UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Facoltà di Ingegneria - Sede di Modena

Corso di Laurea in Ingegneria Informatica

Progettazione UML nell'ambito del Semantic Web

Relatore

Chiar.ma Prof.ssa Sonia Bergamaschi

Tesi di Laurea di

Patrizia Ponchiroli

Anno Accademico 2003-2004

Keywords:

Ontologia
ODLI3
Semantic Web
UML
Metamodello

INDICE

Indice delle Figure

Introduzione

La maggior parte dei contenuti del Web sono, al giorno d'oggi, espressamente rappresentati affinché possano essere letti da utenti umani e non sono progettati per essere elaborati da agenti software. Un browser può, secondo ben precise istruzioni, disporre e visualizzare le informazioni in una certa maniera, ma non ha modo di riconoscerle o attribuire loro un particolare significato per elaborarle. In questo contesto si sta sviluppando una nuova visione del Web, che conferisce ai contenuti presenti in rete una struttura semantica. Questo è ciò che Tim Berners-Lee chiama "Semantic Web", o "Web Semantico", che può essere considerato la terza generazione del Web. E' chiaro che l'HTML, il linguaggio standard per la creazione di pagine Web, non ha le caratteristiche per strutturare i dati in base alla loro semantica, e pertanto deve essere affiancato da altri strumenti che permettano la separazione e l'indipendenza tra il contenuto e il layout delle pagine. Aggiungere la semantica ai contenuti del Web richiede anche la creazione di linguaggi specifici per la rappresentazione della conoscenza (Linguaggi KR), che permettano di esprimere dati e regole per i ragionamenti. Il Semantic Web non risulterà separato dal Web tradizionale, ma, introducendo il significato dei dati, ne costituisce un'estensione, aggiungendo una nuova funzionalità agli elaboratori, che diventano in grado di capire ed elaborare automaticamente le informazioni (*machine-understandable*), che fino ad ora semplicemente visualizzavano (*machine-readable*). Le potenzialità del Semantic Web sarebbero inutili se non ci fossero degli agenti software in grado di raccogliere, elaborare e scambiare con altri programmi le informazioni provenienti da sorgenti di dati distinte ed eterogenee, come sono quelle disponibili sul Web. Questo comporta uno stretto legame tra le problematiche del Semantic Web e quelle dell'Intelligenza Artificiale. Naturalmente, per la natura stessa del Web, è possibile che dati concettualmente diversi siano rappresentati con lo stesso nome. Un programma dovrebbe essere in grado di distinguere la situazione e quindi elaborare le informazioni in maniera appropriata. Perchè questo possa avvenire, si devono creare delle ontologie. Il termine *ontologia* proviene dal linguaggio filosofico, in particolare da Aristotele, dove indica la "Rappresentazione Sistematica dell'Essenza", cioè la teoria che studia quali tipi di cose esistono. Nell'ambito dell'Intelligenza Artificiale esso indica "la descrizione particolareggiata di una concettualizzazione", cioè "l'insieme dei termini e delle relazioni, usati in un particolare dominio applicativo, che denotano in modo univoco una particolare conoscenza e fra i quali non esiste ambiguità poiché sono condivisi dall'intera comunità di utenti del dominio applicativo stesso". La struttura dei dati e la semantica introdotta dalle ontologie migliorano le potenzialità del Web. I programmi di ricerca, infatti, basandosi su un preciso concetto, cercano e trovano le pagine che effettivamente si riferiscono a quel concetto, anziché

quelle che contengono keyword ambigue o generiche, tra le quali l'utente deve ulteriormente cercare quelle d'interesse, come avviene nel Web tradizionale. Un aspetto molto importante per la creazione del Semantic Web consiste nel rendere possibili agli utenti, che non sono esperti di logica, la creazione di ontologie e la realizzazione di contenuti Web che siano leggibili dalle macchine. Per far fronte a queste esigenze il World Wide Web Consortium (W3C) ha progettato una serie di linguaggi che, nel corso del tempo, sono diventati degli standard, e che consentono proprio la strutturazione semantica delle informazioni e la condivisione dei significati dei concetti. Quello che si cerca di proporre in questa tesi è un'alternativa a questi popolari e conosciuti formalismi per la rappresentazione di ontologie, costituita dall'uso di formalismi UML (Unified Modeling Language), associati ad espressioni OCL (Object Constraint Language), un potente strumento per l'espressione di vincoli. La questione che viene posta è se l'UML possa essere visto, oltre che come ormai consolidato elemento di analisi preliminare del software, anche come un efficace linguaggio per rappresentare ontologie, e quindi possa essere inserito nel gruppo dei linguaggi di rappresentazione della conoscenza. Si sta così cercando di formalizzare ulteriormente l'UML per renderlo sempre più *machine-processable*, in modo che i suoi modelli possano essere usati anche in fase di compilazione e run-time, e non solo come notazione grafica nella comunicazione *human-to-human*, come è stato finora.

Si mostra ora l'organizzazione della tesi e il contenuto dei vari capitoli in essa presenti:

Il **Capitolo 1** introduce le premesse al Semantic Web e descrive la struttura a più livelli di questo nuovo modo di concepire il Web. Vengono brevemente illustrati i linguaggi che permettono l'implementazione del Web Semantico, soffermandosi in particolare sugli standard proposti dal W3C: XML (eXtensible Markup Language) e RDF (Resource Description Framework). Viene poi descritto il linguaggio ontologico DAML+OIL, nato dalla fusione di altri due linguaggi per la definizione di ontologie, DAML (DARPA Agent Markup Language) e OIL (Ontology Inference Layer).

Il **Capitolo 2** presenta l'approccio semantico per l'integrazione di sorgenti eterogenee, strutturate o semistrutturate, soffermandosi sul concetto di "ontologia", il cui ruolo è fondamentale per permettere la combinazione e la condivisione di informazioni che provengono da basi di dati distinte. In particolare vengono presentati gli obiettivi, le caratteristiche e l'architettura del sistema MOMIS (Mediator envirOnment for Multiple Information Sources), un progetto il cui sviluppo è iniziato dalla collaborazione tra i gruppi operativi dell'Università di Modena e Reggio Emilia e di Milano, e che attualmente continua nell'ambito di SEWASIE (SEmantic Webs and AgentS in Integrated Economies), un progetto che si propone di implementare un motore di ricerca semantico che permetta di accedere a sorgenti di dati

eterogenee nel Web. Viene inoltre descritto il linguaggio $ODLI_3$, utilizzato per la rappresentazione della GVV (Global Virtual View), la vista integrata degli schemi locali all'interno del sistema MOMIS. In particolare, si sottolineano le estensioni apportate da $ODLI_3$ rispetto al linguaggio standard ODL (Object Definition Language), al fine di rispondere alle problematiche di integrazione di informazioni da fonti eterogenee.

Nel **Capitolo 3** viene descritto l'UML, un linguaggio di modellazione visuale standard per sistemi object-oriented, sviluppato dall'OMG (Object Management Group), con lo scopo di integrare le "Best Practices", cioè le linee di condotta migliori, nel campo della progettazione software orientata agli oggetti, in particolare quelle precedentemente proposte da G. Booch, I. Jacobson e J. Rumbaugh, i "tres amigos" autori di questo linguaggio di progettazione. Vengono poi presentati il concetto di "metamodello", un modello utilizzato per descrive un altro linguaggio di modellazione, e la tecnologia MOF (Meta Object Facility), adottata dall'OMG per la definizione di una sintassi astratta comune per metamodelli. Il metamodello UML, di cui ogni schema UML costituisce un'istanza, ha lo scopo di fornire una definizione univoca, comune e definitiva della sintassi e della semantica dell'UML e viene espresso in notazione UML. Successivamente sono brevemente elencate le caratteristiche di due tool, basati sullo standard UML, che si propongono di fornire un efficace sostegno durante le fasi di analisi e progettazione di un sistema software object-oriented: Rational Rose, prodotto dalla Rational Software Corporation, e ArgoUML, sviluppato presso l'Università della California. Infine, è presentata la sintassi di OCL, un linguaggio sviluppato da IBM per la definizione formale di vincoli, utilizzato nella specifica di UML per esprimere alcune regole di correttezza (*well-formedness rules*).

Il **Capitolo 4** mostra come l'UML possa essere considerato una notazione eccellente nello sviluppo di ontologie per il Semantic Web. Un gruppo di ricercatori, tra i quali K. Baclawski, sta cercando di rimuovere le critiche riguardo l'adattabilità dell'UML nella rappresentazione di modelli formali come le ontologie, e ha proposto un'estensione dell'UML tramite un insieme di stereotipi (specializzazioni di costrutti di modellazione UML) che corrispondano a classi e proprietà in DAML+OIL. Viene inoltre proposta un'estensione al metamodello UML per renderlo compatibile con le proprietà DAML+OIL. Il problema di mappare le specifiche UML in quelle dei linguaggi tipici di rappresentazione della conoscenza sta alla base di molti progetti che si propongono di formalizzare complesse ontologie attraverso l'utilizzo di UML. In particolare, sono descritti in questo capitolo il progetto UBOT (UML Based Ontology Tool-set), sviluppato dalla Lockheed Martin Corporation, e il tool UML-Data-Binding, proposto da S. Cranefield presso l'Università di Otago (Nuova Zelanda).

1. Semantic Web

La maggior parte dei contenuti del Web, attualmente, sono rappresentati in modo che possano essere usufruiti da utenti umani, ma non sono progettati per essere elaborati da agenti software. Un browser può, fornendo ben precise istruzioni, disporre e visualizzare le informazioni in una certa maniera, ma non ha modo, al giorno d'oggi, di riconoscerle o di attribuire loro un particolare significato per elaborarle. In questo contesto si sta sviluppando una nuova visione del Web, che conferisce ai contenuti presenti in rete una struttura semantica. Questo è ciò che Tim Berners-Lee chiama Semantic Web [1, 2], che può essere considerato la terza generazione del Web. La prima generazione del Web è stata quella delle pagine HTML statiche. La seconda, quella attuale, è quella delle pagine dinamiche, generate attraverso l'interrogazione dei dati contenuti in un database. In questo caso le informazioni sono utilizzabili dalle persone, ma non dagli elaboratori, se non dopo una fase di post-elaborazione delle informazioni. L'esigenza di una terza generazione del Web nasce dal continuo aumentare delle informazioni disponibili e dalle crescenti aspettative che la gente ripone in Internet. Se, fino ad ora, il Web è stato un mezzo di trasmissione di documenti per le persone, in questo modo diventa una fonte di dati e informazioni che possono venire elaborate automaticamente. Il Semantic Web non è separato dal Web tradizionalmente inteso, non dovrà cioè essere un'entità separata da ciò che è adesso il Web, ma ne costituisce una sua estensione, nella quale le informazioni hanno associato un ben definito significato, migliorando così la cooperazione uomo-macchina. Introducendo il significato dei dati viene aggiunta una nuova funzionalità alle macchine, che diventano in grado di capire ed elaborare i dati, che prima semplicemente visualizzavano. I dati presenti nella rete, cioè, non saranno più solo *machine-readable* ma diventeranno anche *machine-understandable*. L'idea, infatti, è di generare documenti che possano al tempo stesso essere letti ed apprezzati da esseri umani, ma anche acceduti ed interpretati da agenti automatici alla ricerca di contenuti, così da rendere le informazioni utilizzabili direttamente dagli elaboratori. Nel Web Semantico non si esprimono testi, all'interno dei quali le informazioni sono nascoste e richiedono un intervento umano, ma affermazioni, informazioni non ambigue, che esprimono relazioni tra oggetti, risorse, esseri umani, fatti del mondo reale, e che possono essere utilizzate anche da applicazioni automatiche. Il Web si deve dunque dotare di una sovrastruttura semantica utilizzabile dalle applicazioni, in modo da poter svolgere quelle funzioni che oggi debbono essere fatte a mano o codificate dentro ai programmi. Aggiungere la semantica ai contenuti del Web richiede la creazione di un linguaggio che permetta di esprimere dati e regole per i ragionamenti. Il Semantic Web, facendo riferimento a ogni entità con un URI (Uniform Resource Identifier), permetterà a ciascuno di descrivere nuovi concetti, eventualmente basandosi su altri concetti già esistenti, e l'utilizzo di linguaggi standard consentirà alle

informazioni di essere collegate al resto del Web, permettendo che le regole proprie di un dato sistema di rappresentazione della conoscenza possano essere utilizzate sul Web. Le potenzialità del Semantic Web sarebbero inutili se non ci fossero degli agenti software in grado di raccogliere le informazioni dalle eterogenee sorgenti, elaborarle e scambiarle con altri programmi. La struttura che si verrà a formare dovrà essere un framework dove agenti software, che viaggiano da una pagina ad un'altra, possano eseguire compiti sofisticati per utenti. La rappresentazione delle informazioni sarà imperniata sulla conoscenza e sullo studio di questi agenti software e fornirà una serie di strumenti nuovi, in grado di comunicare e lavorare insieme. Per la natura stessa del Web è possibile che dati concettualmente diversi siano rappresentati con lo stesso nome, o viceversa. Un programma dovrebbe essere in grado di distinguere il contesto ed elaborare le informazioni in maniera appropriata. A tale scopo vengono creati sistemi di integrazione delle informazioni e ontologie [Capitolo 2]. Il raggiungimento di questo obiettivo avrà ripercussioni in molti campi, come i motori di ricerca, l'e-commerce e i diritti di proprietà. Attualmente i motori di ricerca restituiscono un'alta percentuale di documenti che non riguardano l'argomento cercato dall'utente. Fornendo la capacità di capire il significato delle parole si aumenteranno le capacità di ricerca. Un sito di e-commerce potrà capire meglio le preferenze di un cliente se capisce il significato delle parole che esprimono tali preferenze. Si potrà inoltre esprimere in maniera più chiara la proprietà intellettuale di una pagina Web, in quanto sarà chiaro il significato di parole quali: autore, editore, etc.

1° generazione:	**Pagine statiche** HTML
2° generazione: Attuale	**Pagine dinamiche** Pagine generate attraverso l'interrogazione dei dati contenuti in un database Le ricerche avvengono ancora sul semplice match di keyword di cui non si conosce il significato Le informazioni sono *machine-readable*, ma non *machine-understandable*
3° generazione:	**Web Semantico** (Tim Berners-Lee [1, 2]) Le informazioni Web hanno una sovrastruttura semantica → → non sono solo *machine-readable*, ma anche *machine-understandable* Creazione di ontologie Utilizzo di linguaggi standard appropriati Utilizzo di agenti mobili Dati: netta separazione tra mark-up dei contenuti e mark-up di presentazione Precisa formalizzazione dei metadati

Figura 1 - Le tre generazioni del Web

1.1 Web e basi di conoscenza

L'insieme delle conoscenze di un sistema prende il nome di "base di conoscenza" ed è descritto da un linguaggio di rappresentazione. La rappresentazione della conoscenza è uno dei campi dell'Intelligenza Artificiale [3] e si occupa di gestire il modo in cui un programma modella la realtà del mondo in cui è inserito. La conoscenza su una realtà, infatti, per poter essere manipolata deve venire strutturata. Il linguaggio utilizzato per descrivere questa realtà deve essere conciso e non ambiguo. Esso deve indicare, con una sintassi formale e comprensibile alle macchine, come le informazioni sono definite e il tipo di operazioni che sono supportate e deve essere indipendente dal contesto, cioè deve poter raffigurare la conoscenza di qualunque tipo di sistema. Il Web stesso può essere inteso come una base di conoscenza, una fonte di informazioni facilmente reperibili e su cui si possono effettuare interrogazioni. Alcune caratteristiche del Web sono però in contrasto con la conoscenza tradizionale di base di conoscenza. Il Web, infatti, è un immenso contenitore di informazioni e il database che si otterrebbe da esso sarebbe così grande da essere ingestibile dai sistemi di rappresentazione esistenti. Il Web è anche un mondo "aperto" e dinamico, dove le pagine crescono a dismisura ed ognuna contiene una parte infinitesima di quello che può essere raccolto da un agente software, dove nuove pagine vengono continuamente aggiunte e pagine esistenti vengono modificate o rimosse molto rapidamente. Il Web, quindi, è un mondo in continua evoluzione, dove i dati disponibili cambiano in maniera imprevedibile e ciò fa si che le informazioni possano essere, di volta in volta, inconsistenti, inaffidabili o non disponibili. Un sistema di rappresentazione della conoscenza, invece, considera il dominio come un mondo chiuso, ma questa completezza non può essere garantita nella gestione del Web. Questo porterà ad accettare una serie di compromessi. L'introduzione di contenuti semantici, infatti, dovrà cercare di mantenere le caratteristiche peculiari del Web, come la larga quantità di informazioni disponibili e la decentralizzazione delle informazioni, ma contemporaneamente dovrà consentire una ricerca più mirata e precisa e uno scambio di informazioni tra gli agenti software, che le trovano nel Web e le elaborano in base al loro significato. Bisogna precisare che esattamente come il Web tradizionale si è sviluppato senza controlli e senza regole, è difficile immaginare che il Semantic Web si evolva come un insieme ordinato di ontologie, costruite da esperti di Intelligenza Artificiale. Soprattutto nei primi tempi, esso si estenderà con un gran numero di ontologie di dimensioni ridotte, che fanno riferimento l'una all'altra (per riutilizzare o modificare termini comuni) e saranno sviluppate dagli utenti, analogamente a quanto avviene per i contenuti del Web attualmente creati.

1.2 Architettura del Semantic Web

Il Web Semantico può essere visto come un'architettura a 3 livelli, schematizzata in figura 2:

- Livello Logico
- Livello Schema
- Livello Dati

A questi può essere aggiunto un quarto livello, quello ontologico, che si inserisce tra il livello logico e il livello schema. Esso prevede la definizione delle ontologie, ad esempio attraverso il formalismo DAML+OIL.

Per far sì che il Web Semantico rappresenti una svolta è necessario garantire due livelli di interoperabilità, semantica e sintattica. Per stabilire questi due livelli di interoperabilità è necessario un linguaggio che permetta di formalizzare una semantica e di fornire supporto al ragionamento, è necessario cioè prevedere un livello logico nell'architettura del Web Semantico.

- Interoperabilità sintattica: è la capacità di leggere i dati ed ottenere una rappresentazione utilizzabile da un'altra applicazione. Questo obiettivo è già stato da tempo raggiunto grazie all'affermarsi di XML.

- Interoperabilità semantica: è la capacità di comprendere i dati. Questo livello di interoperabilità ha avuto negli ultimi tempi un notevole contributo dallo sviluppo di RDF e RDF-Schema.

Il livello logico è il livello superiore, quello che fornisce gli strumenti per definire la semantica e dare un supporto per "ragionare" con i dati. I linguaggi associati a questo livello sono i linguaggi tipici del Semantic Web [Paragrafo 1.4], cioè i linguaggi di rappresentazione della conoscenza, i cosiddetti Linguaggi KR (Knowledge Representation Languages) [4]. Questi linguaggi, contrariamente a molti linguaggi di modellazione di dati, non hanno una rigida separazione tra metalivelli. Infatti, tutte le affermazioni nel linguaggio sono in un singolo spazio di affermazioni, includendo relazioni come "instanceOf", che attraversa metalivelli. Una catena di "instanceOf" può essere di ogni lunghezza. Quindi questi linguaggi incorporano non solo capacità di modellazione, ma allo stesso tempo includono metamodelli, meta-metamodelli etc. [Capitolo 3].

1. Livello Logico Logical Layer	Semantica formale e supporto al ragionamento Linguaggi KR: linguaggi di rappresentazione della conoscenza • OIL • DAML • DAML+OIL • OWL MOMIS = $ODLI_3$ + OLCD
2. Livello Schema Schema Layer	Definizione di un vocabolario, di concetti Non c'è una logica descrittiva, non fornisce alcun supporto al ragionamento, quindi non siamo al livello logico, ma ad un livello inferiore • RDF-Schema + XML-Schema
3. Livello Dati Data Layer	Sintassi per le istanze dei concetti Modello dei dati e sintassi per i metadati • RDF + XML

Figura 2 - Livelli del Semantic Web

Come in ogni Data Language, i Linguaggi KR hanno l'abilità di definire schemi che definiscono la struttura e i vincoli tra i dati (istanze o oggetti). Uno schema in un Linguaggio KR è detto "ontologia" [Paragrafo 2.2.1]. Esistono tre categorie di Linguaggi KR:

- Linguaggi logici: esprimono conoscenza sotto forma di affermazioni logiche. Un esempio è rappresentato da KIF (Knowledge Interchange Format) [7].

- Linguaggi frame-based, basati su frame: sono simili ai linguaggi dei database object-oriented. Esempi di questi linguaggi sono RDF o DAML [Paragrafo 1.4].

- Linguaggi graph-based, basati su grafi: includono reti semantiche e grafi concettuali. La conoscenza è rappresentata usando nodi e link tra nodi.

I linguaggi logici si fondano su logiche descrittive, DLs (Description Logics), o su logiche del primo ordine. Ricordiamo che le logiche descrittive sono un sottoinsieme delle logiche del primo ordine. Citando F. Baader e W. Nutt [5, 6], DLs "è il nome più recente per una famiglia di formalismi di rappresentazione della conoscenza che rappresentano la conoscenza di un dominio di applicazione (il mondo), prima definendo i concetti rilevanti del dominio (la sua terminologia), e poi usando questi concetti per specificare proprietà di oggetti e individui presenti nel dominio (la descrizione del mondo)". Le logiche descrittive sono linguaggi, elaborati nella ricerca sulle rappresentazioni della conoscenza, che descrivono la conoscenza in termini di concetti e ruoli. Le espressioni possono essere descritte in forma matematica, in modo che si possano fare ragionamenti, basati sulla descrizione dei concetti, e classificazioni automatiche. Nel ambito del sistema MOMIS [Paragrafo 2.3], ad esempio, il livello logico è costituito dall'uso congiunto del linguaggio object-oriented ODLI3 e dalla sua logica descrittiva OLCD [Paragrafo 2.4.3].

Linguaggio	Logica sottesa	
	Logica del primo ordine	Logica descrittiva (DL)
$ODLI_3$		OLCD
OIL		SHIQ [65]
DAML+OIL	KIF	
OWL		SHOIN(D) [5]

1.3 Metadati

Lo scopo dei linguaggi del Semantic Web è di ottenere interoperabilità semantica tra pagina Web, database, programmi, etc. Utilizzare metadati (o meta-informazioni) è di supporto nell'affrontare problematiche relative all'integrazione semantica di sorgenti così eterogenee. I metadati, infatti, sono informazioni che predicano su altre informazioni. Nell'ambito degli schemi, ad esempio, esprimono i dati che descrivono le risorse Web. Il Web Semantico prevede l'utilizzo sistematico di meta-informazioni. I metadati, infatti, permettono di ragionare ad un livello superiore rispetto alle singole fonti eterogenee, in quanto permettono un'astrazione del significato dei singoli dati descritti. Questo permetterà ad agenti software di ragionare sulla conoscenza fornita dalle varie sorgenti e di integrare servizi a run-time, superando il limite di effettuare ricerche sul semplice match di keyword di cui non si conosce il significato. Il linguaggio DAML+OIL, ad esempio, permette di descrivere ontologie, che sono alla base dell'attività degli agenti mobili, e di arricchire e formalizzare le annotazioni riguardo i metadati, in modo da rendere *machine-understable* la conoscenza relativa ad un documento e da supportare il ragionamento e la comunicazione degli agenti mobili.

I metadati includono:

- significati delle classi (o relazioni) e delle loro proprietà (attributi), cioè le annotazioni
- relazioni sintattiche e/o semantiche esistenti tra i modelli delle sorgenti
- caratteristiche delle istanze nelle sorgenti: tipi degli attributi, valori null, etc.

1.4 I Linguaggi del Semantic Web

Un aspetto molto importante per la creazione del Semantic Web consiste nel rendere possibili agli utenti, che non sono esperti di logica, la creazione di ontologie e la realizzazione di contenuti Web che siano leggibili dalle macchine. Per far fronte a queste esigenze il World Wide Web Consortium (W3C) [8] ha progettato una serie di linguaggi che, nel corso del tempo, sono diventati degli standard, e che consentono proprio la strutturazione semantica delle informazioni e la condivisione dei significati dei concetti. Ricordiamo brevemente i linguaggi nati per il Web, per poi soffermarci sui linguaggi specifici per il Web Semantico.

