Capitolo 4 viene introdotto ed analizzato il problema della rivalidazione e della ristrutturazione dei documenti XML. Vengono presentati e descritti gli algoritmi da noi ideati per risolvere il problema includendo esempi e spiegazioni per le scelte che sono state prese.

Il Capitolo 5 presentata l'interfaccia grafica di XEvolution con una veloce descrizione del linguaggio C# (dell'ambiente .NET Framework di Microsoft®) dal momento che è stato utilizzato tale linguaggio per l'implemetazione dell'interfaccia e degli algoritmi.

Nel Capitolo 6 sono presentate i risultati sperimentali generati dal confronto tra i nostri algoritmi e l'algoritmo di validazione *brute-force* offerto dal .NET Framework.

Infine nel Capitolo 7 sono presenti le conclusioni tratte dal lavoro di tesi ed i possibili sviluppi futuri che vi possono essere.

CAPITOLO 1

Documenti e schemi XML

In questo capitolo viene fornita una semplice descrizione dei documenti XML, dei meccanismi che sono alla base del funzionamento degli schemi XML (o XML Schema). Il contenuto del capitolo che riguarda XML Schmema è estratto da *XML Schema Part 0: Primer Second Edition* disponibile in rete all'indirizzo `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`.

Il capitolo è strutturato come segue. Nelle prime tre sezioni paragrafi vengono trattati i documenti XML. In particolar modo nella Sezione 1.1 viene introdotta brevemente la nascita del meta-linguaggio XML, nella Sezione 1.2 sono descritti i componenti che si trovano all'interno di un documento XML (presentando brevemente alcuni linguaggi che servono ad arricchirlo) mentre nella Sezione 1.3 vengono descritte le caratteristiche dei documenti ben formattati e la validazione di tali documenti. Nella sezione 1.4 viene presentato il linguaggio XML Schema, invece nelle sezioni 1.5 e 1.6 sono descritti rispettivamente i tipi semplici ed i tipi complessi di XML Schema includendo in questi ultimi anche gli elementi (sia globali che locali).

1.1 La nascita del linguaggio XML

Contrariamente a quanto comunemente si pensi, l'eXtensible Markup Language (XML) non è l'ennesimo linguaggio di markup e nemmeno l'evoluzione dell'ormai vecchio, ma sempre vivo, Hypertext Markup Language (HTML). Esso è un meta-linguaggio, cioè un linguaggio che permette di definire altri linguaggi di marcatura. A differenza di HTML, XML non ha tag predefiniti e non serve per definire pagine Web ne per programmare. Esso serve esclusivamente per descrivere altri linguaggi. In realtà, XML di per se non è altro che un insieme standard di regole sintattiche per modellare la struttura di documenti. Questo insieme di regole, dette più propriamente specifiche, definiscono le modalità per creare un proprio linguaggio di markup. Le specifiche ufficiali sono state definite dal World Wide Web Consortium (W3C) e sono consultabili a partire dall'indirizzo `http://www.w3.org/XML`.

Internet ed il Web, per la loro stessa natura non centralizzata ed aperta a macchine e persone diverse, hanno bisogno di standard per poter essere utilizzabili e per poter evolvere senza forzature. A questo scopo è stato istituito il W3C nel dicembre del 1994 con l'obiettivo di definire standard accettati dai maggiori produttori di software per il Web, primo fra tutti il linguaggio HTML. Tuttavia, l'assalto commerciale ad Internet degli anni '90 e la rapida diffusione del Web hanno scatenato una delle lotte più agguerrite sul piano tecnico e commerciale: la guerra dei browser tra Netscape®e Microsoft®. Ciascun contendente introduceva, con ogni nuova versione del proprio browser, una estensione proprietaria di HTML ufficiale. Il risultato di tale battaglia era che un sito Web che voleva utilizzare le estensioni proprietarie di un browser rischiava di risultare inaccessibile agli altri browser. La situazione peggiorò con l'introduzione dell'HTML dinamico le cui implementazioni erano quasi totalmente proprietarie.

In questo panorama il W3C era costretto a rincorrere le evoluzioni de facto dell'HTML e doveva scegliere quali caratteristiche standardizzare e quali invece lasciare fuori dalle specifiche ufficiali di HTML. In questo contesto cominciò a delinearsi la necessità di un linguaggio di markup che offrisse maggiore libertà nella definizione dei tag pur rimanendo nell'ambito del rispetto di uno standard. Fu così che nel 1996 si costituì l'XML Working Group nell'ambito del W3C. Lo scopo del gruppo di lavoro era quello di definire un linguaggio che salvasse gli standard e offrisse libertà di estensione. La ricerca partì, come era già accaduto in passato per HTML, dal linguaggio Standard Generalized Markup Language (SGML), un meta-linguaggio per la definizione di linguaggi di markup. Questo linguaggio risultava però troppo complesso per gli scopi della ricerca e pertanto fu snellito da alcune caratteristiche e semplificato in alcuni punti per renderlo adatto allo scopo. Nel dicembre '97 le specifiche di XML venivano pubblicate come Proposed Recommendation.

Tuttavia, anche se gli obiettivi iniziali della nascita di XML erano rivolti alla soluzione di un problema di standard per il Web, ben presto ci si accorse che XML non era limitato al solo contesto Web. Esso risultava abbastanza generale per poter essere utilizzato nei più disparati contesti: dalla definizione della struttura di documenti allo scambio di informazioni tra sistemi diversi, dalla rappresentazione di immagini alla definizione di formati di dati. Questo aspetto rappresentava una rivoluzione. Si cominciò difatti ad applicare XML in numerosi campi e per vari scopi: archiviazione elettronica e gestione dei contenuti documentali, pubblicazione su web, scambio di documenti elettronici, formati interni per strumenti software, commercio elettronico ed in molti altri campi troppo numerosi per essere tutti menzionati in questa sede.

1.2 I documenti XML

Le regole di XML consentono di definire la struttura di documenti ma non altre caratteristiche come il tipo o la presentazione dei dati. Questo compito non è di XML ma è delegato ad altri standard, alcuni dei quali sono basati sullo stesso XML, a dimostrarne la flessibilità e l'universalità. Qui di seguito alcuni esempi:

- Document Type Definition (DTD): linguaggio utile a definire le componenti ammesse nella costruzione di un documento XML. Tramite i DTD si è in

grado di definire gli elementi leciti all'interno di un documento, la struttura di ogni elemento, gli attributi per ogni elemento e quali valori questi ultimi possono o devono assumere.

- XML Schema: linguaggio che serve per descrivere il contenuto di un documento XML con lo scopo di delineare quali elementi sono permessi, quali tipi di dati sono ad essi associati e quale struttura hanno fra loro gli elementi contenuti nel file XML. A differenza del DTD il linguaggio XML Schema è molto più adatto alla definizione dei tipi degli elementi e di contenuti fortemente strutturati. La sua trattazione sarà approfondita più avanti

- Cascading Style Sheets (CSS): è un linguaggio di stile che permette di dare una formattazione ad un documento XML o HTML. Consente di controllare la presentazione dei dati.

- eXtensible Stylesheet Language (XSL): è una famiglia di raccomandazioni per definire trasformazione e presentazione dei documenti XML. Consiste di tre parti:

  - XSL Transformations (XSLT): un linguaggio per le trasformazioni. Consente di trasformare un documenti XML in altri formati.
  - XML Path Language (XPath): un linguaggio di espressione usato per accedere o riferirsi a parti di un documento XML. Definisce espressioni e metodi per formattare in maniera precisa un oggetto trasformato.
  - XSL Formatting Objects (XSL-FO): un vocabolario XML per la specifica semantica di formattazione. Usato per formattare in maniera precisa un oggetto trasformato.

- XML Linking Language (XLink): definisce dei collegamenti tra risorse utilizzando XPath.

- XML Pointer Language (XPointer): ideato per indicizzare precisi punti o porzioni di un documento XML.

- XML Query (XQuery): linguaggio simile per funzionalità allo Structured Query Language (SQL). Si rende utile quando si ha la necessità di recuperare agevolmente le informazioni da documenti e basi di dati XML. Consente di estrarre informazioni secondo determinati criteri.

XML è dunque un meta-linguaggio per definire la struttura di documenti e viene utilizzato nella sua accezione più ampia di contenitore di informazioni. Concretamente, un documento XML è un semplice file di testo che contiene una serie di tag, attributi e contenuti secondo regole sintattiche ben definite. Un documento XML è intrinsecamente caratterizzato da una struttura gerarchica. Esso è composto da componenti denominati elementi. Ciascun elemento rappresenta un componente logico del documento e può contenere altri elementi (sottoelementi) o del testo.

Quest'ultimi, definiti anche nodi, possono avere associati degli attributi il cui scopo è quello di contenere informazioni aggiuntive indicanti le proprietà

dell'elemento. L'organizzazione degli elementi segue un ordine gerarchico che prevede un elemento principale, chiamato root element oppure semplicemente root o radice. Tale nodo contiene l'insieme degli altri elementi del documento. È quindi possibile rappresentare graficamente la struttura di un documento XML tramite un albero, generalmente noto come document tree.

Listato 1.1: movie.xml

```
<?xml version="1.0" encoding="utf-8"?>
<movies>
  <movie>
    <title>Il laureato</title>
    <cast>
      <actor>
        <sex>M</sex>
        <name>Dustin Hoffman</name>
      </actor>
      <actor>
        <sex>F</sex>
        <name>Anne Bancroft</name>
      </actor>
    </cast>
    <release-date>1967-01-01</release-date>
    <genre>drammatico</genre>
    <rating>8</rating>
  </movie>
  <movie>
    <title>Amarcord</title>
    <cast>
      <actor>
        <sex>M</sex>
        <first-name>Ciccio</first-name>
        <last-name>Ingrassia</last-name>
      </actor>
      <actor>
        <sex>F</sex>
        <name>Magali Noel</name>
      </actor>
    </cast>
    <release-date>1974-01-01</release-date>
    <genre>commedia</genre>
    <short-description>...</short-description>
    <rating>8</rating>
  </movie>
</movies>
```

La struttura di un documento XML dipende dalle scelte progettuali difatti non esistono regole universali per l'organizzazione logica degli elementi all'interno del documento se non il buon senso e l'esperienza. Un esempio di documento XML si può osservare dal Listato 1.1

Analizzando il contenuto del documento si nota come la prima riga serva ad identificarlo come un documento XML specificandone la versione (in questo caso la 1.0) e la codifica utilizzata per i caratteri (in questo caso utf-8).

1.3 Documenti ben formati e documenti validi

Un documento XML viene definito ben formato se rispetta i vincoli illustrati nella specifica del W3C. In particolare un documento XML deve avere l'elemento radice, cioè un elemento che contiene tutti gli elementi del documento; i marcatori devono essere bilanciati, cioè ad ogni marcatore di apertura deve corrisponderne uno di chiusura ed i nomi devono corrispondere (fatta eccezione per il caso di elemento vuoto dove non è presente il tag di chiusura ed il tag di apertura ha la particolare rappresentazione "`/ >`").

XML prevede inoltre degli oggetti speciali, detti entità, che consentono di sostituire altri caratteri. Il meta-linguaggio definisce cinque entità predefinite e permette inoltre all'utente di descriverne altre nel documento. Le entità predefinite sono:

- `&` per definire `&`
- `<` per definire `<`
- `>` per definire `>`
- `&apos;` per definire `'`
- `&quot;` per definire `"`

Un documento XML viene definito valido rispetto ad uno schema se è un documento ben formato e se rispetta i vincoli descritti nello schema. Per verificare la validità di un documento XML ocorre processare il documento. Il controllo prima verifica se la radice del documento XML ha un contenuto valido, e poi esamina la conformità di ogni suo sottoelemento, procedendo in questa maniera fino a che l'intero documento è stato processato con successo o fino a quando si riscontra una scorrettezza.

Se un elemento ha tipo semplice, dopo aver verificato che l'elemento non contiene sottoelementi o attributi, viene effettuato il controllo sulle regole del tipo semplice. Questo talvolta comporta un'analisi della sequenza di caratteri rispetto ad un espressione regolare oppure ad un enumeration, e talvolta comporta il controllo che la sequenza di caratteri rappresenti un valore di un determinato insieme.

1.3.1 La validazione di documenti XML

Il linguaggio XML offre la libertà di definire i tag a seconda delle necessità; affinché non si generi confusione si rende implicitamente necessario un meccanismo che vincoli l'utilizzo degli elementi all'interno dei documenti stabilendo quali tag possono essere utilizzati per rispecchiare una struttura logica predefinita.

In altre parole si ha il bisogno di definire una grammatica per il linguaggio di markup ideato. Quest'ultima viene vista come un insieme di regole che indica quali vocaboli (elementi) possono essere utilizzati e con che struttura è possibile comporre frasi (documenti). Sostanzialmente una grammatica definisce quindi uno specifico linguaggio di markup e se un documento XML rispetta le regole in essa definite viene detto valido per un particolare linguaggio.

La caratteristica di documento valido si affianca a quella di documento ben formato per costruire documenti XML adatti ad essere elaborati automaticamente. C'è da sottolineare che un documento ben formato può non essere valido rispetto ad una grammatica, mentre un documento valido è necessariamente ben formato. Tra l'altro, un documento valido per una grammatica può non essere valido per un'altra.

Attualmente gli approcci più diffusi per la creazione di grammatiche per documenti XML sono i DTD e l'XML Schema.

Un documento XML può essere all'origine di diversi tipi di elaborazione come la generazione di altri documenti (eventualmente in formati diversi), il controllo delle impostazioni di programmi e la rappresentazione di immagini. Tutti i possibili impieghi di XML, però, si fondano su due tipi di elaborazione preliminare: la verifica che un documento sia ben formato e la sua validità rispetto ad una grammatica.

I software che si occupano di queste elaborazioni sono detti tecnicamente parser e sono degli strumenti standard disponibili sulle diverse piattaforme. È possibile suddividere i parser in due categorie (talvolta può essere lo stesso parser che assume due ruoli):

- *parser non validante*: è un parser che verifica soltanto se un documento è ben formato ossia se è corretto dal punto di vista sintattico;
- *parser validante*: è un parser che, oltre a controllare che un documento è ben formato, verifica se è corretto rispetto ad una data grammatica.

Ora che sono state descritte brevemente le peculiarità dei documenti XML verranno analizzate e discusse le caratteristiche della tecnologia utilizzata dall'interfaccia per la creazione delle grammatiche: l'XML Schema.

1.4 Caratteristiche del linguaggio XML Schema

XML Schema serve per rappresentare la struttura di un documento XML. Uno schema descrive il contenuto di una classe di documenti XML, infatti spesso viene utilizzato il termine *istanza di uno schema* per indicare un documento XML che sia conforme aa un determinato schema.

Grazie ad XML Schema si possono definire:

- quali elementi/attributi possono apparire in un documento;
- le relazioni di parentela tra i vari elementi;
- l'ordine dei sottoelementi;
- quando un elemento è vuoto o può contenere testo;
- il tipo dei vari elementi/attributi.

XML Schema ha una tipizzazione più raffinata rispetto a quella offerta dai DTD (Document Type Definition), fornendo un insieme di tipi di dato compatibile con quelli usati nelle basi di dati tradizionali; supporta i tipi di dato lista e consente di specificare restrizioni sui dati ad esempio tramite vincoli sui pattern e sulle enumerazioni. Inoltre, XML Schema usa la sintassi di XML e, proprio

per questo, è estendibile; inoltre permette di riusare uno schema in altri schemi e consente di far riferimento a più schemi per validare un documento. Offre un'estrema modularità e riusabilità ed introduce il concetto di ereditarietà.

1.4.1 Il tag "schema"

Gli XML Schema sono dei documenti XML, quindi possono avere essi stessi uno schema che li descrive. Esattamente come succede con i documenti XML, i tag appaiono normalmente in coppia, uno di apertura ed uno di chiusura, eccetto per il caso di elemento vuoto dove non è presente il tag di chiusura ed il tag di apertura viene chiuso tramite la segnatura: "/ >". Ogni elemento dello schema può avere degli attributi che ne descrivono le caratteristiche.

Uno schema ha un preambolo oltre a zero o più definizioni e dichiarazioni. L'elemento root corrisponde a schema, può contenere l'attributo xmlns per fornire il namespace di XML Schema, xml:lang per indicare la lingua utilizzata e targetNamespace per specificare lo spazio dei nomi.

Listato 1.2: tag.xsd

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/1999/XMLSchema"
  xml:lang="en"
  targetNamespace="http://www.exemple.com/exemple">
...
</xs:schema>
```

Esempio 1 *Il Listato 1.2 mostra un esempio di tag iniziale di un XML Schema.*

Per comprendere meglio il significato di namespace immaginiamoci applicazioni di XML dove un singolo documento contiene elementi ed attributi definiti ed usati da diversi moduli software (modularità), nel caso esistesse un vocabolario che illustra il significato di un tag, è meglio riusare questo tag piuttosto che ridefinirlo. I moduli software devono essere in grado di identificare senza ambiguità i tag e gli attributi da processare, e questo comporta che i costrutti dei documenti abbiano nomi universali. Un namespace è una collezione di nomi, identificati da un riferimento URI, che sono utilizzati nei documenti XML come nomi di elementi e di attributi. In breve, tramite un namespace viene fissato un vocabolario in modo da individuare senza ambiguità il significato dei nomi usati per gli elementi e gli attributi dello schema.

1.4.2 Un esempio

Consideriamo il documento XML del Listato 1.1 contenente un semplice archivio di dati relativi a film. L'archivio consiste di un elemento principale, movies, che contiene una sequenza di sottoelementi chiamati movie i quali contengono a loro volta dei sottoelementi come title, cast, genre, short-description, rating. Fra questi elementi cast contiene a sua volta altri sottoelementi, mentre gli altri contengono smplicemente dei valori.

Gli elementi che contengono sottoelementi (e/o attributi) sono di tipo complesso (complex type), mentre gli elementi che contengono valori sono di tipo

semplice (simple type). I tipi complessi ed alcuni tipi semplici sono definiti nello schema, gli altri tipi semplici sono definiti come tipi nativi di XML Schema.

Listato 1.3: movie.xsd

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="movies">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="movie">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string" />
              <xs:element name="cast">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="actor" maxOccurs="unbounded"
                        type="personType" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="release-date" type="xs:date"
                  minOccurs="0" />
              <xs:element name="genre" type="xs:string" />
              <xs:element ref="short-description" minOccurs="0" />
              <xs:element name="rating" type="xs:integer" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="short-description" type="xs:string" />
  <xs:complexType name="personType">
    <xs:sequence>
      <xs:element name="sex" type="xs:string" />
      <xs:choice>
        <xs:element name="name" type="xs:string" />
        <xs:sequence>
          <xs:element name="first-name" type="xs:string" />
          <xs:element name="last-name" type="xs:string" />
        </xs:sequence>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xsschema>
```

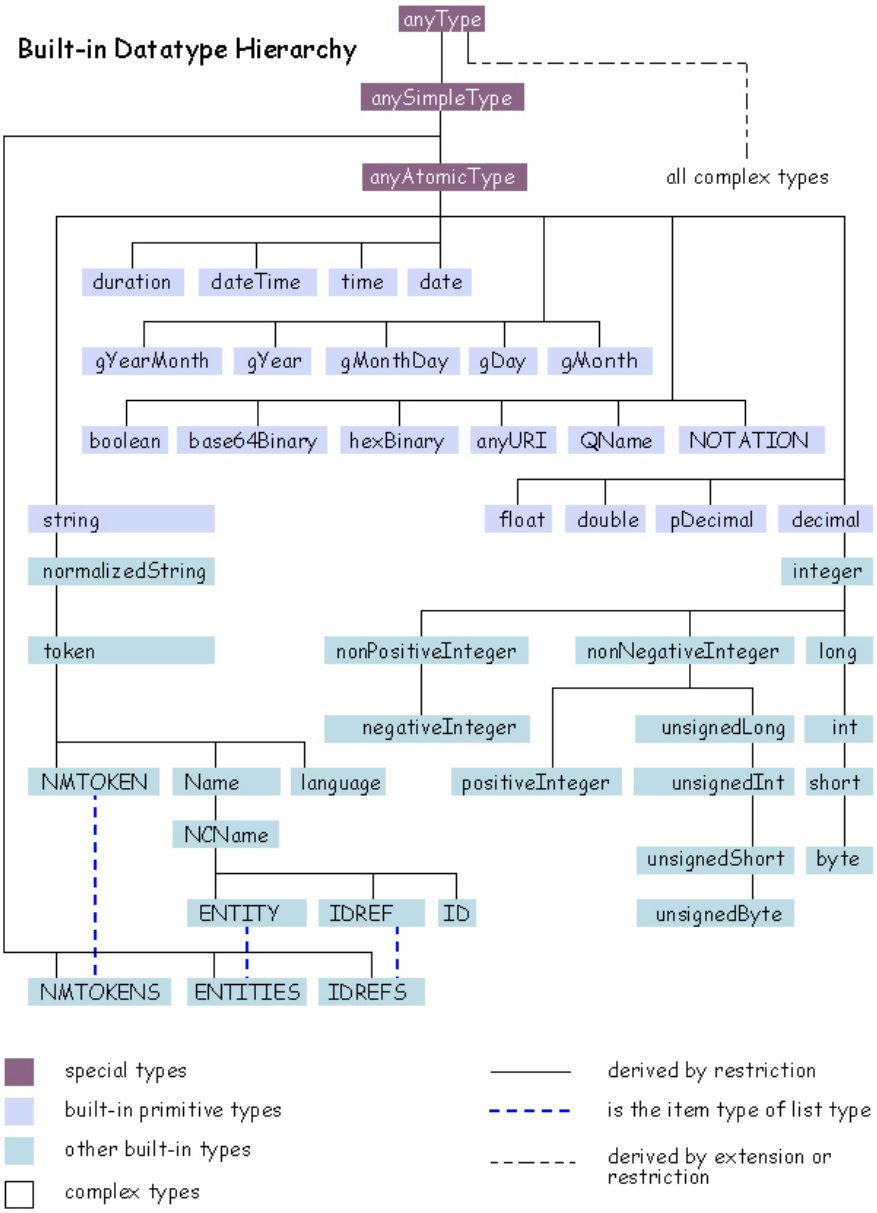
Il Listato 1.3 riporta lo schema movie.xsd che descrive il documento movie.xml presentato nel Listato 1.1.

Nella Figura 1.1 sono illustrati i tipi semplici presenti nello standard e le varie relazioni di sottotipo supportate.