Linguaggi per il Web:

HTML (HyperText Markup Language):

E' lo standard per strutturare i documenti, il linguaggio di formattazione per la creazione di pagine Web che, originariamente, ha permesso un incremento molto rapido del Web. Non ha le caratteristiche per strutturare i dati in base alla loro semantica, poiché specifica solo come devono essere visualizzate le informazioni, e pertanto deve essere affiancato da altri strumenti che permettano la strutturazione semantica. HTML è un linguaggio molto semplice che utilizza delle istruzioni, i tag, per indicare l'aspetto che le informazioni devono assumere a video, ma che non si occupa di gestirle o elaborarle. La soluzione per risolvere questa incapacità di capire la semantica dei concetti contenuti in una pagina Web, consiste di strumenti che permettono di associare ad ogni concetto il suo significato, affinché esso sia disponibile per gli agenti software. In un documento HTML non c'è niente che indichi l'argomento trattato o la fonte delle informazioni. L'unico tipo di ricerca che si può fare su un documento è la ricerca sul contenuto. Questo non è sufficiente nella maggior parte delle volte, usando un motore di ricerca, infatti, si può ottenere un qualche migliaio di hit, la maggior parte dei quali non serve assolutamente a niente.

XML (eXtensible Markup language) [Paragrafo 1.4.1]:

E' il primo linguaggio che separa il mark-up dei contenuti dal mark-up di presentazione. Questo comporta la separazione e l'indipendenza tra il contenuto e il layout delle pagine. XML consente la creazione di documenti strutturati, ma non gestisce la semantica dei contenuti (un programma riconosce i contenuti, ma non è in grado di attribuire loro un significato) e non fornisce meccanismi di classificazione o ragionamento. Esso costituisce una meta-sintassi utilizzabile da ogni applicazione ma fornisce solo un modello dei dati e una convenzione sintattica, quindi deve essere affiancato ad altri linguaggi più potenti.

Linguaggi per il Semantic Web:

RDF (Resource Description Framework) e **RDF-Schema** [Paragrafi 1.4.3 e 1.4.4]:

(Useremo il termine RDF(S) per riferirci a RDF e RDF-Schema)

RDF(S) è un linguaggio per la descrizione delle informazioni delle risorse del Web, sviluppato dal W3C come uno standard, un modello per esprimere affermazioni sul mondo, per la gestione dei metadati, con lo scopo di aggiungere una semantica formale al Web. RDF(S) definisce un meccanismo generale di rappresentazione della conoscenza, neutrale per quanto riguarda il dominio, perché deve essere in grado di descrivere informazioni di qualunque natura. E' un linguaggio per esprimere affermazioni che permette di definire un vocabolario, un insieme di termini e loro definizioni. RDF(S) non descrive solo ciò che è presente nel Web, come le pagine, ma descrive le risorse, tutto quello a cui può essere associato un identificatore Web, chiamato URI (Uniform Resource Identifier). Gli URI (stringhe identificative di qualunque tipo di oggetto o concetto) non sono necessariamente URL (Uniform Resource Locator), indirizzi accessibili di documenti Web.

OIL (Ontology Inference Layer) [17, 65]:

Il progetto OIL è finanziato dal programma dell'Unione Europea IST (Information Society Technologies), sotto il progetto On-To-Knowledge [18]. Poiché questo progetto si basa sull'approccio ontologico, fa uso di ontologie per quanto riguarda i processi di integrazione e di condivisione delle informazioni. OIL è nato come linguaggio per la creazione e la definizione di ontologie, basato anch'esso sugli standard del W3C RDF/RDF-Schema e XML/XML-Schema, per aumentarne l'espressività. Esso combina le primitive di modellazione di linguaggi frame-based con le capacità offerte dalle Logiche Descrittive.

DAML (DARPA Agent Markup Language) [19]:

E' un linguaggio creato per ragionare efficacemente con le ontologie, proposto da DARPA, come base per il Semantic Web. Nasce come un'estensione di RDF e RDF-Schema, ma è capace di esprimere una più ricca varietà di vincoli rispetto a questi ultimi, in modo da supportare l'inferenza logica. DAML permette di inserire il contenuto semantico dei dati, basandosi su ontologie costruite con RDF-Schema.

DAML+OIL [Paragrafo 1.4.5]:

E' un linguaggio ontologico standard, sviluppato da DARPA, di approccio object-oriented, per la realizzazione del Semantic Web. Esso consente la definizione di ontologie e istanze per strutturare le informazioni contenute nelle pagine Web, affinché siano comprensibili non solo dagli utenti umani, ma anche dagli agenti software. Originariamente erano due linguaggi separati: DAML permetteva di inserire il contenuto semantico dei dati, basandosi sulle ontologie

definite con RDF(S), OIL era nato come un altro linguaggio per la creazione di ontologie, basato anch'esso su RDF(S). Ad un certo punto, è stato evidente che i due prodotti potevano essere uniti e il risultato è stato questo linguaggio ontologico, che consente di descrivere la struttura di un dominio. DAML+OIL, come suggerisce il nome, prende molte delle caratteristiche di OIL, per cui le capacità dei due linguaggi sono molto simili. Le differenze più significative sono:

- OIL ha una maggior compatibilità con RDF(S) rispetto a DAML+OIL
- OIL permette l'indicazione di condizioni necessarie e sufficienti in maniera più semplice che in DAML+OIL
- DAML+OIL è più espressivo e permette di esprimere vincoli più complessi di OIL

OWL (Web Ontology Language) [21]:

E' un linguaggio di mark-up semantico, sviluppato dal W3C, per pubblicare e condividere ontologie nell'ambito del World Wide Web. Poiché un'ontologia può includere descrizioni di classi, proprietà e loro istanze, OWL può essere usato per formalizzare un dominio definendo classi e loro proprietà, per definire individui e asserire proprietà su di essi e per ragionare su queste classi e individui per quanto è permesso dalla semantica del linguaggio. OWL è stato sviluppato come un'estensione del vocabolario di RDF, deriva da DAML+OIL ed è influenzato fortemente dalle Logiche Descrittive, dal paradigma a frame e dalla sintassi RDF. In particolare, le specifiche formali del linguaggio OWL risentono delle Logiche Descrittive, la sintassi astratta è influenzata dal paradigma a frame e la sintassi di interscambio è disegnata per essere compatibile con RDF. Esistono tre livelli di OWL di complessità crescente [5]:

- **Lite**: permette la realizzazione di gerarchie di classificazione e semplici vincoli. E' semplice da usare e implementare ma scarsamente espressivo. Impone ulteriori vincoli sull'uso dei costruttori del linguaggio, oltre a quelli imposti da OWL DL
- **DL**: abbastanza espressivo ma comunque decidibile e dotato di procedure di ragionamento di complessità nota, approfonditamente studiate e ormai ben ottimizzate. Impone un certo numero di vincoli sull'uso dei costruttori del linguaggio

- **Full**: consente di definire rappresentazioni che vanno al di là della logica predicativa del primo ordine, è molto espressivo, ma non decidibile, e quindi problematico dal punto di vista della meccanizzazione del ragionamento. Contiene tutti i costruttori OWL e permette l'uso libero e incondizionato dei costruttori RDF(S)

Per una descrizione più dettagliata di OWL si veda [69].

Oltre a questi standard, un gruppo di ricerca dell'Università del Maryland ha realizzato un sistema con funzionalità e intenti analoghi al W3C, proponendo il linguaggio **SHOE** [21].

1.4.1 XML

XML è il primo linguaggio che separa il mark-up dei contenuti dal markup di presentazione. Esso consente la creazione di documenti strutturati ma non gestisce la semantica dei contenuti (un programma riconosce i contenuti, ma non è in grado di attribuire loro un significato) e non fornisce meccanismi di classificazione o ragionamento. La definizione di XML aveva come obiettivo originario il superamento dei limiti di HTML relativamente al mark-up per il Web. Nasce poi l'idea che XML possa servire per qualcosa più, come un linguaggio di markup per trasferire dati, non pensato per la visione umana, ma per essere prodotto ed usato da programmi, un meccanismo per convertire dati dal formato interno dell'applicazione ad un formato di trasporto, facile da convertire in altri formati interni. In quanto metalinguaggio, XML ha la possibilità di esprimere nuovi tag senza bisogno di ridefinire una nuova versione del linguaggio (in questo consiste la sua estensibiltà). Grazie alla sua flessibilità, è diventato uno standard universale per lo scambio di dati sul Web, offrendo un semplice metodo per integrare applicazioni distribuite. La libertà con cui si possono inventare nuovi tags e la netta separazione dei dati dalla loro rappresentazione, fa si che XML sia il linguaggio ideale per la descrizione di informazioni di vario tipo. XML non consente solo di descrivere dati liberamente, ma anche di interpretarli in maniera automatica tramite tool standard. Esso è infatti utilizzato per creare una "struttura digitale" dei documenti, cioè i documenti sono rappresentati in maniera da essere interpretati da un computer, affinché le informazioni contenute possano essere elaborate, memorizzate, ricercate o stampate. Prima di tutto, si deve definire una struttura logica del documento che è comune a tutti i documenti di un certo tipo e individuare i suoi componenti. Ognuno di questi è un elemento XML ed è legato agli altri da vincoli di dipendenza. Tutti gli elementi che compongono un documento hanno una struttura ad albero, dove l'elemento

principale è la radice, gli altri sono i suoi sottoelementi, che sono a loro volta rami o foglie a seconda che abbiano o no dei discendenti. Ad esempio, si consideri come tipo di documento un messaggio di posta elettronica: partendo dall'elemento principale, e-message, si considerano le sue componenti, che a loro volta sono scomposte, ottenendo la struttura rappresentata in figura:

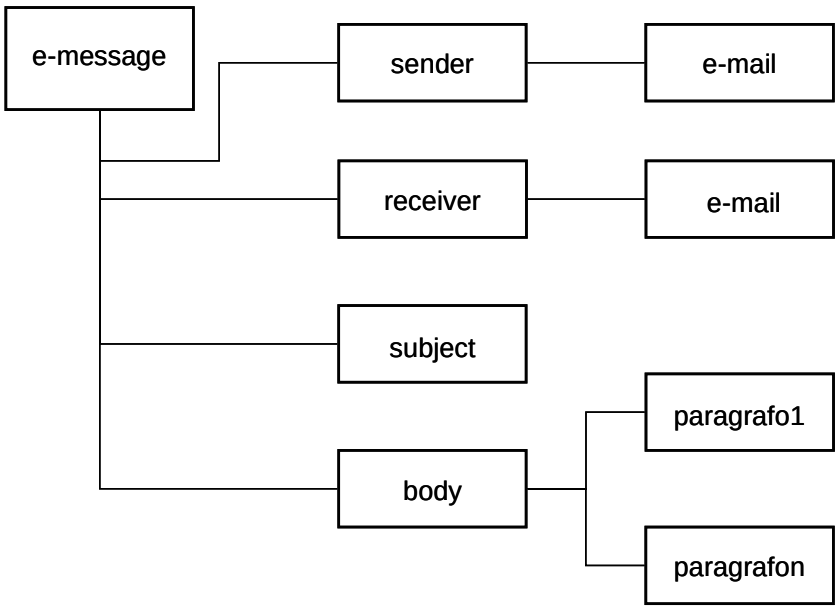

Figura 3 - Documento XML

Ogni elemento può avere degli attributi, che ne descrivono le proprietà. Si noti che la strutturazione di un documento in questi termini non è immediata, alcuni aspetti possono essere rappresentati sia come elementi sia come attributi e da alcuni tipi di documento non è possibile ricavare una struttura perfettamente ad albero. Anche alla presenza di collegamenti fra un elemento dell'albero ed un altro, XML non dà problemi. La struttura fisica di un documento XML è formata da entità. Un'entità è un costrutto che rappresenta una parte del testo, ha una dimensione variabile (da un carattere a tutto il documento) ed ha un nome, per fare i riferimenti ad essa (con un funzionamento analogo a quello delle macro). Come gli elementi indicano i componenti della struttura logica del documento, le entità specificano la posizione di insiemi di byte. Si deve fare in modo che questi due aspetti siano correlati tra loro. L'insieme di comandi che stabiliscono questo legame è il mark-up, costituito da tag. Questi comandi sono distinti dal testo mediante dei delimitatori e in pratica ciò che è racchiuso fra i simboli di minore ("<") e maggiore (">") costituisce il mark-up, che è la parte che è compresa dagli elaboratori. La forma è analoga a quella dei tag HTML, ma, se in questo caso danno le indicazioni per la

formattazione delle informazioni, in un documento XML non ci dovrebbero essere istruzioni per la loro visualizzazione. Quello che non è racchiuso fra i tag, è interpretato come testo e non è generalmente analizzato.

Il documento XML dell'esempio precedente può essere il seguente:

```
<?XML version="1.0"?>
<e-message>
        <sender e-mail=ponchiroli.patrizia@unimo.it> Patrizia Ponchiroli </sender>
        <receiver e-mail=anzola.fabio@tin.it> Fabio Anzola </receiver>
        <subject> Semantic Web </subject>
        <body>
                <primo-Paragrafo> ... testo primo Paragrafo </primo-Paragrafo>
                <secondo-Paragrafo> ... testo secondo Paragrafo </secondo-Paragrafo>
        </body>
</e-message>
```

Si noti come l'indirizzo di mittente e destinatario siano stati considerati come attributi: non ci sono regole precise per scegliere una soluzione piuttosto che un'altra, ma genericamente si considera come attributo ciò che non contiene degli elementi e dà informazioni semplici e non strutturate. Generalmente, gli attributi sono delle stringhe che indicano una proprietà dell'elemento a cui sono associati.

Nell'ottica della portabilità dei dati tra le applicazioni, indipendentemente dalla piattaforma utilizzata, XML può essere considerata una lingua franca per l'interscambio di dati tra le applicazioni e, quindi, per una prima forma di interoperabilità. XML è ottimo come sintassi e struttura dati, elimina ambiguità tra contenuto e mark-up, elimina incertezze e dipendenze da specifiche codifiche carattere e fornisce API (Application Protocol Interface) e modelli concettuali semplici per trattare qualunque tipo di struttura dati. XML non è però perfettamente adatto per il Semantic Web, in quanto esistono troppi modi "linguistici" per esprimere gli stessi concetti e gli attributi e le entità sono retaggio di un passato di linguaggio per documenti pensati per essere letti. XML fornisce solo un modello dei dati e una convenzione sintattica, quindi deve essere affiancato ad altri linguaggi più potenti. E' meglio quindi trovare un modello astratto per esprimere i concetti, RDF(S), e lasciare ad XML il compito di renderli in maniera linguistica.

1.4.1.1 DTD

La libertà di invenzione di nuovi tag permessa dall'XML non crea situazioni anarchiche in quanto essa è regolata da norme ben precise. Un documento XML, infatti, è valido solo se fa riferimento ad una DTD (Document Type Definition), o ad uno Schema, e ne segue le regole grammaticali. La funzione delle DTD è quella di definire formalmente la struttura e la sintassi di ogni tipo di documento XML, specificandone gli elementi, gli attributi e le entità, ma non il loro contenuto. Se il documento non è valido, potrebbe essere ben formato (cioè contenente un mark-up comprensibile), ma non conforme ad una DTD. Un documento XML deve dichiarare a quale DTD si conforma per verificarne la validità, ma tale dichiarazione non è obbligatoria e, se è assente, il documento non è considerato né valido, né non valido. Le DTD hanno una sintassi particolare, diversa dall'XML, perciò sono richiesti strumenti appositi per la validazione. Inoltre, esse non distinguono tra nome del tag e tipo del tag, ed hanno solo due tipi: complesso (strutturato) e semplice (CDATA o #PCDATA).

1.4.1.2 XML-Schema

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

- Utilizzare applicazioni XML per la verifica della validità dei dati espressi
- Specificare il tipo di dati del contenuto di ciascun elemento. XML-Schema fornisce un set complesso di tipi, a cui i tag e il loro contenuto debbono aderire
- Esprimere maggiori vincoli in merito al numero di occorrenze di un elemento in una struttura, e alla sequenza con la quale avvengono tali occorrenze
- Fornire un approccio object-oriented, permettendo di ampliare i tipi disponibili e di estenderne e precisarne le proprietà

- Implementare il concetto di Key e ForeignKey

- Utilizzare il concetto di Namespace, che consente di definire dei domini di elementi

1.4.2 Tecnologie basate su XML

In questo paragrafo verranno presentate tre tecnologie basate su XML:

- Il linguaggio XSLT, lo standard definito dal World Wide Web Consortium per trasformare documenti XML in altri documenti

- XLink, un linguaggio per ottenere link unidirezionali e multidirezionali tra diversi elementi dello stesso documento XML o di documenti XML differenti

- SVG, un formato di grafica vettoriale progettato per il Web

1.4.2.1 XSLT

XSLT (XSL Trasformations) [11] è un linguaggio per la trasformazione di documenti XML in altri documenti. E' stato progettato come parte di XSL, tuttavia risulta del tutto indipendente da esso. **XSL** (eXtensible Stylesheet Language) permette di creare fogli di stile (stylesheet) per specificare come il contenuto di un documento XML debba essere visualizzato sul supporto scelto. Con questo strumento si realizza uno scopo primario dell'XML, mantenere separarti gli elementi di contenuto da quelli di formattazione. XSL aggiunge a XSLT determinate regole di formattazione per documenti XML. XSLT prevede la costruzione di un albero che rispecchia la struttura del documento da processare, attraverso l'associazione di pattern con template. Un template definisce le regole di trasformazione di una determinata porzione del documento XML d'origine. Ogni template è costituito da due parti: un pattern, per individuare i nodi a cui applicare le trasformazioni, ed un template, che può essere istanziato per ottenere una porzione dell'albero risultante. Quando un pattern viene riconosciuto all'interno del documento di partenza, viene istanziato il template relativo ad esso. Quest'ultimo può risultare completamente diverso da quello di partenza, può essere arricchito con strutture arbitrarie e può presentarsi in

forma di testo, pagina HTML o documento XML. Poiché XSLT è parte di XSL, una trasformazione XSLT è chiamata anche stylesheet.

1.4.2.2 XLink

XML non è in grado di realizzare i collegamenti ipertestuali. Per questo motivo è stato affiancato da uno standard, Xlink (XML Linking Language) [12], che permette di inserire elementi in un documento in modo da creare e descrivere collegamenti tra risorse. I link che si possono ottenere non sono solo di tipo unidirezionale e uno a uno, come nell'HTML, ma possono essere bidirizionali, uno a molti, molti a molti. Uno dei motivi del successo del World Wide Web è da ricercare proprio nella capacità di collegare risorse differenti. Tuttavia, i semplici collegamenti unidirezionali forniti dal linguaggio HTML, possono risultare spesso limitanti, soprattutto all'interno di un ambiente ipertestuale basato su XML. La soluzione proposta dal World Wide Web Consortium è rappresentata proprio dall'XLink, che definisce la sintassi e la semantica di link unidirezionali e multidirezionali. Xlink specifica i costrutti utilizzabili all'interno di documenti XML per descrivere collegamenti tra risorse differenti, o tra porzioni di esse. Il concetto di risorsa è definito dal W3C come qualsiasi sorgente di informazione o servizio a cui è possibile fare riferimento in qualche modo. Esempi di risorse sono file, immagini, documenti, programmi, risultati di ricerche. Un riferimento ad una risorsa è sempre esprimibile mediante un URI. I link XLink possono essere di due tipi:

- Link semplici: coinvolgono esattamente due risorse e risultano del tutto simili ai tipici link utilizzati in HTML
- Link estesi: consentono di mettere in relazione un numero arbitrario di risorse differenti. Rappresentano una novità rispetto ai link resi disponibili dal linguaggio HTML e la loro struttura può essere anche molto complessa e articolata

1.4.2.3 SVG

SVG (Scalable Vector Graphics) [13] è un linguaggio di grafica vettoriale bidimensionale per il Web basato su XML. Fu proposto per la prima volta al W3C da Adobe Systems nel 1999. L'obiettivo di SVG è ambizioso e consiste nel divenire il linguaggio universale per la realizzazione di pagine Web, in sostituzione dei linguaggi esistenti, in particolare HTML. Allo

stato attuale, SVG è supportato all'interno dei browser Web mediante appositi plug-in, il più noto dei quali è fornito gratuitamente da Adobe Systems. Gli oggetti grafici bidimensionali descritti da SVG possono essere di tre tipi:

- figure geometriche, basate su grafica vettoriale, composte da linee e curve
- immagini, rappresentate da file binari, solitamente in formato PNG o JPEG
- testo

Questi oggetti grafici possono essere raggruppati e possono subire trasformazioni generiche, inoltre per essi è possibile definire uno stile grafico comune. Il linguaggio SVG rappresenta una tecnologia estremamente promettente perché fornisce uno standard per la rappresentazione di informazioni grafiche. Esso viene utilizzato da ArgoUML [Capitolo 3] per salvare le informazioni grafiche relative ad un diagramma UML.

PGML (Precision Graphics Markup Language) è un altro linguaggio di grafica vettoriale per il Web proposto al W3C da Adobe Systems, che dopo poco tempo è stato sostituito da SVG.

1.4.3 RDF

RDF è un linguaggio per la descrizione delle informazioni delle risorse del Web, sviluppato dal W3C come uno standard, un modello per esprimere affermazioni sul mondo, per la gestione dei metadati, con lo scopo di aggiungere una semantica formale al Web. Se XML è un'estensione del documento, RDF è un'estensione dei dati introdotti da XML ed è infatti definito sopra XML. RDF rende possibile la specificazione della semantica dei dati basati su XML in modo standardizzato e permette di definire un meccanismo per descrivere le risorse, indipendentemente dal dominio di applicazione e senza definire tale dominio a priori. I documenti RDF permettono di stabilire nuove relazioni fra i documenti, le immagini e qualunque altra risorsa Web, consentendo l'interoperabilità tra le applicazioni che si scambiano via Web informazioni a loro comprensibili. RDF è, infatti, un mezzo per rappresentare i significati dei dati in modo interoperabile, descrivendo le relazioni tra le risorse in termini di proprietà e valori. RDF è stato creato con il particolare intento di rappresentare i metadati delle risorse Web come il titolo, l'autore, la data dell'ultima modifica di una pagina Web, le informazioni sulle licenze e sui copyright di un documento Web o la disponibilità di risorse condivise. Comunque, dalla generalizzazione del concetto di risorsa Web, RDF può anche essere usato per rappresentare informazioni riguardanti tutte le cose che possono essere identificate nel

Web. RDF è stato pensato per quelle situazioni in cui le informazioni devono essere elaborate da un'applicazione per essere presentate a una persona. Esso fornisce una struttura per esprimere queste informazioni, che vengono scambiate tra applicazioni, senza perdite di significato. RDF non descrive solo ciò che è presente nel Web, come le pagine, ma anche tutto quello a cui può essere associato un identificatore Web, chiamato URI (Uniform Resource Identifier). Ad esempio, se si può associare un URI ad una persona, tutte le informazioni ad essa relative potranno essere rappresentate mediante RDF. Questo permette a RDF di rappresentare semplici dichiarazioni di risorse come un grafo di nodi e di archi che rappresentano le proprietà e i valori della risorsa. RDF non è un formato XML, ma un modello astratto. Esistono più linearizzazioni in XML di RDF. Caratteristica di queste linearizzazioni è che non sono univoche. Esistono anche linearizzazioni non XML di RDF. Una di queste, Notation3 [22], è stata introdotta da Berners-Lee, ed ha un certo successo.

Caratteristiche fondamentali di RDF sono:

- Indipendenza:
  Ognuno può inventare le proprietà che gli interessano. Una persona può inventare la proprietà "source", un'altra la proprietà "sorgente", etc.

- Interscambio:
  Le proprietà RDF possono essere espresse in XML, quindi sono facilmente interscambiabili.

- Scalabilità:
  Le proprietà RDF sono tuple con tre campi, in questo modo sono facili da gestire e cercare. Questa caratteristica, in una realtà come Internet, in cui la mole di informazione a disposizione è sempre crescente, è molto importante.

RDF ha un sistema a classi, paragonabile a quelle della programmazione ad oggetti, uno schema è una collezione di classi, le classi formano una gerarchia che, con il meccanismo delle sottoclassi, consente la specializzazione dei concetti e la condivisione delle definizioni dei metadati. E' inoltre permessa l'ereditarietà multipla, che permette la vista multipla dei dati e lo sfruttamento di definizioni create da altri.

1.4.3.1 Modello dei dati

RDF fornisce un metodo non ambiguo per la codifica, lo scambio e il riuso dei metadati grazie ad un modello dei dati simile al modello ER (Entità-Relazioni), anche se quest'ultimo è più generale. RDF può essere considerato l'estensione del modello ER al Web perché ha i concetti di entità e relazione, ma il concetto di relazione è stato elevato allo stesso livello di entità (hanno una loro identità indipendente dalle entità che mettono in relazione), grazie all'utilizzo di URI. In questo modo chiunque può dire tutto su tutto e non è necessario che tutte le informazioni su un'entità siano definite in un unico luogo.

Le corrispondenze tra ER e RDF possono essere schematizzate così:

RDF	ER
Risorsa RDF	Riga di una tabella
Nome di una proprietà	Nome di un campo/colonna
Valore della proprietà	Valore del campo

Il modello dei dati, rappresentato con un grafo RDF, permette di esprimere ogni affermazione come una tripla *(Soggetto, Predicato, Oggetto)*, che costituisce la "asserzione", o "statement", cioè l'associazione di una proprietà ad una risorsa. Il modello di RDF è basato su questi tre concetti principali:

- **Soggetto** (Subject)/Risorsa (Resource):
    - Costituita da un URI
    - E' l'entità a cui si applica la proprietà
    - E' tutto quello che è descritto da espressioni RDF, individuato da un URI. Può essere quindi anche un oggetto non accessibile da Web
    - Graficamente è rappresentata da un nodo

- **Predicato** (Predicate)/Proprietà (PropertyType):

    - E' la relazione, o attributo, una caratteristica usata per descrivere una risorsa
    - Ogni proprietà ha un significato, un insieme di valori che può assumere, le risorse a cui può essere associata ed eventuali relazioni con altre proprietà
    - L'insieme delle proprietà e la loro semantica (poca, a dir la verità) sono definiti in uno schema RDF, il quale rappresenta una sorta di vocabolario. In questo vocabolario possono essere indicate anche le risorse che verranno descritte. I Namespace legano l'uso di un termine in un dominio allo schema in cui si trova la definizione attraverso la stessa notazione dei Namespace in XML
    - Le proprietà di una risorsa possono essere altre risorse
    - Graficamente è rappresentata da un arco

- **Oggetto** (Object)/Valore (Value):

    - E' un'altra risorsa, oppure un valore letterale
    - Graficamente è rappresentato da un nodo

Il modello dei dati RDF si può schematizzare graficamente, ad esempio:

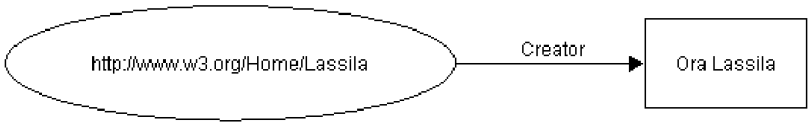

Figura 4 - RDF: Esempio

Questo grafico si legge così:

"La proprietà Creator della risorsa http://www.w3.org/Home/Lassila vale Ora Lassila" oppure

“Ora Lassila è il Creator della risorsa http://www.w3.org/Home/Lassila”.

Questo primo caso diventa in sintassi estesa:

```
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:s="http://description.org/schema/">
        <rdf:Description about="http://www.w3.org/Home/Lassila">
                <s:Creator> Ora Lassila </s:Creator>
        </rdf:Description>
</rdf:RDF>
```

L’elemento *RDF* definisce i confini in un documento XML degli statement RDF. L’elemento *Description* indica l’inizio della descrizione di una risorsa. L’attributo *about* (o *id*) indica la risorsa descritta, se entrambi mancanti si parla di risorsa anonima. All’interno della sezione Description vi è l’elenco delle proprietà della risorsa, che possono essere a loro volta delle risorse:

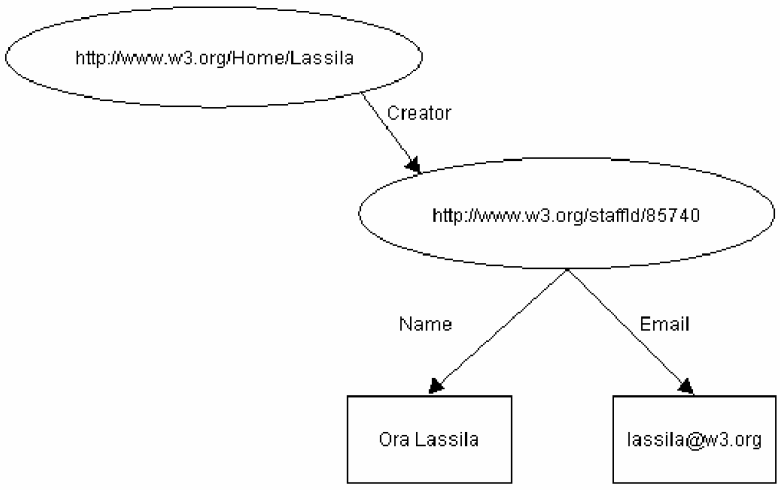

Figura 5 - RDF: Esempio 2

Questo grafico si legge così:

“La proprietà Creator della risorsa http://www.w3.org/Home/Lassila è Ora Lassila, che ha e-mail lassila@w3.org”.

Questo secondo caso diventa in sintassi estesa:

```
<rdf:RDF>
        <rdf:Description about="http://www.w3.org/Home/Lassila">
                <s:Creator rdf:resource="http://www.w3.org/staffId/85740"/>
        </rdf:Description>
        <rdf:Description about="http://www.w3.org/staffId/85740">
                <v:Name> Ora Lassila </v:Name>
                <v:Email> lassila@w3.org </v:Email>
        </rdf:Description>
</rdf:RDF>
```

Per rendere più chiara la relazione fra le due risorse, possiamo scrivere in maniera equivalente:

```
<rdf:RDF>
        <rdf:Description about="http://www.w3.org/Home/Lassila">
                <s:Creator>
                        <rdf:Description about="http://www.w3.org/staffId/85740">
                                <v:Name> Ora Lassila </v:Name>
                                <v:Email> lassila@w3.org </v:Email>
                        </rdf:Description>
                </s:Creator>
        </rdf:Description>
</rdf:RDF>
```

Quest’ultimo è un esempio di sintassi RDF abbreviata, la quale permette di scrivere con un XML più compatto, senza pregiudicare la comprensione degli statement da parte di un processore RDF.

1.4.3.2 Contenitori

A volte è importante fare riferimento ad un insieme di risorse (ad esempio, se un documento è stato creato da più autori, o se lo stesso autore ha fatto più di un documento, etc.). In questo caso tali risorse devono essere inserire all'interno di un contenitore che sarà l'oggetto dello statement. RDF definisce tre tipi di contenitori:

- Bag: E' un insieme di risorse con ripetizioni. L'ordine non è importante
- Sequence: E' un insieme di risorse con ripetizioni ed un ordine definito tra le risorse presenti
- Alternative: E' un insieme di risorse senza ripetizioni in cui può essere scelto uno solo degli elementi. Le risorse, cioè, possono essere scelte come alternative. L'ordine degli elementi può essere usato per esprimere preferenze

1.4.3.3 Statement su statement (Reification)

Oltre alle affermazioni, agli statement su risorse, RDF permette di esprimere anche citazioni, ovvero reificazioni, meta-affermazioni, vale a dire affermazioni su altre affermazioni, statement su altri statement. Ciò equivale a fornire meta-informazioni su una meta-informazione.

Esempio:

"Ralph Swick dice che Ora Lassila è il Creator della risorsa http://www.w3.org/Home/Lassila".

In questo modo non diciamo nulla sulla risorsa http://www.w3.org/Home/Lassila, esprimiamo, invece, un fatto sull'affermazione fatta da Ralph. Questo, in breve, significa attribuire la proprietà "dice" allo statement "Ora Lassila è il Creator della risorsa http://www.w3.org/Home/Lassila". Occorre pertanto considerare la meta-informazione come una risorsa da descrivere. Questa procedura si chiama reificazione (riduzione ad oggetto) dell'asserzione (o statement). Per esprimere questo in RDF, poiché gli statement sono fatti su risorse, dobbiamo creare una risorsa dello statement originale ("Ora Lassila è il Creator della risorsa http://www.w3.org/Home/Lassila") con quattro proprietà: subject, predicate, object e type. Dopo aver reificato l'asserzione, potrò esprimere ulteriori proprietà su di essa. La

reification è usata per esplicitare i vari statement contenuti in una sezione Description. Il meccanismo della reification è stato introdotto nelle specifiche di RDF(S) per aumentarne le capacità di descrizione, in quanto rende descrivibile in RDF qualsiasi elemento definito in RDF.

1.4.3.4 Tipizzazione

E' possibile assegnare ad ogni risorsa un tipo appartenente ad uno schema di meta-informazioni:

```
<rdf:Description about="http://www.w3.org/staffId/85740">
        <rdf:type rdf:resource="/myschema.rdf#Persona"/>
        <v:Name> Ora Lassila </v:Name>
        <v:Email> lassila@w3.org </v:Email>
</rdf:Description>
```

L'attributo *rdf:type* specifica l'URI della definizione del tipo.

1.4.4 RDF-Schema

Il modello di RDF non permette di effettuare la validazione di un valore o la restrizione di un dominio di applicazione di una proprietà. Questo compito è svolto da RDF-Schema. L'insieme delle proprietà RDF e la loro semantica sono definite in uno schema RDF, il quale rappresenta una sorta di vocabolario, in cui possono essere indicate anche i tipi di risorse che verranno descritte. RDF-Schema fornisce un meccanismo di base per un sistema di tipizzazione da utilizzare in modelli RDF, definendo un insieme di risorse RDF da usare per descrivere caratteristiche di altre risorse e proprietà RDF. A differenza di XML-Schema o di una DTD, RDF-Schema non vincola la struttura del documento, ma fornisce informazioni utili all'interpretazione del documento stesso. Lo schema è definito in termini di RDF stesso e si trova a un livello superiore rispetto RDF. In RDF ogni predicato è astratto, senza connessioni né riferimenti, senza relazione con altri predicati. RDF-Schema permette, invece, di esprimere relazioni e vincoli tra predicati, permette di segnalare l'esistenza di proprietà caratteristiche di un concetto, che esprimono in maniera organizzata e sistematica affermazioni simili su risorse simili. Esso esprime il significato comune dei termini utilizzati in uno statement, definisce i termini che vengono usati e il loro contenuto semantico. Lo schema è identificabile con

un’ontologia e consente la definizione di classi e risorse, delle proprietà caratteristiche di una classe, dei vincoli di dominio e di condominio (range) sulle proprietà, delle relazioni (comprese quelle di sottoclasse di una risorsa e sottotipo di una proprietà), dei vincoli e della documentazione. RDF-Schema somiglia ad un linguaggio di programmazione object-oriented, in quanto ha il concetto di classe e proprietà. A differenza di un linguaggio object-oriented, però, non definisce le classi e le proprietà che le istanze di tali classi hanno, ma definisce le proprietà e le classi alle quali si possono applicare (property-oriented). In questo modo si possono aggiungere nuove proprietà ad una classe senza doverla modificare. RDF-Schema può essere effettivamente pensato come linguaggio ontologico e/o di rappresentazione della conoscenza.

1.4.4.1 Classi

Quelle che seguono sono le descrizioni delle classi che costituiscono le fondamenta di RDF-Schema:

rdfs:Resource	Tutto quello che è descritto attraverso statement RDF è chiamato resource, istanza della classe rdfs:Resource.
rdf:Property	Un sottoinsieme di risorse RDF che sono proprietà. E’ sottoclasse di rdfs:Resource.
rdfs:Class	E’ un concetto simile a quello di classe di un qualsiasi linguaggio object-oriented, come Java. Le classi RDF possono essere definite per rappresentare pagine Web, persone, tipi di documento, database o concetti astratti. Quando viene definita una nuova classe, la risorsa che la rappresenta deve avere la proprietà rdf:type impostata a rdfs:Class.
rdfs:Literal	Corrisponde all’insieme dei Literals (valori atomici), le stringhe ad esempio. E’ sottoclasse di rdfs:Resource.

La gerarchia delle classi RDF-Schema è la seguente:

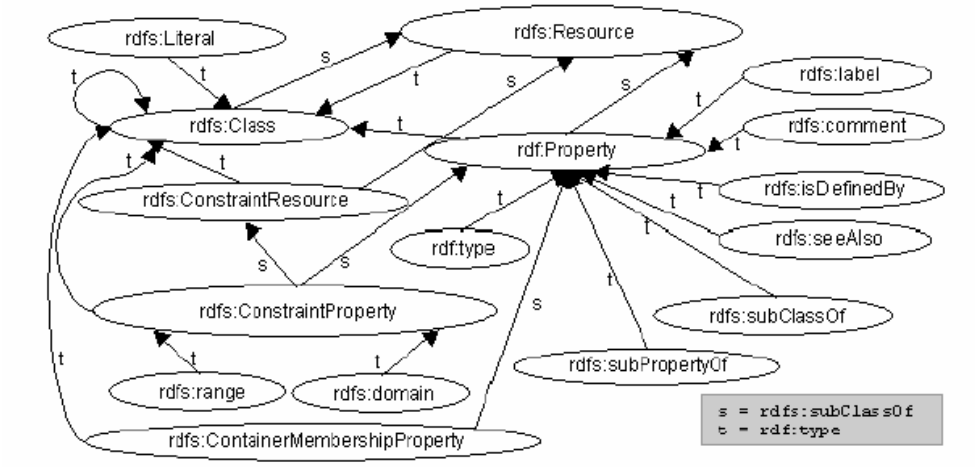

Figura 6 - RDF-Schema: Gerarchia della classi

1.4.4.2 Proprietà

Le proprietà sono usate per definire delle relazioni fra classi o con superclassi:

rdf:type	Introduce una relazione di tipo *is-member of*. Indica l'appartenenza di una risorsa ad una certa classe, quindi ha tutte le caratteristiche che sono previste per un membro di tale classe. La risorsa è un'istanza della classe. Il valore di rdf:type deve essere una risorsa di tipo rdfs:Class. Una risorsa può essere istanza di più di una classe.
rdfs:subClassOf	Introduce una relazione di inclusione tra classi *is-a*. Specifica la relazione di ereditarietà fra classi. Questa proprietà può essere assegnata solo a istanze di rdfs:Class. Una classe può essere sottoclasse di una o più classi (ereditarietà multipla), cioè può avere più di una superclasse. Una classe non può essere sottoclasse di se stessa né di una sua sottoclasse. E' transitiva.

rdfs:subPropertyOf	E' una proprietà istanza di rdf:Property. Indica che una proprietà è la specializzazione di un'altra. Una proprietà non può essere sottoproprietà di se stessa né di una sua sottoproprietà. Ogni proprietà può essere la specializzazione di zero o più proprietà.
rdfs:seeAlso	Indica una risorsa che può fornire ulteriori informazioni sulla risorsa in cui è presente, sul soggetto dell'asserzione.
rdfs:isDefinedBy	E' una sottoproprietà di rdfs:seeAlso. Indica la risorsa che definisce la risorsa in cui è usata, il soggetto di un'asserzione.

1.4.4.3 Vincoli

I predicati più utilizzati per esprimere vincoli su altre proprietà sono i seguenti.

rdfs:ConstraintResource	E' una sottoclasse di rdfs:Resource, le cui istanze sono primitive di RDF-Schema coinvolte nell'espressione di vincoli. Questa risorsa rappresenta un meccanismo per permettere l'introduzione di nuovi vincoli nelle future versioni di RDF senza perdere la compatibilità con le vecchie versioni.
rdfs:ConstraintProperty	E' una sottoclasse di rdf:Property, le sue istanze sono proprietà che indicano dei vincoli. E' una sottoclasse di rdfs:ConstraintResource. Rdfs:domain e rdfs:range sono istanze di rdfs:ConstraintProperty.
rdfs:domain (dominio)	E' un'istanza di rdfs:ConstraintProperty. Viene usata per dichiarare che tutte le risorse che hanno la proprietà sono istanze di una o più classi. Indica le classi che possono avere tale proprietà. La terna P rdfs:domain C dichiara che P è una istanza della classe rdf:Property, che C è una istanza della classe rdfs:Class e che la risorsa denotata dal soggetto della terna che ha per predicato P è una istanza della classe C. Una proprietà può avere zero uno o più domain.

	Zero domain significa che la proprietà può essere usata in qualsiasi risorsa, un solo domain che la proprietà può essere usata solo nelle istanze di una classe, più di un domain significa che la proprietà può essere usata in una qualunque delle classi indicate nei vari domain. Il range di rdfs:domain è la classe rdfs:Class. Questo significa che tutte le risorse che sono valore della proprietà rdfs:domain è una istanza di rdfs:Class. Il domain di rdfs:domain è la classe rdf:Property. Questo significa che tutte le risorse con una proprietà rdfs:domain sono istanze di rdf:Property. La proprietà rdfs:domain può essere applicata a se stessa.
rdfs:range (codominio)	E' un'istanza di rdfs:ConstraintProperty. Viene usata per dichiarare che il valore di una proprietà è un'istanza di una classe. La terna P rdfs:range C dichiara che P è una istanza della classe rdf:Property, che C è una istanza della classe rdf:Class e che le risorse denotate dalle terne che hanno come oggetto P sono delle istanze di C. Una proprietà può avere al massimo una proprietà range. Il range di rdfs:range sono delle istanze di rdfs:Class. Questo significa che tutte le risorse che sono valori della proprietà rdfs:range sono istanze della classe rdfs:Class. Il domain di rdfs:range è la classe rdf:Property. Questo significa che tutte le risorse che sono valori della proprietà rdfs:range sono istanze della classe rdf:Property. La proprietà rdf:range può essere applicata a se stessa.

1.4.4.4 Documentazione

rdfs:comment	Fornisce una descrizione di una risorsa.
rdfs:label	Fornisce una versione comprensibile dalle persone del nome di una risorsa.

1.4.4.5 XML e RDF(S)

Possiamo affermare che XML e RDF(S) nascono entrambi per facilitare la condivisione delle informazioni sul Web, ma hanno caratteristiche complementari.

XML:

- XML è un ottimo veicolo per lo scambio di metadati sul Web
- XML-Schema fornisce una serie di primitive per vincolare la struttura di un documento e definire il numero di occorrenze e il tipo dei dati
- XML è carente per quanto riguarda la scalabilità
- XML-Schema non fornisce primitive per specificare la semantica dei concetti espressi

RDF(S):

- Fornisce alcune primitive per definire la semantica dei concetti espressi (Class, Property, subClassOf, subPropertyOf, type, etc.), permette, ad esempio, di definire gerarchie (che non sono previste in XML)
- Ha come caratteristica fondamentale la scalabilità
- Non vincola la struttura del documento, ma fornisce informazioni utili all'interpretazione del documento stesso

Anche il modello dei dati utilizzato è diverso.

- XML è basato su un modello ad albero
- RDF è basato su un modello a grafo orientato ed etichettato (Direct Labeled Graph), simile al relazionale, con risorse (entità) e proprietà (attributi)

Sebbene XML sia ineguagliabile come formato di scambio sul Web (per cui il suo utilizzo congiunto a RDF per l'interscambio dei metadati è necessario) e RDF(S) rappresenti un buon strumento per la definizione di un linguaggio di mark-up per il Semantic Web (ad esempio, per determinare le relazioni semantiche tra termini differenti), questi due strumenti non sono sufficienti. RDF(S) non consente di specificare le proprietà delle proprietà e le condizioni necessarie e sufficienti per l'appartenenza alle classi. Gli unici vincoli che permette di definire sono quelli di dominio e range delle proprietà. Non permette, inoltre, di specificare meccanismi

di ragionamento, ma rappresenta semplicemente un sistema a frame. C'è quindi bisogno di meccanismi di ragionamento definiti a un livello superiore, come, ad esempio, in DAML+OIL.

1.4.5 DAML+OIL

DAML+OIL è un linguaggio ontologico standard che consente la rappresentazione delle informazioni del Web in modo che il loro significato sia comprensibile alle macchine, un linguaggio di mark-up semantico per risorse Web, sviluppato da DARPA, di approccio object-oriented, che fornisce i costrutti per la creazione di ontologie ed è stato progettato per descrivere la struttura di un dominio. Il livello ontologico prevede la possibilità di trarre conclusioni dalle affermazioni. Diventa quindi necessario un linguaggio per esprimere inferenze, ovvero la creazione di informazioni nuove attraverso la manipolazione automatica di informazioni già acquisite. DAML+OIL precisa l'esistenza di relazioni tra proprietà. Ad esempio, permette di dire che A e B sono l'una l'inverso dell'altra, o che C e D sono equivalenti. Inoltre è necessario aggiungere un po' di ulteriori concetti (sostanzialmente i quantificatori universali ed esistenziali) per arrivare veramente ad un linguaggio dei predicati del primo ordine che permetta di esprimere vere e proprie inferenze sulle affermazioni RDF.

DAML+OIL si basa su altre discipline:

- Linguaggi basati sui frame: modellano la realtà in classi (frame), ognuna delle quali ha delle proprietà (slot). Le classi sono legate dalle relazioni di sottoclasse e superclasse. DAML+OIL si basa su questi concetti, consentendo la definizione di classi, delle loro sottoclassi e delle loro proprietà.

- Description Logics (DLs) [Paragrafo 1.2]: è un linguaggio, elaborato nella ricerca sulle rappresentazioni della conoscenza, che descrive la conoscenza in termini di concetti (paragonabili ai frame) e ruoli (paragonabili agli slot). DAML+OIL sfrutta questo supporto per i ragionamenti.

- XML e RDF: la sintassi di DAML+OIL è quella di XML e RDF. Inoltre, lo schema RDF fornisce un insieme di primitive (come la relazione di sottoclasse) e le regole sintattiche per definire le gerarchie (non previste in XML). Una base di conoscenza in DAML+OIL è un insieme di triple RDF.

DAML+OIL, con un approccio object-oriented, struttura il dominio in classi e proprietà, conformemente alle specifiche memorizzate in un'ontologia. L'ontologia è un insieme di assiomi che stabiliscono la classificazione e dichiarano le relazioni tra classi e proprietà. Se una risorsa è un'istanza di classe C, allora è di tipo C.

1.4.5.1 La definizione di un'ontologia

La definizione di un'ontologia è racchiusa fra i tag:

```
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:daml="http://www.daml.org/2001/03/daml+oil-ex#"
        xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">
        ...
</rdf:RDF>
```

Nel tag di apertura vengono specificati i Namespace ed i prefissi con cui, nel documento, si fa riferimento agli elementi in essi contenuti.

L'ontologia DAML+OIL è formata da:

- Zero o più header/intestazioni, con informazioni a scopo di documentazione. Ognuno contiene la versione del documento e le ontologie importate.

  Ad esempio:

```
<daml:ontology rdf:about="">
        <daml:versionInfo>
                $Id: daml-ex.daml,v. 1.0 2002/05/22 18:04:20 mdean Exp $
        </daml:versionInfo>
        <rdfs:comment>
                Esempio di ontologia
        </rdfs:comment>
        <daml:imports
                rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
```

</daml:ontology>

- definizione di concetti/classi/frame e proprietà/slot

1.4.5.2 Definizione di classi

La definizione di una classe è introdotta da daml:**Class**.

Esempio:

```
<daml:Class rdf:ID="Person" >
        <rdfs:label> Person </rdfs:label>
        <rdfs:comment> Classe delle persone </rdfs:comment>
</daml:Class>
```

Riassumiamo le componenti della definizione di una classe, che la specificano ulteriormente:

rdfs:**subClassOf**	La classe è sottoclasse di un'altra, rappresentata come rdf:**resource**. Definisce la classe C, sottoclasse della class-expression definita. A differenza delle specifiche di RDF(S), DAML+OIL non vieta la ciclicità che può derivare da tale relazione.
daml:**disjointWith**	La classe non ha istanze in comune con un'altra specificata.
daml:**disjointUnionOf**	La classe è l'unione di altre, che sono a due a due disgiunte tra di loro.
daml:**sameClassAs**	La classe è equivalente a un'altra, ha le stesse istanze.
daml:**equivalentTo**	Equivalente a daml:sameClassAs.