1.5 Tipi semplici

Esiste una forte distinzione tra la definizione di un tipo complesso e quella di un tipo semplice: nel contenuto della prima sono generalmente presenti dichia-

Figura 1.1: Datatype



razioni di nuovi elementi/attributi o riferimenti ad elementi esistenti, mentre un tipo semplice non può contenere nella sua definizione dei sottoelementi, ma deve rappresentare direttamente un valore. Un tipo semplice può essere o un tipo nativo di XML Schema oppure un tipo derivato dai tipi semplici dichiarati (sia nativi che derivati).

È possibile infatti introdurre una nuova definizione di tipo semplice derivando un tipo semplice esistente ed estendendo o restringendo l'insieme dei valori consentiti. I nuovi tipi semplici sono introdotti tramite l'elemento simpleType all'interno del quale viene definita la tecnica di derivazione, che può essere `list`, `restriction` oppure `union`.

1.5.1 Tipi nativi

La Tabella 1.1 elenca i tipi semplici di XML Schema. Notiamo che questi tipi sono compatibili con quelli dei database tradizionali e che rispetto ai DTD si può effettuare una tipizzazione decisamente più raffinata. I tipi nativi si distinguono in primitivi e derivati: si catalogano come primitivi i tipi che non sono definiti in termini di altri tipi, e come derivati i tipi la cui definizione dipende da un tipo base. Per esempio float è un ben definito concetto matematico che non può esser espresso in termini di altri tipi di dato, mentre integer è un caso speciale del tipo più generale decimal.

Esiste inoltre un tipo di dato concettuale il cui nome è anySimpleType, che può esser considerato il tipo base di tutti i tipi di dato primitivi. L'insieme dei valori validi per anySimpleType è l'unione dei valori validi di tutti i tipi di dato primitivi.

1.5.2 Derivare un tipo semplice per `restriction`

Nel caso della `restriction` l'insieme dei valori validi per il tipo base contiene l'insieme dei valori validi per il nuovo tipo. XML Schema prevede un insieme di sfaccettature che possono esser usate per restringere i valori validi del tipo base. Ci sono in tutto 12 sfaccettature, ma non tutte possono esser usate su un dato tipo base. Per esempio si possono usare combinazioni di minExclusive, minInclusive, maxExclusive e minInclusive per restringere l'insieme dei valori permessi di un tipo numerico, length, maxLength e minLength per controllare la lunghezza di un tipo alfanumerico. Si possono usare più enumeration per specificare un fissato insieme di valori; è possibile imporre esplicitamente tramite espressioni regolari dei vincoli lessicali su un valore.

Listato 1.4: restriction.xsd

```
<xs:simpleType name="myInteger">
  <xs:restriction base="xsd:integer">
    <xs:minInclusive value="10" />
    <xs:maxInclusive value="100" />
  </xs:restriction>
</xs:simpleType>
```

Tabella 1.1: Tipi semplici di XML Schema

Tipi semplici	Esempi (delimitati dalle virgole)	Note
string	commedia	
normalizedString	commedia	vedere (3)
token	commedia	vedere (4)
base64Binary	GpM7	
hexBinary	0FB7	
integer	...-1, 0, 1, ...	vedere (2)
positiveInteger	1, 2, ...	vedere (2)
negativeInteger	...-2, -1	vedere (2)
nonNegativeInteger	0, 1, 2, ...	vedere (2)
nonPositiveInteger	...-2, -1, 0	vedere (2)
long	-9223372036854775808, ...-1, 0, 1, ...9223372036854775807	vedere (2)
unsignedLong	0, 1, ...18446744073709551615	vedere (2)
int	-2147483648, ...-1, 0, 1, ...2147483647	vedere (2)
unsignedInt	0, 1, ...4294967295	vedere (2)
short	-32768, ...-1, 0, 1, ...32767	vedere (2)
unsignedShort	0, 1, ...65535	vedere (2)
byte	-128, ...-1, 0, 1, ...127	vedere (2)
unsignedByte	0, 1, ...255	vedere (2)
decimal	-1.23, 0, 123.4, 1000.00	vedere (2)
float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	equivale alla singola precisione a 32 bit in floating point, NaN è "non un numero", vedere (2)
double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	equivale alla doppia precisione a 64 bit in floating point, vedere (2)
boolean	true, false, 1, 0	
duration	P1Y2M3DT10H30M12.3S	1 anno, 2 mesi, 3 giorni, 10 ore, 30 minuti, e 12.3 secondi
dateTime	1999-05-31T13:20:00.000-05:00	31 maggio 1999 all'1.20pm dell'Eastern Standard Time che è 5 ore indietro il Co-Ordinated Universal Time, vedere (2)
date	1999-05-31	vedere (2)
time	13:20:00.000, 13:20:00.000-05:00	vedere (2)
gYear	1999	1999, vedere (2) (5)
gYearMonth	1999-02	febbraio 1999, senza tenere conto del numero dei giorni, vedere (2) (5)
gMonth	`--05--`	maggio, vedere (2) (5)
gMonthDay	`--05-31`	31 maggio, vedere (2) (5)
gDay	`---31`	il giorno 31, vedere (2) vedere (5)
Name	cast	XML 1.0 Name type
QName	my:cast	XML Namespace QName
NCName	cast	XML Namespace NCName, cioè un QName senza il prefisso e due punti
anyURI	http://www.example.com/	
language	en-GB, en-US, fr	valori validi per xml:lang come definiti in XML 1.0
ID		vedere (1)
IDREF		vedere (1)
IDREFS		vedere (1)
ENTITY		vedere (1)
ENTITIES		vedere (1)
NOTATION		vedere (1)
NMTOKEN	Gran Bretagna, Scandinavia	vedere (1)
NMTOKENS	Inghilterra Galles Scozia, Norvegia Svezia Danimarca	cioè una lista di NMTOKEN separata da uno spazio bianco, vedere (1)

Note: (1) Per mantenere la compatibilità tra XML Schema e XML 1.0 DTDs, i tipi semplici ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS dovrebbero essere usati solo negli attributi. (2) Un valore di questo tipo può essere rappresentato da più di un formato lessicale, p. es. 100 e 1.0E2 sono entrambi validi formati float che rappresentano "cento". (3) Un carattere di nuova linea, di tab ed di ritorno a capo in un tipo normalizedString sono convertiti in spazio. (4) Come la normalizedString, e gli spazi adiacenti sono collassati in un singolo carattere di spazio, e gli spazi d'interliea sono rimossi. (5) Il prefisso "g" segnala un arco di tempo nel calendario Gregoriano.

Esempio 2 *Nel Listato 1.4 è mostrata la definizione del nuovo tipo* myInteger *che rappresenta gli interi compresi tra* 10 *e* 100. *Il tipo* myInteger *è in grado di validare ad esempio gli elementi:* `<x>30</x>` *e* `<x>100</x>`.

Sfaccettature

Le sfaccettature consentono di formulare vincoli sull'insieme di valori ammessi (valuespace). Affinché il vincolo espresso sia compatibile con il tipo occorre che vengano rispettate alcune regole di correttezza: ovviamente un vincolo su un generico tipo deve esser espresso tramite sfaccettature consentite per tale tipo. Inoltre, a seconda dei casi, devono esser verificate le seguenti ulteriori proprietà:

- length: è il numero di unità di lunghezza, ovvero il numero di caratteri per i tipi string ed anyURI, il numero di byte per hexBinary e base64Binary, il numero di item per i tipi lista. Non può essere un valore negativo, non può coesistere con minLength o maxLength ed è un errore se il valore non è uguale a quello indicato nella definizione di un tipo antenato.
- minLength: è il numero minimo di unità di lunghezza. Non può essere un valore negativo, non può coesistere con length e nel caso i cui è specificato anche maxLength, deve aver valore minore o uguale a maxLength. Inoltre è un errore se il valore è minore a quello indicato nella definizione di un tipo antenato.
- maxLength: è il numero massimo di unità di lunghezza. Non può essere un valore negativo, non può coesistere con length e nel caso i cui è specificato anche minLength, deve aver valore maggiore o uguale a minLength. Inotre, è un errore se il valore è maggiore a quello indicato nella definizione di un tipo antenato.
- enumeration: i valori specificati devono essere valori ammessi dal tipo base.
- maxInclusive: è il limite superiore inclusivo per i valori di un tipo. Il valore specificato deve essere un valore ammesso dal tipo base. Inoltre, è un errore se il valore specificato per minInclusive è più grande del valore specificato per maxInclusive; se maxInclusive è maggiore del valore specificato in un tipo antenato per maxInclusive o se maxInclusive è maggiore o uguale al valore specificato in un tipo antenato per maxExclusive.
- maxExclusive: è il limite superiore esclusivo. È un errore se maxInclusive e maxExclusive sono specificati nella stessa derivazione di tipo; se il valore specificato per minExclusive è maggiore del valore specificato per maxExclusive; se maxExclusive ha un valore maggiore di quello specificato per un tipo antenato.
- minExclusive: è il limite inferiore esclusivo. È un errore se sia minInclusive che minExclusive sono specificati per lo stesso tipo; se il valore specificato per minExclusive è maggiore del valore specificato per maxInclusive; se minExclusive ha un valore minore di quello specificato per minExclusive in un tipo antenato.
- minInclusive: è il limite inferiore inclusivo. È un errore se sia minInclusive che minExclusive sono specificati per lo stesso tipo; se minInclusive è

maggiore o uguale al valore specificato nella stessa definizione di tipo per maxExclusive; se minInclusive ha un valore minore di quello specificato per minInclusive in un tipo antenato.

- totalDigits: è il massimo numero di cifre per un tipo derivato dai decimal. Il valore deve essere un positiveInteger. È un errore se totalDigits ha un valore maggiore di quello specificato per totalDigits in un tipo antenato.
- fractionDigits: è il massimo numero di cifre per la parte frazionale di un tipo derivato dai decimal. Il valore specificato deve essere un nonNegativeInteger. È un errore se fractionDigits è maggiore di totalDigits.
- pattern: il valore specificato per pattern deve essere un'espressione regolare.
- whiteSpace: il valore di whiteSpace deve essere preserve, replace o collapse.
  - *replace*: tutte le occorrenze di tab e return sono sostituite con spazi;
  - *preserve*: nessuna normalizzazione viene effettuata, il valore rimane invariato;
  - *collapse*: dopo il processo del replace, sequenze di spazi sono collassati in un unico spazio, e vengono rimossi gli spazi in testa ed in coda.

  È un errore se il valore è replace o preserve mentre per un tipo antenato è stato specificato collapse o se il valore è preserve mentre per un tipo antenato è stato specificato replace.

Una sfaccettatura è definita come un elemento e può apparire solo una volta all'interno di una definizione di tipo, fatta esclusione del caso di enumeration e del caso dei pattern.

Nella Tabella 1.2 sono indicate per ogni tipo primitivo di XML Schema le sfaccettature ammesse.

1.5.3 Derivare un tipo semplice per `list` oppure per `union`

I costrutti `list` ed `union` permettono di definire un nuovo tipo semplice estendendo il range di valori descritti da un tipo esistente: l'insieme dei valori validi per il nuovo tipo contiene l'insieme dei valori validi per il tipo base. Con `list` si può definire una sequenza di valori di un tipo semplice. Il tipo base deve essere un tipo atomico o un tipo derivato tramite `union` e viene indicato dall'attributo itemType.

Listato 1.5: list.xsd

```
<xs:simpleType name="myIntegerList">
  <xs:list itemType="myInteger" />
</xs:simpleType>
```

Tabella 1.2: Sfaccettature applicabili

Tipi primitivi	**Sfaccettature**
string	length, minLength, maxLength, pattern, enumeration, whiteSpace
boolean	pattern, whiteSpace
float	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
double	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
decimal	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
precisionDecimal	totalDigits, maxScale, minScale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
duration	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
dateTime	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
time	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
date	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
gYearMonth	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
gYear	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
gMonthDay	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
gDay	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
gMonth	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
hexBinary	length, minLength, maxLength, pattern, enumeration, whiteSpace
base64Binary	length, minLength, maxLength, pattern, enumeration, whiteSpace
anyURI	length, minLength, maxLength, pattern, enumeration, whiteSpace
QName	length, minLength, maxLength, pattern, enumeration, whiteSpace
NOTATION	length, minLength, maxLength, pattern, enumeration, whiteSpace

Esempio 3 *Nel Listato 1.5 viene mostrata la definizione del nuovo tipo* myIntegerList *che rappresenta liste di interi compresi tra* 10 *e* 100. *Il tipo* myIntegerList *è in grado di validare ad esempio i seguenti elementi:* `<x>30</x>` *e* `<x>100 30 58</x>`.

I tipi atomici e i tipi lista permettono al valore di un elemento di essere una o più istanze di un unico tipo atomico. Un tipo definito come `union`, invece, ha come dominio l'unione dei domini dei tipi base. L'elenco dei tipi base viene indicato tramite l'attributo memberTypes.

Listato 1.6: union.xsd

```
<xs:simpleType name="myTypeUnion">
  <xs:union memberTypes="string myIntegerList" />
</xs:simpleType>
```

Esempio 4 *Nel Listato 1.6 viene definito un nuovo tipo in grado di rappresentare liste di interi compresi tra* 10 *e* 100 *e/o generiche stringhe. Il tipo* myTypeUnion *è in grado di validare ad esempio gli elementi:* `<x>una stringa</x>` *e* `<x>100 30 58</x>`.

1.6 Tipi complessi e dichiarazioni di elementi

In XML Schema c'è una sostanziale differenza tra tipi complessi che contengono sottoelementi ed attributi e tipi semplici che non possono contenere né sottoelementi né attributi. C'è anche una basilare distinzione tra una definizione che crea un nuovo tipo ed una dichiarazione che consente ad elementi con uno specifico nome e tipo di apparire nel documento XML.

I nuovi tipi complessi sono definiti usando l'elemento complexType e questa definizione contiene tipicamente un insieme di dichiarazione di elementi, di riferimenti ad elementi e di dichiarazione di attributi. Gli elementi sono dichiarati usando l'elemento element, mentre gli attributi sono dichiarati tramite l'elemento attribute.

1.6.1 Gli elementi

Vi è una sostanziale differenza tra il concetto di elemento globale e quello di elemento locale: un elemento globale (così come un attributo globale) è creato tramite una dichiarazione che appare come figlia dell'elemento schema, mentre un elemento è locale quando la sua dichiarazione appare all'interno di una definizione di tipo. Un elemento globale può esser riferito in una o più dichiarazioni usando l'attributo ref. La dichiarazione di un elemento globale permette all'elemento di apparire nel top-level del documento XML. Nello schema movie.xsd sono stati dichiarati due elementi globali: movies e short-description. Il primo appare nel top-level del documento XML, mentre il secondo viene richiamato nella definizione del tipo associato all'elemento movie tramite `<xs:element ref="short-description" />`. La conseguenza di questa dichiarazione è che un elemento chiamato short-description può apparire come sottoelemento di movie e che il suo contenuto deve essere di tipo string.

1.6.2 Vincoli sulle occorrenze

L'elemento short-description è opzionale perché il valore dell'attributo minOccurs è 0. In generale un elemento è obbligatorio quando il valore di minOccurs è 1 o più di 1. Il massimo numero di volte che l'elemento può apparire è determinato dal valore di maxOccurs. Questo valore deve essere o un intero positivo o il termine unbounded, per indicare che non esiste un limite massimo al numero di occorrenze dell'elemento.

Il valore di default per entrambi gli attributi è 1. Quindi, quando un elemento come short-description è dichiarato senza specificare il valore di maxOccurs, l'elemento può apparire nel documento XML al massimo una volta. Ovviamente il valore di maxOccurs non può essere minore del valore di minOccurs. Inoltre, si può dichiarare un valore di default o un valore fisso.

Quando si specifica un valore di default tramite l'attributo default, l'elemento che appare nel documento istanza avrà il valore che appare al suo interno, ma se l'elemento appare senza contenuto XML Schema provvederà a porre il valore specificato con default. Si può specificare un valore di default anche per gli attributi, con la differenza che il valore viene applicato quando l'attributo non esiste, mentre per gli elementi viene applicato quando l'elemento è vuoto.

Il valore fisso è specificato tramite l'attributo fixed ed impone agli elementi e agli attributi di avere un particolare valore. Notiamo che il concetto di valore fisso e di valore di default sono mutualmente esclusivi, e che quindi sarebbe un errore dichiarare qualcosa specificando sia fixed che default.

Le dichiarazioni globali non possono contenere né riferimenti né vincoli sulla cardinalità delle occorrenze, ovvero in una dichiarazione globale non possono apparire gli attributi href, minOccurs e maxOccurs.

1.6.3 Struttura di un tipo complesso

Lo schema movie.xsd mostrato nel Listato 1.3 impone all'elemento movie di avere una sequenza ben precisa di sottoelementi: l'elemento title, l'elemento cast, eventualmente l'elemento release-date, l'elemento genre, eventualmente l'elemento short-description ed infine l'elemento rating. Analogamente l'elemento actor deve avere un sottoelemento sex seguito o dall'elemento name o dalla sequenza first-name e last-name.

I costruttori `all`, `choice`, `group` e `sequence` permettono la definizione di un tipo complesso: `all` indica una sequenza in cui non conta l'ordine; `choice` impone che solo uno dei suoi sottoelementi sia presente nel documento istanza; `group` permette di dare un nome ad una collezione di elementi in modo da poterla richiamare (riferire) in più definizioni; `sequence` indica una sequenza ordinata di elementi. È possibile specificare minOccurs e maxOccurs anche per i gruppi.

Esistono dei vincoli che impongono delle restrizioni ad un gruppo costruito tramite una `all`: il gruppo deve apparire esclusivamente come top-level di un content, i membri di una `all` devono necessariamente essere elementi e non gruppi e gli elementi possono apparire al massimo una volta sola, ovvero gli unici valori permessi per gli attributi minOccurs e maxOccurs dei vari sottoelementi sono 0 e 1. Combinando i vari tipi di gruppi offerti dal linguaggio XML Schema e settando i valori di minOccurs e maxOccurs è possibile rappresentare qualsiasi

struttura esprimibile tramite un DTD. Inoltre il costrutto `all` e l'indicazione della cardinalità ammessa permettono un ulteriore potenziamento espressivo.

Listato 1.7: group.xsd

```
<xs:group name="actorGroup">
  <xs:sequence>
    <xs:element name="actor" maxOccurs="unbounded" type="personType
        " />
  </xs:sequence>
</xs:group>
<xs:element name="movies">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="movie">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string" />
            <xs:element name="cast">
              <xs:complexType>
                <xs:group ref="actorGroup">
              </xs:complexType>
            </xs:element>
            <xs:element name="release-date" type="xs:date"
                minOccurs="0" />
            <xs:element name="genre" type="xs:string" />
            <xs:element ref="short-description" minOccurs="0" />
            <xs:element name="rating" type="xs:integer" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Lo schema illustrato nel Listato 1.7 dal punto di vista dei vincoli imposti ai documenti istanza è equivalente allo schema movie.xsd; l'unica differenza è data dalla presenza della definizione del gruppo actorGroup che può esser eventualmente riusato tramite riferimento anche in un altro punto dello schema (o in un altro schema).

1.6.4 Derivare un tipo complesso

Analogamente a quanto visto per i tipi semplici, è possibile derivare un tipo complesso da un tipo esistente. Questo viene realizzato tramite l'elemento complexType che contiene il sottoelemento simpleContent o complexContent a seconda se il nuovo tipo complesso è derivato da un tipo semplice o da un altro tipo complesso. Si estende un tipo semplice in un tipo complesso quando si vogliono associare degli attributi ad un tipo semplice esistente. Gli elementi restriction e extension indicano se si deriva per restrizione o per estensione. Derivando per restrizione, un'istanza del tipo derivato è sempre un'istanza valida per il tipo base.

Derivando per estensione non è consentito modificare le sfaccettature del tipo base, è possibile solo estendere il suo contenuto aggiungendo alla sequenza di elementi del tipo base degli elementi aggiuntivi. In questo caso, istanze del

Tabella 1.3: Esempi di restriction

Base	Restriction	Note
	default="1"	assegna un valore di default dove non precedentemente dato
	fixed="100"	assegna un fissato valore dove non precedentemente dato
	type="string"	specificazione di un tipo dove precedentemente dato
(minOccurs, maxOccurs)	(minOccurs, maxOccurs)	
(0, 1)	(0, 0)	esclusione di un componente opzionale
(0, 1)	(1, 1)	creazione di un componente opzionale richiesto
(0, ∞)	(0, 0) (0,37) (1, 37)	
(1, 9)	(1, 8) (4, 9) (3, 3)	
(1, ∞)	(1, 12) (3, ∞) (6, 6)	
(1, 1)	(1, 1)	non può ulteriormente limitare i minOccurs o maxOccurs

tipo derivato sono generalmente istanze non valide per il tipo base, e solo nel caso in cui gli elementi aggiunti sono tutti opzionali le istanze del tipo base sono valide anche per il nuovo tipo.

Listato 1.8: extension.xsd

```
<xs:complexType name="personType">
  <xs:sequence>
    <xs:element name="sex" type="xs:string" />
    <xs:choice>
      <xs:element name="name" type="xs:string" />
      <xs:sequence>
        <xs:element name="first-name" type="xs:string" />
        <xs:element name="last-name" type="xs:string" />
      </xs:sequence>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="newPersonType">
  <xs:complexContent>
    <xs:extension base="personType">
      <xs:sequence>
        <xs:element name="nationality" type="xs:string">
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Esempio 5 *Nel Listato 1.8 viene definito il nuovo tipo complesso* newPersonType *che estende il tipo* personType *includendo un elemento per la nazionalità dell'attore.*

1.6.5 Definizione di tipi anonimi

Per render più leggibile uno schema si evita di definire come globali tipi che sono riferiti esclusivamente da un unico elemento. In questi casi si utilizza una

definizione anonima includendo direttamente nel tag dell'elemento la definizione del tipo.

Esempio 6 *In riferimento allo schema* movie.xsd*, l'elemento* cast *ha un tipo complesso anonimo che consiste in una sequenza di elementi con nome* actor *e di tipo* personType*.*

1.6.6 Conflitto di nomi

Abbiamo descritto come definire nuovi tipi complessi, come dichiarare elementi e attributi. Cosa succede se viene dato lo stesso nome a due entità differenti? In determinate situazioni si può verificare un conflitto che rende scorretto lo schema. Il conflitto si viene a creare solo nel caso in cui lo stesso nome viene associato a due concetti equivalenti. Se viene definito un tipo complesso chiamato personType ed un tipo semplice con lo stesso nome nasce un conflitto. Ma se viene definito un tipo chiamato Person e un elemento o un attributo con lo stesso nome non c'è conflitto, poiché un elemento ed un attributo sono due concetti distinti; se due elementi locali appaiono in due distinte definizioni di tipo con lo stesso nome ma con tipo diverso non c'è conflitto, poiché XML Schema individua il tipo di un elemento a seconda del contesto; se viene definito dall'utente un tipo con lo stesso nome di un tipo nativo (ad esempio integer) la possibilità di ridefinire i tipi nativi offerta dal linguaggio XML Schema fa si che non si crei un conflitto.

CAPITOLO 2

Rappresentazione dei documenti e degli schemi XML

Come punto di partenza del nostro lavoro abbiamo effettuato una rappresentazione formale sia dei documenti XML che degli XML Schema in modo da avere delle strutture facilmente trattabili per le operazioni che dobbiamo eseguire.

Al fine di semplificarne la trattazione si è deciso di trascurare alcune caratteristiche dei documenti e degli schemi. In particolare abbiamo trascurato gli attributi, gli elementi con contenuto misto e le entità, inoltre gli schemi devono essere *conflict-free*[1]. Dopotutto la maggior parte degli schemi utilizzati sul Web sono schemi conflict-free.

Per rappresentare di un documento XML è sufficiente l'utilizzo di un albero etichettato dal momento che ci interessa considerare solo la struttura gerarchica di un documento XML e non le caratteristiche di ogni singolo nodo specificate attraverso gli attributi.

La rappresentazione di un XML Schema la struttura adottata è più complicata. È necessario rappresentare sia le dichiarazioni di elementi sia le definizioni di tipi semplici e complessi. La possibilità di dichiarare elementi e tipi globali o locali richiede di mantenere la distinzione tra elementi/tipi che possono essere utilizzati in più contesti, all'interno dello schema, dagli elementi/tipi che possono essere utilizzati solo nella definizione in cui compaiono.

Il capitolo è strutturato nel seguente modo. Nella Sezione 2.1 viene introdotto il concetto di albero etichettato che verrà utilizzato per la definizione di un documento XML e per specificare la struttura di un elemento/tipo complesso. Nella Sezione 2.2 viene presentata la definizione formale di un documento XML e vengono introdotte delle operazioni che permettono di manipolare documenti XML. Nella Sezione 2.3 introduciamo la rappresentazione dei tipi semplici di XML Schema. Nella Sezione 2.4 ci occupiamo della rappresentazione dei tipi complessi e delle operazioni atomiche che ci permettono di manipolarla. Nella

[1] Precisiamo che uno schema è definito conflict-free quando nelle definizioni dei tipi i nomi dei sottoelementi appaiono solo una volta

Sezione 2.5 utilizzando i concetti presentati nelle Sezioni 2.3 e 2.4, presentiamo la definizione formale di XML Schema.

2.1 Alberi etichettati

Il concetto di albero etichettato è stato utilizzato sia per la definizione di documento XML che per specificare la struttura di un elemento/tipo complesso.

La rappresentazione adottata è basata sulla classica definizione di albero etichettato.

Definizione 1 (Albero) *Un albero su un insieme di nodi $\mathcal{V}$ è definito per induzione come segue:*

- $\nu \in \mathcal{V}$ *è un albero;*
- *se* $T_1, \ldots, T_n$ *sono alberi allora* $(\nu, [T_1, \ldots, T_n])$ *è un albero.*

Definizione 2 (Albero etichettato) *Sia $\mathcal{V}_T \subseteq \mathcal{V}$ l'insieme dei nodi dell'albero T e sia $\mathcal{L}$ un insieme di etichette. Si definisce albero etichettato la coppia (T, φ) dove T è un albero e φ è una funzione tale che $\forall \nu \in \mathcal{V}_T, \varphi(\nu) \in \mathcal{L}$.*

Tabella 2.1: Notazioni riguardo utili per gli alberi etichettati

$\varphi(\nu)$	Etichetta del nodo ν
$root(T)$	Radice dell'albero T
$children(\nu)$	Figli del nodo ν
$parent(\nu)$	Genitore del nodo ν

In Tabella 2.1 sono riportate alcune notazioni adottate per gli alberi etichettati con la relativa spiegazione.

2.2 Rappresentazione di un documento XML

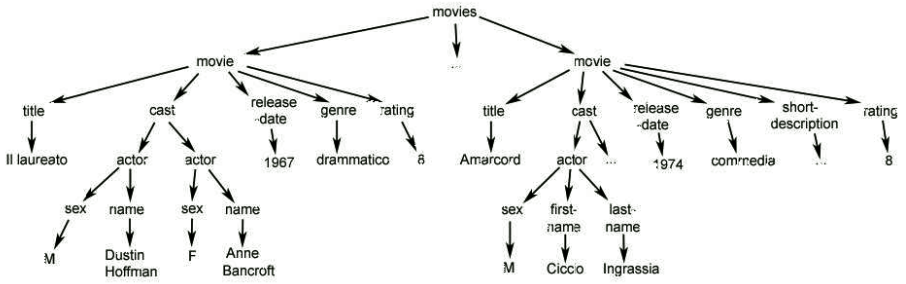
A questo punto risulta semplice la Definizione 3 di come viene rappresentato un documento XML.

Definizione 3 (Documento XML) *Sia $\mathcal{C}$ l'insieme di valori che possono essere all'interno degli elementi ed $\mathcal{EN}$ l'insieme dei nomi degli elementi. Un documento XML viene rappresentato formalmente come un albero etichettato (T, φ) dove $\forall \nu \in \mathcal{V}_T, \varphi(\nu) \in \mathcal{EN} \cup \mathcal{C}$ con le seguenti proprietà:*

- *se ν è un nodo interno allora $\varphi(\nu) \in \mathcal{EN}$;*
- *se ν è una foglia allora $\varphi(\nu) \in \mathcal{C}$.*

Precisiamo che un nodo etichettato con un valore viene interpretato come un documento XML formato unicamente da una foglia e che un elemento con contenuto vuoto viene rappresentato come un documento con un nodo padre etichettato con il nome dell'elemento e un nodo figlio etichettato con lo speciale valore $\emptyset$ appartenente a $\mathcal{C}$.

Figura 2.1: Raffigurazione della rappresentazione di un documento XML



D'ora in avanti verrà indicato con $\mathcal{D}$ l'insieme di tutti i documenti XML (d indica un solo documento), con e un elemento di un documento XML e con l l'etichetta di un elemento.

Esempio 7 *Intanto in Figura 2.1 viene mostrata la rappresentazione formale del documento XML* movie.xml *visualizzato nel Listato 1.1.*

2.3 Rappresentazione dei tipi semplici

Un tipo semplice è un tipo nativo di XML Schema (vedere Tabella 1.1) oppure è un tipo derivato dai tipi semplici dichiarati tramite `list`, `restriction` od `union`.

Come illustrato in Tabella 1.2 per ogni tipo semplice sono presenti delle sfaccettature che consentono di formulare vincoli sull'insieme di valori ammessi. Si rende così necessaria la presenza di una funzione che verifichi le regole di correttezza.

$$is\_compatible(pred, \tau) = \begin{cases} \textbf{true} & \text{se } pred \text{ è ammissibile per } \tau \\ \textbf{false} & altrimenti \end{cases}$$

Assumiamo così la presenza della funzione *is_compatible* che dato in input un predicato $pred \in \mathcal{PRED}$ ed un tipo $\tau \in \mathcal{T}$, possa determinare se un valore corrisponda ai vincoli delle sfaccettature.

Definizione 4 (Tipi semplici) *L'insieme dei tipi semplici di XML Schema è definito induttivamente come segue:*

- *i tipi nativi $\mathcal{NT}$ (string, decimal, boolean ...) sono tipi semplici;*
- *se τ è un tipo semplice allora $\texttt{list}(\tau)$ è un tipo semplice;*
- *se τ è un tipo semplice e risulta verificata $is\_compatible(pred, \tau)$ allora $\texttt{restriction}(\tau, pred)$ è un tipo semplice;*
- *se $\tau_1, \ldots, \tau_n$ sono tipi semplic allora $\texttt{union}(\tau_1, \ldots, \tau_n)$ è un tipo semplice.*

La definizione $\tau = \texttt{list}(\tau_N)$ indica che il tipo τ è derivato tramite il costrutto `list` dal tipo τ_N. Se $\tau = \texttt{restriction}(\tau_N, pred)$ allora il tipo τ è derivato per `restriction` dal tipo τ_N. In fine la definizione $\tau = \texttt{union}(\tau_1, \ldots, \tau_n)$ indica che il tipo τ è derivato tramite il costrutto `union` dai tipi $\tau_1, \ldots, \tau_n$. Ricordiamo che l'insieme dei valori ammessi dal tipo τ è l'unione degli insiemi di valori ammessi per ciascuno dei tipi $\tau_1, \ldots, \tau_n$.

2.4 Rappresentazione dei tipi complessi

Per la rappresentazione dei tipi complessi utilizziamo gli alberi etichettati specificando alcuni vincoli che un albero deve rispettare al fine di rappresentare in maniera corretta un tipo complesso.

Viene introdotto l'insieme degli operatori $\mathcal{OP} = \{\texttt{all}, \texttt{choice}, \texttt{sequence}\}$ al fine di poter rappresentare le sequenze non ordinate (`all`), gli insiemi di elementi alternativi (`choice`) e le sequenze ordinate (`sequence`).

In più viene introdotto l'insieme dei vincoli $\Gamma = \{(min, max) | min, max \in \mathbb{N}, min \leq max\}$ dove min rappresenta l'attributo minOccurs ed indica il numero minimo di volte che l'elemento deve apparire, max rappresenta l'attributo maxOccurs ed indica il numero massimo di volte che l'elemento può apparire. Ricordiamo che, se non specificato, il valore di default di (min, max) corrisponde ad $(1, 1)$. Inoltre inidichiamo con $\varphi_{|_i}(\nu)$, con $i = 1, 2, \ldots, n$ denota l'i-esima componente dell'etichetta del nodo ν.

Definizione 5 (Struttura dei tipi complessi) *La struttura di un tipo complesso è un albero etichettato T definito su un insieme di etichette $(\mathcal{EN} \cup \mathcal{OP}) \times \Gamma$ con le seguenti proprietà:*

- $\varphi(T) \in \mathcal{OP} \times \Gamma$*;*
- *per ogni sottoalbero* $(\nu, [T_1, \ldots, T_n])$ *di* $T, \varphi(\nu) \in \mathcal{OP} \times \Gamma$*;*
- *per ogni foglia* ν *di* $T, \varphi(\nu) \in \mathcal{EN} \times \Gamma$*;*
- *per ogni sottoalbero* $(\nu, [T_1, \ldots, T_n])$ *di* T*, se* $\varphi(\nu) = \langle \texttt{all}, (min, max) \rangle$ *allora* $\forall i, j \in \{1, \ldots, n\} \varphi(T_i), \varphi(T_j) \in \mathcal{EN} \times \Gamma$ *e* $i \neq j \Rightarrow \varphi_{|_1}(T_i) \neq \varphi_{|_1}(T_j)$ *con* $0 \leq min_i \leq max_i \leq 1$ *dove* $\varphi(T_i) = \langle l_i, (min_i, max_i) \rangle$*;*
- *se* $\varphi(\nu) = \langle \texttt{all}, (min, max) \rangle$ *allora* $\nu = root(T)$*.*

2.5 Rappresentazione di un XML Schema

Gli XML Schema, diversamente dai DTD, permettono ad un elemento di avere differenti tipi a seconda del contesto; tuttavia, un unico tipo è assegnato ad ogni elemento dello schema dipendendo dal contesto (globale o locale).

Definizione 6 (XML Schema) *Un XML Schema è rappresentato come la 4-tupla $(\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)$, dove:*

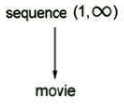
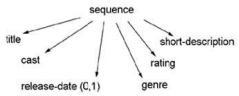
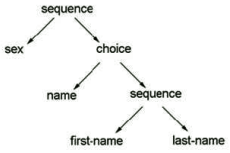
- $\mathcal{EN}_G \subseteq \mathcal{EN}$ *è l'insieme delle etichette degli elementi globali;*
- $\mathcal{T} = (\mathcal{TT} \cup \mathcal{AT})$ *è l'insieme dei nomi dei tipi;*
- ρ *associa ad ogni tipo* $\tau \in \mathcal{T}$ *con la sua dichiarazione, che è:*
  - *se* τ *è un tipo semplice,* $\rho(\tau) \in \mathcal{NT} \cup \{\texttt{list}(\tau_1), \texttt{union}(\tau_1, \ldots, \tau_n), \texttt{restriction}(\tau_1, pred) | \tau_1, \ldots, \tau_n$ *tipi semplici*$\}$*;*
  - *se* τ *è un tipo complesso,* $\rho(\tau) \in (\mathcal{EN}_\mathcal{T}, \mathcal{S}_\mathcal{T}, \mathcal{R}_\mathcal{T})$*, dove:*
    - $\mathcal{EN}_\mathcal{T} \subseteq \mathcal{EN}$ *è l'insieme dei nomi di elementi locali per il tipo* $\tau \in \mathcal{T}$*;*

  - $*$ $\mathcal{S}_\mathcal{T}$ *è l'insieme delle strutture dichiarate per il tipo* $\tau \in \mathcal{T}$ *specificate nella Definizione 5;*
  - $*$ $\mathcal{R}_\mathcal{T} : \mathcal{EN}_\mathcal{T} \rightarrow \mathcal{T}$ *è una funzione che assegna ad ogni elemeno locale di* $\tau \in \mathcal{T}$ *il corrispondente tipo;*

- $\mathcal{R}_G : \mathcal{EN}_G \rightarrow \mathcal{T}$ *è una funzione che assegna ad ogni elemeno globale il corrispondente tipo.*

In particolare l'insieme $\mathcal{T}$ e formato dagli insiemi $\mathcal{TT}$ ed $\mathcal{AT}$, dove $\mathcal{TT}$ è l'insieme dei nomi di tipo assegnati in modo esplicito nello schema ed $\mathcal{AT}$ è, invece, l'insieme dei nomi di tipo assegnati dal sistema in modo da identificare i tipi anonimi dello schema.

Tabella 2.2: Rappresentazione formale di uno schema XML

$\mathcal{EN}_G = \{short\text{-}description, movies\}$ $\mathcal{T} = \mathcal{TT} \cup \mathcal{AT}$ $\{personType\} \subseteq \mathcal{TT}$ $\mathcal{AT} = \{t1, t2, t3\}$			
$\rho(t1)$	*movie*		$movie \rightarrow t2$
$\rho(t2)$	*title* *cast* *release-date* *genre* *rating*		$title \rightarrow string$ $cast \rightarrow t3$ $release\text{-}date \rightarrow date$ $genre \rightarrow string$ $rating \rightarrow integer$
$\rho(t3)$	*actor*		$actor \rightarrow personType$
$\rho(personType)$	*sex* *name* *first-name* *last-name*		$sex \rightarrow string$ $name \rightarrow string$ $first\text{-}name \rightarrow string$ $last\text{-}name \rightarrow string$
$\mathcal{R}_G(movies) = t1$ $\mathcal{R}_G(short\text{-}description) = string$			

Esempio 8 *La Tabella 2.2 mostra la rappresentazione dello schema* movie.xsd *presente nel Listato 1.3.*

Tabella 2.3: Notazioni riguardo agli schemi

$\tau_O \sqsubseteq \tau_N$	Tutti i valori ammissibili per τ_O sono ammissibili anche per τ_N
$type(t_s)$	Restituisce il tipo associato ad una struttura t_s
$ST(\tau)$	Restituisce la struttura del tipo τ
$[\![\tau]\!]$	Insieme dei valori di tipo τ

In Tabella 2.3 sono riportate alcune notazioni adottate per gli schemi con la relativa spiegazione.

Quindi per evitare confusione riassumiamo (utilizzando simboli e notazioni presentati già presentati):

- $\tau \in \mathcal{T}$
- $\tau = \mathcal{R}(l)$
- $\tau = type(t_s)$
- $t_s \in \mathcal{S}_\mathcal{T}$
- $t_s = ST(\tau)$
- $\rho(\tau) = (\mathcal{EN}_\tau, t_s, \mathcal{R}_\tau)$
- $(\mathcal{EN}_\tau, t_s, \mathcal{R}_\tau) \in (\mathcal{EN}_\mathcal{T}, \mathcal{S}_\mathcal{T}, \mathcal{R}_\mathcal{T})$
- $[\![\tau]\!] \subseteq \mathcal{C}$
- $\tau_O \sqsubseteq \tau_N \iff [\![\tau_O]\!] \subseteq [\![\tau_N]\!]$
- $l = \varphi(e)$
- $e = root(d)$

CAPITOLO 3

Primitive di evoluzione dello schema

In questo capitolo vengono presentate le primitive proposte al fine di consentire l'evoluzione di uno schema e ne viene specificata la semantica. In particolare la Sezione 3.1 mostra una classificazione generale delle primitive, la Sezione 3.2 illustra le primitive riguardo ai tipi semplici. Le Sezioni 3.3, 3.4 e 3.5 riguardano le primitive per i tipi, si occupano rispettivamente della rinomina, della cancellazione e della migrazione. La Sezione 3.6 descrive le primitive per i tipi semplici ed infine la Sezione 3.7 si occupa delle primitive sugli elmenti.

Per ogni primitiva sono state esplicitate le condizioni per l'applicabilità in modo da garantire la consistenza dello schema risultante.

3.1 Classificazione e presentazione delle primitive

Sono state disposte tre categorie di primitive per l'evoluzione: inserimento, modifica e cancellazione dei componeti XML Schema (tipi semplici, tipi complessi ed elementi). Le modifiche possono essere ulteriormente classificate in tre sottocategorie: strutturali, rinomina e modifiche migratorie. Le modifiche strutturali permettono di modificare il tipo di un (sotto)elemento ed i suoi vincoli di cardinalità. La rinomina permette di cambiare il nome di un elemento/tipo. Le modifiche migratorie includono lo spostamento di un sottoelemento da un elemento ad un'altro e la trasformazione di un elemento/tipo locale in un elemento/tipo globale (o viceversa).

La Tabella 3.1 riporta le primitive facendo affidamento alla classificazione proposta. Con $\mathcal{P}^*$ si identificano le primitive la cui applicazione non alterano sicuramente la validità dei documenti, mentre con $\mathcal{P}^{\mathcal{S}_\mathcal{T}}$ si identificano le primitive che vanno a modificare la struttura di un tipo.

Tabella 3.1: Classificazione delle primitive di evoluzione

	Insertion	Modification	Deletion
Simple Type	$insert\_glob\_simple\_type^*$ $insert\_new\_member\_type^*$	$change\_restriction$ $change\_base\_type$ $rename\_glob\_type^*$ $change\_item\_type$ $glob\_to\_local^*$ $local\_to\_glob^*$	$remove\_type^*$ $remove\_member\_type^*$
Complex Type	$insert\_glob\_complex\_type^*$ $insert\_local\_elem^{\mathcal{ST}}$ $insert\_ref\_elem^{\mathcal{ST}}$ $insert\_operator^{\mathcal{ST}}$	$rename\_local\_elem$ $rename\_glob\_type^*$ $change\_type\_local\_elem$ $change\_cardinality^{\mathcal{ST}}$ $change\_operator^{\mathcal{ST}}$ $glob\_to\_local^*$ $local\_to\_glob^*$	$remove\_elem$ $remove\_operator^{\mathcal{ST}}$ $remove\_substructure^{\mathcal{ST}}$ $remove\_type^*$
Element	$insert\_glob\_elem^*$	$rename\_glob\_elem$ $change\_type\_glob\_elem$ $local\_to\_ref^*$ $ref\_to\_local^*$	$remove\_glob\_elem$

Per ogni primitiva sono state definite delle condizioni di applicabilità (AC). Se esse non sono soddisfatte, l'operazione non è applicata e lo schema rimane inalterato.

3.2 Primitive per i tipi semplici

Vengono considerate primitive sia per tipi con nome che per tipi anonimi. Ricordiamo che in quest'ultimo caso viene associato lo stesso un nome al tipo.

3.2.1 Inserimento di un tipo semplice

La primitiva per essere applicata prevede che non deve esistere un altro tipo con lo stesso nome. Come parametro di input viene presa una definizione di tipo che può essere un tipo `list`, `restriction` o `union`. Quando inseriamo un tipo `restriction` la sfaccettatura che è stata dichiarata dovrebbe essere compatibile con il tipo base ed il tipo base dovrebbe essere già dichiarato. Quando inseriamo un tipo `union` tutti i nomi dei tipi membri nella lista dovrebbero essere distinti e già dichiarati. Quando inseriamo un tipo `list`, il tipo base dovrebbe essere già dichiarato e dovrebbe essere un tipo semplice anonimo oppure un tipo `union`.

$insert\_glob\_simple\_type : \mathcal{TN} \times \mathcal{ST} \times \mathcal{PRED} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$insert\_glob\_simple\_type(\tau_N, st, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}_N, \rho_N, \mathcal{R}_G)$

$$\mathcal{T}_N = \mathcal{T} \cup \{\tau_N\}, \rho_N(x) = \begin{cases} st & \text{se } x = \tau_N \\ \rho(x) & altrimenti \end{cases}$$

AC: $\tau_N \notin \mathcal{T}$, $\tau 1$ e $\{\tau_1, \ldots, \tau_n\}$ sono tipi semplici

- se $st = \texttt{list}(\tau 1)$, $\tau 1 \in \mathcal{TT}$, $\tau 1$ è un tipo semplice atomico oppure `union`
- se $st = \texttt{restriction}(\tau 1, pred)$, $\tau 1 \in \mathcal{T}$, $is\_compatible(pred, \tau 1) =$ **true**
- se $st = \texttt{union}(\tau_1, \ldots, \tau_n)$, $\tau_1, \ldots, \tau_n \in \mathcal{TT}$, $\forall i, j \in \{1, \ldots, n\}, \tau_i = \tau_j \Rightarrow i = j$

3.2.2 Modifica di un tipo semplice

Differenti primitive sono state separate a seconda della specie di tipo semplice: `list`, `restriction`, `union`. Per i tipi `list` c'è solo un tipo di modifica: cambio di un tipo.

$change\_item\_type : \mathcal{T} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$change\_item\_type(\tau, \tau_N, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} \texttt{list}(\tau_N) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

AC: $\tau \in \mathcal{T}$, $\tau_N \in \mathcal{T}$, τ_N è un tipo nativo od un tipo `union`

Per i tipi `restriction` ci sono due specie di modifiche: cambio del tipo base e cambio della sfaccettatura.