Una class-expression può indicare:

- Il nome di una classe
- Un elenco di istanze: la classe è definita elencandone gli elementi, le istanze che ne fanno parte (daml:oneOf)
- Una restrizione di una proprietà
- Una combinazione booleana di class-expressions: la classe deve essere uguale alla classe derivante dalla combinazione di due o più class-expressions

Esempi:

- La classe degli uomini, sottoclasse delle persone:

```
<daml:Class rdf:ID="Man" >
        <rdfs:subClassOf rdf:resource="#Person"/>
</daml:Class>
```

- La classe delle donne, sottoclasse delle persone e disgiunta dalla classe degli uomini:

```
<daml:Class rdf:ID="Woman" >
        <rdfs:subClassOf rdf:resource="#Person"/>
        <daml:disjointWith rdf:resource="#Man"/>
</daml:Class>
```

Combinazioni di classi:

daml:**intersectionOf** <lista di class-expression>	La classe è ottenuta dall'intersezione di altre classi o restrizioni di esse. Definisce la classe degli oggetti in comune alle class-expressions della lista. Congiunzione logica
daml:**unionOf** <lista di class-expression>	La classe è ottenuta dall'unione di altre classi o restrizioni di esse. Definisce la classe i cui membri sono l'unione degli oggetti appartenenti alle class-expressions della lista.

	Disgiunzione logica
daml:**complementOf** <class-expression>	La classe è complementare di un'altra. Definisce la classe degli oggetti che non appartengono alla class-expression. Negazione logica limitata ai soli oggetti

1.4.5.3 Vincoli sulle proprietà

Una restrizione di una proprietà definisce implicitamente una classe anonima, la classe delle istanze che ne soddisfano i vincoli. I vincoli possono riguardare proprietà che mettono in relazione due classi o che mettono in relazione una classe e un tipo di dato XML-Schema. La classe è ottenuta facendo una restrizione su un'altra in base ad una specifica proprietà.
Un elemento di tipo daml:**Restriction** deve indicare:

- la proprietà daml:**onProperty**, alla quale va applicato il vincolo
- il tipo di restrizione, il vincolo sulla proprietà, che può essere uno di quelli elencati nella tabella sotto

Tipi di restrizione:

daml:**toClass** <class-expression>	La classe è formata da tutti gli elementi il cui valore della proprietà appartiene solo ad un'altra classe specificata. Definisce la classe degli oggetti x per i quali se (x,y) è un'istanza della proprietà, allora y è un'istanza della class-expression indicata da daml:toClass. Quantificatore universale dei predicati logici
daml:**hasClass** <class-expression>	La classe è formata da tutti gli elementi che hanno almeno un valore della proprietà appartenente ad un'altra classe. Definisce la classe degli oggetti x per i quali esiste almeno un'istanza y della class-expression tale che (x,y) è un'istanza

	della proprietà. Questo non esclude che possa esistere un'istanza (x,y1) della proprietà, con y1 non appartenente alla class-expression. Quantificatore esistenziale dei predicati logici
daml:**hasValue** <istanza o valore di un tipo di dato>	La classe è formata dagli oggetti che hanno come valore della proprietà un oggetto o un valore specificato. Se y è l'istanza, definisce la classe degli oggetti x per i quali (x,y) è un'istanza della proprietà.

Elenchiamo ora i vincoli di cardinalità:

daml:**cardinality** <intero non negativo> N	Definisce la classe degli oggetti che hanno esattamente N distinti valori per la proprietà a cui si applica la restrizione.
daml:**minCardinality** <intero non negativo> N	Definisce la classe degli oggetti che hanno almeno N distinti valori per la proprietà a cui si applica la restrizione.
daml:**maxCardinality** <intero non negativo> N	Definisce la classe degli oggetti che hanno al massimo N distinti valori per la proprietà a cui si applica la restrizione.
daml:**cardinalityQ** <intero non negativo> N	Definisce la classe degli oggetti che hanno esattamente N distinti valori per la proprietà a cui si applica la restrizione. Tali valori sono istanze della classe o del tipo di dato indicato nell'elemento daml:hasclassQ.
daml:**minCardinalityQ** <intero non negativo> N	Definisce la classe degli oggetti che hanno almeno N distinti valori per la proprietà a cui si applica la restrizione. Tali valori sono istanze della classe o del tipo di dato indicato nell'elemento daml:hasclassQ.
daml:**maxCardinalityQ** <intero non negativo> N	Definisce la classe degli oggetti che hanno al massimo N distinti valori per la proprietà a cui si applica la restrizione. Tali valori sono istanze della classe o del tipo di dato indicato nell'elemento daml:hasclassQ.

Esempi:

- La classe delle donne che hanno esattamente 3 figli:

```
<daml:Class rdf:about="#Woman">
        <rdfs:subClassOf>
                <daml:Restriction daml:cardinality="3">
                        <daml:onProperty rdf:resource="#hasSons"/>
                </daml:Restriction>
        </rdfs:subClassOf>
</daml:Class>
```

- La classe delle donne che hanno almeno un figlio maschio:

```
<daml:Class rdf:about="#Woman">
        <rdfs:subClassOf>
                <daml:Restriction daml:cardinalityQ="1">
                        <daml:onProperty rdf:resource="#hasSons"/>
                        <daml:hasClass rdf:resource="#Man"/>
                </daml:Restriction>
        </rdfs:subClassOf>
</daml:Class>
```

1.4.5.4 Proprietà

Le proprietà definiscono delle relazioni binarie fra due entità.

Esistono due casi:

daml:**ObjectProperty** rdf:ID=”nome-della-proprietà”	Tra due classi La proprietà lega un oggetto con un altro oggetto.
daml:**DataTypeProperty** rdf:ID=”nome-della-proprietà”	Tra una classe ed un tipo di dato XML-Schema La proprietà lega un oggetto con un valore di un certo tipo specificato.

Gli elementi per la definizione di una proprietà P sono:

rdfs:**subPropertyOf** <nome di una proprietà>	La proprietà è una sottoproprietà di un'altra indicata. Se (x,y) e un'istanza di P allora lo è anche della proprietà indicata da subPropertyOf.
rdfs:**domain** <class-expression>	Dominio della proprietà, la classe alle cui istanze si può applicare. Se (x,y) è un'istanza di P, allora x è un'istanza della class-expression. La presenza di più di un rdfs:domain definisce il dominio di P come l'intersezione di ogni class-expression, diversamente dalle specifiche di RDF(S).
rdfs:**range** <class-expression>	Insieme dei valori che la proprietà può assumere, che sono istanze di una classe indicata. Se (x,y) è un'istanza di P, allora y è un'istanza della class-expression. La presenza di più di un rdfs:range definisce il range di P come l'intersezione di ogni class-expression. Anche in questo caso si è andato contro le specifiche di RDF(S), che permettono l'indicazione di un solo range.
daml:**samePropertyAs** <nome di una proprietà>	La proprietà è equivalente ad un'altra indicata.
daml:**equivalentTo**	Equivalente a daml:samePropertyAs.
daml:**inverseOf** <nome di una proprietà>	La proprietà è l'inverso di un'altra. Se (x,y) è istanza di P allora (y,x) è istanza della proprietà indicata.
daml:**TransitiveProperty** rdf:ID="nome-della-proprietà"	Sottoclasse di ObjectProperty. La proprietà è transitiva. Se (x,y) $\in$ P e (y,z) $\in$ P allora (x,z) $\in$ P
daml:**UniqueProperty** rdf:ID="nome-della-proprietà"	Ogni istanza deve avere un unico valore per questa proprietà. Restrizione della cardinalità: ogni oggetto identifica univocamente il valore della sua proprietà. Se (x,y1) $\in$ P e (x,y2) $\in$ P allora y1=y2
daml:**UnambigousProperty** rdf:ID="nome-della-proprietà"	Due istanze diverse non possono avere lo stesso valore della proprietà. Restrizione della cardinalità: il valore della proprietà

	identifica l'oggetto che caratterizza. Se (x1,y) ∈ P e (x2,y) ∈ P allora x1=x2
	daml:UniqueProperty e daml:UnambiguosProperty definiscono vincoli globali per una proprietà, indipendenti dalla classe a cui la proprietà è applicata.

Esempi:

- La proprietà di avere un'età:

```
<daml:DataTypeProperty rdf:ID="age">
        <rdf:type
                rdf:resource="http://www.daml.org/2001/03/
                        daml+oil#UniqueProperty"/>
        <rdfs:range
                rdf:resource="http://www.w3.org/2000/10/
                        XMLSchema#nonNegativeInteger"/>
</daml:DataTypeProperty>
```

- La proprietà di una persona di avere un genitore:

```
<daml:ObjectProperty rdf:ID="hasParent">
        <rdfs:domain rdf:resource="#Person"/>
        <rdfs:range rdf:resource="#Person"/>
</daml:ObjectProperty>
```

- La proprietà di una persona di avere una madre, che è sottoproprietà dell'avere un genitore:

```
<daml:ObjectProperty rdf:ID="hasMother">
        <rdfs:subPropertyOf rdf:resource="#hasParent"/>
        <rdfs:range rdf:resource="#Woman"/>
</daml:ObjectProperty>
```

- La proprietà di una persona di avere figli:

```
<daml:ObjectProperty rdf:ID="hasSons">
        <daml:inverseOf rdf:resource="#hasParent"/>
</daml:ObjectProperty>
```

1.4.5.5 Tipi di dati

Un elemento chiave di DAML+OIL è il supporto per i tipi di dato XML-Schema, supporto non presente in RFD(S). La soluzione avanzata in DAML+OIL vuole essere un'idea per il W3C per includerlo nelle specifiche di RDF(S).

Questa caratteristica è facilitata dalla separazione tra:

- istanze delle classi e
- istanze dei tipi di dato

In particolare, si mantengono separati i domini di interpretazione, in modo che l'istanza di una classe non sia mai interpretata come il valore di un tipo e l'insieme delle proprietà degli oggetti sia separato da quello delle proprietà dei tipi di dato:

- Istanza di una classe / valore di un tipo
- Proprietà degli oggetti / proprietà dei tipi.

In questo modo il sistema dei tipi può essere esteso, senza che questo coinvolga il linguaggio dell'ontologia e viceversa. Infatti, il linguaggio dell'ontologia può specificare dei vincoli sui valori dei dati, ma questi, come non possono essere istanze di classi, non possono applicare vincoli addizionali agli elementi del dominio degli oggetti. Da un punto di vista pratico, le implementazioni di DAML+OIL possono scegliere se supportare oppure no tutti i tipi di dati dello schema XML. Per fare questo, esse devono implementare i meccanismi di controllo e validazione, oppure contare su componenti esterne.

1.4.5.6 Istanze/oggetti

Una volta definita l'ontologia, si possono specificare le istanze/gli oggetti, utilizzando la sintassi RDF. Le istanze sono descritte sia all'interno di un'ontologia, sia all'esterno. L'istanza può

anche essere utilizzata per descrivere una risorsa Web. Nel primo caso, la specifica di un individuo si ottiene assegnando un identificativo all'oggetto e attribuendo i valori alle proprietà che caratterizzano la classe a cui l'oggetto appartiene.

Ad esempio, una persona di nome "Patrizia" è indicata:

```
<Person rdf:ID="Patrizia">
        <rdfs:label> Patrizia </rdfs:label>
        <rdfs:comment> Patrizia e' un'istanza della classe delle Persone </rdfs:comment>
        <age> <xsd:integer rdf:value="26"/> </age>
        ...
</Person>
```

L'indicazione del valore numerico deve essere preceduta dalla specificazione del tipo.

1.4.5.7 Costruttori e assiomi

Il potere espressivo di DAML+OIL è stimato in base a due aspetti: i costruttori e i tipi di assiomi supportatati.

I costruttori:

I costruttori permettono la costruzione di classi intese come espressioni, cioè come il prodotto di un certo numero di operatori. Partendo da qualcosa che esiste già, si costruisce qualcosa di nuovo. La complessità con cui questi costruttori possono essere combinati è arbitraria e i tipi di dati primitivi, definiti nello schema XML (stringhe, interi, float), possono essere usati in qualunque punto richieda il nome di una classe.

Sono costruttori:

- intersectionOf
- unionOf
- complementOf
- oneOf
- toClass

- hasClass
- hasValue
- cardinalityQ
- minCardinalityQ
- maxCardinalityQ

Gli assiomi:

Gli assiomi migliorano l'espressività del linguaggio permettendo la dichiarazione di relazioni di classificazione o equivalenza tra classi e/o proprietà, la disgiunzione tra classi, l'equivalenza o meno di oggetti diversi, le proprietà delle proprietà. Non creano qualcosa di nuovo, ma stabiliscono delle proprietà su classi già costruite.

Sono assiomi:

- subClassOf
- sameClassAs
- subPropertyOf
- samePropertyAs
- disjointWith
- sameIndividualsAs
- differentIndividualsFrom
- inverseOf
- transitiveProperty
- uniqueProperty
- unambiguousProperty

I primi due dell'elenco possono essere applicati ad espressioni di classe arbitrarie. Ciò garantisce un incremento del potere espressivo rispetto ai linguaggi basati su frame, in cui questi assiomi dovevano avere necessariamente una forma in cui la parte sinistra era un nome, in cui ad ogni assioma era associato un solo nome e in cui non erano permessi cicli (cioè la classe a destra non poteva fare riferimenti, diretti o indiretti, al nome della classe del lato sinistro).

2. L’integrazione delle Informazioni

2.1 Metodi di integrazione

La crescita delle sorgenti di dati, sia all'interno di un’azienda che sulla rete, ha reso possibile l'accesso ad una quantità molto ampia di informazioni. La probabilità di reperire un dato di interesse è di conseguenza aumentata vertiginosamente ma, allo stesso tempo, bisogna registrare una diminuzione altrettanto importante della probabilità di venirne in possesso nei tempi e soprattutto nei modi desiderati. Questo principalmente a causa della grande eterogeneità dei dati disponibili, sia per quanto riguarda la natura (testi, immagini, etc.), sia il modo in cui vengono descritti i diversi tipi di sorgenti, che possono essere semplici pagine HTML, DBMS relazionali o ad oggetti, file system, etc. Gli standard esistenti (TCP/IP, ODBC, CORBA, SQL, etc.) risolvono solo parzialmente i problemi relativi alle diversità hardware e software dei protocolli di rete e di comunicazione tra i moduli. Rimangono invece irrisolti i problemi relativi alla modellazione delle informazioni. I modelli dei dati e gli schemi, infatti, si differenziano tra loro per quanto riguarda la definizione di una struttura logica per i numerosi generi di dati da immagazzinare e questo crea una eterogeneità semantica (o logica) non risolvibile da questi standard. Un'importante area, sia di ricerca che di applicazione, riguarda proprio l'integrazione di database (basi di dati) e di datawarehouse (magazzini di dati) eterogenei. Questi lavori studiano la possibilità di materializzare presso l'utente finale delle viste, ovvero delle porzioni di sorgenti, replicando fisicamente i dati. Ciò avviene affidandosi a complicati algoritmi di mantenimento al fine di garantire la consistenza dei dati trattati a fronte di modifiche nelle sorgenti originali e di rendere possibile l'accesso ad una quantità molto ampia di informazioni.
L’approccio all’integrazione può essere di due tipi:

- **Strutturale** (o **sintattico**): è quello tradizionale, tuttora dominante tra i sistemi presenti sul mercato, caratterizzato dal fatto di usare un “self-describing model” per rappresentare gli oggetti da integrare, limitando l'uso delle informazioni semantiche alle regole predefinite dall'operatore. In pratica, il sistema non conosce a priori la semantica di un oggetto che va a recuperare da una sorgente (e dunque di questa non possiede alcuno schema descrittivo), bensì è l'oggetto stesso che, attraverso delle etichette, si autodescrive, specificando tutte le volte, per ogni suo singolo campo, il significato che ad esso è associato.

- **Semantico**: è quello seguito dal sistema I3. L'integrazione, in questo caso, si avvale di un Mediatore, "un modulo software che sfrutta la conoscenza su un certo insieme di dati per creare informazioni utilizzabili da un'applicazione di livello superiore" [23]. Esso deve conoscere, per ogni sorgente, lo schema concettuale (i metadati). Le informazioni semantiche sono codificate in questi schemi. Deve essere inoltre disponibile un modello comune per descrivere le informazioni da condividere (e dunque per descrivere anche i metadati). Infine, deve essere possibile una integrazione (parziale o totale) delle sorgenti di dati. In questo modo, sfruttando opportunamente le informazioni semantiche che necessariamente ogni schema sottintende, il mediatore può individuare concetti comuni a più sorgenti e le relazioni che li legano.

2.1.1 Il sistema I_2

I sistemi I2 (Integration of Information) sono i sistemi che, basandosi sulle descrizioni dei dati, combinano tra loro informazioni provenienti da diverse sorgenti senza dover ricorrere alla duplicazione fisica. L'approccio all'integrazione è *virtuale*, consiste cioè nel non replicare i dati, ma nell'eseguire a run-time la decomposizione della query globale, l'interrogazione delle sorgenti e la fusione delle informazioni.

2.1.2 Il sistema I_3

Il programma I3 (Intelligent Integration of Information) è stato sviluppato dall'ARPA (Advanced Research Projects Agency), agenzia che fa capo al Dipartimento di Difesa americano [24], e costituisce il frutto di un'ambiziosa ricerca finalizzata ad indicare un'architettura di riferimento che realizzi l'integrazione di sorgenti eterogenee in maniera automatica. Si parla di Integrazione Intelligente delle Informazioni quando l'integrazione delle informazioni fa uso di tecniche di Intelligenza Artificiale (AI) [3]. Al contrario di quanto accade con l'uso di tecniche tradizionali (che si limitano ad una semplice aggregazione), questa forma di integrazione ha lo scopo di accrescere il valore delle informazioni gestite, ottenendone anche di nuove (derivandole dai dati utilizzati). Per ottenere risultati selezionati sono richieste conoscenza ed intelligenza volte alla scelta delle sorgenti e dei dati, alla loro fusione e alla loro sintesi. Le tecniche sviluppate

dall'Intelligenza Artificiale, potendo dedurre informazioni utili dagli schemi delle sorgenti, diventano uno strumento prezioso per la costruzione automatica di soluzioni integrate flessibili e riusabili. Progettare la costruzione di supersistemi ad hoc che interessino una grande quantità di sorgenti non correlate semanticamente risulta molto complesso ed il risultato ottenuto è scarsamente manutenibile o adattabile, strettamente finalizzato alla risoluzione dei problemi per cui è stato implementato. L'introduzione di architetture modulari sviluppabili secondo i principi proposti da uno standard cerca invece di abbassare i costi di sviluppo e di mantenimento. Per la riusabilità è fondamentale l'esistenza di interfacce ed architetture standard. Il paradigma suggerito per la suddivisione dei servizi e delle risorse nei diversi moduli, si articola su due dimensioni:

- Orizzontalmente:
    - livello utente
    - livello intermedio che fa uso di tecniche di IA
    - livello delle sorgenti di dati
- Verticalmente:
    - diversi domini in cui raggruppare le sorgenti

I domini dei vari livelli non sono strettamente connessi, ma si scambiano dati ed informazioni la cui combinazione avviene a livello dell'utilizzatore, riducendo la complessità totale del sistema e permettendo lo sviluppo di applicazioni con finalità diverse. Il sistema I3 si concentra sul livello intermedio, che media tra gli utenti e le sorgenti. In questo livello sono presenti vari moduli, tra i quali evidenziamo:

Facilitator e Mediatore (le differenze tra i due sono sottili ed ancora ambigue in letteratura): ricercano le fonti interessanti e combinano i dati da esse ricevuti.

Query Processor: riformula le query aumentando le loro probabilità di successo.

Data Miner: analizza i dati per estrarre informazioni intensionali implicite.

Due sono le ipotesi che rappresentano la base del progetto, che rendono il processo di integrazione molto complesso. La prima è connessa con la difficoltà che si ha nel ricercare delle informazioni all'interno della molteplicità delle sorgenti di informazioni. La seconda è legata al

fatto che le fonti di informazione e i sistemi informativi, pur essendo spesso semanticamente correlati tra di loro, non lo sono in una forma semplice né preordinata. Pertanto è necessario proporre una architettura di riferimento per i sistemi I3 che rappresenti alcuni dei servizi che un integratore di informazioni deve contenere e le possibili interconnessioni fra di essi. Tale architettura è riportata in figura 7 e prevede cinque famiglie di servizi:

Servizi di Coordinamento: sono quei servizi di alto livello che permettono l'individuazione delle sorgenti di dati interessanti, ovvero che probabilmente possono dare risposta ad una determinata richiesta dell'utente.

Servizi di Amministrazione: sono servizi usati dai Servizi di Coordinamento per localizzare le sorgenti utili, per determinare le loro capacità, e per creare ed interpretare Template, strutture dati che descrivono i servizi, le fonti ed i moduli da utilizzare per portare a termine un determinato task. Fanno parte di questa categoria di servizi il Browsing, con cui si permette all'utente di "navigare" attraverso le descrizioni degli schemi delle sorgenti, recuperando informazioni su queste. Il servizio si basa sulla premessa che queste descrizioni siano fornite esplicitamente tramite un linguaggio dichiarativo leggibile e comprensibile all'utente.

Servizi di Integrazione e Trasformazione Semantica: sono il cuore dell'architettura I3, i servizi che supportano le manipolazioni semantiche necessarie per l'integrazione e la trasformazione delle informazioni. Essi si occupano dell'integrazione vera e propria delle informazioni, cercando di risolvere le differenze tra gli schemi, i modelli e i tipi di dati. Il tipico input per questi servizi è rappresentato da una o più sorgenti di dati, e l'output è la vista integrata o trasformata di queste informazioni. Tra questi servizi si distinguono quelli relativi alla trasformazione degli schemi (ovvero di tutto ciò che va sotto il nome di metadati) e quelli relativi alla trasformazione dei dati. Sono spesso indicati come servizi di Mediazione, essendo tipici dei moduli mediatori, e includono:

- Servizi di integrazione degli schemi, che supportano la trasformazione e l'integrazione degli schemi e delle conoscenze derivanti da fonti di dati eterogenee. Ne fanno parte i servizi di trasformazione dei vocaboli e dell'ontologia, usati per arrivare alla definizione di un'ontologia unica che combini gli aspetti comuni alle singole ontologie usate nelle diverse fonti.

- Servizi di integrazione delle informazioni, che provvedono alla traduzione dei termini da un contesto all'altro, ovvero dall'ontologia di partenza a quella di destinazione.

- Servizi di supporto al processo di integrazione, utilizzati nel momento in cui una query è scomposta in molte subquery da inviare a fonti differenti e nel momento in cui i risultati provenienti dalle singole subquery devono essere integrati.

Servizi di Wrapping: sono utilizzati per fare sì che le fonti di informazioni aderiscano ad uno standard. Si comportano come traduttori dai sistemi locali ai servizi di alto livello dell'integratore e viceversa quando si interroga la sorgente di dati. In particolare, si prefiggono di permettere ai servizi di coordinamento e di mediazione di manipolare in modo uniforme il numero maggiore di sorgenti locali (anche se queste non sono state esplicitamente pensate come facenti parte del sistema di integrazione) e di essere il più riusabili possibile (per fare ciò, dovrebbero fornire interfacce che seguano gli standard più diffusi). In pratica, compito di un wrapper è modificare l'interfaccia, i dati ed il comportamento di una sorgente per facilitarne la comunicazione con il mondo esterno. Il vero obiettivo è quindi standardizzare il processo di wrapping delle sorgenti, permettendo la creazione di una libreria di fonti accessibili. Inoltre, il processo stesso di realizzazione di un wrapper dovrebbe essere standardizzato, in modo da poter essere riutilizzato da altre fonti.

Servizi Ausiliari: aumentano le funzionalità degli altri servizi descritti precedentemente. Sono prevalentemente utilizzati dai moduli che agiscono direttamente sulle informazioni. Vanno dai semplici servizi di monitoraggio del sistema, ai servizi di propagazione degli aggiornamenti e di ottimizzazione.