$change\_base\_type : \mathcal{T} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$change\_base\_type(\tau, \tau_N, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} \texttt{restriction}(\tau_N, pred) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

AC: $\tau \in \mathcal{T}$, $\tau_N \in \mathcal{T}$, $is\_compatible(pred, \tau_N) = \textbf{true}$

Queste modifiche possono essere eseguite quando il nuovo tipo è compatibile con il predicato esistente ed il nuovo predicato è compatibile con il tipo esistente (questo tipo di controllo è eseguito attraverso la funzione *is_compatible*).

$change\_restriction : \mathcal{T} \times \mathcal{PRED} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$change\_restriction(\tau, pred_N, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} \texttt{restriction}(\tau, pred_N) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

AC: $\tau \in \mathcal{T}$, $is\_compatible(pred_N, \tau) = \textbf{true}$

Per i tipi `union` ci sono due specie di modifiche: inserimento di un tipo membro e rimozione di un tipo membro.

$insert\_new\_member\_type : \mathcal{T} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$insert\_new\_member\_type(\tau, \tau_N, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} \texttt{union}(\tau_1, \ldots, \tau_n, \tau_N) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

AC: $\tau_N \in \mathcal{T}$ è tipo semplice, $\rho(\tau) = \texttt{union}(\tau_1, \ldots, \tau_n)$, $\forall i \in \{1, \ldots, n\}, \tau_N \neq \tau_i$

Per rimuovere un tipo membro la posizione del tipo membro da eliminare nella lista deve essere specificata.

$remove\_member\_type : \mathcal{T} \times \mathbb{N} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$remove\_member\_type(\tau, i, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} \texttt{union}(\tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots, \tau_n) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

AC: $\tau \in \mathcal{T}$, $\rho(\tau) = \texttt{union}(\tau_1, \ldots, \tau_n), n > 1$, $1 \leq i \leq n$

3.3 Rinomina di tipi

La primitiva che stiamo introducendo permette di modificare il nome di un tipo (ovviamente globale) e funziona sia per i tipi semplici che per quelli complessi. Affinché la modifica non comprometta la correttezza dello schema occorre assicurarsi che il nuovo nome non crei conflitto, ovvero non deve esser stato definito un altro tipo (globale) con lo stesso nome. Inoltre occorre aggiornare le eventuali associazioni al tipo in questione.

$rename\_type : \mathcal{T} \times \mathcal{TN} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$rename\_type(\tau, \tau_N, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}_N, \rho_N, \mathcal{R}_{G\,N})$
$\mathcal{T}_N = \mathcal{T} \setminus \{\tau\} \cup \{\tau_N\}, \rho_N = \rho \setminus \{(\tau, \_)\} \cup \{(\tau_N, \_)\}$

$$\mathcal{R}_{\tau\,N}(x) = \begin{cases} \tau_N & \text{se } \mathcal{R}_G(x) = \tau \\ \mathcal{R}_G(x) & \textit{altrimenti} \end{cases}$$

AC: $\tau \in \mathcal{TT}$, $\tau_N \notin \mathcal{T}$

3.4 Cancellazione di tipi

Questa primitiva può essere applicata per la cancellazione di entrambi i tipi semplici e complessi, e per i tipi anonimi e globali. La primitiva può essere applicata se nessun elemento (locale o globale) è del tipo che sta per essere cancellato.

$remove\_type : \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$remove\_type(\tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T} \setminus \{\tau\}, \rho \setminus \{(\tau, \_)\}, \mathcal{R}_G)$
AC: $\tau \in \mathcal{T}$, $\nexists l \in \mathcal{EN}$ tale che $\mathcal{R}_G(l) = \tau$, $\forall \tau \in \mathcal{T} \nexists l \in \mathcal{EN}$ tale che $\mathcal{R}_\tau(l) = \tau$

3.5 Migrazione di tipi

Nella definizione di uno schema un tipo che è inizialmente definito come globale può succedere che venga spostato in un tipo locale. Nello stesso modo, un tipo locale può essere spostato in un tipo globale. Queste primitive possono essere applicate sia a tipi semplici che complessi. Nel primo caso (da globale a locale) la primitiva prende come input il tipo globale ed il nome del sottoelemento che ha lo stesso tipo. Una volta trovato il sottoelemento, viene introdotto un nuovo tipo locale che è la dupplicazione del tipo globale e viene aggiornata l'associazione con il sottoelemento. In questo modo si possono applicare localmente delle variazioni che non devono influenzare gli altri sottoelementi dello schema associati al tipo globale.

$glob\_to\_local : \mathcal{T} \times \mathcal{EN} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$glob\_to\_local(\tau, l, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}_N, \rho_N, \mathcal{R}_G)$

$$\mathcal{AT}_N = \mathcal{AT} \cup \{\tau_N\}, \rho_N(x) = \begin{cases} \rho(\mathcal{R}_\tau(l)) & \text{se } x = \tau_N \\ (\mathcal{EN}_\tau, t_s, \mathcal{R}_\tau \setminus \{(l, \_)\} \cup \{(l, \tau_N)\}) & \text{se } x = \tau \\ \rho(x) & \textit{altrimenti} \end{cases}$$

AC: $\tau \in \mathcal{T}$, $\rho(\tau) = (\mathcal{EN}_\tau, t_s, \mathcal{R}_\tau)$, $l \in \mathcal{EN}_\tau$, $\mathcal{R}_\tau(l) \in \mathcal{TT}$

Nel secondo caso (da locale a globale), un nuovo nome di tipo globale (univoco rispetto ai tipi già esistenti) deve essere specificato. Grazie a questa operazione sarà possibile in seguito associare il tipo in questione a sottoelementi presenti in diverse definizioni di tipo.

$local\_to\_glob : \mathcal{TN} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$local\_to\_glob(\tau_N, \tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_{GN})$

$$\mathcal{T}_N = (\mathcal{TT} \cup \{\tau_N\}) \cup (\mathcal{AT} \setminus \{\tau\}), \mathcal{R}_{GN}(x) = \begin{cases} \tau_N & \text{se } \mathcal{R}_G = \tau \\ \mathcal{R}_G(x) & altrimenti \end{cases}$$

$$\forall t \in \mathcal{T}, \mathcal{R}_{tN}(x) = \begin{cases} \tau_N & \text{se } \mathcal{R}_t(x) = \tau \\ \mathcal{R}_t(x) & altrimenti \end{cases}$$

AC: $\tau_N \notin \mathcal{T}$

3.6 Primitive per i tipi complessi

Denotiamo con $\mathcal{POS} = \mathbb{N} \times \mathbb{N}$ l'insieme dei valori che individuano una posizione in un albero etichettato. Il primo valore serve per indicare un nodo dell'albero, in modo da individuare la radice del sottoalbero al quale verrà apportata la modifica, il secondo valore serve per indicare la posizione in cui apportare la modifica all'interno del sottoalbero.

3.6.1 Inserimento di un tipo complesso

La primitiva può essere applicata prevedendo che l'univocità del nome del tipo non sia violata.

$insert\_glob\_complex\_type : \mathcal{TN} \times \mathcal{CT} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$insert\_glob\_complex\_type(\tau_N, ct, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}_N, \rho_N, \mathcal{R}_G)$
$\mathcal{T}_N = (\mathcal{TT} \cup \{\tau_N\}) \cup \mathcal{AT}$, $\rho_N = \rho \cup \{(\tau_N, ct)\}$
AC: $\tau_N \notin \mathcal{T}$, $\forall l \in \mathcal{EN}_\tau, \mathcal{R}_\tau(l) \in \mathcal{T}$

3.6.2 Inserimento di un sottoelemento

L'inserimento di un nuovo sottoelemento locale permette di aumentare le informazioni contenute in un elemento. Per poter inserire un nuovo sottoelemento si richiede di specificare il nome dell'elemento, i vincoli sulla cardinalità dell'elemento e il tipo che vogliamo associargli. Il tipo specificato deve essere già definito nello schema e non devono esser presenti nella definizione del tipo oggetto della modifica sottoelementi con nome uguale a quello specificato. Ovviamente non è accettabile un valore per minOccurs minore del valore indicato per maxOccurs. L'inserimento avviene per mezzo della funzione ausiliaria *tree_insert_node*

$insert\_local\_elem : \mathcal{EN} \times \Gamma \times \mathcal{T} \times \mathcal{POS} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$insert\_local\_elem(l, (min, max), t, (p, j), \tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau \cup \{l\}, t_{sN}, \mathcal{R}_\tau \cup \{(l, t)\}) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

$t_{sN} = tree\_insert\_node(l, (min, max), (p, j), t_s)$
AC: $l \notin \mathcal{EN}_\tau$, $min \leq max$, $t \in \mathcal{T}$, $\tau \in \mathcal{T}$, $t_{sN} \neq t_s$

3.6.3 Inserimento per riferimento

Per inserire un sottoelemento in una definizione di tipo complesso senza specificare esplicitamente il tipo, occorre farlo per riferimento. Questo ci consente di gestire sia il caso in cui si vuole inserire un riferimento ad un elemento globale, sia il caso in cui si vuole inserire un elemento già presente nella definizione del tipo oggetto della modifica. Quindi, nel momento in cui si aggiunge un nuovo sottoelemento senza specificare il tipo, deve essere possibile determinare il tipo da assegnare all'elemento. Occorre verificare se esiste già un elemento locale con lo stesso nome e, se l'esito del controllo è negativo, occorre riscontrare se è stato indicato il nome di un elemento globale già dichiarato nello schema.

$insert\_ref\_elem : \mathcal{EN} \times \Gamma \times \mathcal{POS} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$insert\_ref\_elem(l, (min, max), (p, j), \tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau, t_{sN}, \mathcal{R}_\tau) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

$t_{sN} = tree\_insert\_node(l, (min, max), (p, j), t_s)$
AC: $l \in \{\mathcal{EN}_\tau \cup \mathcal{EN}_G\}$, $min \leq max$, $\tau \in \mathcal{T}$

3.6.4 Inserimento di operatore

La struttura di un tipo complesso può essere affetta delle modifiche degli operatori usati dalla nella rappresentazione ad albero. In questa primitiva bisogna specificare la posizione nell'albero di rappresentazione della struttura del tipo complesso dove il nuovo operatore deve essere inserito (l'inserimento avviene con l'utilizzo della funzione *tree_insert_node*).

$insert\_operator : \mathcal{OP} \times \Gamma \times \mathcal{POS} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$insert\_operator(op, (min, max), (p, j), \tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau, t_{sN}, \mathcal{R}_\tau) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

$t_{sN} = tree\_insert\_node(op, (min, max), (p, j), t_s)$
AC: $op \in \mathcal{OP}$, $min \leq max$, $\tau \in \mathcal{T}$

3.6.5 Cambio di operatore

Si può modificare un operatore in una `all` solo quando si tratta della radice della struttura, comunque non devono esser presenti sottostrutture ed i sottoelementi non devono avere cardinalità maggiore di uno.

$change\_operator : \mathcal{OP} \times \mathcal{POS} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$change\_operator(op, (p, j), \tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau, t_{sN}, \mathcal{R}_\tau) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

$t_{sN} = tree\_rename\_node(op, (min, max), (p, j), t_s)$
AC: $op \in \mathcal{OP}$, $min \leq max$, $\tau \in \mathcal{T}$

3.6.6 Cambio di cardinalità

Si possono imporre vincoli sulla cardinalità delle occorrenze sia dei sottoelementi che degli operatori. La primitiva che stiamo introducendo consente di effettuare entrambe le modifiche. Ovviamente i valori specificati per minOccurs e maxOccurs devono esser sensati, ovvero minOccurs e maxOccurs $\in \mathbb{N}$, minOccurs $\leq$ maxOccurs. Nel caso in cui si sta modificando il vincolo sulla cardinalità ammessa dei figli di una `all` occorre verificare che minOccurs = maxOccurs = 1. Questo controllo è effettuato attraverso la funzione ausiliaria *tree_rename_node* che restituisce la struttura inalterata in caso di scorrettezza.

$change\_cardinality : \Gamma \times \mathcal{POS} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$change\_cardinality((min, max), (p, j), \tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau, t_{sN}, \mathcal{R}_\tau) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

$t_{sN} = tree\_rename\_node(l, (min, max), (p, j), t_s)$
AC: $min \leq max$, $\tau \in \mathcal{T}$

3.6.7 Rinomina di un elemento locale

Questa primitiva consente di modificare il nome di un sottoelemento. Il sottoelemento in questione non può essere un riferimento ad un elemento globale. La primitiva prende come parametri il nome da sostituire, il nome che deve esser sostituito e il nome del tipo oggetto della modifica e va a modificare nella definizione del tipo il nome di tutti i sottoelementi che hanno come nome il nome che si sta rinominando. I problemi che possono nascere da questa operazione sono dovuti al fatto che potrebbe esistere già un sottoelemento con nome uguale a quello specificato come nuovo nome.

$rename\_local\_elem : \mathcal{EN} \times \mathcal{EN} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$rename\_local\_elem(l_O, l_N, \tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau \setminus \{l_O\} \cup \{l_N\}, t_{sN}, \mathcal{R}_\tau \setminus \{(l_O, \_)\} \cup \{(l_N, \_)\}) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

$t_{sN} = tree\_rename\_subelemens(l_O, l_N, t_s)$
AC: $l_O \in \mathcal{EN}_\tau$, $\tau \in \mathcal{T}$

3.6.8 Cambio del tipo di un elemento locale

La primitiva che verrà illustrata consente di modificare il tipo di un elemento locale. Inoltre il tipo specificato deve esser già stato definito nello schema. Infine se il vecchio tipo è un tipo anonimo, lo si può eliminare, insieme agli eventuali tipi anonimi discendenti dal tipo, effettuando *garbage collection* di tipi anonimi. Quest'ultima operazione richiede comunque dei controlli che potrebbero esser ritenuti superflui nei casi in cui la velocità dell'operazione ha importanza primaria.

$change\_type\_local\_elem : \mathcal{T} \times \mathcal{EN} \times \mathcal{S}_\mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$change\_type\_local\_elem(\tau_N, l, t_s, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau, t_s, \mathcal{R}_\tau \setminus \{(l, \tau_O)\} \cup \{(l, \tau_N)\}) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

AC: $\tau_N \in \mathcal{T}$, $l \in \mathcal{EN}_\tau$, $\tau \in \mathcal{T}$

3.6.9 Cancellazione di un elemento

Nel momento in cui si elimina un sottoelemento occorre verificare che questo non sia l'unico nodo figlio di una sottostruttura. La funzione ausiliaria *tree_remove_node* ci supporta per questo tipo di intervento e si occupa del controllo della validità della struttura modificata: essa restituisce la struttura inalterata in caso di scorrettezza. Infine, analogamente a quanto precedentemente discusso, per una questione di pulizia, nel caso in cui il sottoelemento in questione aveva un tipo anonimo si può effettuare *garbage collection* di tipi anonimi.

$remove\_elem : \mathcal{POS} \times \mathcal{T} \times \mathcal{SX} \to \mathcal{SX}$
$remove\_elem((p,j),\tau,(\mathcal{EN}_G,\mathcal{T},\rho,\mathcal{R}_G)) = (\mathcal{EN}_G,\mathcal{T},\rho_N,\mathcal{R}_G)$

$$\mathcal{EN}_{\tau N} = \begin{cases} (\mathcal{EN}_{\tau N} \setminus \{l\} & \text{se } \nexists \nu 1, \nu 2 \in \mathcal{V}_{t_s} : \varphi_{|_1}(\nu 1) = \varphi_{|_1}(\nu 2) = l \\ \mathcal{EN}_\tau & \textit{altrimenti} \end{cases}$$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_{\tau N}, t_{sN}, \mathcal{R}_{\tau N} & \text{se } x = \tau \\ \rho(x) & \textit{altrimenti} \end{cases}$$

$t_{sN} = tree\_remove\_node((p,j),t_s)$

$$\mathcal{R}_{\tau N} = \begin{cases} (\mathcal{R}_{\tau N} \setminus \{(l,\_)\} & \text{se } \nexists \nu 1, \nu 2 \in \mathcal{V}_{t_s} : \varphi_{|_1}(\nu 1) = \varphi_{|_1}(\nu 2) = l \\ \mathcal{EN}_\tau & \textit{altrimenti} \end{cases}$$

AC: $\tau \in \mathcal{T}$, $t_{sN} \notin t_s$

Ricordiamo che la notazione $\mathcal{V}_{t_s}$ individua l'insieme dei nodi della struttura e precisiamo che con $\varphi_{|_1}(\nu)$ si indica la prima componente dell'etichetta del nodo ν. Si ha sempre che $\varphi_{|_1}(\nu) \in \mathcal{EN} \cup \mathcal{OP}$.

3.6.10 Cancellazione di una sottostruttura

Questa primitiva consente di rimuovere un nodo e tutti i suoi figli da una struttura. Questa modifica utilizza la funzione ausiliaria *tree_remove_subtree*. Ovviamente questa primitiva se richiamata su un nodo foglia è equivalente alla primitiva *remove_elem*.

$remove\_substructure : \mathcal{POS} \times \mathcal{T} \times \mathcal{SX} \to \mathcal{SX}$
$remove\_substructure((p,j),\tau,(\mathcal{EN}_G,\mathcal{T},\rho,\mathcal{R}_G)) = (\mathcal{EN}_G,\mathcal{T},\rho_N,\mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau \setminus \{l_1,\ldots,l_n\}, t_{sN}, \mathcal{R}_\tau \setminus \{(l_1,\_),\ldots,(l_n,\_)\} & \text{se } x = \tau \\ \rho(x) & \textit{altrimenti} \end{cases}$$

$t_{sN} = tree\_remove\_subtree((p,j),t_s)$
AC: $\tau \in \mathcal{T}$, $t_{sN} \notin t_s$

3.6.11 Cancellazione di un operatore

La primitiva permette di rimuovere un operatore dalla struttura del tipo complesso. Questa modifica utilizza la funzione ausiliaria *tree_remove_node*

effettuando il tentativo di spostare le sottostrutture dell'operatore come figlie dell'operatore superiore. Ovviamente questa primitiva non può esser richiamata sulla radice di una struttura ed il nodo indicato deve avere come etichetta un operatore.

$remove\_operator : \mathcal{POS} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$remove\_operator((p,j),\tau,(\mathcal{EN}_G,\mathcal{T},\rho,\mathcal{R}_G)) = (\mathcal{EN}_G,\mathcal{T},\rho_N,\mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau, t_{sN}, \mathcal{R}_\tau) & \text{se } x = \tau \\ \rho(x) & altrimenti \end{cases}$$

$t_{sN} = tree\_remove\_node((p,j), t_s)$
AC: (p,j) non indica una foglia, $\tau \in \mathcal{T}$, $t_{sN} \notin t_s$

3.7 Primitive per gli elementi

Per concludere le primitive che si riferiscono agli elementi dello schema. Solitamente si tratta di elementi globali (quelli locali si riferiscono ai tipi complessi) ad eccezion fatta per i riferimenti.

3.7.1 Inserimento di un elemento globale

Dichiarando un elemento come globale è poi possibile far riferimento ad esso in una definizione di tipo complesso. Nel momento in cui si aggiunge un nuovo elemento globale occorre verificare che il nome specificato non corrisponda al nome di un elemento globale già dichiarato, altrimenti verrebbe a crearsi un conflitto. Inoltre il nome specificato per il tipo da associare all'elemento deve essere già definito all'interno dello schema.

$insert\_glob\_elem : \mathcal{EN} \times \mathcal{T} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$insert\_glob\_elem(l,\tau,(\mathcal{EN}_G,\mathcal{T},\rho,\mathcal{R}_G)) = (\mathcal{EN}_G \cup \{l\},\mathcal{T},\rho,\mathcal{R}_G \cup \{(l,\tau)\})$
AC: $l \in \mathcal{EN}_G$, $\tau \in \mathcal{T}$

3.7.2 Rinomina di un elemento globale

Affinché l'operazione non introduca errori occorre verificare che il nuovo nome specificato non corrisponda al nome di un elemento globale già presente nello schema, altrimenti verrebbe introdotto un conflitto di nomi. Inoltre occorre modificare il nome anche in tutte le definizioni di tipo che contengono un riferimento ad esso, altrimenti verrebbe introdotta una inconsistenza nello schema.

$rename\_glob\_elem : \mathcal{EN}_G \times \mathcal{EN} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$rename\_glob\_elem(l_O,l_N,(\mathcal{EN}_G,\mathcal{T},\rho,\mathcal{R}_G)) = (\mathcal{EN}_{GN},\mathcal{T},\rho_N,\mathcal{R}_{GN})$
$\mathcal{EN}_{GN} = \mathcal{EN}_G \setminus \{l_O\} \cup \{l_N\}$

$$\forall\tau \in \mathcal{T}\,\rho_N(\tau) = \begin{cases} (\mathcal{EN}_\tau, t_{sN}, \mathcal{R}_\tau) & \text{se } \rho(\tau) = (\mathcal{EN}_\tau, t_s, \mathcal{R}_\tau) \\ & \text{e } l_O \notin \mathcal{EN}_\tau \\ & \text{e } \exists\nu \in \mathcal{V}_{t_s} : \varphi(\nu) = l_O \\ \rho(\tau) & altrimenti \end{cases}$$

$t_{sN} = tree\_rename\_subelems(l_O, l_N, t_s)$
$\mathcal{R}_{GN} = \mathcal{R}_G \setminus \{(l_O, \tau)\} \cup \{(l_N, \tau)\}$
AC: $l \in \mathcal{EN}_G$, $l_N \notin \mathcal{EN}_G$

3.7.3 Cambio del tipo di un elemento globale

Affinché la modifica del tipo di un elemento globale restituisca uno schema ben formato occorre assicurarsi che il nuovo nome di tipo sia stato dichiarato in precedenza, altrimenti verrebbe introdotta una inconsistenza nello schema. Per una questione di pulizia, nel caso in cui l'elemento in questione aveva un tipo anonimo si può effettuare l'eliminazione del tipo locale, nonché l'eliminazione degli eventuali tipi anonimi annidiati nel tipo. Quest'ultima operazione permette di effettuare *garbage collection* della struttura dati che contiene i tipi. Questo controllo può esser eliminato nel caso in cui si voglia porre l'attenzione sulla velocità del sistema, trascurando la pulizia delle strutture dati.