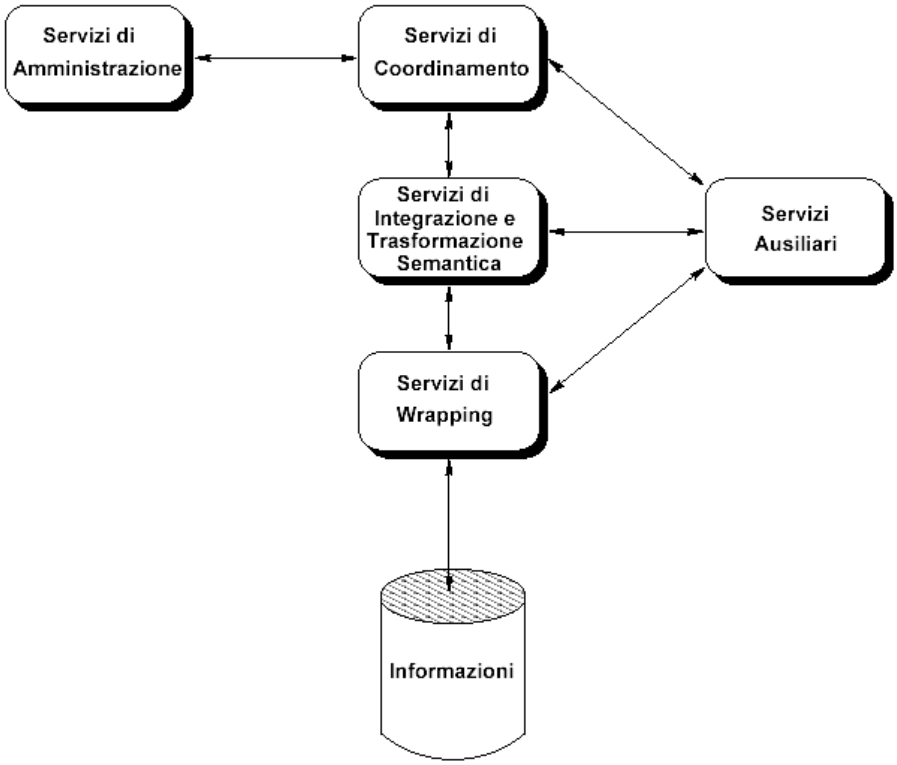

Figura 7 - Diagramma dei servizi I_3

2.2 Problemi

Il concetto su cui si basa l'integrazione semantica di sorgenti, autonome nel fine ed eterogenee nella struttura, riguarda la sovrapposizione dei relativi domini, che si traduce in corrispondenze fra gli schemi sorgenti. I problemi legati al processo di individuazione delle similitudini tra questi schemi, che complicano la generazione di un sistema di integrazione di informazioni, sono essenzialmente:

- problemi ontologici
- problemi semantici

2.2.1 Problemi ontologici (Ontologie)

Il termine *ontologia* proviene dal linguaggio filosofico, in particolare da Aristotele, dove indica la "systematic explanation of Existence", cioè la teoria che studia quali tipi di cose esistono [25]. Nell'ambito dell'Intelligenza Artificiale esso indica "la descrizione particolareggiata di una concettualizzazione", cioè "l'insieme dei termini e delle relazioni, usati in un particolare dominio applicativo, che denotano in modo univoco una particolare conoscenza e fra i quali non esiste ambiguità poiché sono condivisi dall'intera comunità di utenti del dominio applicativo stesso". Essenzialmente, il ruolo di un'ontologia è di permettere la combinazione e la comparazione di informazioni che provengono da basi di dati distinte. Essa costituisce un documento condiviso, che contiene la descrizione formale, esplicita e dichiarativa dei concetti e degli oggetti di un dato dominio, ne identifica le classi più importanti, le organizza in una gerarchia, specifica le loro proprietà e ne descrive le relazioni più significative. L'ontologia definisce così la "struttura di un mondo", di un dominio. Essa si presenta come una collezione di asserzioni che definiscono le relazioni tra i concetti e specificano le regole logiche per ragionare su di essi, un insieme di assiomi che stabiliscono la classificazione e dichiarano le relazioni tra classi e proprietà.

Le ontologie includono generalmente:

- definizioni di concetti e proprietà
- termini di vocabolario
- gerarchie di classi e sottoclassi, tassonomie
- relazioni tra classi, regole, asserzioni
- definizioni di attributi di classe
- assiomi che specificano vincoli

Non includono normalmente:

- istanze/annotazioni

Le ontologie rappresentano una possibilità per raggiungere l'obiettivo di un Web più *machine-friendly*. Attraverso un'ontologia è possibile condividere conoscenza attraverso l'interpretazione comune di un dominio che può essere comunicato tra persone e applicazioni. Le convenzioni usate per presentare la descrizione di un dominio da parte di un'ontologia vanno dal linguaggio naturale a formalismi logici. Per permetterne la comprensione software è ovviamente necessaria

una regolare e specifica codifica formale. Gli agenti software comprendono il significato di un dato riferendosi all'ontologia indicata. La struttura dei dati e la semantica introdotte dalle ontologie aumentano le potenzialità del Web a diversi livelli. I programmi di ricerca che, nel Web tradizionale, cercano e trovano pagine che contengono keyword ambigue o generiche, tra le quali l'utente deve ulteriormente cercare quelle di interesse, con le funzionalità del Web Semantico, invece, cercano e trovano pagine che si riferiscono effettivamente ad un preciso concetto. Le ontologie sono un elemento chiave del Web Semantico, in quanto risolvono il problema di dover individuare i "significati" comuni tra i database che un programma di integrazione di informazioni incontra. L'informazione infatti, in questo tipo di sistema, è distribuita in differenti siti Web o in diversi database, ed è necessario che un programma confronti e combini le informazioni delle varie sorgenti a disposizione. Un esempio di come le prestazioni delle applicazioni basate sul Web Semantico siano superiori è rappresentato dai siti di e-commerce, dove l'introduzione di una o più ontologie, che uniformano le informazioni disponibili dei diversi "negozi", permette un confronto più immediato dei cataloghi e un'analisi via software dei dati. L'ambizioso obiettivo è quello di realizzare un "mondo" nuovo, in cui le persone interagiscono con i loro dispositivi elettronici in un modo completamente nuovo da quello attuale. Immaginiamo che una persona sia in uno studio medico per fissare alcune visite ma abbia già diversi appuntamenti. Essa potrà istruire, attraverso il browser Web del palmare, il proprio agente software per il Web Semantico e potrà effettuare una ricerca per trovare i posti disponibili ad una distanza massima di 20 Km dalla propria casa e solo in un determinato orario del giorno. L'agente software, quindi, comincerà ad interagire con altri agenti presenti nei siti Web dei dottori dell'area circostante. In poco tempo l'agente presenterà un programma di appuntamenti per le visite. Se il programma è soddisfacente si potranno inserire gli appuntamenti in agenda ed inviare una e-mail all'istruttore di tennis per disdire gli allenamenti per il mese seguente e, se i risultati non sono soddisfacenti, si potranno aggiungere nuove preferenze ed effettuare una nuova ricerca. Le ontologie giocano un ruolo fondamentale nella definizione della terminologia che gli agenti usano per nello scambio di informazioni. Scegliere il linguaggio giusto per rappresentarle è fondamentale per la progettazione di agenti.

Riportiamo qui brevemente la classificazione delle ontologie fatta da Nicola Guarino [26, 27, 28]:

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

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

- **Application ontology**: descrivono concetti che dipendono sia da un particolare dominio sia da un particolare obiettivo.

2.2.2 Problemi semantici

Consideriamo sorgenti diverse che condividono una visione similare del problema da modellare, quindi un insieme di concetti comuni. Poiché esse sono state progettate e modellate da persone differenti, è molto improbabile che condividano la stessa concettualizzazione del mondo esterno Possiamo affermare quindi che non esiste una semantica univoca a cui chiunque possa riferirsi. Le diverse concettualizzazioni del mondo esterno che le persone distinte possono avere è sicuramente la causa principale delle differenze semantiche tra le fonti di informazioni, ma non è l'unica. Le differenze nei sistemi di DBMS, ad esempio, possono portare all'uso di differenti modelli per la rappresentazione della porzione di mondo in questione. Partendo così dalla stessa concettualizzazione, infatti, determinate relazioni tra concetti avranno strutture diverse a seconda che siano realizzate attraverso un modello relazionale o ad oggetti. L'obiettivo di fornire un accesso integrato ad un insieme di sorgenti si traduce nell'identificazione di concetti comuni all'interno di queste sorgenti e nella risoluzione delle differenze semantiche presenti tra di loro.
Possiamo classificare le contraddizioni semantiche in tre gruppi principali:

- Eterogeneità tra i nomi delle classi di oggetti: due classi che rappresentino lo stesso concetto nello stesso contesto possono usare nomi diversi per attributi o per i metodi, oppure avere gli stessi attributi con domini di valori differenti o ancora avere regole diverse su questi valori.

- Eterogeneità tra le strutture delle classi: sono le differenze nei criteri di specializzazione, nelle strutture per realizzare un'aggregazione, ed anche le discrepanze semantiche, quando cioè valori di attributi sono parte dei metadati di un altro schema (ad esempio, l'attributo "sesso", presente esplicitamente in uno schema, può essere presente solo implicitamente in un altro schema attraverso la divisione della classe "persone" in "maschi" e "femmine").

- Eterogeneità nelle istanze delle classi: al esempio, l'uso di diverse unità di misura per i domini di un attributo, o la presenza/assenza di valori nulli.

2.3 Il sistema MOMIS

MOMIS (Mediator envirOnment for Multiple Information Sources) [29] è il progetto di un sistema I3 per l'integrazione di sorgenti di dati strutturati e semistrutturati, il cui sviluppo è iniziato all'interno del progetto MURST 40% INTERDATA dalla collaborazione tra i gruppi operativi dell'Università di Modena e Reggio Emilia e di Milano, e che attualmente continua nell'ambito del progetto SEWASIE (SEmantic Webs and AgentS in Integrated Economies), che si propone di progettare e implementare un motore di ricerca semantico che permetta di accedere a sorgenti di dati eterogenee nel Web [30]. MOMIS è nato dalla necessità di automatizzare la fase di reperimento e integrazione delle informazioni provenienti da una varietà di sorgenti difficilmente integrabili, data la recente esplosione nella quantità e varietà dei dati accessibili. L'idea del progetto MOMIS è fornire un accesso integrato a queste informazioni eterogenee, memorizzate sia in database tradizionali o in file system, sia in sorgenti di tipo semistrutturato, mettendo l'utente in condizione di porre una singola query ed ottenere una risposta unificata dai dati proveniente dalle singole sorgenti. Il contributo innovativo di questo progetto risiede nella fase di analisi ed integrazione, realizzata in modo semi-automatico, nell'impiego di un approccio semantico (oltre che virtuale) all'integrazione e nell'uso di Logiche Descrittive.

2.3.1 L'architettura di MOMIS

L'architettura adottata dal MOMIS è schematizzata in figura 8. Descriviamo qui brevemente i livelli che la costituiscono:

- **Sorgenti** locali di dati: sono le sorgenti informative strutturate o semistrutturate da integrare, possono essere database o parte di essi, file system, etc.

- **Wrapper**: sono i moduli che rappresentano l'interfaccia tra il mediatore vero e proprio e le sorgenti locali di dati. Servono a rendere disponibili i dati della

sorgente in esame in un formato che permetta l'integrazione. La loro funzione è duplice:

- In fase di integrazione, forniscono la descrizione delle informazioni delle sorgenti in linguaggio ODLI3 (descritto nel Paragrafo 2.4).

- In fase di query processing, traducono la query ricevuta dal mediatore (espressa quindi nel linguaggio comune di interrogazione, OQLI3) in una query comprensibile e realizzabile dalla sorgente stessa. Devono inoltre esportare i dati ricevuti in risposta all'interrogazione, presentandoli al mediatore attraverso il modello comune dei dati utilizzato dal sistema.

- **Mediatore**: è il cuore del sistema. E' costituito da due moduli distinti:

  - **GSB** (Global Schema Builder): è il modulo di integrazione degli schemi locali che, partendo dalle descrizioni delle sorgenti, espresse attraverso il linguaggio ODLI3, genera un unico schema globale, la GVV (Global Virtual View), una vista integrata degli schemi locali da presentare all'utente.

  - **QM** (Query Manager): è il modulo di gestione delle interrogazioni che, dalla query dell'utente sullo schema globale, genera la query in OQLI3 da inviare ai wrapper delle singole sorgenti.

- **Utente**: L'utilizzatore del sistema dovrà poter interrogare lo schema globale e ricevere un'unica risposta, come se stesse interrogando un'unica sorgente di informazioni.

I componenti del GSB, che interagiscono con strumenti software esterni, sono:

- SIM1: componente che si occupa della generazione del Thesaurus comune, un insieme di relazioni terminologiche (lessicali e strutturali).

- SI-Designer: ha il duplice scopo di interfacciarsi fra il sistema e il progettista (fornendo a quest'ultimo un'interfaccia "user-friendly" per l'interazione) e di coordinare l'esecuzione dei diversi software che partecipano all'integrazione. In particolare, interagisce con WordNet per estrarre automaticamente relazioni lessicali tra i termini usati per dare nome a classi ed attributi. Utilizza Artemis per svolgere il calcolo delle affinità e assiste il progettista nelle ultime fasi della costruzione dello schema globale, generando in linguaggio ODLI3 il codice che lo descrive.

- Parser ODLI3: effettua l'analisi degli schemi sorgenti, verificandone la consistenza, analizza e archivia, inoltre, tutte le informazioni sullo schema integrato.

Gli strumenti fondamentali utilizzati dal sistema MOMIS sono:

- **ODB-Tools Engine** [31]: è un tool, sviluppato presso il Dipartimento di Scienze dell'Ingegneria dell'Università di Modena, che compie la validazione di schemi e l'ottimizzazione semantica di query rivolte a Basi di Dati Orientate agli Oggetti (OODB). E' basato sulla logica descrittiva OLCD [Paragrafo 2.4.3].

- **WordNet** [32]: è un database lessicale on-line in lingua inglese, capace di individuare relazioni lessicali e semantiche fra termini, sviluppato dal Cognitive Science Laboratory alla Princeton University. Sostantivi, verbi, aggettivi e avverbi della lingua inglese vengono organizzati in insiemi di sinonimi (synset), ognuno dei quali rappresenta un determinato concetto lessicale. Vari tipi di relazioni collegano fra loro i synset.

- **ARTEMIS-Tool Environment**: è un tool che compie analisi e clustering di schemi. Esso riceve in ingresso il thesaurus, cioè l'insieme delle relazioni terminologiche (lessicali e strutturali) precedentemente generate, e sulla base di queste, assegna ad ogni classe coinvolta nelle relazioni un coefficiente numerico indicante il suo grado di affinità. I coefficienti sono utilizzati per raggruppare le classi locali in modo che ogni gruppo (cluster) comprenda solo classi con coefficienti di affinità maggiori.

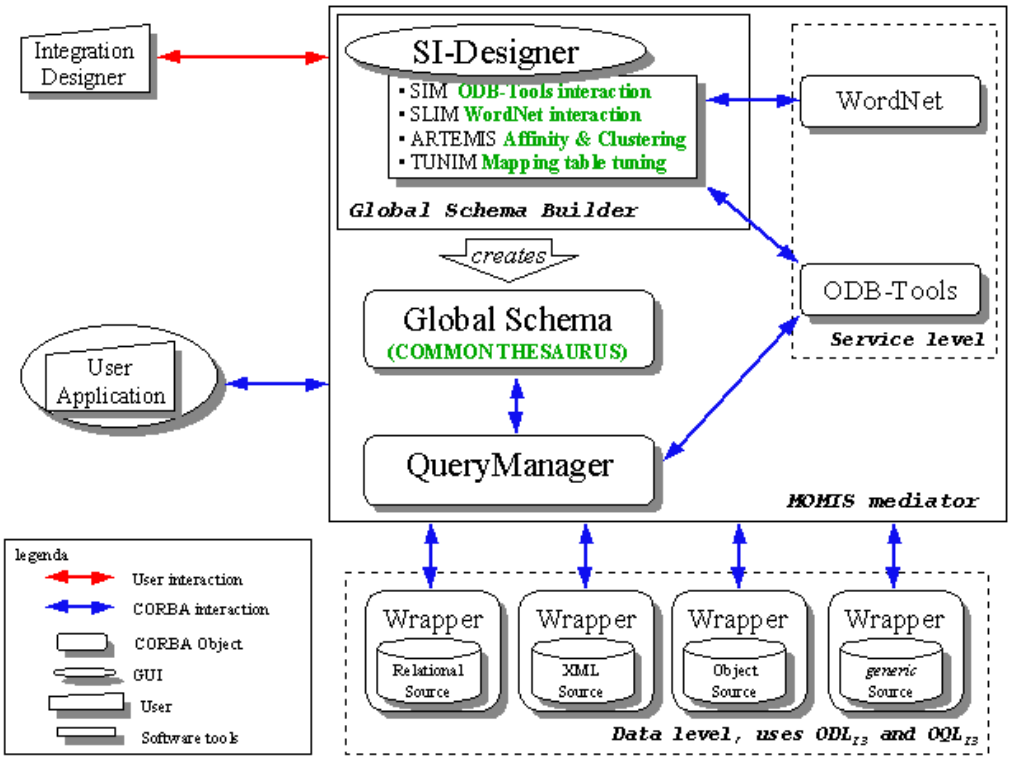

Figura 8 - Architettura del sistema MOMIS

2.3.2 Il processo d'integrazione

L'integrazione delle sorgenti informative strutturate e semistrutturate, schematizzata in figura 9, viene compiuta in modo semi-automatico, utilizzando le descrizioni degli schemi locali in linguaggio ODLI3 e combinando le tecniche di Description Logics e di clustering. Le attività compiute sono le seguenti:

Generazione del Thesaurus Comune:

Durante questo passo viene costruito un Thesaurus Comune di relazioni terminologiche. Le relazioni terminologiche esprimono la conoscenza inter-schema su sorgenti diverse. Le relazioni terminologiche sono derivate in modo semi-automatico, a partire dalle descrizioni degli schemi in ODLI3, attraverso l'analisi strutturale (utilizzando ODB-Tools e le tecniche di Description Logics) e di contesto (attraverso l'uso di WordNet) delle classi coinvolte.

Generazione dei cluster di classi ODLI3:

Con il supporto dell'ambiente ARTEMIS-Tool, le relazioni terminologiche contenute nel Thesaurus vengono utilizzate per valutare il livello di affinità tra le classi ODLI3 in modo da identificare le informazioni che devono essere integrate a livello globale. A tal fine, ARTEMIS calcola i coefficienti che misurano il livello di affinità delle classi ODLI3 basandosi sia sui nomi delle stesse, sia sugli attributi. Le classi ODLI3 con maggiore affinità vengono raggruppate utilizzando le tecniche di clustering.

Costruzione dello schema globale del mediatore, della GVV:

I cluster di classi ODLI3 affini sono analizzati per costruire lo schema globale del Mediatore. Per ciascun cluster viene definita una classe globale ODLI3 che rappresenta tutte le classi locali che sono riferite al cluster, e che è caratterizzata dall'unione dei loro attributi. L'insieme delle classi globali definite costituisce lo schema globale del Mediatore che deve essere usato per porre le query alle sorgenti locali integrate.

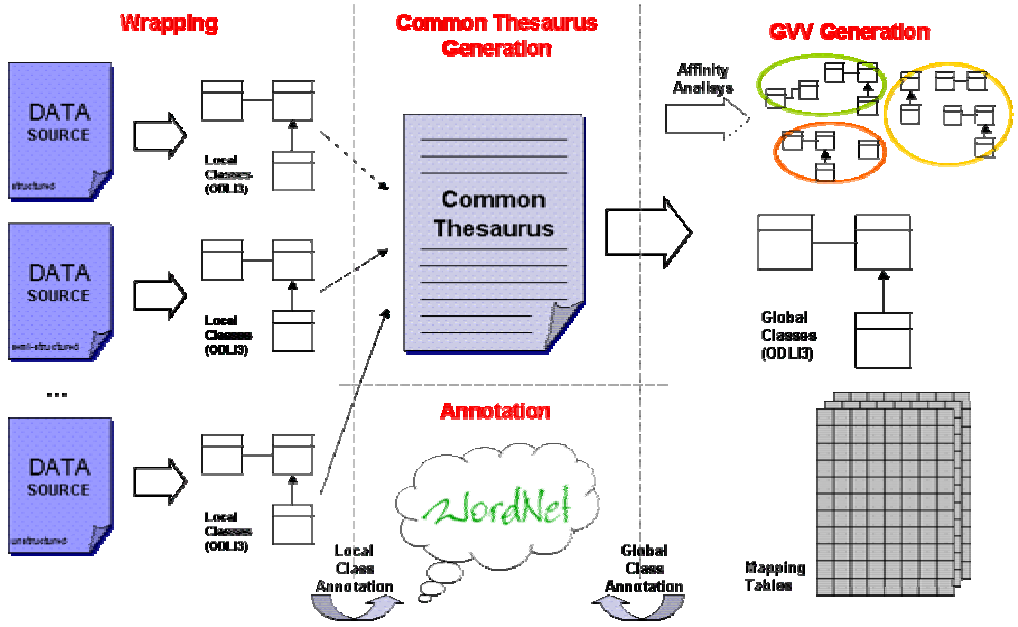

Figura 9 - Il processo d'integrazione in MOMIS

Riassumendo, in MOMIS i wrapper si incaricano di tradurre in ODLI3 le descrizioni delle sorgenti, i moduli del mediatore di costruire lo schema globale, cioè la GVV, che costituisce l'ontologia MOMIS, composta dalle classi globali a cui è associata l'annotazione. Tutte le interrogazioni sono poste dall'utente su questo schema utilizzando il linguaggio OQLI3, disinteressandosi dell'effettiva natura ed organizzazione delle sorgenti. Sarà il sistema stesso che si incaricherà di tradurre le interrogazioni in un linguaggio comprensibile alle singole sorgenti e di riorganizzare le risposte ottenute in una unica, corretta e completa, da fornire all'utilizzatore. Alcune delle problematiche del sistema MOMIS hanno portato alla constatazione della necessità di una nuova generazione del Web, il Web Semantico [Capitolo 1], che mira a rendere i dati presenti nelle rete non solo *machine-readable* ma anche *machine-understandable*.

- Web Semantico: nasce dalla necessità di elaborare automaticamente le informazioni per poter fare un ampio uso di agenti software.

- MOMIS: nasce dalla necessità di rendere trasparente all'utente le diversità delle informazioni che ha a disposizione. MOMIS va oltre le problematiche affrontate nell'ambito del Semantic Web e mira ad una reale integrazione delle sorgenti.

2.4 Il linguaggio ODLI$_3$

In MOMIS l'integrazione di sorgenti di dati strutturati e semistrutturati viene compiuta in modo semi-automatico, utilizzando le descrizioni degli schemi locali in linguaggio ODLI3 [33]. L'ODLI3 è un linguaggio di descrizione dei dati, object-oriented, ad alto livello, che si avvale dello standard di modelli ad oggetti ODMG-93 ed è indipendente dal formato della sorgente dei dati stessi. ODLI3 deriva direttamente dal linguaggio ODL (Object Definition Language) [34] e ne rappresenta un'estensione, in accordo con le raccomandazioni della proposta di standardizzazione per i linguaggi di mediazione elaborata presso l'Università del Maryland dal gruppo I3 costituitosi in tale ambito. ODLI3 è un linguaggio in grado di supportare sia dei sistemi complessi, quali possono essere quelli ad oggetti, sia quelli semplici, quali i file strutturati.

2.4.1 ODLI$_3$ e ODL

L'ODL, pur essendo ben progettato per la rappresentazione della conoscenza relativa ad un singolo schema ad oggetti, risulta insufficiente per la descrizione relativa ad un insieme di sorgenti di basi di dati eterogenee. Nel progetto MOMIS, quindi, il linguaggio ODL può risultare solamente una buona base di partenza. Le estensioni apportate ad ODL da parte di ODLI3 riguardano fondamentalmente:

- La possibilità di definire strutture di **dati semistrutturati** che, per definizione, non devono seguire una rigida formattazione. I dati semistrutturati, infatti, si prestano a rappresentare lo stesso aspetto della realtà attraverso contenuti fortemente differenziati a livello strutturale. Attributi che generalmente assumono questa caratteristica potrebbero essere l'indirizzo, l'ora, o la data. Per quest'ultimo caso, ad esempio, ci si potrebbe trovare di fronte a un formato dato solamente da una stringa, in cui sono racchiuse tutte le informazioni, oppure ad un formato composto da un insieme di campi, quali giorno, mese ed anno.