$change\_type\_glob\_elem : \mathcal{EN} \times \mathcal{TN} \times \mathcal{SX} \to \mathcal{SX}$
$change\_type\_glob\_elem(l, \tau_N, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}_N, \rho_N, \mathcal{R}_{GN})$

$$\mathcal{T}_N = \begin{cases} \mathcal{T} \setminus \mathcal{R}_G(l) & \text{se } \mathcal{R}_G(l) \in \mathcal{AT} \\ \mathcal{T} & \text{se } \mathcal{R}_G(l) \in \mathcal{TT} \end{cases}$$

$$\rho_N = \begin{cases} \rho \setminus \{(\tau_O, \_)\} & \text{se } \tau_O \in \mathcal{AT} \\ \rho & \text{se } \tau_O \in \mathcal{TT} \end{cases}$$

$t_{sN} = tree\_remove\_node((p, j), t_s)$
$\mathcal{R}_{GN} = \mathcal{R}_G \setminus \{(l, \tau_O)\} \cup \{(l, \tau_N)\}$
AC: $l \in \mathcal{EN}_G$, $\tau \in \mathcal{T}$

3.7.4 Primitive migratorie

La necessità può sorgere nel trasformare i riferimenti ad elementi globali in elementi locali, oppure un elemento locale in un riferimento ad un elemento globale.

$ref\_to\_local : \mathcal{EN} \times \mathcal{T} \times \mathcal{SX} \to \mathcal{SX}$
$ref\_to\_local(l, \tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G, \mathcal{T}, \rho_N, \mathcal{R}_G)$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau \cup \{l\}, t_s, \mathcal{R}_\tau \cup \{(l, \mathcal{R}_G(l))\}) & \text{se } x = \tau \\ \rho(x) & \textit{altrimenti} \end{cases}$$

AC: $l \notin \mathcal{EN}_\tau$, $\tau \in \mathcal{T}$

Si può traformare un elemento locale in un riferimento se non esiste un elemento globale con lo stesso nome oppure l'elemento globale esiste ma il suo tipo e quello dell'elemento locale sono identici.

$local\_to\_ref : \mathcal{EN} \times \mathcal{T} \times \mathcal{SX} \to \mathcal{SX}$
$local\_to\_ref(l, \tau, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_{GN}, \mathcal{T}, \rho_N, \mathcal{R}_{GN})$

$$\mathcal{EN}_{GN} = \begin{cases} (\mathcal{EN}_G \cup \{l\} & \text{se } l \notin \mathcal{EN}_G \\ \mathcal{EN}_G & \textit{altrimenti} \end{cases}$$

$$\rho_N(x) = \begin{cases} (\mathcal{EN}_\tau \setminus \{l\}, t_s, \mathcal{R}_\tau \setminus \{(l, \_)\}) & \text{se } x = \tau \\ \rho(x) & \textit{altrimenti} \end{cases}$$

$$\mathcal{R}_{G\,N} = \begin{cases} \mathcal{R}_G \cup \{(l, \mathcal{R}_\tau(l))\} & \text{se } l \notin \mathcal{EN}_G \\ \mathcal{R}_G & \textit{altrimenti} \end{cases}$$

AC: $l \in \mathcal{EN}_\tau$, $\tau \in \mathcal{T}$

3.7.5 Cancellazione di un elemento globale

La cancellazione di un elemnto globale è permesso ogni volta che non esiste un riferimento a tale elemento all'interno del documento.

$remove\_glob\_elem : \mathcal{EN} \times \mathcal{SX} \rightarrow \mathcal{SX}$
$remove\_glob\_elem(l, (\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)) = (\mathcal{EN}_G \setminus \{l\}, \mathcal{T}_N, \rho_N, \mathcal{R}_G \cup \{(l, \_)\})$

$$\mathcal{T}_N = \begin{cases} \mathcal{T} \setminus \{\mathcal{R}_G(l)\} & \text{se } \mathcal{R}_G(l) \in \mathcal{AT} \\ \mathcal{T} & \text{se } \mathcal{R}_G(l) \in \mathcal{TT} \end{cases}$$

$$\rho_N(x) = \begin{cases} \rho \setminus \{(\mathcal{R}_G(l), \_)\} & \text{se } \mathcal{R}_G(l) \in \mathcal{AT} \\ \rho & \text{se } \mathcal{R}_G(l) \in \mathcal{TT} \end{cases}$$

AC: $l \in \mathcal{EN}_G$, $\nexists \nu \in \mathcal{V}_{t_s} : \varphi(\nu) = l \wedge \varphi(\nu) \notin \mathcal{EN}_\tau$

CAPITOLO 4

Validazione incrementale ed adattamento al nuovo schema

Dopo aver trattato le varie evoluzioni sugli schemi, in questo capitolo parleremo delle ripercussioni che hanno tali evoluzioni hanno sui documenti XML che risultavano validi prima dell'esecuzione delle primitive. In particolare nella Sezione 4.1 parleremo di come poter controllare piuttosto velocemente se il documento XML è ancora valido rispetto al nuovo schema modificato, invece nella Sezione 4.2 diremo come è possibile adattare il documento al nuovo schema in modo che risulti nuovamente valido.

4.1 La validazione incrementale dei documenti XML

Dato un documento d valido per uno schema sx vogliamo capire se d continua ad essere valido dopo che su sx è stata applicata una primitiva di evoluzione riassunti in Tabella 3.1.

Un approccio naïf al problema è quello di applicare un algoritmo standard di validazione tra il documento d ed il nuovo schema sx_N, ottenuto dal vecchio sx_O applicando una primitiva p_e. Questo approccio però non cosidera due aspetti:

1. non tutte le operazioni che alterano lo schema vanno ad incidere sulla validità del documento e la validità può essere verificata andando a controllare solo una piccola e ridotta parte del documento;

2. applicando pedissequamente un'algoritmo standard non si tiene in considerazione l'informazione semantica che il documento d era valido rispetto ad sx_O.

Per questo secondo motivo il nostro algoritmo viene considerato come un approccio di validazione incrementale di un documento verso uno schema che contiene una variazione rispetto allo schema per il quale il documento era valido.

4.1.1 Concetti preliminari

La struttura di un tipo complesso stabilisce quali elementi e in quale ordine devono apparire come sottoelementi di un dato elemento. Quindi la struttura di un tipo complesso può essere vista come una grammatica che stabilisce le sequenze corrette di sottoelementi. Per tale motivo introduciamo le seguenti due operazioni:

- $init : \mathcal{S}_\mathcal{T} \rightarrow 2^{\mathcal{EN} \cup \{\lambda\}}$. Dato un struttura di tipo la funzione *init* restuisce l'insieme dei nomi di elementi inizialmente attesi in una alla struttura di tipo. L'insieme può contenere il simbolo λ ad indicare che la struttura di tipo ammette che l'elemento abbia contenuto vuoto.
- $nextElems : \mathcal{S}_\mathcal{T} \times \mathcal{EN} \rightarrow 2^{\mathcal{EN} \cup \{\lambda\}}$. Questa funzione restituisce il prossimo insieme di nomi di elementi attesi dopo aver incontrato un elemento imposto dalla struttura di tipo. Il suo funzionamento è analogo alla funzione *init*, infatti anche in questo caso l'insieme ottenuto può contenere il simbolo λ ad indicare che la struttura di tipo ammette che l'elemento abbia contenuto vuoto.

Esempio 9 *Si consideri la funzione init e si consideri il tipo* t2 *della Tabella 2.2. Applicando la funzione init alla struttura del tipo* t2 *si ottiene l'insieme* $S = \{title\}$.

Esempio 10 *Si consideri la funzione nextElems e si consideri il tipo* t2 *della Tabella 2.2. Applicando la funzione nextElems alla struttura del tipo* t2 *ed all'elemento* genre *si ottiene l'insieme* $S = \{short - description, rating\}$. *Se invece l'elemento passato alla funzione fosse* rating *all'ora si otterrebbe l'insieme* $S = \{\lambda\}$.

Per convenzione introduciamo una funzione *is_valid* che valida un documento rispetto ad uno schema o un elemento rispetto ad un tipo utilizzando una qualsiasi tecnica standard di validazione.

$$valids([T_1, \ldots, T_n], S, t_s) = \begin{cases} valids([T_2, \ldots, T_n], nextElems(t_s, \varphi(T_1)), t_s) & \text{se } \varphi(T_1) \in S \wedge \\ & is\_valid(T_1, \mathcal{R}(\varphi(T_1))) \\ \textbf{true} & \text{se } n = 0 \wedge \lambda \in S \\ \textbf{false} & altrimenti \end{cases}$$

Sulla base delle funzioni *init* e *nextElems* opera la funzione *valids*, riportata sopra, che prende in input una lista di alberi $L_T = [T_1, \ldots, T_n]$ che sono fratelli nel documento (avendo lo stesso genitore), l'insieme di nomi di elementi attesi in base a t_s (inizialmente quelli restituiti dalla funzione *init*, nelle chiamate ricorsive quelli restituiti dalla funzione *nextElems*) ed una struttura di tipo t_s. Se l'etichetta del primo elemento T_1 di L_T appartiene ad S ed il contenuto di T_1 è valido per il suo tipo, allora la funzione *valids* è invocata sul resto della lista L_T e sui successivi elementi della struttura t_s. Se invece la lista L_T è vuota e tra gli elementi attesi c'è il simbolo λ allora la lista rispetta i vincoli imposti da t_s e la funzione restituise **true**. In tutti gli altri casi viene restituito **false** ad indicare che la lista non rispetta la struttura t_s.

Esempio 11 *Si consideri la funzione valids e si consideri il tipo* t2 *della Tabella 2.2. Aquesto punto passiamo alla funzione valids i figli del primo nodo* movie *della Figura 2.1, la init*(ts) *e la stessa struttura* t2. *A questo punto avremo la*

lista degli alberi $L_T = [T_1, \ldots, T_5]$ *dei cinque sottoelementi di* movie*, tali che* $\varphi(T_1) = title$ *e* $\varphi(T_5) = rating$*, oltre all'insieme* $S = \{title\}$ *ottenuto dalla chiamata di init. Siccome* $\varphi(T_1) \in S$ *e risulta vera la funzione* $is\_valid(T_1, string)$ *allora la valids sarà invocata su* $[T_2, \ldots, T_5]$ *con* $\varphi(T_2) = cast$*, sul nuovo insieme insieme* $nextElems(t_s, title) = \{cast\}$ *e sempre sulla struttura* t_s*. Alla fine se tutti gli alberi saranno risultati validi allora la funzione valids restituirà* ***true****.*

La funzione *getPaths* può prendere in input diversi tipi di valori (il nome di un elemento globale, il nome di un tipo, la struttura di un tipo) oltre ad uno schema, e restituisce l'insieme dei cammini (desumibili dallo schema e rappresentati utilizzando Xpath) che portano ad individuare elementi con tale nome, con tale tipo, con tale struttura. La funzione *getPaths* opera sullo schema e non va a toccare in alcun modo il documento. La funzione *getElems*, invece, opera sul documento e restituisce l'insieme di elementi contenuti nel documento d che sono raggiunti da una delle espressioni Xpath contenuti indicate in input.

Esempio 12 *Si consideri la funzione getPaths e lo schema sx rappresentato in Tabella 2.2. Applicando la funzione getPaths alla struttura di tipo* t_2 *lo schema sx si ottiene l'insieme* $\{/movies/movie\}$*.*

Esempio 13 *Si consideri la funzione getElems ed il documento d di Figura 2.1. Applicando la funzione getElems all'insieme* $\{/movies\}$ *ed al documento d si ottengono gli elementi* $\{movie, movie\}$ *dove il primo movie riguarda il film "Il laureato", mentre il secondo movie si riferisce ad "Amarcord".*

4.1.2 L'algoritmo di validazione incrementale

L'algoritmo di validazione incrementale che abbiamo sviluppato prende in input uno schema sx, un documento d valido per sx e l'operazione di modifica $p_e \in \mathcal{P}$. L'algoritmo restituisce **true** qualora il documento continui ad essere valido per lo schema sx_N ottenuto da sx applicando la primitiva p_e, **false** altrimenti. L'algoritmo si basa sulla pre-condizione che la primitiva di evoluzione sia applicabile allo schema in oggetto come discusso nel capitolo precedente.

La verifica di validità viene effettuata a seconda della primitiva utilizzata tra quelle riportate in Tabella 3.1. L'algoritmo è pensato per decidere la validità del documento direttamente dall'analisi dello schema sx, fintanto che è possibile e di considerare il documento d sono in caso di effettiva necessità. Questa scelta rende il nostro algoritmo molto più efficiente delle normali tecniche di validazione. Inoltre, per semplicità di esposizione, l'algoritmo viene presentato per lavorare su un singolo documento alla volta, la versione che abbiamo implementato invece consente di lavorare su un insieme di documenti rendendo quindi ancora più efficiente il nostro approccio.

Se la primitiva p_e è tra quelle marcate con * la sua applicazione non altera sicuramente la validità di d ed è possibile dire immediatamente che d continua ad essere valido.

Quando la primitiva p_e richiede di modificare il nome di un elemento (sia esso globale o locale) o di rimuovere un elemento, la validità del documento dipende dall'occorrenza di tale elemento nel documento. Per questo motivo si identificano le occorrenze di tale elemento nel documento e se non ce ne sono

Algoritmo 1: revalidate

Dati: p_e, d, sx
p_e: primitiva di evoluzione
d: documento XML
sx: schema XML
$sx \times p_e \rightarrow sx_N$ `// ` sx_N `è lo schema ottenuto applicando` p_e `a` sx
Risulta: **true** $\iff$ $is\_valid(d, sx_N)$

```
1  switch p_e do
2      case p_e ∈ P*
3          return true
4
5      case p_e ∈ {rename_glob_elem, rename_local_elem, remove_elem}
6          if getElems(getPaths(l_O, sx), d) = ∅ then return true
7          else return false
8
9      case p_e = remove_glob_elem(l, sx)
10         if φ(root(d)) = l then return false
11         else return true
12
13     case p_e = {change_type_local_elem, change_type_glob_elem} ∧ τ_N ∈ CT
14         if p_e = change_type_local_elem then path ← getPaths(ST(t_s), sx)
15         else path ← getPaths(l, sx)
16         E ← getElems(path, d)
17         if ∃e ∈ E : valids(children(e), init(t_sN), t_sN) = false then
18             return false
19         end
20         else return true
21
22     case p_e ∈
       {change_restriction, change_[base|item]_type, change_type_[local|glob]_elem}
23         if τ_O ⊑ τ_N then return true
24         C ← getContents(getPaths(τ_O, sx), d)
25         if ∃c ∈ C : c ∉ [[τ]] then return false
26         else return true
27
28     case p_e ∈ P^{S_T}
29         return validateStructureType(t_s, p_e, d, sx)
30     end
31 end
```

viene restituito che d continua ad essere valido. Se esiste una sola occorrenza di tale elemento in d il documento non è più valido.

Se la primitiva di evoluzione richiede di rimuovere un elemento globale, basta verificare che la radice del documento non abbia lo stesso nome per affermare che il documento continua ad essere valido. Questo è l'unico controllo da effettuare perché, nelle condizioni di applicabilità della primitiva di rimozione di un elemento globale si richiede, per effettuare la cancellazione, che non ci siano elementi nello schema che riferiscano a tale elemento.

Nel caso in cui la primitiva p_e richieda di cambiare il tipo di un elemento e globale o locale ed il nuovo tipo sia di tipo complesso, bisogna identificare nel documento tutti gli elementi e e si verifica per ogni elemento, attraverso la funzione *valids*, che la sequenza dei suoi figli corrisponda alla sequenza imposta dal nuovo tipo. Se esiste un elemento per cui la condizione non è verifica il documento non è più valido.

Se la primitiva di evoluzione richiede di modificare un tipo semplice definito dall'utente attraverso i costrutti `list`, `restriction` ed `union`, oppure viene cambiato il tipo di un elemento (sia locale che globale) assegnando ad esso un nuovo tipo semplice. Per prima cosa dobbiamo verificare, dall'analisi dei tipi, se è possibile affermare che i valori validi per il vecchio tipo sono contenuti nei valori validi per il nuovo tipo. In tal caso la validità del documento è preservata. Se così non è, occorre individuare nel documento i contenuti testuali definiti sul tipo precedente e verificare che continuino ad essere validi per il nuovo tipo. In caso contrario, il documento non è più valido.

Algoritmo 2: validateStructureType

Dati: t_s, p_e, d, sx
t_s: struttura da modificare
p_e: primitiva di evoluzione
d: documento XML
sx: schema XML
$sx \times p_e \rightarrow sx_N$ `//` sx_N `è lo schema ottenuto applicando` p_e `a` sx
Risulta: **true** $\iff$ $is\_valid(d, sx_N)$

```
1  if ∀en ∈ EN, R(en) ≠ type(t_s) then return true
2  if p_e = change_cardinality(min_N, max_N, p, t_s, sx) then
3      if (min_N ≤ min_O ∧ max_N ≥ max_O) then return true
4  end
5  if p_e = change_operator(op_N, p, t_s, sx) then
6      if (op_O = sequence ∧ op_N = all) ∨ (|children(op_O)| = 1) then return true
7  end
8  E ← getElems(getPaths(t_s, sx), d)
9  if ∃e ∈ E : valids(children(parent(e)), init(t_s), t_s) = false then
10     return false
11 end
12 else return true
```

Se la primitiva di evoluzione richiede di effettuare la modifica della struttura di un tipo tra quelle contenute nell'insieme $\mathcal{P}^{\mathcal{S}_T}$, il controllo di validità viene rinviato all'Algoritmo 2 *validateStructureType* che effettua controlli sulla base della struttura del tipo oggetto della modifica.

Questo algoritmo effettua un insieme di controlli sullo schema che, se verificati, permettono di affermare la validità di d senza dover accedere al documento. Si verifica se il tipo associato alla struttura da modificare è utilizzato per dichiarare elementi nello schema. Se non è utilizzato allore le modifiche allo schema non riguardano alcun elemento del documento per cui il documento continua ad essere valido. Se la modifica alla struttura va a cambiare le cardinalità (minime o massime), nel caso in cui la cardinalità minima iniziale sia maggiore della nuova e la cardinalità massima iniziale sia minore della nuova, allora il documento continua ad essere valido. Se la modifica alla struttura riguarda un operatore, nel caso si passasse da una `sequence` ad una `all` oppure il gruppo dell'operatore è composto da un solo elemento, il documento rimane valido senza bisogno di modifiche.

Se nessuno dei precedenti casi è verificato occorre andare ad identificare nel documento tutti gli elementi che presentano la struttura iniziale (t_s) e si verifica per ogni elemento, attraverso la funzione *valids*, che la sequenza dei suoi figli corrisponda alla sequenza imposta dalla nuova struttura (t_{sN}). Se esiste un elemento per cui la condizione non è verifica il documento non è più valido.

4.2 L'adattamento dei documenti rispetto allo schema nuovo

In questa sezione presentiamo l'algoritmo che permette di ristrutturare un documento quando questo non risulta più valido per il nuovo schema. L'algortimo è una estensione dell'algoritmo di validazione presentato nella sezione precedente in cui, nel momento in cui si determina che il documento non è più valido, si apportano le modifiche minime per rendere il documento nuovamente valido. Le operazioni di modifica del documento possono riguardare il cambiamento del nome di un elemento, la cancellazione di un elemento con tutto il suo contenuto, l'inserimento di un elemento. Nel caso venga inserito un elemento, a questo viene assegnato un valore di default per il tipo dell'elemento nel caso di un elemento testuale, una struttura minima nel caso si tratti di un elemento con una struttura. Le modifiche sono minime perché vengono effettuate solo sulla parte del documento oggetto della primitiva di modifica e perché elimina (o aggiunge) il minimo numero di elementi necessario per riottenere la validità del documento.

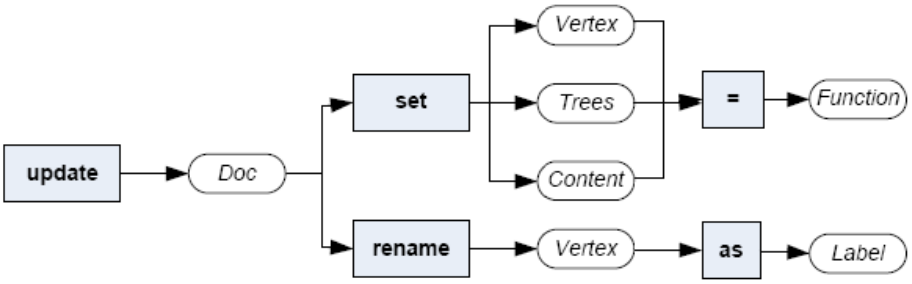
Nel resto della sezione, prima presentiamo alcuni concetti preliminari utili per comprendere il funzionamento dell'algoritmo di ristrutturazione e poi l'algoritmo di ristrutturazione stesso.

4.2.1 Concetti preliminari

Al fine di semplificare la presentazione dell'algoritmo di ristrutturazione abbiamo introdotto un semplice linguaggio basato su SQL per specificare le modifiche da effettuare su un documento XML.

La Figura 4.1 riporta l'albero sintattico del linguaggio sviluppato. I nodi rettangolari rappresentano parole chiavi mentre i nodi ovali rappresentano parametri specificati dagli utenti. Il linguaggio permette di modificare un documento andando a cambiare un elemento di un documento XML, una lista di sottoalberi (figli dello stesso elemento) oppure il contenuto testuale di un

Figura 4.1: Linguaggio dichiarativo per la modifica di un documento XML



elemento testuale. Il nuovo valore può essere specificato attraverso opportune funzioni che restituiscono valori appropriati a secondo dell'oggetto della modifica. Inoltre, il linguaggio permette di rinominare elementi specificando la nuova etichetta.

Come l'algoritmo di rivalidazione, quello di ristrutturazione (Algoritmo 3) considera un documento d valido per uno schema sx sul quale viene invocata la primitiva di evoluzione p_e. A seconda della primitiva invocata, l'algoritmo stabilisce se d continua ad essere valido così com'è o effettua delle modifiche su d per portarlo ad essere nuovamente valido. L'algoritmo viene eseguito solo se le condizioni di applicabilità delle primitive sono verificate, altrimenti le modifiche non sono realizzate.

4.2.2 L'algoritmo di ristrutturazione dei documenti

Se la primitiva p_e appartiene all'insieme di primitive $\mathcal{P}^*$ che non alterano la validità dei documenti, il documento d non necessita di modifiche per essere valido per il nuovo schema sx_N.

Nel cao in cui la primitiva p_e richiedesse di rinominare un elemento etichettato l_O (sia esso locale o globale) al fine di mantenere la validità occorre individuare nel documento tutti gli elementi che riferiscono ad l_O e procedere nel cambiare loro l'etichetta con l_N. Questa operazione rende valido il documento d per il nuovo schema sx_N.