- Sono previsti due nuovi costrutti:

  - **union**: introdotto per esprimere strutture dati alternative nella definizione di una classe ODLI3, in modo da poter catturare le caratteristiche dei dati semistrutturati.

  - **optional**: per indicare che un determinato attributo è opzionale per un'istanza, cioè potrebbe non venire specificato nell'istanza. Anche questo costrutto è stato introdotto per catturare requisiti dei dati semistrutturati.

- Estensioni ai tipi valore ed ai tipi classe.

- E' stata data al wrapper la possibilità di indicare, per ogni classe, il nome della sorgente di appartenenza ed il relativo tipo.

- Nel caso di sorgenti relazionali è possibile definire, per le classi appartanenti a tali sorgenti, delle **foreign key**.

- E’ stata introdotta la possibilità di definire attributi globali, oltre ai normali attributi locali.

- Vengono supportate le **Mapping Rule**, le regole di mapping, fra grandezze locali e globali. Questo tipo di regole è stato introdotto per esprimere le relazioni che esistono tra la descrizione dello schema integrato e la descrizione ODLI3 degli schemi delle sorgenti originali.

- E’ possibile la definizione di **regole di integrità** denominate *if-then rule*. Questo tipo di regola è stato introdotto perché si potesse esprimere, in maniera dichiarativa, vincoli di integrità di tipo if-then, sia all’interno della stessa sorgente, che tra sorgenti differenti, sia sullo schema globale, che sui singoli schemi locali.

- ODLI3 può essere automaticamente tradotto nella logica descrittiva OLCD, usata dagli ODB-Tools.

- È possibile definire le **relazioni terminologiche**, intensionali ed estensionali, di:

    - SYN (SYNonym-of): sinonimia
    - BT (Border Terms): ipernimia
    - RT (Related Terms): associazione
    - NT (Narrow Terms): iponimia

- **Annotazioni** rispetto a WordNet.

2.4.2 Sintassi $ODLI_3$

Vengono adesso illustrate le caratteristiche principali della sintassi ODLI3, soprattutto per quanto riguarda le estensioni che sono state introdotte rispetto alla sintassi ODL. In Appendice 1 è riportata la sintassi BNF del linguaggio ODLI3, comprendente sia gli elementi sintattici del linguaggio standard ODL che le estensioni.

2.4.2.1 Tipi di dati

Si distinguono in:

- **ValueType**, **tipi valore**: caratteristici delle variabili e degli attributi semplici. Ogni istanza di questo tipo non ha un proprio identificatore, la sua unica proprietà è il suo valore. Sono insiemi di valori complessi, limitatamente raggruppati, senza identità di oggetto.

- **Interface**, **tipi classe**: gli attributi complessi, e gli oggetti in generale, sono istanze di classe, caratterizzate da un proprio OID (Object IDentifier), una propria interfaccia e un proprio comportamento. I tipi classe sono denominati più brevemente Classi.

I **tipi valore** di distinguono in:

- SimpleType
- ConstrType

Ai SimpleType appartengono:

- I tipi predefiniti (BaseType): int, short, long, string, float, char, boolean, octect, any. Sono i tipi semplici di base. I tipi int, short e long possono anche essere dichiarati unsigned. Il tipo any è il tipo da cui derivano tutti gli altri presenti nel linguaggio ODLI3, sia i tipi valore che quelli classe. Il contenuto di un oggetto any può essere in pratica qualsiasi cosa. Ai tipi semplici di base presenti dell'ODL è stato aggiunto in ODLI3 un nuovo tipo che può essere

utilizzato per rappresentare un valore intero compreso tra un estremo inferiore e uno superiore: il tipo range.

- I tipi collezione (CollectionType o TemplateType): sono una collezione di istanze di tipo SimpleType tutte omogenee tra loro:

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

- I tipi DefinedType definiti dall'utente: un tipo definito possiede un nome proprio, un ValueType mappato ed, eventualmente, un insieme di valori interi, nel caso si tratti di un array, che stanno ad indicare la dimensione dell'array stesso. Il fatto che il tipo riferito sia un generico ValueType, e non un SimpleType, consente la dichiarazione di variabili strutturate, ad esempio, tramite una sintassi semplificata: prima si definisce un nuovo tipo, poi le variabili secondo la sintassi dei tipi atomici.

I ConstrType possono essere dettagliati in:

- StructType (tipi struttura): servono per definire strutture di tipi valore. I tipi struttura sono caratterizzati da un tagName opzionale che dà la possibilità all'utente di dichiarare diverse variabili dello stesso tipo senza essere costretto tutte le volte a riscrivere l'intera struttura. Tramite l'utilizzo del termine opzionale tagName le dichiarazioni devono sempre essere accompagnate dall'utilizzo della parola chiave *struct*.

- UnionType (tipi unione): come StructType, sono caratterizzati da un tagName opzionale. Il tipo unione serve per scegliere, in base al valore di una variabile in una clausola switch, il tipo di appartenenza di una variabile. Come per il caso dei tipi struttura, le variabili che compaiono in un'unione devono essere ValueType, e non tipi classe.

- EnumType (tipi enumerazione): servono per restringere il campo dei valori possibili di un SimpleType a quelli presenti in un elenco. In essi non c'è altro che un semplice elenco definito di valori e la variabile associata dovrà assumere solamente uno dei valori elencati nella dichiarazione.

I **tipi classe** sono identificati dalla parola chiave *interface* e sono costituiti da un insieme di diverse dichiarazioni. Ogni classe è caratterizzata da due parti fondamentali:

- Interface Header (l'intestazione dell'interfaccia), in cui si specifica il nome della classe (OID), le superclassi da cui eredita (se presenti) ed una lista di proprietà (PropertyList).

- Interface Body (il corpo dell'interfaccia), che ne descrive la struttura interna (l'insieme di attributi e di metodi che ne fanno parte).

Interface Header:

Ereditarietà (inheritance): Per ogni Interface è ammessa l'ereditarietà multipla, cioè la possibilità di avere una o più superclassi. Per ogni superclasse vengono ereditati tutti gli attributi e tutti i metodi. Questa proprietà viene rappresentata attraverso la collezione complessa inheritance che mappa sulla stessa classe Interface.

Source: E' un costrutto per la dichiarazione della sorgente, introdotto da ODLI3, che contiene la descrizione della classe. Tale costrutto indica che ogni classe appartiene a una sorgente, caratterizzata da un nome (unico) e da un tipo (relazionale, a oggetti, file, semistrutturato). Il nome e il tipo della sorgente vengono definiti all'interno della PropertyList. Poiché non viene a priori esclusa la presenza di classi caratterizzate da uno stesso nome, per ottenere una identificazione univoca della classe stessa occorre ricorrere alla coppia nameSource-nameInterface, cioè al mantenimento, anche all'interno dell'oggetto Source, della lista di tutte le classi che ne fanno parte. Questo meccanismo risulta efficace anche dal punto di vista di una migliore navigazione nella struttura dei dati.

Extent: La proprietà di estensione, specificata nella PropertyList, rappresenta l'insieme di tutte le istanze di una particolare classe all'interno di un database. Serve per facilitare il reperimento indicizzato delle informazioni da parte del DBMS. Nella definizione di un Interface Header è consentita una sola definizione di extent.

Key e ForeignKey: All'interno della lista di proprietà è possibile definire anche una chiave (key). Nella sintassi ODLI3, infatti, è stata inserita la possibilità di definire foreign key, semplici e composte. Ogni chiave può mappare un singolo attributo presente nel corpo dell'interfaccia (chiave semplice), oppure può mappare un insieme di attributi (chiave composita).

Interface Body:

Nel Body dell'interfaccia vengono descritti tutti i metodi e gli attributi che sono caratteristici della classe (o vista). Diversamente da quanto previsto dall'ODL, in ODLI3 è prevista la possibilità di definire strutture di dati semistrutturati che, per definizione, non seguono una rigida formattazione. L'ODLI3 mette a disposizione, per questi casi, lo speciale costrutto *union*, che permette di definire più specifiche alternative per una stessa interfaccia e di confrontare degli oggetti che rappresentano due istanze della medesima realtà, ma che sono descritte attraverso scelte strutturali differenti. Ad una stessa classe possono appartenere più attributi con lo stesso nome, uno diverso per ogni implementazione consentita. Sempre per rappresentare dati semistrutturati è stato inserito il costrutto *optional*. Postponendo un asterisco (*) alla dichiarazione di un attributo, all'interno del corpo di un'interfaccia, si afferma che in una istanza della classe stessa l'apparizione dell'attributo in questione è opzionale. Un Interface Body può essere composto dai seguenti elementi, che devono avere nomi unici:

- Operazioni
- Attributi
- Relazioni
- Costanti

Operazioni: Sono i metodi della classe, o meglio, la loro signature, quindi descrivono il comportamento della classe stessa. La classe Operation contiene le informazioni riguardanti il nome e il tipo restituito dal metodo (SimpleType) ed una lista di parametri passati (OpVar) ognuno avente un nome, un tipo (SimpleType) ed un elenco di eventuali dimensioni di array.

Attributi: Sono previste due categorie di attributi:

- Semplici, di tipo valore
- Composti, di tipo classe

I tipi di attributi possibili sono:

- SimpleAttribute, attributi semplici, locali: sono caratterizzati da un tipo generico, denominato type, un booleano, per indicare se si tratta di attributi di sola lettura, ed una dimensione di array.

- Relationship (relazioni), attributi complessi: servono per mappare un'interfaccia e contengono informazioni sulla inversa attraverso un puntatore alla stessa classe Relationship. Qui il tipo (type) indica se l'attributo è una collezione o no, mentre orderBy serve per definire il criterio di ordinamento. Le relazioni possono fare riferimento solamente ad interfacce, mai a tipi valore.

- GlobalAttribute, attributi composti, globali: sono stati introdotti dall'ODLI3 per l'operazione di integrazione delle sorgenti. Hanno le stesse caratteristiche degli attributi locali ma in più possiedono la proprietà di avere un collegamento con oggetti MappingRule, in grado di legare grandezze appartenenti allo schema integrato con grandezze degli schemi locali.

MappingRule: Gli oggetti MappingRule servono per esprimere relazioni tra l'ontologia globale e le ontologie locali sottostanti. Qualora la dichiarazione di un attributo sia seguita da una MappingRule (un insieme di regole di mapping), l'attributo è automaticamente considerato globale. Possono presentarsi vari casi:

- Corrispondenza tra l'attributo globale ed un solo attributo locale (AttributeOnly): il sistema possiede le informazioni necessarie per realizzare in modo automatico il meccanismo di mapping e fa coincidere il nome con quello dell'attributo locale.

- Corrispondenza tra l'attributo globale ed un insieme di diversi attributi locali:

    - Se gli attributi locali sono concatenati tramite una condizione and (AndList) allora l'attributo globale è ottenuto dal concatenamento degli attributi locali stessi.

    - Se gli attributi locali sono concatenati tramite una condizione union (UnionList) ciò viene risolto tramite la corrispondenza tra l'attributo globale ed uno solo alla volta degli attributi locali in union. Per poter individuare con

certezza quale di questi parametri locali debba essere mappato si ricorre ad un terzo attributo locale (onParameter).

- Non c'è corrispondenza tra l'attributo globale e gli attributi locali: potrebbe essere necessario esprimere un metaconcetto tramite un valore di default (DefaultValue).

Costanti: All'interno del corpo dell'interfaccia possono essere definite delle costanti indicando tre componenti:

- Il nome della costante
- Il tipo della costante, che deve essere un tipo semplice BaseType
- Il valore che assume

2.4.2.2 Relazioni terminologiche

Un'altra aggiunta nella sintassi di ODLI3 riguarda la possibilità di inserire relazioni terminologiche tra classi, tra attributi, oppure miste tra gli attributi e le classi. Le relazioni terminologiche vengono definite, durante il processo di integrazione, nella fase di generazione del Thesaurus.

I tipi di relazioni terminologiche intensionali, tra classi e nomi di attributi, sono:

- **SYN** (SYNonym-of): sinonimia tra due termini. Comporta un vincolo di uguaglianza tra le classi o gli attributi coinvolti.
- **BT** (Border Terms): ipernimia, un termine ha un significato più generale di un altro. Comporta un vincolo di ereditarietà tra le classi.
- **NT** (Narrow Terms): iponimia, opposto di BT. Comporta un vincolo di ereditarietà tra le classi.
- **RT** (Related Terms): associazione positiva. Due termini sono generalmente usati insieme nello stesso contesto.

Esistono diversi casi per l'archiviazione della struttura dati:

- Creazione di un oggetto InterfaceRel, per una relazione che coinvolge due classi
- Creazione di un oggetto AttributeRel, per una relazione che coinvolge due attributi
- Creazione di un oggetto AttrIntRel, per una relazione che coinvolge una classe e un attributo

Un'ulteriore importante funzionalità nel processo di integrazione delle sorgenti è la possibilità di esprimere, sotto forma di proposizioni, relazioni estensionali tra gli oggetti di classi distinte. Questi assiomi esprimono le relazioni insiemistiche tra le estensioni di classi definite in sorgenti autonome (indipendentemente dalla loro intensione). Ogni classe dello schema integrato può avere associati gli assiomi estensionali, che predicano le relazioni di inclusione, equivalenza e disgiunzione. Esse vengono espresse sottintendendo l'intersezione in quanto si presume che classi tra loro affini abbiano almeno estensioni la cui intersezione non è nulla. Gli assiomi estensionali sono espressi come Rule nel linguaggio ODLI3, senza ulteriori estensioni alla sintassi. La ragione che consente di ricorrere alla sintassi delle Rule risiede nel fatto che assiomi estensionali e Rule sono semanticamente equivalenti.

2.4.2.3 Regole di Integrità

Un'altra particolarità della sintassi ODLI3 è data dalle regole di integrità di tipo *if-then*, che devono essere verificate per ogni istanza del database. Tale sintassi contiene due forme distinte:

- rule *nomerule*
  forall *iteratore* in *collezione* : *antecedente* then *conseguente*
- rule *nomerule*
  [{ case of *identifier* : *caselist* }]

Attributi della classe Rule:

- Nomerule: nome della regola d'integrità
- Iteratore: identificatore che rappresenta un'istanza di un elemento fra quelli appartenenti a collezione

- Collezione: insieme di istanze, una classe o parte di essa
- Antecedente: condizione *if* della regola, che si ottiene ponendo in and una serie di predicati booleani
- Conseguente: parte *then* della regola, affermazione valida per tutte le istanze che verificano l'antecedente

Descriviamo in dettaglio le condizioni che vengono poste in and tra loro, cioè gli oggetti classe RuleBody:

- In: confronta due oggetti (ad esempio dotName1 e dotName2) e ritorna vero se dotName1 è contenuto nella collezione dotName2.

- Forall e Exist: sintassi identica a quanto già visto per la prima parte della Rule, in quanto è prevista una chiamata ricorsiva a predicati booleani della classe RuleBody. La differenza tra i due predicati consiste nel numero di match necessari perché la condizione ritornata sia vera:

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

- Compare: classico operatore di confronto tra il valore di una variabile (ad esempio un attributo) e il valore di un letterale. L'attributo CastType (SimpleType) serve per fare, quando richiesto, il cast di un valore letterale per rendere possibile il confronto con la variabile specificata.

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

    - Name: nome dell'operazione
    - OpType: tipo ritornato dall'operazione, un SimpleType
    - Args: lista di parametri (oggetti RuleOpArg), di ognuno dei quali si conosce il type (SimpleType): DotName o Value a

seconda che il parametro sia, rispettivamente, una variabile/attributo/costante o un letterale.

2.4.2.4 Annotazioni rispetto a WordNet

Una fase molto importante del processo di integrazione delle sorgenti del sistema MOMIS è quella di annotazione, durante la quale si può selezionare manualmente uno o più significati per ogni elemento dello schema sorgente locale utilizzando l'ontologia lessicale di WordNet. Successivamente si avrà la fase di generazione del Thesaurus comune. Per poter rappresentare le annotazioni rispetto a WordNet, il linguaggio ODLI3 aggiunge il costrutto WNAnnotation. Se per un termine sono previste più annotazioni diverse vengono ripetute diverse dichiarazioni di WNAnnotation.

2.4.3 OLCD

Le Logiche Descrittive (DLs) costituiscono una restrizione delle logiche del primo ordine. Esse consentono di esprimere i concetti sotto forma di formule logiche, usando solamente predicati unari e binari, contenenti solo una variabile (corrispondente alle istanze del concept). La logica descrittiva che sottende a ODLI3 è OLCD (Object Language with Complements allowing Descriptive cycles). Derivante dalla famiglia KL-ONE, estende la logica descrittiva OLC (Object Language with Complements) [5]. OLCD è proposto come formalismo comune per esprimere descrizioni di classi, vincoli di integrità ed interrogazioni ed è dotato di tecniche di inferenza basate sul calcolo della sussunzione introdotte per le Logiche Descrittive nell'ambito dell'Intelligenza Artificiale [65].

3. UML e Metamodello

3.1 Il paradigma Object-Oriented

Nel corso dell'ultimo decennio, il paradigma Object-Oriented è divenuto il fondamento della maggior parte dei processi di sviluppo del software. Fu introdotto per far fronte alle limitazioni del paradigma strutturato, che, tra il 1975 e il 1985, ha riscosso un notevole successo nella realizzazione di prodotti software di piccola e media dimensione. Tuttavia si è rivelato inadeguato quando applicato a sistemi di grandi dimensioni. Al contrario, il paradigma Object-Oriented si è dimostrato estremamente efficace nella realizzazione di sistemi software di piccola, media e anche grande dimensione. A partire dalla fine degli anni Ottanta, si è affermato come il paradigma dominante e ha dato origine a diverse metodologie e modelli di sviluppo che hanno rivoluzionato il modo di produrre software. Verso la fine degli anni Sessanta, la crescente complessità dei sistemi software ha determinato la nascita di quella disciplina nota come "Ingegneria del Software". Il suo scopo è la produzione di software di qualità, in tempi brevi e al minor costo possibile. L'idea su cui si fonda è quella di applicare al processo di produzione del software gli stessi paradigmi progettuali comuni ad altre discipline ingegneristiche. Il paradigma strutturato rappresenta il primo tentativo di introdurre delle tecniche derivanti dall'Ingegneria del Software, in grado di fornire un valido supporto durante le diverse fasi in cui si articola il processo di produzione del software. Queste tecniche coprono l'intero ciclo di vita e furono inizialmente applicate con successo, soprattutto perché, prima dell'introduzione del paradigma strutturato, la maggior parte delle aziende non utilizzava nessuna tecnica specifica. Il software veniva semplicemente "scritto" e l'abilità di un programmatore era misurata in termini di linee di codice prodotte. Il paradigma strutturato si è dimostrato inadeguato quando applicato a prodotti di grandi dimensioni, che cominciarono a diffondersi verso la fine degli anni Ottanta. La ragione è attribuita al fatto che questo paradigma è una collezione di tecniche orientate solamente ai dati o solamente alle azioni. Invece il paradigma Object-Oriented attribuisce la stessa importanza sia ai dati che alle azioni, infatti si basa sul concetto di "oggetto", un'entità che incapsula sia i dati che le azioni che operano su di essi. Più precisamente, un oggetto è un'istanza di una classe, ossia un tipo di dato astratto che supporta l'ereditarietà. La dichiarazione di classe incapsula i campi (dati) e i metodi (azioni) degli oggetti creabili come istanza della classe. L'incapsulamento rappresenta un meccanismo per dominare la complessità, infatti viene definito come il risultato del processo di nascondere i dettagli di un oggetto che non contribuiscono alla sua natura essenziale visibile dall'esterno. La comunicazione tra oggetti avviene mediante scambio di messaggi, quindi un oggetto risulta essere un'unità indipendente

con un'interfaccia ben definita dalla classe che istanzia. Questo agevola la manutenzione e il riuso di oggetti esistenti, quindi rappresenta un altro punto di forza del paradigma Object-Oriented. Viene spontaneo domandarsi se in un futuro prossimo esisterà un paradigma migliore di quello Object-Oriented. Certamente il paradigma Object-Oriented non rappresenta la soluzione definitiva a tutti i problemi dell'Ingegneria del Software. Tuttavia è opinione comune che attualmente le tecniche Object-Oriented rappresentano il meglio che l'Ingegneria del Software possa offrire.

3.1.1 Metodologie Object-Oriented

In tutti i processi produttivi del software, le fasi di analisi e progettazione sono tra le più importanti e costose in termini di tempo. Se affrontate correttamente, riducono al minimo l'attività di manutenzione che, come è noto, risulta essere quella di gran lunga più dispendiosa. L'obiettivo dell'attività di analisi è stabilire che cosa un'applicazione deve fare, mentre la progettazione si occupa di definire come deve essere fatto. Nell'ambito del paradigma Object-Oriented, sono nate diverse metodologie di analisi e progettazione orientata agli oggetti. La più nota e diffusa è rappresentata dall'UML, un linguaggio di modellazione visuale che ha avuto origine dall'integrazione di tre metodologie sviluppate rispettivamente da G. Booch, I. Jacobson e J. Rumbaugh.

3.1.1.1 OMT (Object Modeling Tecnique)

Questa metodologia è stata messa a punto da **James Rumbaugh** nel 1991 [35]. Essa propone di costruire prima un modello del dominio dell'applicazione e di aggiungere solo successivamente i dettagli implementativi. Le attività di analisi e progettazione vengono suddivise in quattro fasi:

- Analisi: consiste nel modellare il sistema nel mondo reale, esaminando i requisiti e analizzando le loro implicazioni. Successivi raffinamenti portano alla costruzione di un modello rigoroso del dominio del sistema, costituito da tre diversi sottomodelli (a Oggetti, Dinamico e Funzionale).

- Progetto del Sistema: consiste nello stabilire una strategia ad alto livello che risolva efficacemente il problema. Si prendono decisioni concettuali e progettuali sull'architettura complessiva del sistema.

- Progetto a Oggetti: vengono fornite definizioni complete di classi ed associazioni che devono essere usate nell'implementazione, insieme alle interfacce e agli algoritmi dei metodi utilizzati allo scopo di implementare le operazioni.

- Implementazione: è considerata la fase meno importante dell'intero processo in quanto la codifica del sistema è un'attività diretta, quasi meccanica, se lo sviluppo del progetto è stato condotto in modo idoneo e coerente.

La metodologia OMT combina tre modi di vedere la descrizione di un sistema. A tale scopo, utilizza i tre seguenti modelli:

- **Modello ad Oggetti:** rappresenta gli aspetti statici e strutturali del sistema. È una modello puramente orientato ai dati, che utilizza diagrammi simili ai diagrammi UML delle classi.

- **Modello Dinamico**: relativo agli aspetti temporali e di controllo. È una modello puramente orientato alle azioni, che utilizza diagrammi simili ai diagrammi UML di sequenza e di attività.

- **Modello Funzionale**: riguarda gli aspetti di trasformazione, ossia il modo in cui un certo risultato viene ottenuto dal sistema. Utilizza diagrammi di data-flow.

3.1.1.2 OOSE (Object-Oriented Software Engineering)

È' una metodologia messa a punto da **Ivar Jacobson** nel 1992 [36]. Essa pone l'enfasi sull'utilizzo dei diagrammi dei casi d'uso, dai quali derivano tutti gli altri diagrammi. Un sistema viene sviluppato svolgendo tre diverse attività:

- **Analisi**: si interpreta il sistema secondo i suoi requisiti funzionali, vengono identificati gli oggetti e si descrivono le loro interazioni.
- **Costruzione**: consiste di due sotto-attività:

    - **Progettazione**: viene definita l'architettura del sistema. Inizialmente viene elaborato un modello di progetto in cui gli oggetti identificati nella fase di Analisi sono adattati all'ambiente di sviluppo. Usando i diagrammi dei casi d'uso, le interazioni tra i vari oggetti sono tradotte nelle interfacce degli stessi.

    - **Implementazione**: si procede all'implementazione di ogni oggetto.

- **Testing**: il sistema viene verificato, controllandone la correttezza, coerentemente alla sue specifiche.

3.1.1.3 Metodologia di Booch

È una metodologia ideata da **Grady Booch** nel 1994 [37], che può essere definita "evolutiva", in quanto procede attraverso raffinamenti successivi. Il processo di sviluppo è suddiviso in micro e macro-processo. Il primo è iterativo, il secondo rigidamente sequenziale.