Se la primitiva richiede di eliminare un elemento globale, occorre verificare se tale elemento è la radice del documento d. In tale caso d viene posto a **null** e quindi l'intero documento rimosso. Ricordiamo che nelle condizioni di applicabilità di questa primitiva c'è quella che richiede che non devono esistere elementi che riferiscono all'elemento globale da cancellare per poter effettuare l'operazione.

Nel cao in cui la primitiva p_e richiedesse di rimuovere un elemento etichettato l dalla struttura di un tipo t_s, non è sufficiente individuare tutti gli elementi e definiti in accordo alla struttura t_s e rimuovere tutti i sottoelementi etichettati l. Infatti, l potrebbe essere legato attraverso l'operatore `choice` ad un altro elemento l_1 nella struttura t_s con il vincolo che uno dei due deve obbligatoriamente essere presente nel documento. Quindi l'eliminazione di l impone l'introduzione di l_1.

Per affrontare questa situazione, viene invocata per ogni elemento e, la funzione *updNode* (Algoritmo 4) su tutti i sottoelementi di e, sull'insieme di ele-

Algoritmo 3: restructure

Dati: p_e, d, sx
p_e: primitiva di evoluzione
d: documento XML
sx: schema XML
$sx \times p_e \rightarrow sx_N$ // sx_N è lo schema ottenuto applicando p_e a sx
Risulta: $d : is\_valid(d, sx_N)$

```
1  switch p_e do
2      case p_e ∈ P*
3          return d
4
5      case p_e ∈ {rename_glob_elem, rename_local_elem}
6          foreach e ∈ getElems(getPaths(l_O, sx), d) do update d rename e as l_N
7
8      case p_e = remove_glob_elem(l, sx)
9          if φ(root(d)) = l then d ← null
10
11     case p_e = remove_elem((p, j), τ, sx)
12         foreach e ∈ getElems(getPaths(τ, sx), d) do
13             update d set children(e) ← updNode(children(e), init(t_sN), t_sN, del)
14         end
15
16     case p_e = {change_type_local_elem, change_type_glob_elem} ∧ τ_N ∈ CT
17         if p_e = change_type_local_elem then path ← getPaths(ST(t_s), sx)
18         else path ← getPaths(l, sx)
19         foreach e ∈ getElems(path, d) do
20             if valids(children(e), init(t_sN), t_sN) = false then
21                 update d set children(e) ← updNode([], init(t_sN), t_sN, ins)
22             end
23         end
24
25     case
       p_e ∈ {change_restriction, change_[base|item]_type, change_type_[local|glob]_elem}
26         if τ_O ⋢ τ_N then
27             foreach e ∈ getElems(getPaths(τ_O, sx), d) : content(e) ∉ [[τ_N]] do
28                 update d set content(e) ← defaultContent(τ_N)
29             end
30         end
31
32     case p_e ∈ P^ST
33         if ST(t_sO) ⋢ ST(t_sN) then
34             foreach e ∈ getElems(getPaths(t_sO, sx), d) do
35                 if delElems(p_e) then
36                     update d set children(e) ← updNode(children(e), init(t_sN), t_sN, del)
37                 end
38                 if insElems(p_e) then
39                     update d set children(e) ← updNode(children(e), init(t_sN), t_sN, ins)
40                 end
41             end
42         end
43
44 end
45 return d
```

Algoritmo 4: updNode

Dati: $e, \mathcal{L}, S, t_s, sx, action$
L_T: $[T_1, \ldots, T_n]$ lista di alberi figli dello stesso elemento
S: insieme degli elementi che ci si attende
t_s: struttura modificata
$action$: specifica se occorre inserire (`ins`) o rimuore (`del`) figli di e
Risulta: $children(e) : is\_valid(e, t_s)$

```
1  l_1 ← φ(T_1)
2  τ_{l_1} ← R(l_1)
3  if l_1 ∈ S then
4      if is_valid(T_1, τ_{l_1}) = false then T_1 ← genTree(l_1, τ_{l_1})
5      return T_1 · updNode([T_2, ..., T_n], nextElems(t_s, l_1), t_s, action)
6  end
7  if l_1 ∉ S then
8      choose l ∈ S il cui tipo è τ
9      if action = ins then return updNode([T_1, ..., T_n], S, t_s, action)
10     else
11         return genTree(l, τ) · updNode([T_2, ..., T_n], nextElems(t_s, l), t_s, action)
12     end
13 end
14 while λ ∉ S do
15     choose l ∈ S il cui tipo è τ
16     L_T ← L_T · genTree(l, τ)
17     S ← nextElems(t_s, l)
18 end
19 return L_T
```

menti attesi dalla struttura di tipo t_{sN} e sulla stessa t_{sN} specificando che si richiede l'eliminazione di sottoelementi.

Questa funzione verifica che i figli di e appaiano come indicato dalla struttura modificata t_{sN} e nel caso un sottoelemento di e non rispetti la grammatica questo viene eliminato. Se invece la grammatica richiede un elemento non presente tra i figli di e questo viene prima generato e poi inserito.

Se la primitiva p_e richiede di cambiare il tipo di un elemento (globale o locale) ed il nuovo tipo è complesso, si identificano tutti gli elementi del documento d del tipo originale. Per ogni elemento e si verifica se i figli rispettino i vincoli imposti dal nuovo tipo. In caso negativo, si elimina il contenuto dell'elemento e si genera un nuovo contenuto sulla base del nuovo tipo.

La cancellazione del contenuto e la creazione del nuovo contenuto vengono realizzate anch'esse attraverso la funzione *updNode* passando come parametro una lista vuota di figli di e, utilizzando un'espressione del linguaggio di update che permette di specificare i nuovi figli di un elemento. La validità del documento viene garantita perché si possono verificare due casi:

1. ogni elemento e continua ad essere valido per il nuovo schema (quindi non si effettua alcuna modifica);

2. gli elementi e non sono validi per il nuovo schema, allora si butta via il vecchio contenuto e si inserire un contenuto nuovo che rispetta il nuovo tipo.

Se invece il nuovo tipo è semplice oppure la primitiva p_e richiede di effettuare modifiche su un tipo semplice definito per `list`, `restriction` oppure `union`, per prima cosa si verifica se siamo nella situazione in cui anche il tipo originale

era semplice e i valori validi per il tipo originale sono contenuti nei valori validi per il nuovo tipo. In tal caso il documento non richiede modifiche. Altrimenti, occorre individuare tutti gli elementi del documento del tipo originale che non vanno bene per il nuovo tipo e assegnare loro un valore di default per il nuovo tipo. Anche in questo caso viene garantita la validità del documento modificato in quanto o i valori continuano ad essere validi per il nuovo tipo o gli si assegna un valore di default che rispetta i vincoli del nuovo tipo.

Tabella 4.1: Valori assegnati di default ai nuovi elementi

Tipo	**Valore**
string	
normalizedString	
token	
base64Binary	B64A
hexBinary	0A
integer	0
positiveInteger	1
negativeInteger	-1
nonNegativeInteger	0
nonPositiveInteger	0
long	0
unsignedLong	0
int	0
unsignedInt	0
short	0
unsignedShort	0
byte	0
unsignedByte	0
decimal	0
float	0
double	0
boolean	0
duration	PT1S
dateTime	2000-01-01T00:00:00
date	2000-01-01
time	00:00:00
gYear	2000
gYearMonth	2000-01
gMonth	`--01--`
gMonthDay	`--01-01`
gDay	`---01`
anyURI	http://www.w3.org/
language	en-GB

I valori di defaul da asseganre ad un nuovo elemento (elencati in Tabella 4.1) sono stati scelti arbitrariamente facendo in modo che il loro valore possa influenzare il meno possibile il contenuto del documento, per esempio agli elementi di

tipo stringa conterranno una stringa vuota, invece quelli di tipo numerico il numero 0.

Se la primitiva di evoluzione p_e richiede di effettuare una modifica della struttura di un tipo tra quelle contenute in $\mathcal{P}^{\mathcal{S}_\mathcal{T}}$, la modifica del documento dipende da diversi fattori che vengono analizzati dalle funzioni *delElem* e *insElem*. Le modifiche che un documento può subire riguardano la diminuzione o l'aumento di elementi in modo che esso risulti nuovamente valido. Per esempio se nello schema vengono inseriti nuovi elementi (con occorrenza minima maggiore di 1) oppure viene modificata una `choice` in una `all`, od in una `sequence`, nel documento dovranno essere inseriti il minimo numero di elementi necessari a far risultare il documento ancora valido. Nel caso in cui, invece, venisse diminuita l'occorrenza massima ad un elemento o venisse addirittura eliminato un elemento stesso, bisogna accedere al documento e cancellare il minimo numero di elementi necessari a far rendere il di nuovo il documento valido per lo schema.

Una volta effettuati i controlli sull'operatore di modifica e applicate le eventuali modifiche al documento, il documento ottenuto viene restituito. Tale documento risulta essere valido per il nuovo schema.

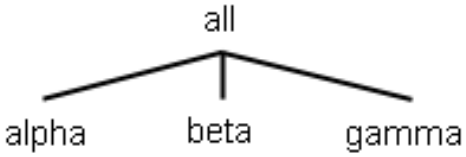
4.2.3 Aumento/diminuzione degli elementi

La *delElems* è una funzione che data una primitiva $p_e \in \mathcal{P}^{\mathcal{S}_\mathcal{T}}$ ed uno schema restituisce **true** se l'esecuzione di tale primitiva comporta una diminuzione del numero degli elementi sul documento per rendere tale documento ancora valido per lo schema modificato, altrimenti restiruisce **false**. Il comportamento della *insElems* è analogo, solo che restituisce **true** nel caso gli elementi debbano essere aumentati e **false** altrimenti.

Prendendo come riferimento le primitive $\mathcal{P}^{\mathcal{S}_\mathcal{T}}$ della Tabella 3.1, la funzione *delElems* restituirà **true** per le primitive che si trovano sotto la colonna Deletion, invece la funzione *insElems* restituirà **true** per quelle primitive poste nella colonna Insertion (eccezion fatta per la *insert_operator* nel caso in cui il nuovo operatore sia un `choice`, in questo caso a restituire **true** sarà la *delElems*, perché passando ad un operatore opzionale bisogna tenere solo uno dei suoi sottoelementi e cancellare gli altri). Discorso a parte meritano le primitive sotto la colonna Modification, più specificatamente si tratta delle primitive *change_operator* e *change_cardinality*.

La *change_operator* necessita una diminuzione di elementi nel documento (*delElems* restituisce **true**) nel caso in cui il nuovo operatore sia una `choice`, infatti se da un operatore `all` oppure da una `sequence` si passasse ad una `choice` vanno eliminati tutti i sottoelementi dell'operatore ad eccezione di uno. Se il nuovo operatore fosse una `sequence` bisognerebbe controllare il vecchio operatore che viene rimpiazzato, infatti se quello vecchio è una `choice` allora dovranno essere inseriti nuovi elementi e si avrà *insElems* che restituirà **true**, nel caso in cui quello vecchio fosse una `all` a restituire **true** sarebbe la *delElems* perché verrebbero eliminati gli elementi non in sequenze, poi rimpiazzati da nuovi elementi grazie al ciclo finale della *updNode* che inserisce nuovi elementi nel caso non ci fossero più figli e $\lambda \notin S$. Unica nota negativa è che in quest'ultimo caso si correrebbe il rischio di perdere il contenuto di alcuni nodi che verrebbero rimpiazzati da altri con il contenuto di default. In caso che il nuovo operatore sia una `all` gli elementi vanno sempre aumentati quindi a restituire **true** sarà sempre la *insElems*.

Figura 4.2: Esempio di struttura per cambio operatore



Listato 4.1: operatorBegin.xml

```
...
<alpha>1</alpha>
<gamma>2</gamma>
<beta>3</beta>
...
```

Listato 4.2: operatorLast.xml

```
...
<alpha>1</alpha>
<beta>3</beta>
<gamma>0</gamma>
...
```

Esempio 14 *Dato un elemento che ha come tipo la Figura 4.2, nel caso in cui al posto della* `all` *venisse assegnata una* `sequence` *ed il documento Listato 4.1 da ristrutturare. Verrebbe eliminato l'elemento* gamma *ed assegnato uno nuovo in fondo avendo come risultato finale il Listato 4.2. A questo punto l'elemento risulta valido per la struttura ad esso assegnata, ma come detto prima si è perso il contenuto del vecchio gamma. Abbiamo supposto che gli elementi siano di tipo numerico quindi in base alla Tabella 4.1 di default a* gamma *viene assegnato il valore* 0.

Anche la *change_cardinality* è un caso complesso e le funzioni *delElems* ed *insElems* restituiscono un risultato diverso a seconda dei valori di occorrenza dello schema vecchio rispetto ai valori nuovi.

In Tabella 4.2 sono riportate le azioni da compiere nel documento a seconda che i valori di *min* e *max* aumentano, diminuiscano o rimangono costanti. Nel caso in cui il valore di *min* diminuisca e contemporaneamente aumenti il valore di *max* la primitiva viene richiamata due volte. La prima volta prendendo come valore di *max* il nuovo valore e come *min* il vecchio valore (così che a risultare **true** sarà la *delElems*), la seconda volta invece prendendo i nuovi valori sia di *min* che di *max* (in questo caso sarà la *delElems* a risultare **true**). Così facendo prima vengono diminuiti gli elementi in sovrappiù e poi aumentati nei punti dove è necessario. Si è deciso di procedere prima alla diminuzione e poi all'aumento degli elementi per cercare di evitare che le strutture su cui lavoriamo diventino troppo grandi con inevitabile decadimento di prestazioni. Nella tabella non è riportata nessuna azione nei casi in cui il documento rimane valido per il nuovo schema e quindi non subisce nessuna modifica (sia la *insElems* che la *delElems* restituiscono **false**).

Tabella 4.2: Azione sui nodi per l'aumento o la diminuzione di occorrenze

min	*max*	*action*
$>$	$>$	`ins`
$>$	$=$	`ins`
$>$	$<$	`del` & `ins`
$=$	$<$	`del`
$=$	$>$	
$=$	$>$	
$=$	$=$	
$<$	$>$	
$<$	$=$	
$<$	$<$	`del`

Tabella 4.3: Semplificazione dell'azione sui nodi

min	*max*	*action*
$>$	$<$	`del` & `ins`
$\leq$	$\geq$	
$>$	$\geq$	`ins`
$\leq$	$<$	`del`

Nella Tabella 4.3 è riportata una semplificazione della Tabella 4.2, comprimendo i casi simili. Ricordiamo che non può essere assegnato a *min* un valore superiore a quello assegnato a *max*.

Listato 4.3: occursBegin.xml

```
...
<pippo>1</pippo>
...
<pippo>1</pippo>
<pippo>2</pippo>
<pippo>3</pippo>
<pippo>4</pippo>
...
```

Listato 4.4: occursFirst.xml

```
...
<pippo>1</pippo>
...
<pippo>1</pippo>
<pippo>2</pippo>
<pippo>3</pippo>
...
```

Listato 4.5: occursLast.xml

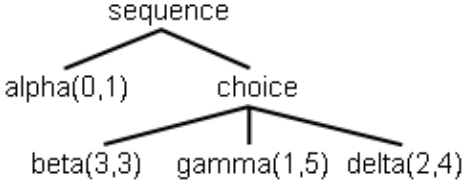
```
...
<pippo>1</pippo>
<pippo>0</pippo>
...
<pippo>1</pippo>
<pippo>2</pippo>
<pippo>3</pippo>
...
```

Esempio 15 *Nel caso in cui si avesse nello schema un elemento dichiarato:*
`<xs:element name="pippo" type="int" minOccurs="1" maxOccurs="4" />`
ed il documento del Listato 4.3 con l'elemento pippo che in un punto è ripetuto 1 *volta sola ed in un'altro punto* 4 *volte. Supponiamo che venga effettuata sullo schema una change_cardinality dell'elemento di nome* pippo *in modo che il min passasse a* 2 *ed il max passasse a* 3. *La restructure prima diminuisce le occorezze di* pippo *dove necessario, e nel documento si avrebbe la situazione del Listato 4.4. Dopodiché verrebbero aumentati gli elementi dove mancano avendo alla fine la situazione del Listato 4.5. A questo punto le occorrenze di* pippo *nel documento risultano valide per la nuova dichiarazione dell'elemento:*
`<xs:element name="pippo" type="int" minOccurs="2" maxOccurs="3" />`
nello schema. Ricordiamo che in base alla Tabella 4.1 ai nuovi elementi di tipo int *viene assegnato di default il valore* 0.

4.2.4 Aggiornamento del nodo

La funzione *updNode* prende in input una lista di alberi L_T che sono fratelli nel documento (avendo lo stesso genitore), un insieme di etichette S che ci si aspetta occorrano come prossimo elemento in base alla struttura di tipo t_s, e un'opzione che stabilisce se si vogliono introdurre nuovi elementi (qualora la struttura del tipo li richieda e non siano presenti nella lista L_T) oppure bisogna eliminarli. Quando si ha la necessità di estrarre l'etichetta di un elemento contenuta nell'insieme S, c'è prima una selezione tra tutti quelli con il $min > 0$ minore poi tra essi quello con il max minore (nel caso ce ne fossero più di uno, va bene, indifferentemente, qualsiasi di essi). In questo modo si evitano di inserire elementi con occorenza minima uguale a 0 la cui presenza è superflua, dopodiché scegliendo quelli con il min minore permette di inserire un elemento che necessita di essere presente un numero di volte minore rispetto a quello di tutti gli altri. Il primo criterio di scelta risulta particolarmente utile per elementi figli di operatori `all` e `sequence`, mentre il secondo risulta importante nel caso di operatori `choice`.

Figura 4.3: Esempio di struttura per la scelta di un'etichetta in S



Esempio 16 *Dato un elemento che ha come struttura quella la Figura 4.3, nel caso in cui questo elemento fosse vuoto, per riempirlo si avrebbe a disposizione l'insieme* $S = \{alpha, beta, gamma, delta\}$*. La funzione di scelta selezionerebbe l'elemento gamma (da inserire all'interno dell'elemento genitore) perché esso è quello con il* $min > 0$ *minore di tutti gli altri. A questo punto l'elemento risulta valido per la struttura ad esso assegnata.*

Per il resto la funzione *updNode* opera in modo ricorsivo come segue. Come prima cosa verifica se l'etichetta l_1 della radice del primo albero $T_1 \in L_T$ è contenuta in S. In caso affermativo questa è l'etichetta di un elemento richiesto da t_s e presente nella lista. Se T_1 non è valido per il corrispondente tipo nello schema, viene sostituito T_1 con un albero generato sulla base del tipo di l_1. Dopodiché si restituisce T_1 concatenato al risultato della valutazione ricorsiva di *updNode* sul resto della lista di alberi e sull'insieme dei prossimi elementi attesi da t_s.

Se invece l'etichetta l_1 della radice del primo albero $T_1 \in L_T$ non è contenuta in S si effettuano due comportamenti diversi a seconda dell'opzione con cui è stata invocata la primitiva.

Se è stata invocata con $action =$ `ins` occorre generare un nuovo elemento con etichetta scelta tra quelle presenti in S in base alle politiche precedentemente discusse.

Viene restituito tale elemento concatenato al risultato della valutazione ricorsiva di *updNode* sul resto della lista di alberi e sull'insieme dei prossimi elementi attesi da t_s. Se invece è stata invocata con $action =$ `del` si restituisce il risultato della valutazione ricorsiva di *updNode* sul resto della lista di alberi e sull'insieme S stesso perché non è stata trovata l'etichetta di un elemento atteso.

L'ultimo caso che si può verificare è che la lista L_T sia vuota ma $\lambda \notin S$. In questo caso occorre generare elementi da accodare nella lista fintanto che t_s non accetti λ come prossimo elemento valido.

CAPITOLO 5

Presentazione dell'interfaccia grafica

Il presente capitolo si pone l'obiettivo di descrivere in modo generico gli elementi che compongono l'interfaccia e quali sono le loro funzionalità.

Nella Sezione 5.1 viene fatta una panoramica sul linguaggio di programmazione utilizzato per implementare la grafica e gli algoritmi dell'applicazione. Nella Sezione 5.2 vengono presentati i Window Form ed i relativi controlli, mentre nella Sezione 5.3 sono presentate le librerie GDI+. Nella Sezione 5.4 viene descritta la struttura dell'interfaccia, infine nella Sezione 5.5 è presentato il motore grafico con i relativi componenti.

5.1 Il linguaggio C#

C# è il linguaggio di riferimento della Microsoft®, sia per lo sviluppo di applicazioni Windows® sia per applicazioni orientate al web, che mette insieme potenza e facilità d'utilizzo. È un linguaggio orientato agli oggetti, di sintassi molto simile a Java™, che offre alte prestazioni per operazioni di I/O, per l'interazione con il web e le basi di dati, oltre che per quanto riguarda l'utilizzo e la gestione della memoria.

Le principali caratteristiche semantiche e sintattiche di questo linguaggio orientato agli oggetti sono le seguenti:

- supporta tecniche di ereditarietà e polimorfismo;
- consente un'elevata tipizzazione dei dati;
- può girare su tutte le macchine che hanno installato il .NET Framework (un po' come la Java Virtual Machine del linguaggio SUN™);
- ha implementate tecniche di gestione ottimale della memoria;
- è case-sensitive;

- i blocchi di codice sono delimitati da parentesi graffe;
- ogni istruzione, ad eccezione di cicli o blocchi di codice, viene "chiusa" con un punto e virgola.

Una caratteristica fondamentale del linguaggio, come già detto, è la gestione della memoria che è eseguita in modo automatico. L'allocazione ed il rilascio degli oggetti è mediato dal .NET Framework tramite il garbage collector. Quando un oggetto viene allocato viene riservata la memoria necessaria nella memoria dinamica. Si può pensare a questa operazione come a qualcosa di simile alla chiamata *malloc*() del linguaggio C. Il rilascio della memoria non avviene però esplicitamente, ma viene eseguito automaticamente dal garbage collector quando non esistono più riferimenti a quell'oggetto. Questo vuol dire che il programma non ha più modo di accedere all'oggetto e per questo viene rimosso dalla memoria.

C# è un linguaggio ad alto livello molto simile a Java™ dal punto di vista sintattico e semantico. Entrambi hanno infatti bisogno del supporto di un framework ed una volta compilati, prima di passare a linguaggio macchina, attraversano un livello intermedio che, nel caso di C#, si chiama Intermediate Language (IL). L'utilizzo del linguaggio C# e dell'ambiente di sviluppo Visual Studio® ha sicuramente semplificato la progettazione e l'implementazione dell'interfaccia grafica qui in analisi.

5.2 Windows Form e controlli

I Windows Form rappresentano la nuova piattaforma, basata sul .NET Framework, per lo sviluppo di applicazioni Microsoft® Windows®, applicazioni che definiscono esattamente quelle che sono le interfacce utente utilizzate al giorno d'oggi nei sistemi operativi targati Microsoft®. I Windows Form possono inoltre fungere da interfaccia utente locale all'interno di una soluzione distribuita su più livelli.

Volendo progettare e successivamente modificare l'interfaccia utente delle soluzioni create, si rende necessario aggiungere, allineare e posizionare i controlli. I controlli sono oggetti contenuti all'interno di oggetti form. Ogni tipo di controllo presenta un insieme di proprietà, metodi ed eventi che lo rendono adatto a un particolare scopo. Per fare un esempio banale, un controllo tipico di un Windows Form è la casella di testo, definiti dalla classe TextBox, oppure il generico pulsante, definito dalla classe Button.

Scendendo nel dettaglio si può assumere che un form non è altro che una porzione dell'area visualizzata sullo schermo, in genere di forma rettangolare, che si può utilizzare per presentare informazioni all'utente e per accettarne l'input. I form possono essere finestre standard, finestre di interfaccia a documenti multipli MDI (Multiple Document Interface), finestre di dialogo o aree di visualizzazione di routine grafiche. Il modo più semplice per definire l'interfaccia utente per un form è posizionare controlli sulla superficie.

I form sono oggetti che espongono proprietà che ne definiscono l'aspetto, metodi che ne definiscono il comportamento ed eventi che definiscono l'interazione con l'utente. Impostando le proprietà del form e scrivendo il codice per rispondere agli eventi che vengono generati, è possibile personalizzare l'oggetto

in modo da adeguarlo ai requisiti dell'applicazione. I form, come ogni altro oggetto in .NET Framework, rappresentano istanze di classi; quando si visualizza un'istanza del form durante l'esecuzione del programma, la classe è utilizzata come modello per la creazione del form. Grazie al framework, inoltre, è possibile ereditare da form esistenti per aggiungere funzionalità o modificare un comportamento.

Quando si aggiunge un form al progetto, è possibile stabilire che il form erediti dalla classe Form messa a disposizione dal framework o da un form precedentemente creato. I form sono inoltre controlli, in quanto ereditano dalla classe Control. All'interno di un progetto di Windows Form, il form rappresenta il tramite primario per l'interazione con l'utente. Combinando opportunamente codice e diversi gruppi di controlli è possibile ottenere informazioni dall'utente e rispondere, lavorare con differenti archivi di dati, eseguire query, scrivere nel file system e nel registro di sistema.

5.3 I componenti GDI+

GDI+ (Graphic Design Interface) è un'insieme di interfacce grafiche che permettono ai programmatori di scrivere applicazioni indipendenti dalla piattaforma su cui vengono eseguite. Queste interfacce sono presenti come sottosistema di Microsoft® Windows® (dalla versione XP in poi), dove sono responsabili della visualizzazione di informazioni su schermi e stampanti.

Progettato in modo da garantire prestazioni elevate e caratterizzato da un'estrema facilità d'uso, GDI+ può essere utilizzato per eseguire il rendering di immagini grafiche in Windows Form e controlli. GDI+ ha sostituito l'ormai datato GDI e rappresenta l'unico metodo per eseguire il rendering di grafica a livello di codice in applicazioni Windows®. L'utilità di queste API (Application Programming Interface) sta nel fatto che ogni programmatore può scrivere applicazioni che stampano a video delle informazioni senza conoscere i dettagli tecnici di alcun strumento di visualizzazione (come schermi, touch-screen o stampanti). I servizi offerti da GDI+ coprono questi tre campi d'utilizzo.

- Vettori grafici a 2 dimensioni: sono presenti delle classi che implementano le varie primitive conosciute (come linee, curve e figure) specificate semplicemente come insiemi di punti segnati su di un sistema di coordinate ben definito; quindi una linea retta può essere rappresentata dal suo punto di partenza e da quello di fine, mentre per un rettangolo è necessario semplicemente definire il punto del suo angolo sinistro e i suoi valori di altezza e larghezza. GDI+ offre sia classi che contengono le informazioni sulle primitive stesse, sia classi che contengono informazioni su come queste primitive devono essere disegnate, sia classi che effettivamente effettuano il disegno. Per esempio, la struttura Rectangle dà la possibilità di leggere e impostare proprietà quali il punto dove il rettangolo è disegnato, con Location, oppure la sua grandezza, grazie a Size, definita come coppia di valori larghezza/altezza; poi esiste la classe Pen che contiene informazioni sul colore, la grandezza e lo stile della linea; invece la classe Graphics espone i metodi pubblici per disegnare linee, rettangoli, poligoni od altre figure.

- Manipolazioni di immagini: ortogonale alla rappresentazione vettoriale delle immagini è possibile (e a volte necessario) rappresentare attraverso bitmap (rappresentazione raster delle immagini). Ciò è possibile attraverso la classe Bitmap (e nelle classi che ereditano da questa), che espone metodi e proprietà per visualizzare, manipolare e salvare immagini.
- Manipolazioni di testi: consente di visualizzare e manipolare testi secondo una varietà di colori, font e stili differenti.

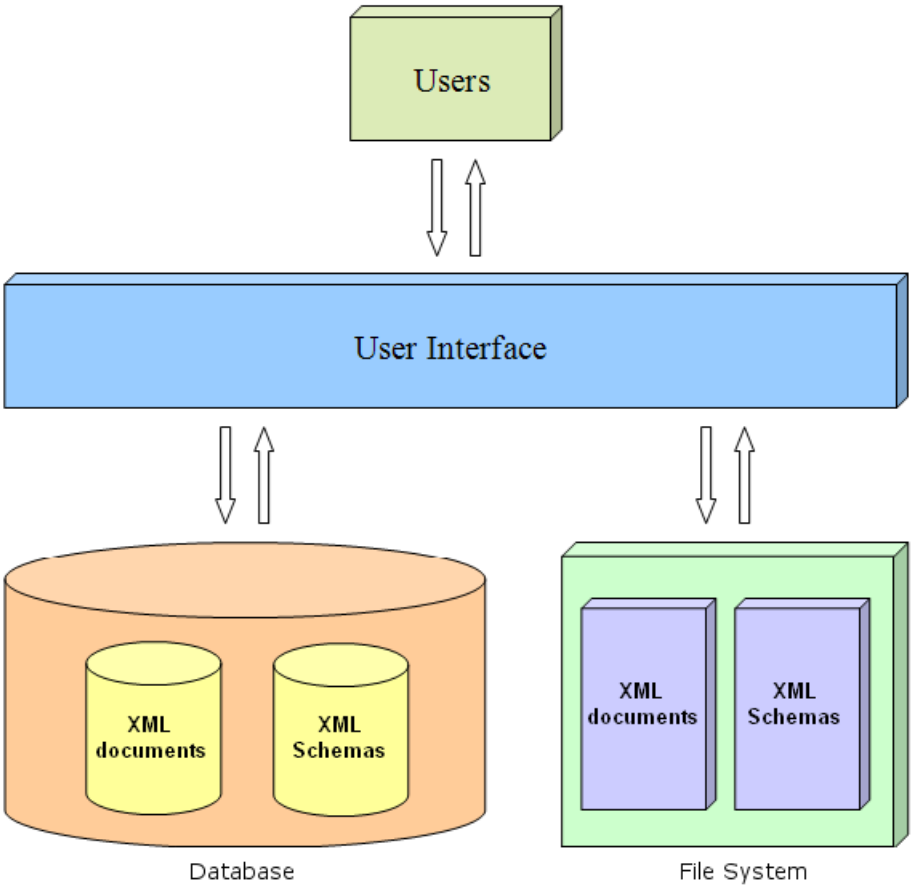
GDI+ conta 60 classi, 50 enumerazioni e 8 strutture. La classe Graphics è quella più rappresentativa per le funzionalità di GDI+ e la maggiorparte delle altre classi che interagiscono con essa. La classe Graphics consente di incapsulare una superficie di disegno GDI+, questa superficie di disegno è un oggetto ed i metodi e/o proprietà utilizzati per realizzare la grafica operano utilizzando le coordinate dell'area client del modulo. Vediamo quindi i principali metodi offerti da questa classe, proprio per verificarne l'interazione con il resto delle classi.

- Clear: ripulisce completamente l'intera superficie di disegno e la riempie con il colore di sfondo passato come parametro.
- DrawArc: disegna un arco rappresentante una porzione di un'ellisse specificata da due coordinate spaziali, un'altezza e una larghezza.
- DrawBezier: disegna una curva di Bézier definita da quattro strutture di punti.
- DrawCurve: disegna una curva da un array di punti.
- DrawImage: disegna un oggetto Image, passato come parametro, in una posizione specifica e mantenendo le dimensioni originali.
- DrawLine: disegna una linea specificandone il punto di inizio e il punto di fine.
- DrawPolygon: disegna un poligono da un array di punti.
- DrawRectangle: disegna un rettangolo dato il punto del suo angolo sinistro, ricavato da Location, la sua larghezza e altezza.
- DrawString: disegna la strina passata come parametro, in una posizione specifica, utilizzando gli specifici oggetti di tipo Brush e Font, utili a settare lo stile del testo.

Molti dei metodi su esposti accettano come parametri oltre che oggetti di tipo Paint o Brush anche oggetti di tipo Point o Rectangle.

Un oggetto Point è una coordinata nello spazio a due dimensioni; può essere creato, ad esempio, invocando il costruttore a due parametri, il primo parametro rappresenta l'ascissa del punto, il secondo l'ordinata.

Figura 5.1: L'architettura dell'applicazione



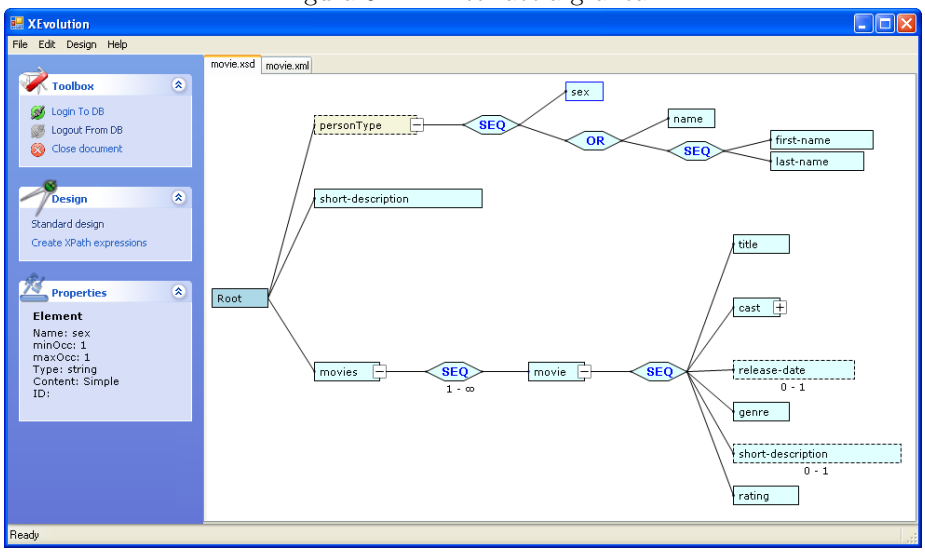
5.4 La struttura di "XEvolution"

L'applicazione permette di aprire sia documenti che schemi XML per eseguire le operazioni che si ritengono necessarie. I file che verranno aperti possono trovarsi sia sul file system che all'interno di una base di dati, dando all'utente la libertà di effettuare le modifiche dove meglio crede.

La Figura 5.1 mostra l'architettura dell'applicazione con l'interfaccia posta tra l'utente ed i file XML, siano essi semplici documenti o schemi, operando indistintamente tra quelli presenti nel database e quelli che si trovano nel file system.

La struttura dell'applicazione è stata progettata pensando di dare il maggior spazio possibile all'area di visualizzazione grafica dei file, scegliendo di permettere l'apertura di più documenti e schemi tramite un sistema di tabulazione.

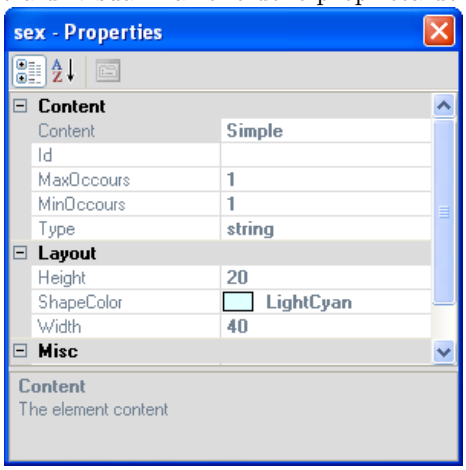
Figura 5.2: L'interfaccia grafica



Come si può notare dalla Figura 5.2 lo stile è quello classico adottato dalla maggior parte delle applicazioni per Windows® XP; stile che comprende un menu principale, dei box espandibili con contenuto dinamico, una barra di stato, menu contestuali e finestre aggiuntive a supporto della principale.

All'area centrale è affiancata una fascia laterale contenente vari box espandibili, ognuno dedicato ad una serie di operazioni prestabilite. Il primo box, intitolato Toolbox, contiene i collegamenti per effettuare il login e il logout dal database e per chiudere il documento corrente. Il box Design è quello che si occupa delle politiche di controllo dell'accesso sui documenti XML, funzionalità presente nel programma ma non verrà trattata in questa tesi. Il box Properties è il box contestuale per le proprietà di ogni elemento selezionato. Questo infatti espone le principali proprietà dell'elemento corrente, espandendosi in automatico, una volta fatto il click su un qualsiasi elemento del grafico. Nel nostro esempio si è selezionato l'elemento sex (come si può notare dalla diversa colorazione del contorno) ed il box ne contiene le principali caratteristiche.

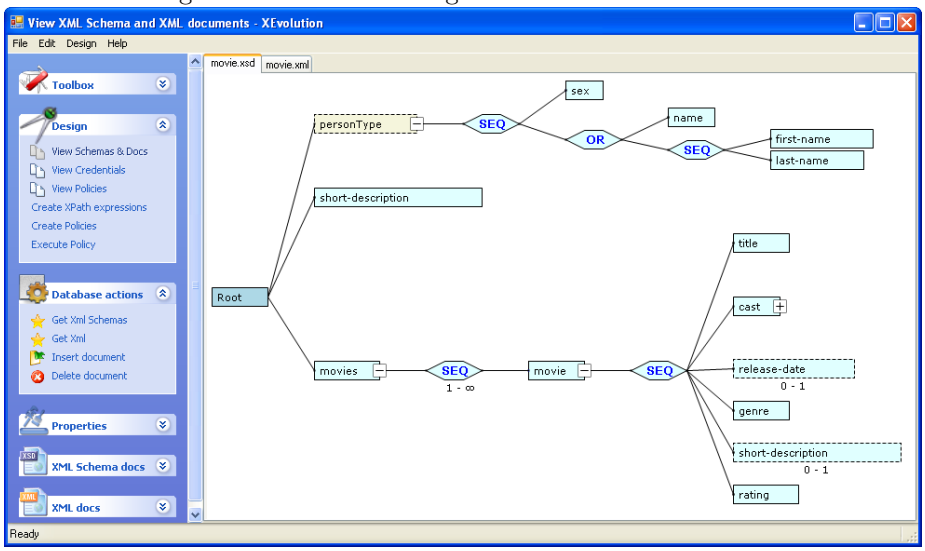
Figura 5.3: Finestra di visualizzazione delle proprietà del nodo selezionato



La visualizzazione delle proprietà, oltre ad essere presente in uno dei box laterali, ha anche un'altra locazione; dal menu contestuale infatti viene aperta una finestra a supporto della principale, contenente le informazioni richieste. Tramite questa finestra, visibile in Figura 5.3, è possibile anche modificare alcune delle proprietà visive dell'elemento, come le sue dimensioni o il suo colore di sfondo.

Tornando ai box laterali essi variano di contenuto e quantità in base al fatto che l'utente abbia avuto accesso o meno al database.

Figura 5.4: L'interfaccia grafica connessa al database



Osservando la Figura 5.4 si salta subito all'occhio la presenza di nuovi box necessari alla gestione dei file presenti nel database oltre all'aumento delle opzioni presenti in Design. Grazie al box Database actions è possibile prelevare documenti e schemi XML, oltre a cancellarne o inserirne di nuovi. Se l'utente

decidesse di estrarre gli schemi (tramite l'opzione Get Xml Schemas) oppure i documenti (utilizzando Get Xml) dalla base di dati, questi verranno inseriti rispettivamente all'interno dei box XML Schema docs ed XML docs, popolando delle liste di nomi, dalle quali sarà poi possibile selezionarli per la visualizzazione grafica.

5.5 Il motore grafico

Il motore grafico rappresenta il vero cuore dell'interfaccia. Esso infatti descrive tutti i metodi di visualizzazione e di interazione dei grafici, tutto ciò è reso possibile grazie alla creazione di un insieme di classi, collezioni di classi e controlli legati fra loro da vincoli d'ereditarietà o, più direttamente, attraverso l'uso di specifiche proprietà. Per applicare modifiche a tutti gli elementi dell'albero, basta solamente modificare le classi da cui questi ereditano.

L'albero XML viene stampato all'interno di un controllo grafico che dispone di metodi e proprietà per gestire le collezioni di elementi visualizzati al suo interno, per le operazioni legate ai click e ai movimenti del mouse e per le operazioni di disegno vere e proprie. L'elemento grafico generico è un oggetto chiamato Entity, in quanto rappresenta una generica entità dell'albero, da esso andranno poi ad ereditare tutti gli altri elementi che comporranno vari punti dell'albero. Da questa classe padre, sono state implementate tre diverse tipologie di oggetti, utili per rappresentare graficamente l'albero: le shape, i connettori e le connessioni.

5.5.1 Le shape

Le shape sono quegli oggetti che rappresentano tutti i tipi di elementi di documenti e schemi XML, perciò si è creata una classe Shape che definisce metodi e proprietà, più o meno utili, a tutti i successivi tipi di shape. I figli, a loro volta, hanno proprietà aggiunte e metodi ridefiniti, in modo tale da poter effettuare una diversificazione vera e propria tra tutti gli oggetti possibili, che andranno a formare i vari alberi.

La classe Shape implementa quasi tutti i metodi definiti dalla classe Entity ed aggiunge altre caratteristiche utili a diversificare un oggetto generico dell'albero da un elemento vero e proprio. Una volta definita questa classe, che definisce una generica shape dell'albero, si è cercato di definire tutte le classi per rappresentare tutti i tipi di elementi contenuti sia in documenti che in schemi XML, come ad esempio gli XmlSchemaComplexType, gli XmlElement, le XmlSchemaSequence. Le classi che rappresentano questi elementi implementano due connettori, uno destro e uno sinistro, utili per la creazione di connessioni con altre shape.

5.5.2 I connettori

I connettori sono quegli oggetti del grafico, la cui visualizzazione è legata a quella della shape a cui appartengono, e che permettono la creazione delle connessioni tra una shape e l'altra. I connettori sono rappresentati dalla classe Connector, anch'essa sottoclasse di Entity.

5.5.3 Le connessioni

Le connessioni sono quegli oggetti del grafico che collegano le shape tra loro, tramite i loro connettori. Graficamente sono rappresentati da una linea continua che va da una shape all'altra e che ne segue i movimenti, se queste vengono spostate col mouse. Sono rappresentate dallla classe Connection sottoclasse anch'essa di Entity.

CAPITOLO 6

Validazione sperimentale

In questo capitolo tratteremo gli esperimenti effettuati per confrontare i tempi di esecuzione tra gli algoritmi da noi ideati e la classica validazione totale di un documento rispetto allo schema.

Nella sezione 6.1 indicheremo dove sono stati reperiti i documenti e gli schemi per gli esperimenti, nella sezione 6.2 parleremo dei tipi di esperimenti effettuati, nella 6.3 il confronto tra i vari algoritmi. Infine, nella sezione 6.4 saranno presentati i risultati ed i grafici ricavati.

6.1 Raccolta di documenti XML e schemi dal Web

Per effettuare i nostri esperimenti abbiamo raccolto diversi schemi e documenti dal Web ed altri (soprattutto quelli di piccole dimensioni) li abbiamo creati noi manualmente. Ad uno schema potevano far riferimento uno o più documenti e per i documenti a cui non era associato nessuno schema abbiamo provveduto a crearlo manualmente con un editor di testo oppure utilizzando vari tool di generazione automatica della schema.

Un sito molto interessante dove reperire documenti è l'ibiblio dove all'indirizzo `http://www.ibiblio.org/xml/examples/` sono presenti vari documenti XML di dimensione poco inferiore al Mega tra cui varie opere di Shakespeare oppure un insieme di statistiche del campionato americano di baseball. Un'altro sito interessante è la famosa raccolta di bibliografie della Digital Bibliography & Library Project (DBLP), tra i vari formati disponibili all'indirizzo `http://dblp.uni-trier.de/xml/` è presente anche una versione XML di tale raccolta.

6.2 Tipologia di esperimenti condotti

Una volta avuti a disposizione un buon numero di documenti sono stati divisi in 3 catogorie in base alla dimensione ed in altre 3 a seconda della profondità. La

caratteristiche degli schemi non sono state prese in considerazione perché sono tutti molto simili e comunque la parte "pesante" della computazione rigurda l'accesso e la validazione/adattamento dei documenti, mentre le modifiche allo schema avvengono in tempo costante.

Per i nostri esperimenti abbiamo deciso di racchiudere nella categoria dei documenti di piccole dimensioni tutti quelli minori di 1 KB, i documenti la cui dimensione è tra 1 KB ed 1 MB sono stati considerati di medie dimensioni, invece sono stati considerati come documenti di grandi dimensioni tutti quelli maggiori di 1 MB.

Per quel che riguarda la profondità i documenti con un massimo di 5 livelli sono stati dichiarati di piccola profondità, quelli oltre i 5 e fino ai 10 livelli vengono considerati di media profondità, invece quelli oltre i 10 livelli fanno parte dei documenti di grande profondità.

6.3 Caratteristiche confrontate

Per l'esecuzione di questi eseperimenti sono state sviluppate delle piccole versioni a riga di comando nelle quali sono state copiate le implemetazioni degli algoritmi interessati, presentati in precedenza.

La macchina sulla quale sono stati svolti gli esperimenti è un semplice computer "Desktop" con processore AMD Athlon™ XP 2000+ da 1.67 GHz e 512 MB di RAM, il tutto su sitema operativo Microsoft® Windows®XP Service Pack 2.