Il **micro-processo** rappresenta le attività giornaliere condotte dal singolo sviluppatore. Consiste di quattro passi fondamentali (non sequenziali):

- identificazione delle classi, relativamente ad un particolare livello di astrazione
- identificazione della semantica, in termini di attributi e metodi, delle classi identificate nel passo precedente
- identificazione delle relazioni esistenti tra le classi
- implementazione delle classi

Il **macro-processo** ha lo scopo di controllare le fasi del micro-processo, incorporando le attività di competenza di tutto il team di sviluppo del sistema, da svolgersi in un arco di tempo ampio.

Tali attività sono:

- analisi dei requisiti
- specifica dei requisiti
- progettazione, in cui viene definita l'architettura del sistema
- evoluzione, dedicata ai dettagli implementativi, che sono sviluppati per raffinamenti successivi
- manutenzione, per gestire le eventuali modifiche, successive alla conclusione dello sviluppo del sistema ed alla sua consegna all'utente finale.

3.2 UML

L'UML (Unified Modeling Language), è un linguaggio di modellazione visuale che unifica le "Best Practices" nel campo della progettazione software orientata agli oggetti, in particolare quelle di Booch, Jacobson e Rumbaugh, i "tres amigos" autori del linguaggio. Il suo ambizioso obiettivo è quello di diventare il "linguaggio per la creazione di modelli software ad oggetti", ossia un linguaggio universale per la progettazione di sistemi object-oriented. L'UML è stato proposto per la prima volta nel gennaio del 1997 nella versione 1.0. Oggi rappresenta il linguaggio di modellazione più utilizzato nell'analisi e nella progettazione di sistemi object-oriented. Con il crescere della complessità dei sistemi, infatti, è cresciuta anche l'importanza di buone tecniche di modellazione. Vi sono molti fattori addizionali che servono al successo di un progetto, ma disporre di un linguaggio di modellazione standard e rigoroso è un fattore essenziale. Una delle motivazioni principali nella mente di chi ha sviluppato l'UML era di creare un insieme di costrutti semantici e di notazione che potessero adattarsi in modo adeguato a tutti i livelli di complessità architetturale, in tutti i domini applicativi. L'UML non è un linguaggio proprietario, ma è aperto al contributo di tutti. Sono due gli aspetti di "unificazione" che l'UML permette di ottenere. Innanzitutto, pone termine a molte tra le differenze, spesso non necessarie, tra i linguaggi di modellazione dei metodi precedenti, selezionandone e integrandone le migliori idee. In secondo luogo, ma con un'importanza forse maggiore, unifica le prospettive tra molte diverse tipologie di sistemi (business e software), fasi di sviluppo (analisi dei requisiti, disegno e progettazione) e concetti implementativi. L'UML è un linguaggio di modellazione standard, non un processo standard, un metodo. La diversità delle organizzazioni e la diversità degli ambiti di utilizzo (problem domains) richiedono infatti processi di produzione differenziati. Pertanto gli sforzi si sono concentrati innanzitutto su un metamodello comune (che rende univoci i concetti

semantici) e secondariamente su una notazione comune (che fornisce agli esseri umani una visualizzazione di questi concetti). Le finalità e gli obiettivi dell'UML sono molteplici:

- Integrare le "Best Practices", cioè le linee di condotta migliori, nel campo della progettazione orientata agli oggetti. Prima della nascita dell'UML i metodi per l'analisi e la progettazione ad oggetti superavano la cinquantina. L'UML si propone di essere un linguaggio standardizzato e flessibile che permette di unificare i metodi migliori, in particolare quelli precedentemente proposti dagli autori dell'UML (OMT, OOSE e Booch).

- Fornire un linguaggio di modellazione che risulti di facile comprensione, che si possa imparare ed applicare velocemente nello sviluppo di modelli ad oggetti.

- Essere il più possibile universale, cioè produrre specifiche di modelli indipendenti da ogni particolare linguaggio di programmazione e da qualsiasi processo di sviluppo. L'UML, essendo un linguaggio di progettazione, e non di programmazione, può rappresentare sistemi molto diversi, senza risentire delle differenze legate alla tecnologia.

- Fornire meccanismi di estensione e specializzazione per estendere i concetti di base del linguaggio. Infatti l'UML definisce un insieme ristretto di concetti di base, che possono essere estesi per adattare il linguaggio a specifici domini applicativi. Per l'analisi e la progettazione delle applicazioni più comuni è sufficiente utilizzare i concetti di base, tuttavia è possibile aggiungerne di nuovi e introdurre anche nuove notazioni.

- Fornire una base semi-formale per ottenere specifiche più rigorose. La maggior parte dei metodi object-oriented sono poco rigorosi, le relative notazioni vengono presentate intuitivamente senza l'utilizzo di alcun formalismo. Tuttavia, molti di questi metodi sono stati utilizzati con successo perché ritenuti utili e facilmente comprensibili. Per non perdere queste qualità, ma per aumentare il rigore del formalismo, l'UML ha adottato una specifica semi-formale. Essa consiste nel descrivere il metamodello del linguaggio, ossia un diagramma delle classi che definisce la notazione.

- Incentivare lo sviluppo di strumenti per la progettazione orientata agli oggetti. La disponibilità di un linguaggio universale per la progettazione object-oriented costituisce un beneficio per il mercato dei tool dedicati allo sviluppo ad oggetti, perché fornisce un linguaggio comune e favorisce l'interoperabilità tra applicazioni differenti.

La specifica del linguaggio UML [38] è composta dai seguenti documenti:

- UML **Summary**: è il documento che serve come introduzione all'UML e come mappa per orientarsi tra gli altri documenti.

- UML **Semantics**: è il documento che descrive il metamodello semi-formale che sta alla base della semantica dell'UML. Il metamodello UML viene presentato in notazione UML e in linguaggio naturale sintetico. Lo scopo del metamodello è di fornire una definizione univoca, comune e definitiva della sintassi e della semantica dell'UML. La presenza di questo metamodello ha reso possibile ai suoi autori di trovare un accordo sugli aspetti semantici, separandoli dalle modalità con cui i significati vengono resi accessibili agli esseri umani. Gli autori si aspettano che il contributo di altri studiosi possa esprimere questo metamodello in termini ancor più precisi, tramite il ricorso a tecniche formali [Capitolo 4].

- UML **Notation Guide**: è il documento che descrive la notazione UML e fornisce esempi, che descrive in modo informale la notazione (sintassi concreta) e le sue connessioni con la sintassi astratta. La notazione grafica e la sintassi testuale costituiscono la parte più visibile dell'UML (la visione "pubblica"), utilizzata da esseri umani e strumenti per modellare i sistemi. Si tratta di rappresentazioni di un modello a livello di utilizzatore (user-level model), che dal punto di vista semantico costituisce un'istanza del metamodello UML.

- UML **Process-Specific Extensions**: è il documento che descrive i meccanismi di estensione dell'UML relativamente agli aspetti legati al processo di produzione del software.

3.2.1 Storia di UML

L'UML è stato sviluppato presso la Rational Software Corporation [39], da Booch, Jacobson e Rumbaugh, ma è stato l'OMG (Object Management Group) [40], un consorzio no-profit che produce e mantiene specifiche per l'industria software, che ha fatto dell'UML uno standard. La sua evoluzione è a carico dell'OMG stesso ed è soggetta a procedure ben definite per ogni cambiamento. Allo stato attuale lo sviluppo dell'UML è affidato alle varie "task force" appartenenti all'OMG, le RTF (Revision Task Force). Obiettivo di tale gruppo è accogliere ed analizzare suggerimenti provenienti dalle industrie, correggere inevitabili imperfezioni, colmare eventuali lacune, e così via. Linguaggi distinti per la modellazione object-oriented iniziarono ad apparire tra la metà degli anni '70 e la fine degli '80. Lo sviluppo dell'UML iniziò nell'ottobre 1994, quando Booch e Rumbaugh, della Rational Software Corporation, cominciarono a lavorare per unificare i metodi Booch e OMT. Dato che i metodi Booch e OMT stavano già crescendo nella stessa direzione ed erano considerati a livello mondiale tra i più importanti metodi object-oriented, Booch e Rumbaugh unirono le forze per dare forma a una completa unificazione del loro lavoro e rilasciarono una bozza del Metodo Unificato (Unified Method). Nel 1995, Jacobson si unì a questo sforzo di unificazione, portando con sé il metodo OOSE. Booch, Rumbaugh e Jacobson erano motivati a creare un linguaggio di modellazione unificato per tre ragioni. Innanzitutto, questi metodi stavano già evolvendo, in modo indipendente, verso una direzione comune. Aveva così più senso continuare tale evoluzione insieme, piuttosto che separatamente, eliminando possibili differenze non necessarie che avrebbero ulteriormente confuso gli utilizzatori. Secondariamente, unificando i significati e le notazioni, avrebbero potuto apportare una certa stabilità al mercato object-oriented, permettendo ai progetti di utilizzare un unico linguaggio di modellazione più maturo. In terzo luogo, si aspettavano che la loro collaborazione avrebbe portato dei miglioramenti a tutti e tre i precedenti metodi, aiutandoli ad affrontare correttamente dei problemi che nessuno dei loro metodi gestiva ancora in modo soddisfacente.

‘70/’80		Linguaggi distinti per la modellazione object-oriented
1994	Booch + OMT	Unified Method Notation 0.8
1995	+ OOSE	UML 0.9
Gennaio 1997		Rational: UML 1.0
Novembre 1997	OMG	UML 1.1
	RTF	UML 1.2
1999		UML 1.3
2001		UML 1.4
2002		UML 1.5
2003		UML 2.0

3.2.2 Diagrammi UML

L’UML prevede la definizione dei diagrammi elencati in tabella. Ciascuno di questi concetti ha altri predecessori e molte altre derivazioni. L’UML, infatti, è il prodotto di una lunga storia di idee nell’informatica e nell’area del “software engineering”.

Categoria	Diagrammi
Diagrammi per analisi dei requisiti	Diagrammi dei casi d’uso (use-case)
Diagrammi di struttura statica	Diagrammi dei package Diagrammi delle classi (class-diagram) Diagrammi degli oggetti (object-diagram)
Diagrammi d’interazione	Diagrammi di sequenza (sequence) Diagrammi di collaborazione (collaboration)
Diagrammi di stato	Diagrammi di stato (state-chart) Diagrammi di attività (activity)
Diagrammi di implementazione	Diagrammi dei componenti (component) Diagrammi di dislocamento (deployment)

Descriviamo brevemente le funzioni di ciascun diagramma:

Diagrammi dei casi d'uso: Un diagramma dei casi d'uso è un grafo che mostra le interazioni tra un utente e un caso d'uso di un sistema. Risulta particolarmente utile durante la fase di specifica dei requisiti. Gli elementi che compongono un diagramma dei casi d'uso sono gli Attori e i Casi d'uso.

Diagrammi delle classi: Un diagramma delle classi è un grafo che descrive i tipi di oggetti in un sistema e le diverse relazioni statiche tra di essi. Queste ultime possono essere di due tipi: associazioni e sottotipi. Le classi sono rappresentate da un rettangolo suddiviso in tre sezioni che indicano rispettivamente il nome della classe, i suoi attributi e le sue operazioni. Le associazioni rappresentano relazioni tra istanze di classi. Ogni associazione ha due ruoli, ognuno dei quali è una direzione dell'associazione. Un ruolo ha anche una molteplicità, che indica quanti oggetti possono partecipare all'associazione. Due tipi particolari di associazioni sono rappresentate dalle aggregazioni e dalle composizioni. Le prime sono caratterizzate da un diamante bianco ad una delle due estremità, le seconde presentano un diamante di colore nero.

Diagrammi degli oggetti: I diagrammi degli oggetti rappresentano istanze dei diagrammi delle classi. Sono composti da due elementi: gli oggetti, che rappresentano le istanze delle classi, ed i link, che rappresentano le relazioni che intercorrono tra due o più oggetti.

Diagrammi di sequenza: Un diagramma di sequenza è un grafo che mostra la sequenza di messaggi scambiati tra diversi oggetti per l'esecuzione di un compito specifico. Gli elementi che compongono un diagramma di sequenza sono: il ruolo di un oggetto all'interno di una interazione, la linea di vita di un oggetto, che rappresenta il suo tempo di vita, l'attivazione, ossia il tempo durante il quale un oggetto è coinvolto nel compimento di un lavoro ed i messaggi scambiati tra gli oggetti.

Diagrammi di collaborazione: Analogamente ai diagrammi di sequenza, i diagrammi di collaborazione mostrano la sequenza di messaggi scambiati tra diversi oggetti. Per individuare la giusta sequenza dei messaggi, essi vengono numerati. Gli elementi che partecipano ad una collaborazione sono: il ruolo di un oggetto all'interno di una interazione, il ruolo di associazione, ossia il ruolo svolto dalle associazioni dell'interazione ed i messaggi scambiati tra gli oggetti.

Diagrammi di stato: Sono grafi in cui i nodi rappresentano lo stato in cui si può trovare un oggetto, mentre gli archi rappresentano transizioni di stato. Gli elementi che compongono un diagramma degli stati sono gli stati, in cui può trovarsi un oggetto durante il suo ciclo di vita, e

le transizioni, etichettate con eventi, che rappresentano le relazioni tra i diversi stati in cui può trovarsi un oggetto e indicano le condizioni necessarie per il verificarsi di una transizione di stato.

Diagrammi di attività: Rappresentano un caso particolare di diagramma di stato in cui ogni stato rappresenta una precisa azione da eseguire. Solitamente vengono utilizzati per descrivere attività che si svolgono in parallelo e possono sincronizzarsi in qualche modo. Gli elementi che costituiscono un diagramma di attività sono le corsie, che dividono le azioni in gruppi, i quali vengono assegnati agli oggetti destinati a compiere una determinata attività, le azioni, che rappresentano azioni atomiche solitamente corrispondenti ai passi di un algoritmo, i flussi di azioni, che rappresentano le relazioni tra azioni differenti ed i flussi di oggetti, che rappresentano le relazioni tra un'azione e gli oggetti coinvolti nella sua esecuzione.

Diagrammi dei package: Un diagramma dei package mostra l'organizzazione delle classi in determinati moduli e le dipendenze tra di essi. Una dipendenza tra due elementi esiste quando il cambiamento della definizione di uno implica dei cambiamenti sull'altro. Un sistema software complesso da un punto di vista architetturale consiste di un numero decisamente elevato di classi in relazione tra esse. I diagrammi dei package forniscono uno strumento per dominare questa complessità, poiché consentono di organizzare le classi in unità logiche differenti e permettono di ottenere viste differenti a diversi livelli di astrazione.

Diagrammi dei componenti: Un diagramma dei componenti è un grafo che mostra le dipendenze e le relazioni tra i componenti di un sistema, che rappresentano moduli di codice eseguibile.

Diagrammi di dislocamento: Un diagramma di dislocamento mostra la struttura run-time dei componenti, ovvero del sistema dei processi e degli oggetti presenti in essi. I nodi rappresentano un componente hardware, all'interno del quale sono presenti uno o più componenti software. Gli archi indicano flussi di comunicazione tra componenti software diversi.

3.2.3 Meccanismi di estensione

Gli obiettivi dei lavori di unificazione erano di mantenere l'UML semplice, di eliminare dai metodi Booch, OMT e OOSE gli elementi poco efficaci nella pratica, di aggiungerne di più efficaci derivati da altri metodi, e di inventare qualcosa di nuovo solo quando nessun'altra soluzione esistente risultasse disponibile. Dal momento che gli autori dell'UML stavano di fatto definendo un linguaggio (anche se grafico), avevano la necessità di trovare un corretto punto di equilibrio tra un approccio minimalista (solo testo e rettangoli) ed uno iper-specializzato (un'icona diversa per ciascun elemento di modellazione concepibile). A questo fine, sono stati molto cauti nell'introduzione di nuove cose, perché non volevano rendere l'UML più complesso del necessario. Ci sono però parecchi nuovi concetti inclusi in UML, tra cui:

- Meccanismi di estensione:
    - Vincoli (constraint)
    - Valori etichettati (tagged value)
    - Stereotipi (stereotype)
- Threads e processi
- Patterns/collaborazioni
- Separazione chiara di tipo, classe, istanza
- Affinamenti (per gestire i legami tra diversi livelli di astrazione)
- Interfacce e componenti

3.2.3.1 Vincoli e commenti

Un vincolo è una relazione semantica tra gli elementi del modello che specifica condizioni e proposizioni che vanno mantenute come vere, altrimenti il sistema descritto dal modello non è valido. Alcuni tipi di vincoli sono predefiniti in UML, altri vengono definiti dall'utilizzatore. Un vincolo definito dall'utilizzatore è descritto con parole in un dato linguaggio (ad esempio OCL [Paragrafo 3.6]), racchiuse tra parentesi graffe ("{}"), la cui sintassi e interpretazione è responsabilità di un tool. Un vincolo rappresenta un'informazione semantica attribuita a un elemento del modello, non a una vista di esso, un predicato che riduce le variazioni consentite a livello semantico, e che può essere associato a qualsiasi metaclasse o stereotipo. Un commento è una stringa di testo (inclusi riferimenti a documenti *human-readable*) attribuita direttamente a un elemento del modello. Questo è sintatticamente equivalente a un vincolo scritto in linguaggio testo, il cui significato è interpretabile dagli utenti umani, ma che non è concettualmente

eseguibile. Un commento può così assegnare arbitrariamente informazione testuale a ogni elemento del modello di presunta importanza generale. I vincoli sono caratterizzati da due aspetti fondamentali: constano di espressioni formali o non formali e non devono contraddire le semantiche di base ereditate.

3.2.3.2 Tagged value

Gli utenti, per aggiungere informazioni addizionali utili per implementare il modello, possono definire nuove proprietà di elementi usando il meccanismo dei tagged value, singoli modificatori con una semantica definita dall'utente, che possono essere associati ad ogni metaclasse o stereotipo. Tipicamente vengono usati per modellare gli attributi di uno stereotipo, ma possono anche essere usati indipendentemente dagli stereotipi. Una stringa può essere usata per visualizzare proprietà attribuite a un elemento di modellazione. Questo include proprietà rappresentate da attributi nel metamodello, così come tagged value, sia predefiniti che definiti dall'utente. Notiamo che viene qui usato il termine "proprietà" in senso generale, per indicare ogni valore attribuito a un elemento del modello, inclusi attributi, associazioni, e tagged value. Un tagged value è una coppia (keyword/tag, valore), che può essere associata a ogni tipo di elemento di modellazione. Ogni tag rappresenta un particolare tipo di proprietà applicabile a uno o più tipi di elementi di modellazione. Sia il tag che il valore sono codificati come stringhe. Una proprietà (attributo del metamodello o tagged value) è visualizzata come una sequenza di "property specifications", delimitate da una virgola, tutte racchiuse in parentesi graffe, con la seguente forma:

keyword = valore,

dove:
keyword: nome della proprietà, attributo del metamodello o tag arbitrario
valore: stringa arbitraria rappresentante il valore della proprietà

Se il tipo della proprietà è booleano, nel caso il valore fosse omesso, il valore di default è "true".

Esempi:

{ author = "Joe Smith", deadline = 31 March 1997, status = analysis }
{ abstract }

3.2.3.3 Stereotipi

Uno stereotipo costituisce un elemento di modellazione che definisce valori addizionali (basati su definizioni di tag), vincoli addizionali e, eventualmente, una nuova rappresentazione grafica. Gli stereotipi sono utilizzati per raffinare metaclassi (o altri stereotipi) definendo semantiche supplementari. Tutti gli elementi di modellazione che sono marcati da uno o più stereotipi ricevono questi valori e vincoli, che si aggiungono agli attributi, associazioni e superclassi che questi elementi hanno nello standard UML. Uno stereotipo, in effetti, è una nuova classe di elemento del metamodello, che viene introdotta a tempo di modellazione. Esso rappresenta una sottoclasse di un elemento del metamodello esistente, con la stessa forma (attributi e relazioni), ma con un intento diverso. Generalmente uno stereotipo rappresenta una "distinzione di utilizzo", utilizzata per definire modelli di elementi specializzati, basati su un modello UML di base Un elemento stereotipato può avere vincoli addizionali rispetto alla classe base del metamodello.

Gli stereotipi si usano quando:

- I vincoli semantici addizionali non possono essere specificati con gli strumenti di modellazione standard UML
- Le semantiche addizionali hanno significati al di là degli obiettivi dell'UML

Gli stereotipi sono definiti da:

- Metaclassi di base (o stereotipi):
  Quale elemento viene specializzato?
- Vincoli:
  Cosa caratterizza questo stereotipo?
- Etichette necessarie (ad es., 0...*):
  Quali/Quanti valori questo stereotipo deve conoscere?
- Icone:
  Come deve apparire questo stereotipo in un modello?

La notazione di uno stereotipo può essere costituita da:

- Virgolette:
  <<keyword>>, dove *keyword* rappresenta lo stereotipo

- Un’icona
- Una forma iconificata

L’icona può essere usata al posto o in aggiunta alla keyword dello stereotipo, come parte del simbolo per l’elemento di modellazione base su cui lo stereotipo è basato. In alternativa, l’intero simbolo dell’elemento di modellazione base viene “collassato” nell’icona contenente il nome dell’elemento o con il nome sopra o sotto l’icona. Le altre informazioni contenute nel simbolo dell’elemento di modellazione base, in questo caso, sono soppresse.

I raffinamenti sono specificati al livello del modello, ma si applicano al livello di metamodellazione, descritto nel paragrafo seguente.

3.3 Il metamodello UML

L’UML è un linguaggio che permette di costruire un modello di un sistema da analizzare o da progettare, ovvero permette di realizzare una rappresentazione astratta di un sistema basata su alcuni concetti fondamentali, come classi, associazioni, etc. Se si descrive il sistema in linguaggio naturale lo si esprime in un modo che può essere ambiguo, non rigoroso. L’UML, invece, definisce un linguaggio che permette di costruire un modello standard di qualsiasi sistema che si voglia descrivere o progettare. I concetti base dell’UML vengono usati anche per descrivere la semantica del linguaggio stesso. Si ha quindi un modello per un linguaggio di modellazione, cioè un metamodello. Tra i metodologi si è ritenuto opportuno definire un linguaggio comune denominato MOF (Meta Object Facility) [41], che costituisce la tecnologia adottata dall’OMG per la definizione di una sintassi astratta comune per la definizione di metamodelli attraverso dei metamodelli. MOF definisce un framework di metamodellazione basato su un’architettura a quattro livelli, schematizzata in figura 10, dove, leggendo dall’alto verso il basso, si assiste ad una graduale diminuzione del livello di astrazione:

Metalivello	Termine MOF	MOF come modello di rappresentazione di ontologie	UML
M3	Meta-metamodello	framework	Modello MOF
M2	Metamodello	Linguaggi di rappresentazione della conoscenza	Metamodello UML
			Istanza del modello MOF
M1	Metadati	Ontologia	Modello/schema UML definito dell'utente
			Istanze del metamodello: Classi
M0	Dati	Database	Sistema reale
			Istanze del modello: Oggetti

Figura 10 - Architettura MOF a quattro livelli

- **Meta-metamodello (M3)**: costituisce l'infrastruttura per l'architettura di modellazione. La responsabilità principale di questo livello è definire un linguaggio per la specifica di metamodelli. Un meta-metamodello definisce un modello che si trova ad un livello di astrazione più alto rispetto ad un metamodello ed è generalmente più compatto dei metamodelli che descrive. Un meta-metamodello può definire molteplici metamodelli. Alcuni dei concetti tipici che si manipolano a questo livello sono quelli di *MetaClasse, MetaAttributo* e *MetaOperazione*.

- **Metamodello (M2**): un metamodello è un'istanza di un meta-metamodello. Questo livello si occupa della definizione di un linguaggio per specificare modelli. Solitamente i metamodelli sono più complessi dei meta-metamodelli da cui sono descritti, specialmente quando definiscono una semantica dinamica. Alcuni concetti tipici di questo livello sono: *Classe*, *Attributo* e *Operazione*. L'UML possiede una definizione rigorosa di metamodello, descritto per mezzo dello UML stesso. Il metamodello dell'UML definisce la struttura dei modelli UML, non fornisce alcuna regola su come esprimere

concetti del mondo object-oriented, quali ad esempio classi, interfacce, relazioni e così via. Così come avviene nella relazione che lega le classi agli oggetti, una volta definita una classe si possono avere svariate istanze della stessa (oggetti), analogamente è possibile progettare un numero infinito di varianti dell'UML (istanze del metamodello), che si conformano al metamodello stesso.