Il confronto è stato fatto tra il nostro algoritmo di rivalidazione, quello di validazione totale utilizzato dal .NET Framework ed il nostro algoritmo di ristrutturazione. Le rilevazioni sono state fatte su schemi e documenti caricati in memoria quindi nel calcolo dei tempi non viene considerata la scrittura su disco, ma sono tutte compresive del tempo utilizzato per le modifiche allo schema.

In un primo momento abbiamo separato i documenti in base alla profondità dopodiché abbiamo fatto una media della dimensione in modo da avere un'indicazione sulla grandezza dei documenti che andavamo ad analizzare. Alla fine della separazione le dimensioni ottenute sono state le seguenti:

- documenti poco profondi:
  - piccoli ≈ 256 byte
  - medi ≈ 1.3 KB
  - grandi ≈ 5 MB

  documenti di media profondità:
  - piccoli ≈ 736 byte
  - medi ≈ 232 KB
  - grandi ≈ 137 MB

  documenti molto profondi:
  - piccoli ≈ 640 byte
  - medi ≈ 924 KB

– grandi ≈ 30 MB

Giunti a questo punto abbiamo effettuato una lunga serie di primitive differenziando quelle che operavano sulle radici dei documenti, quelle che interessavano i nodi intermedi, quelle riferite alle foglie ed infine le primitive* che non intaccano la validità dei documenti (per modifiche ai nodi foglia intendiamo anche strutture che al loro interno non hanno elementi di tipo complesso). Alla fine abbiamo fatto anche una rilevazione sulla veolocità delle varie primitive nell'esecuzione della rivalidazione/ristrutturazione dei documenti, in questo caso le primitive sono state divise in primitive di inserimento, di modifica, di cancellazione e primitive*. Invece i documenti non sono stati separati per profondità (perché si è visto che il risultato non ne è particolarmente influenzato) ma solo in base alla loro dimensione, escludendo però quelli molto grandi (superiori ai 10 MB).

6.4 Risultati ottenuti

Analizzando i risultati ottenuti, riportati nelle tabelle e nei grafici in fondo al capitolo, balza subito all'occhio che le primitive* vengono eseguite sempre in tempo costante, infatti non necessitano di accedere ai documenti, durante la loro esecuzione viene modificato solo lo schema, ma essendo gli schemi tutti grossomodo simili la differenza tra una modifica od un'altra è risultata pressoché nulla.

La validazione dei documenti attraverso il .NET Framework è costante per quelli delle stesse dimensioni (la profondità risulta ininfluente) e cresce all'aumentare delle dimensioni del file. La rivalidazione attraverso XEvolution risulta sempre la più veloce con un miglioramento intorno al 20%, i casi particolarmente favorevoli (oltre alle già citate primitive*) si hanno con documenti molto grandi, in questo caso il fatto di lavorare su un piccola parte di documento aiuta ad alleggerire molto il lavoro. La profondità dei documenti non pare cambiare di molto la resa dell'algoritmo, le differenze sono dovute dal tempo impiegato per ricercare i cammini interessati alla modifica all'interno del documento e dalla composizione dei sottoalberi da analizzare rispetto alla primitiva richiamata.

La ristrutturazione ha un comportamento analogo a quello della rivalidazione , solo leggermente più lento, si vede che l'inserimento e la sottrazione di nodi all'interno di un documento è un'operazione non molto gravosa rispetto al controllo dei nodi già presenti, ed il fatto che il tempo impiegato per scrivere su disco non è stato oggetto di valutazione ha aiutato ad avvicinare i valori dei nostri due algoritmi. Si nota un peggioramento delle prestazioni della ristrutturazione per documenti molto grandi e soprattutto nel caso si debbano modificare le foglie di tali documenti. Questo comportamento può essere dovuto al fatto che per modificare i documenti bisogna comunque caricare in memoria oltre al documento anche una parte di esso per sottoporla a modifica, quindi con documenti molto grandi anche le sottoparti da modificare risultano grandi rendendo alta le probabilità di "swapping" che fanno decadere le prestazioni. Inoltre dai grafici si può notare che in generale l'aggiornamento di nodi profondi richiede più tempo rispetto a quelli a livelli più alti.

Dopo queste prime rilevazioni si è proceduto ad analizzare le varie primitive in base al tipo di modifica effettuata. Si è deciso di non separare più in

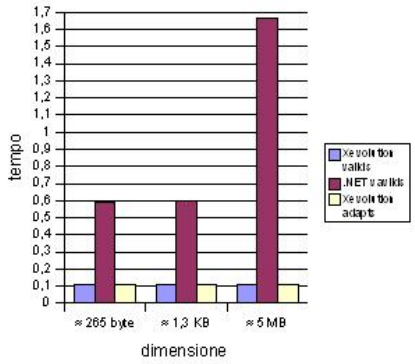
base alla profondità perché, come visto in precedenze, il risultato non ne viene molto influenzato, anche se sulle foglie il tempo impiegato risulta leggermente maggiore. Anche i documenti particolarmente grandi non sono stati inclusi, per evitare che casi di "swapping" possano falsarne l'andamento. Esattamente come prima, il caso migliore risultano le primitive* che lavorano in tempo lineare. L'andamento delle varie primitive aumentano piuttosto costantemente all'aumento delle dimensioni dei file. Le primitive che richiedono il maggior tempo sono quelle di modifica, mentre risultano molto veloci quelle di cancellazione, che per documenti medio/piccoli ha risultati addirittura vicini alle primitive*. Com in precedenza anche in questo caso l'algoritmo di ristrutturazione ha prestazioni simili a quello di rivalidazione, anche se rimane leggermente leggermente peggiore.

Tabella 6.1: Documenti con piccola profondità

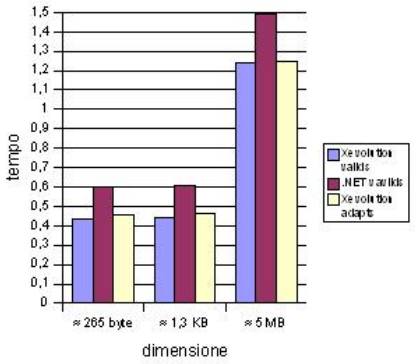
dimensione	rivalidazione	validazione .NET	ristrutturazione
Primitive*			
≈ 256 byte	0.11	0.594	0.11
≈ 1.3 KB	0.11	0.597	0.11
≈ 5 MB	0.11	1.664	0.11
Primitive sulla radice			
≈ 256 byte	0.432	0.599	0.475
≈ 1.3 KB	0.439	0.605	0.462
≈ 5 MB	1.244	1.494	1.251
Primitive sui nodi intermedi			
≈ 256 byte	0.443	0.603	0.469
≈ 1.3 KB	0.49	0.608	0.517
≈ 5 MB	1.345	1.624	1.836
Primitive sulle foglie			
≈ 256 byte	0.5	0.605	0.502
≈ 1.3 KB	0.46	0.608	0.488
≈ 5 MB	1.235	1.82	1.791

Figura 6.1: Documenti con media profondità

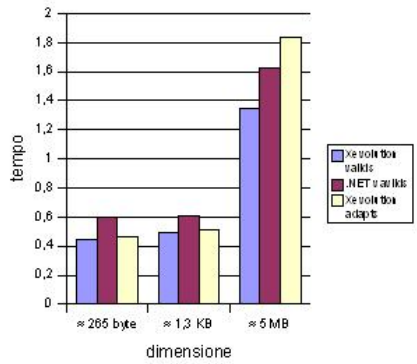
(a) Primitive*



(b) Primitive sulla radice



(c) Primitive sui nodi intermedi



(d) Primitive sui nodi foglia

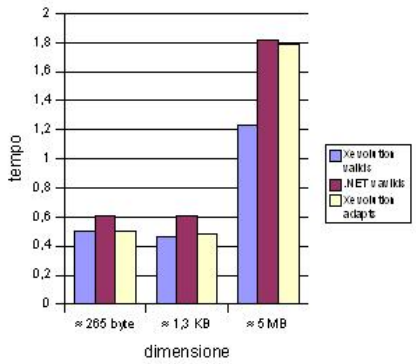
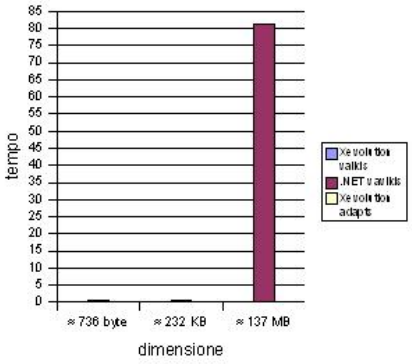


Tabella 6.2: Documenti con media profondità

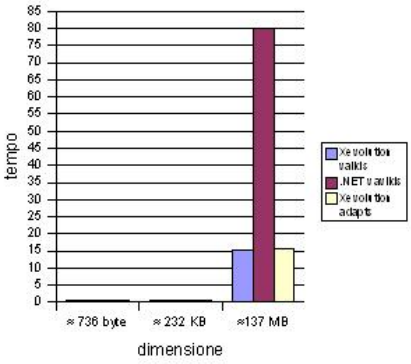
dimensione	rivalidazione	validazione .NET	ristrutturazione
Primitive*			
≈ 736 byte	0.11	0.58	0.11
≈ 232 KB	0.11	0.654	0.11
≈ 137 MB	0.11	81.3	0.11
Primitive sulla radice			
≈ 736 byte	0.379	0.587	0.402
≈ 232 KB	0.533	0.659	0.563
≈ 137 MB	15.42	80.275	15.64
Primitive sui nodi intermedi			
≈ 736 byte	0.551	0.597	0.579
≈ 232 KB	0.516	0.67	0.599
≈ 137 MB	30.821	81.9	70.5
Primitive sulle foglie			
≈ 736 byte	0.415	0.598	0.44
≈ 232 KB	0.44	0.684	0.581
≈ 137 MB	30.5	81.998	80.5

Figura 6.2: Documenti con media profondità

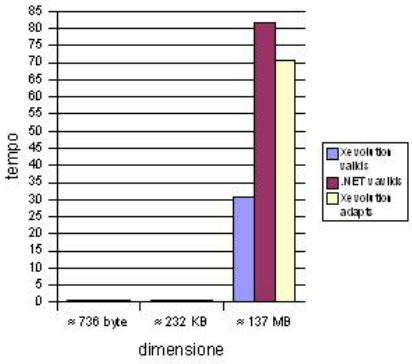
(a) Primitive*



(b) Primitive sulla radice



(c) Primitive sui nodi intermedi



(d) Primitive sui nodi foglia

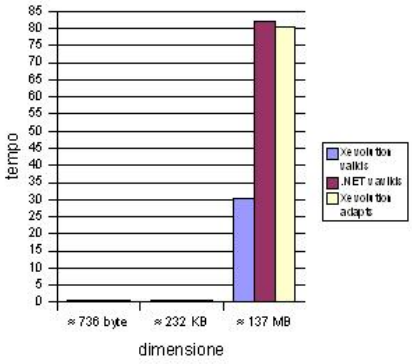
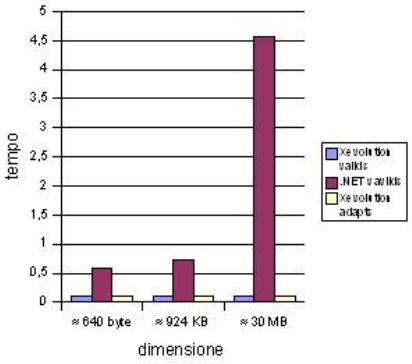


Tabella 6.3: Documenti con grande profondità

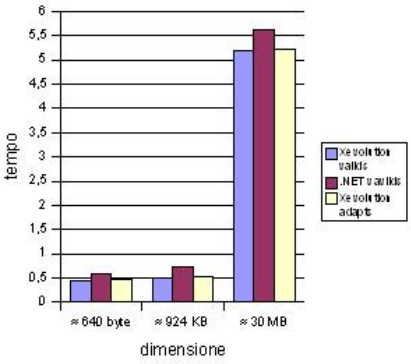
dimensione	rivalidazione	validazione .NET	ristrutturazione
Primitive*			
≈ 640 byte	0.11	0.593	0.11
≈ 232 KB	0.11	0.736	0.11
≈ 137 MB	0.11	4.568	0.11
Primitive sulla radice			
≈ 640 byte	0.457	0.597	0.481
≈ 924 KB	0.492	0.722	0.536
≈ 30 MB	5.2	5.63	5.21
Primitive sui nodi intermedi			
≈ 640 byte	0.389	0.604	0.39
≈ 924 KB	0.512	0.736	0.746
≈ 30 MB	3.1	4.49	3.101
Primitive sulle foglie			
≈ 640 byte	0.546	0.606	0.573
≈ 924 KB	0.549	0.74	0.787
≈ 30 MB	4.005	4.704	7.19

Figura 6.3: Documenti con grande profondità

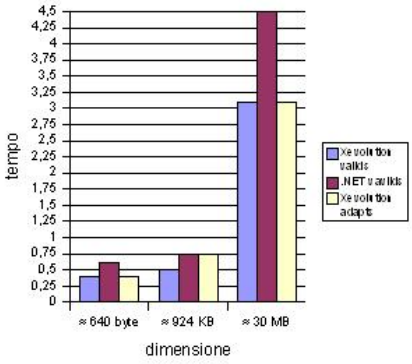
(a) Primitive*



(b) Primitive sulla radice



(c) Primitive sui nodi intermedi



(d) Primitive sui nodi foglia

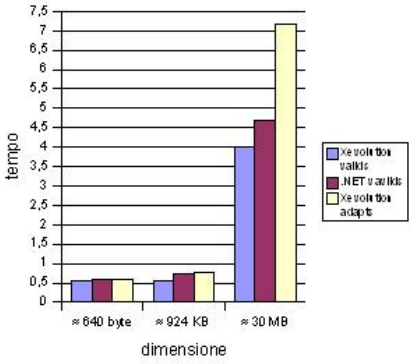


Tabella 6.4: Velocità di rivalidazione per primitive

dimensione	Primitive*	Inserisci	Modifica	Cancella
0 < files ≤ 1 KB	0.11	0.42	0.601	0.121
1 KB < files ≤ 1 MB	0.11	0.735	0.885	0.132
1 MB < files ≤ 10 MB	0.11	1.097	1.987	0.5

Figura 6.4: Velocità di validazione per primitive

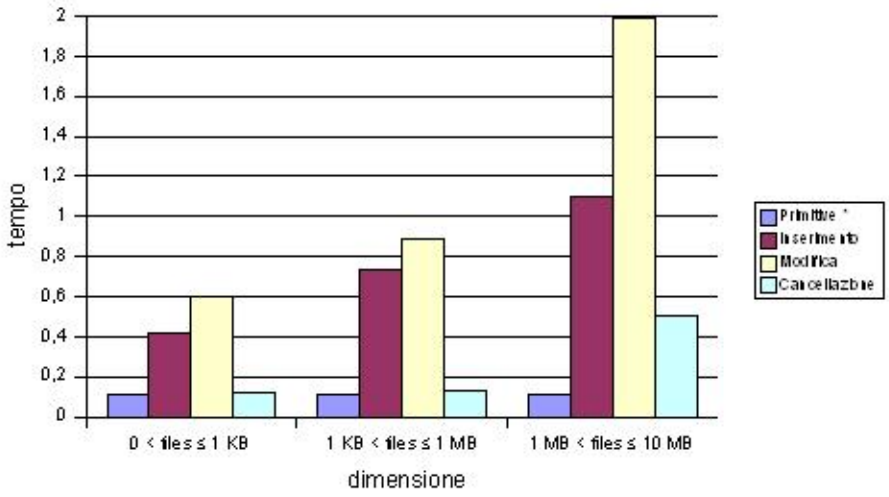
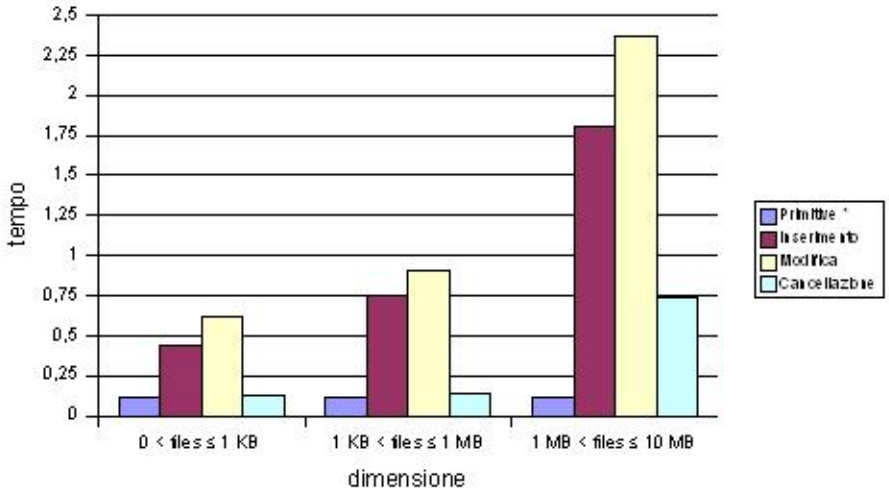


Tabella 6.5: Velocità di ristrutturazione per primitive

dimensione	Primitive*	Inserisci	Modifica	Cancella
0 < files ≤ 1 KB	0.11	0.44	0.618	0.124
1 KB < files ≤ 1 MB	0.11	0.745	0.91	0.135
1 MB < files ≤ 10 MB	0.11	1.8	2.363	0.743

Figura 6.5: Velocità di adattamento per primitive



CAPITOLO 7

Conclusioni e sviluppi futuri

L'applicazione XEvolution è stata studiata per poter adattare e validare in modo facile e veloce documenti che devono essere modificati in base alle nuove e diverse esigenze del constesto nel quale essi si trovano ad operare. Tanto è vero che con una singola modifica ad uno schema si possono modificare in un colpo solo innumerevoli documenti.

Questa tesi può anche contribuire per la soluzione delle notevoli problematiche legate alla recente tendenza delle comunità scientifiche e industriali a sviluppare sistemi di basi di dati che supportano XML come tipo di dato nativo. La natura semistrutturata dei documenti XML e la grande quantità di documenti, provenienti da un ambiente flessibile ed eterogeneo come il Web, fanno nascere l'esigenza di poter evolvere frequentemente la struttura degli schemi, al fine di rappresentare fedelmente i documenti che vengono memorizzati nella base di dati.

In particolare, nella tesi ci siamo occupati di identificare le primitive che permettono di specificare l'evoluzione di uno schema, formalizzando la struttura di un XML Schema in modo da specificare chiaramente e dettagliatamente gli operatori di evoluzione, in modo da indicare le condizioni di applicabilità al fine di garantire la correttezza dello schema evoluto.

È stato introdotti ed analizzato il problema della rivalidazione dei documenti XML che erano conformi allo schema prima dell'evoluzione, proponendo una soluzione che consenta di evitare una rivalidazione *brute-force*. Infine, abbiamo introdotto e analizzato il delicato problema della ristrutturazione dei documenti la cui validità è stata compromessa nel corso dell'evoluzione dello schema, cercando di identificare una soluzione che ha fornito dei buoni risultati ma non vuole essere la soluzione perfetta, ma un punto dal quale partire per continuare a migliorarsi.

Per esempio, invece di eseguire ogni singola primitiva separatemante, si può sviluppare in futuro la possibilità di eseguire un numero arbitrario di primitive e solo alla fine di esse accedere ai documenti per la rivalidazione/ristrutturazione. Un'altro possibile sviluppo può essere quello di cercare di evitare ulteriormente la perdita di informazioni all'interno del documento nella fase di ristrutturazio-

ne, come avviene nella *change_operator* quando un'operatore `sequence` prende il posto di una `all`.

Un'altro sviluppo interessante può essere quello di assegnare ai nuovi elementi dei valori immessi in input dall'utente invece di utilizzare dei valori di default, questo fatto può causre una notevole perdita di tempo (oltre che di pazienza) nel caso la primitiva immetta un gran numero di elementi in una volta sola. Pensiamo al caso che si aumenti di molto la occorrenza minima di un elemento, oppure che da un operatore `choice` si passi ad una `sequence` e nella sottostruttura siano presenti molti elementi con un valore alto per l'occorrenza minima. In questi casi bisognerebbe anche decidere se assegnare un unico valore per ogni elemento dello stessi tipo oppure inserire valori diversi, aumentando di conseguenza la richiesta di input all'utente.

Come ulteriore sviluppo futuro si prevedere di arricchire la struttura dati in modo da accettare anche schemi che non siano *conflict-free*.

BIBLIOGRAFIA

[1] A. Bonifati and D. Lee, editors. *Seventh ACM International Workshop on Web Information and Data Management (WIDM 2005), Bremen, Germany, November 4, 2005*. ACM, 2005.

[2] H. Davis. *Programmazione in Visual C# .NET*. McGraw-Hill, first edition, Jan. 2003.

[3] D. Ghelli. Un'interfaccia grafica per la specifica di politiche di controllo dell'accesso su documenti xml. Laurea in comunicazione digitale, Università degli Studi di Milano, 2005. Supervisor: Dott. Marco Mesiti.

[4] G. Guerrini, M. Mesiti, and D. Rossi. Impact of xml schema evolution on valid documents. In Bonifati and Lee [1], pages 39–44.

[5] G. Guerrini, M. Mesiti, and M. A. Sorrenti. Xml schema evolution: Incremental validation and efficient document adaptation. Technical report, Dipartimento di Informatica e Scienze dell'Informazione - Università degli Studi di Genova.

[6] Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, editors. *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*. Springer, 2006.

[7] G. Marchi. Creazione di un'interfaccia per la visualizzazione di documenti e schemi xml. Laurea in informatica, Università degli Studi di Milano, 2005. Supervisor: Dott. Marco Mesiti.

[8] M. Mesiti, R. Celle, M. A. Sorrenti, and G. Guerrini. X-evolution: A system for xml schema evolution and document adaptation. In Ioannidis et al. [6], pages 1143–1146.

[9] D. Rossi. Evoluzione dello schema dei documenti xml. Laurea in informatica, Università degli Studi di Genova, 2004. Supervisors: Prof.ssa Giovanna Guerrini and Dott. Marco Mesiti.

[10] J. Sharp and J. Jagger. *Microsoft Visual C# .NET Passo per Passo.* Mondadori Informatica, first edition, Apr. 2002.

[11] M. Williams. *Programmare Microsoft Visual C# .NET.* Mondadori Informatica, first edition, Sept. 2002.