- **Modello (M1)**: un modello è un'istanza di un metamodello. La responsabilità principale di questo livello è definire un linguaggio che descrive un dominio di informazione. Un concetto tipico di questo livello è quello di una *classe specifica*, ad esempio la classe Person, dotata degli attributi Nome ed Età.

- **Oggetti (M0)**: sono le istanze dei modelli e servono a descrivere uno specifico dominio di informazione. Il concetto tipico di questo livello è quello di *oggetto*, inteso come istanza delle classi definite al livello superiore, ad esempio l'oggetto Pippo, istanza della classe Person.

Il Modello MOF è definito utilizzando se stesso come linguaggio di modellazione, poiché è posto al livello più alto. In altri termini, il Modello MOF è metamodello di se stesso. MOF definisce tre concetti di base per la modellazione di metadati: Classe, Associazione e Package. Il loro significato è analogo a quello che ritroviamo nel linguaggio UML, con alcune semplificazioni. Altri concetti chiave definiti da MOF sono quelli di DataType e Constraint. I primi vengono utilizzati per definire il tipo di un attributo, i secondi esprimono vincoli semantici che determinano la correttezza di un metamodello. I vincoli sono espressi sia in linguaggio naturale che in OCL.

- Una Classe può comprendere Attributi e Operazioni, su cui è possibile definire vincoli di unicità e cardinalità. Inoltre, le Classi supportano l'ereditarietà multipla.

- Le Associazioni sono sempre binarie e consentono di specificare l'ordinamento, il tipo di aggregazione, la cardinalità.

- Un Package è una collezione di Classi e Associazioni. Inoltre, un Package può essere composto da altri Package.

3.3.1 Architettura del metamodello

La specifica ufficiale dell'OMG [38] descrive la semantica di UML scomponendo l'architettura in package. All'interno di ogni package gli elementi del modello sono definiti nei seguenti termini in modo semi-formale:

- **Sintassi astratta**: si presenta, attraverso diagrammi di classe espressi con la notazione UML, il metamodello UML, i suoi concetti (cioè le metaclassi), le sue relazioni e vincoli. A questi si aggiungono parti di testo scritto in linguaggio naturale (inglese).

- **Semantica**: è fornita in linguaggio naturale, comprende la descrizione degli elementi che compongono il metamodello UML e delle loro relazioni.

- **Well-formedness rules**: sono le regole ed i vincoli per definire modelli che siano validi. Questi sono espressi utilizzando sia un linguaggio formale, OCL, sia il linguaggio naturale.

La complessità del metamodello UML è gestita organizzandolo in tre package: Foundation, Behavioral Elements e Model Management. I primi due sono ulteriormente decomposti in package, ognuno dei quali contiene elementi semanticamente correlati.

Riportiamo una sintetica descrizione di ogni package:

Behavioral Elements: questo package specifica la struttura necessaria alla definizione dei comportamenti di dinamici (behavior) di un modello. È composto da cinque subpackage.

Foundation: questo package rappresenta l'infrastruttura del linguaggio che specifica la struttura statica dei modelli. È suddiviso in tre subpackage.

Model Management: questo package definisce, tra gli altri, gli elementi Model, Package e Subsystem, necessari ad organizzare diversi modelli e a raggruppare elementi che presentano caratteristiche in comune.

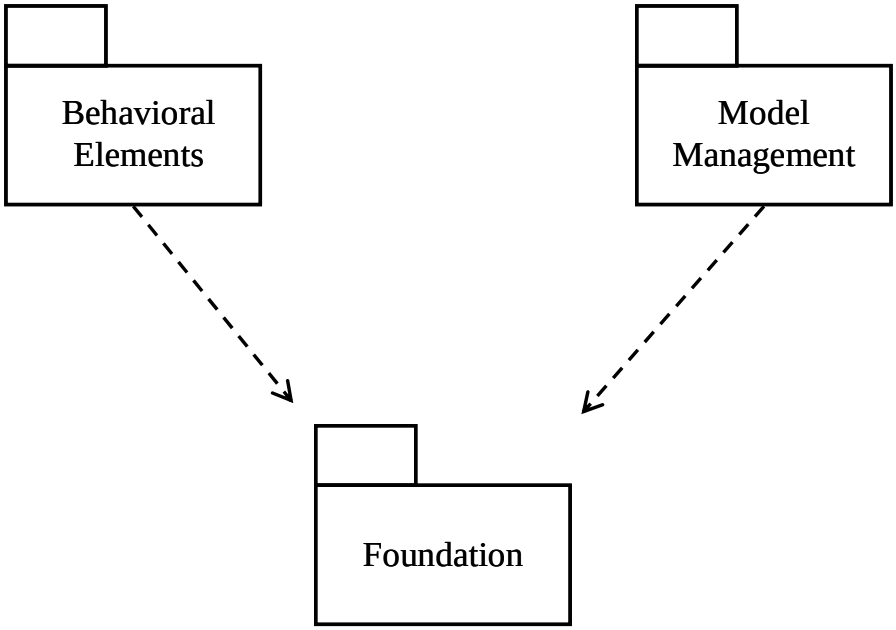

Figura 11 - Struttura dei package del metamodello di UML

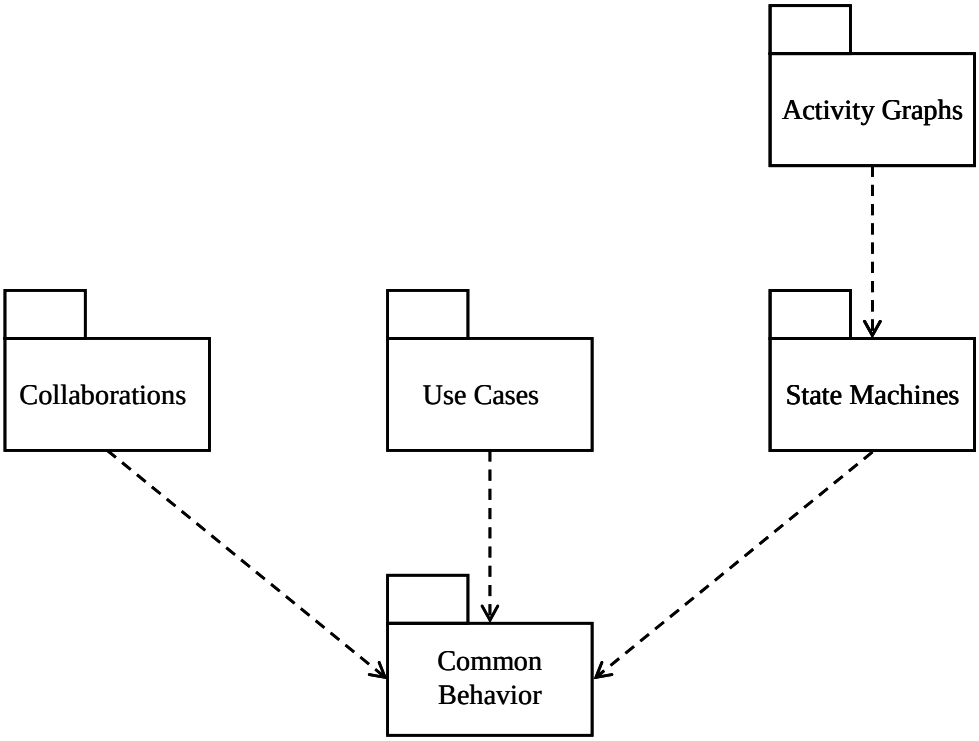

Figura 12 - Behavioral Elements Packages

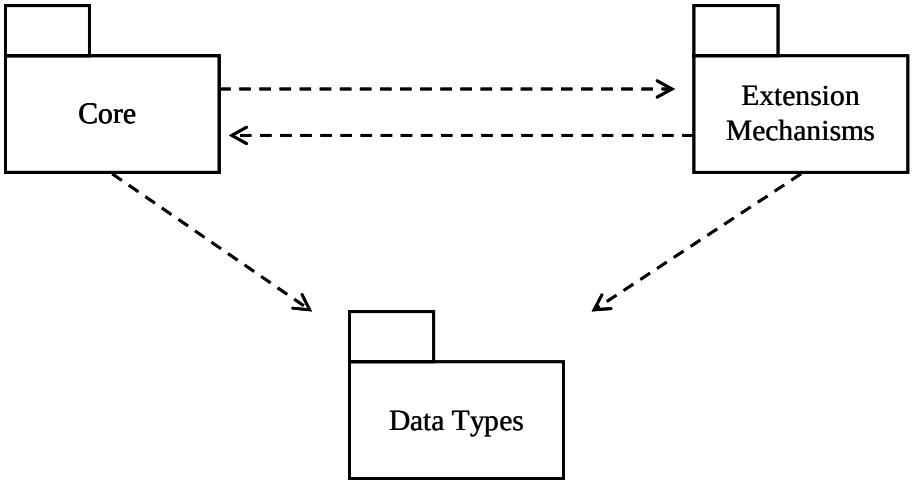

Figura 13 - Foundation Packages

Riportiamo una sintetica descrizione di ogni subpackage:

Common Behavior: è il package più importante, poiché definisce i concetti di base per modellare comportamenti dinamici. Esso fornisce l'infrastruttura di supporto per i package Collaborations, Use Cases e State Machines.

Collaborations: questo package specifica i concetti necessari ad esprimere il modo secondo il quale i diversi elementi di un modello interagiscono tra loro da un punto di vista strutturale. Una Collaborazione è la definizione di un modo di utilizzare gli elementi di un modello e le associazioni tra di essi per l'esecuzione di un determinato compito.

Use Cases: è il package utilizzato per specificare i concetti utilizzati nella definizione dei casi d'uso di un sistema. Include le metaclassi Actor e UseCase.

State Machines: questo package specifica l'insieme dei concetti che possono essere utilizzati per modellare le transizioni di stato che coinvolgono le diverse parti di un sistema. Include le definizioni dei concetti di Evento, Stato e Transizione.

Activity Graphs: estende il package State Machines aggiungendo nuovi concetti. Entrambi i package definiscono i concetti necessari a modellare transizioni di stato e condividono molti

elementi del metamodello. Questo package definisce i concetti che sono specifici dei diagrammi di attività, come quelli di Activity e Activity graph.

Core: è il package più importante tra quelli che compongono il package Foundation. Definisce le metaclassi di base del metamodello, necessarie per definire un modello ad oggetti.

Data Types: questo package contiene la definizione dei tipi di dato che sono usati per definire l'UML. Nel metamodello essi vengono utilizzati per dichiarare il tipo di un attributo di una metaclasse. Questo package ha una struttura più semplice degli altri package perché si suppone che la semantica di questi concetti base sia ben conosciuta. Nei diagrammi i DataType appaiono come stringhe. Ogni occorrenza di un particolare nome di un tipo dei dati denota sempre lo stesso tipo dei dati. Da notare che questi sono i tipi di dato che vengono usati per definire il metamodello UML, e non i dati che sono usati da un utente dell'UML. Questi ultimi saranno istanze delle metaclassi DataType definite nel metamodello.

Extension Mechanisms: questo package definisce il modo secondo il quale gli elementi di base di un modello possono essere estesi introducendo nuovi concetti semantici come Stereotype, Constraint e Tagged Value [Paragrafo 3.2.3].

3.3.2 Core

Il package Core è il sottopackage principale di quelli che, nel loro insieme, compongono il package Foundation dell'UML. Questo definisce i costrutti base astratti e concreti del metamodello necessari per lo sviluppo dei modelli degli oggetti. I costrutti astratti non sono istanziabili e sono comunemente usati per descrivere costrutti chiave, strutture condivise e per organizzare il metamodello dello UML. I costrutti concreti sono invece istanziabili e corrispondono ai costrutti che i disegnatori di modelli ad oggetti usano per il proprio lavoro. I costrutti astratti definiti nel package Core comprendono il ModelElement, il GeneralizableElement, il Classifier, etc. I costrutti concreti del Core comprendono Class, Attribute, Operation, Association, etc. Nei diagrammi le classi astratte hanno il nome scritto in corsivo. Il package Core specifica i costrutti chiave necessari per costruire un metamodello base e definisce uno scheletro (backbone) dell'architettura per supportare costrutti addizionali al linguaggio. Inoltre il package Core è sufficiente da solo a definire il resto della specifica UML, quindi costituisce la struttura principale del suo metamodello. In altri package il Core è esteso aggiungendo metaclassi allo scheletro usando gerarchie di specializzazione e associazioni. I

diagrammi delle figure sottostanti descrivono graficamente la sintassi astratta delle parti del package Core relative allo scheletro (Backbone) e alle relazioni (Relationships). Fanno parte del package Core anche le dipendenze (Dependencies), i classificatori (Classifiers) e gli elementi ausiliari (Auxiliary Elements) [38].

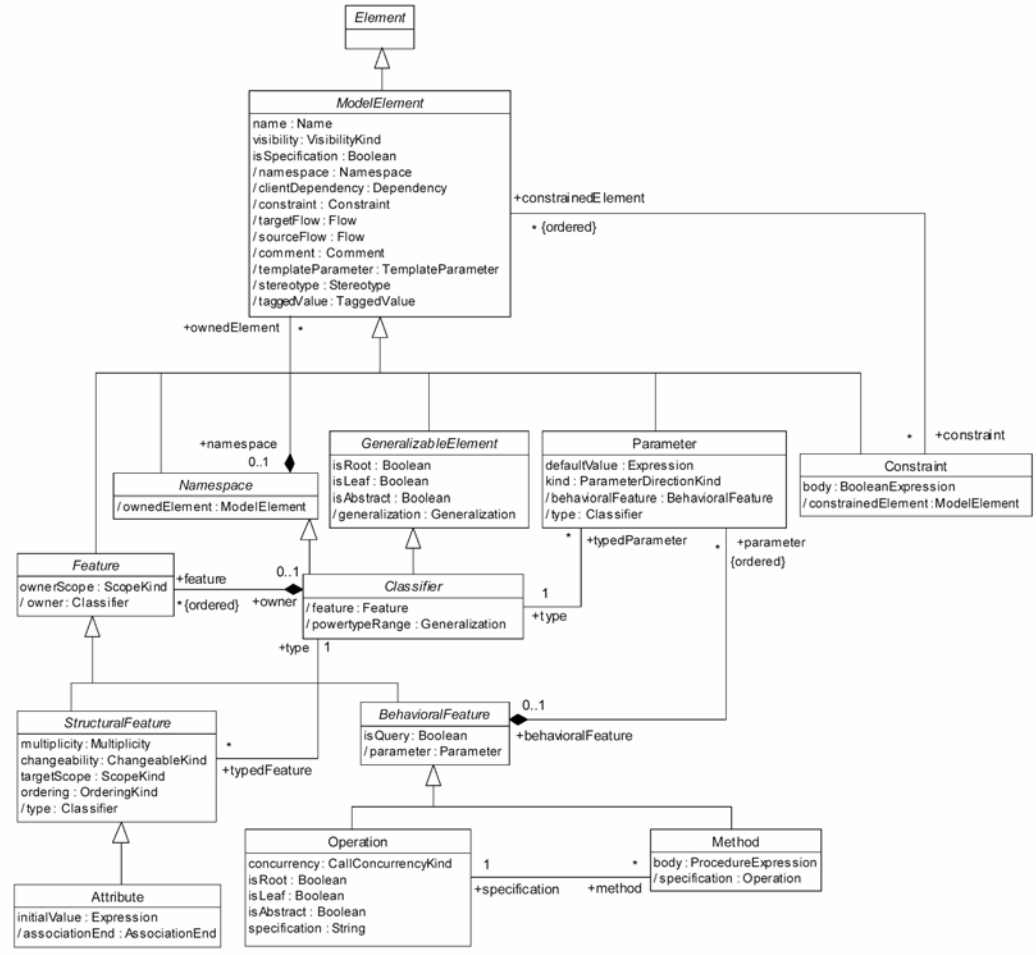

Figura 14 - Il package Core: scheletro (Backbone)

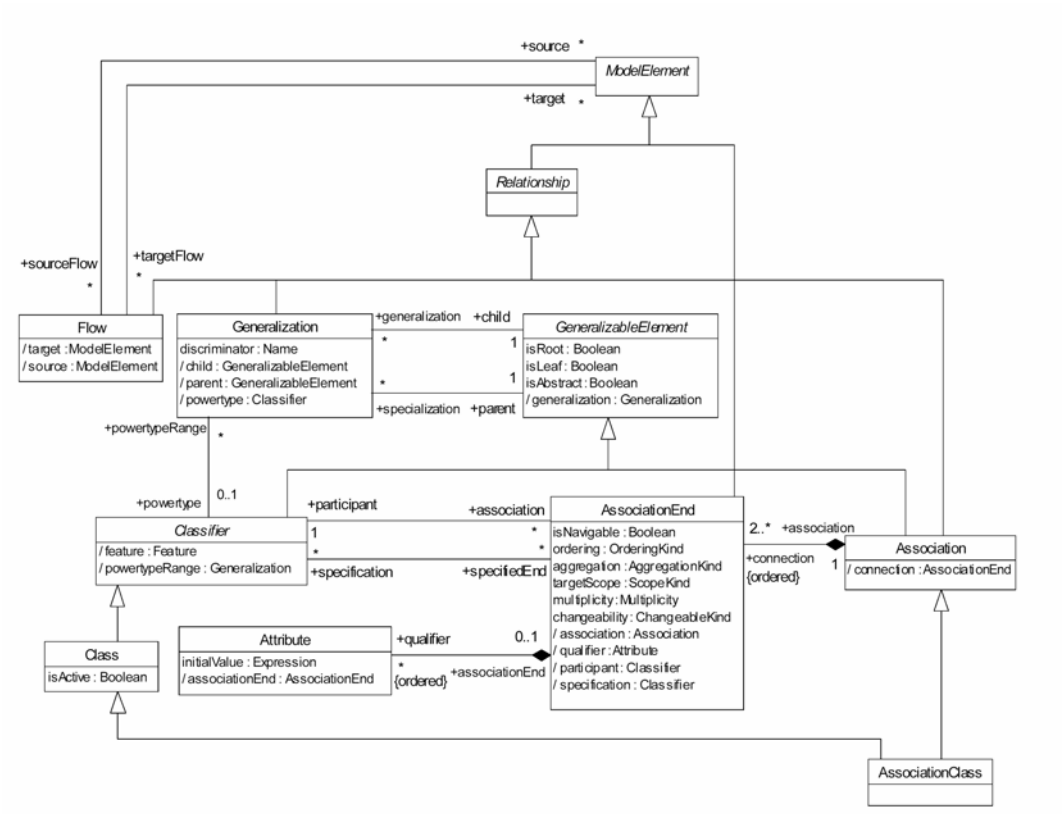

Figura 15 - Il package Core: relazioni (Relationships)

3.4 XMI

XMI (XML Metamodel Interchange) [14] è un formato di scambio per metadati basato su XML, proposto dall'OMG. Esso consente di generare una DTD partendo da un metamodello, mediante l'applicazione di determinate regole di trasformazione. Poiché XMI viene proposto come un formato di scambio per metadati e MOF è la tecnologia adottata dall'OMG per rappresentare metadati, è naturale che l'architettura di riferimento su cui si basa XMI sia proprio MOF. Infatti, XMI rappresenta la tecnologia in grado di fornire un supporto per la serializzazione di metamodelli definiti mediante MOF, mappando il metamodello MOF in una DTD. Il metamodello UML è definito come uno dei quattro livelli di MOF, quindi è possibile utilizzare XMI per la produzione di una DTD, relativa al metamodello UML, per lo scambio di modelli

UML. Una DTD [Paragrafo 1.4] definisce la grammatica a cui un documento XML deve aderire per essere sintatticamente corretto. Mediante una DTD non è però possibile esprimere tutti i vincoli necessari a stabilire se un documento XML è semanticamente corretto. Di conseguenza, la verifica della correttezza semantica è compito di chi importa il documento, solitamente un tool specifico che deve conoscere il metamodello di riferimento. Diversi case-tool basati su UML hanno integrato al loro interno il supporto per XMI. Tuttavia, allo stato attuale, l'unico strumento che supporta il formato XMI esattamente come specificato è ArgoUML [Paragrafo 3.5], che usa XMI per salvare le informazioni relative ad un modello UML. In linea di principio, XMI consente di importare un modello creato con ArgoUML all'interno di altri tool che supportano il formato XMI. Purtroppo, nella realtà questo non è sempre possibile. In primo luogo XMI è uno standard recente e ArgoUML è tra i primi tool ad adottarlo, inoltre XMI non fornisce il supporto per salvare le informazioni grafiche relative ad un modello UML.

3.5 Case-tool

I case-tool sono quegli strumenti che hanno lo scopo di supportare il processo di produzione del software, fornendo allo sviluppatore una serie di strumenti utili, in grado di facilitare lo svolgimento di alcuni lavori. La parola "case" è l'acronimo di "Computer Aided (o Assisted) Software Engineering" e si riferisce a quella disciplina che si occupa di tali strumenti. Alcuni case-tool supportano tutte le fasi di un processo produttivo, tuttavia la maggior parte di essi fornisce sostegno solo per alcune fasi, in particolare quelle di analisi, progettazione ed implementazione. Nel corso degli anni, sono state definite alcune caratteristiche che un case-tool ideale dovrebbe fornire. Ne riportiamo una breve descrizione di seguito.

Caratteristiche architetturali:

- Utilizzo di interfacce standard: poiché la maggior parte dei case-tool non supporta tutte le fasi del processo produttivo, sarebbe auspicabile che applicazioni diverse possano condividere informazioni mediante un formato di scambio comune.

- Espandibilità: un case-tool dovrebbe basarsi su un'architettura aperta che consenta di aggiungere nuove funzionalità senza dover ricorrere ad una reimplementazione.

- Portabilità: se un case-tool è in grado di funzionare su piattaforme differenti, è possibile cambiare ambiente di sviluppo ed utilizzare ancora quel tool, evitando di dover imparare ad utilizzare altri strumenti.

Caratteristiche funzionali:

- Utilizzo di un repository (o dizionario dei dati): è uno degli strumenti più importanti di un case-tool. Esso contiene la lista di tutti i dati relativi agli elementi che appartengono al sistema che si sta progettando.

- Consistency checking: se esiste un dizionario dei dati, è possibile verificarne la consistenza attraverso l'analisi semantica dei dati contenuti in esso. L'analisi consiste nel verificare che tutte le dipendenze, le relazioni, i vincoli e le condizioni imposte nel sistema siano rispettate.

- Report generation e screen generation: sono strumenti detti di "rapid prototyping", che permettono di generare codice in maniera automatica per realizzare alcune porzioni del sistema, come la generazione di report (report generation) o le interfacce per l'inserimento di dati (screen generation).

- Supporto per notazioni grafiche: un case-tool dovrebbe fornire il supporto per le notazioni più comuni, in particolare UML.

- Supporto per il riuso: mediante delle librerie di oggetti e componenti che possano essere facilmente integrati col sistema che si sta progettando.

- Controllo delle versioni: durante il ciclo di vita di un prodotto, vengono prodotte diverse versioni del sistema. Risulta quindi indispensabile uno strumento per la gestione delle versioni, delle revisioni e delle variazioni.

- Web Publishing: il World Wide Web rappresenta lo strumento principale per la diffusione delle informazioni, quindi riteniamo che sia di fondamentale importanza fornire la possibilità di pubblicare su Web la documentazione prodotta.

Il case-tool più popolare e più utilizzato è senza dubbio Rational Rose, prodotto dalla Rational Software Corporation. È uno strumento, basato su UML, per l'analisi e la progettazione di sistemi software object-oriented. Un altro case-tool molto noto è ArgoUML, un tool open-source, sviluppato presso l'Università della California, che presenta diversi caratteristiche interessanti e innovative. Entrambi comunque si propongono di fornire un efficace sostegno durante le fasi di analisi e progettazione di un sistema software object-oriented. Vengono descritti proprio questi due tool per diverse ragioni. Rational Rose è senza dubbio lo strumento più popolare, più utilizzato e tra quelli che offrono il maggior numero di funzionalità. ArgoUML è stato scelto perché presenta diverse caratteristiche fortemente innovative che lo rendono unico, differenziandolo in maniera sostanziale dalla maggior parte dei case-tool tradizionali. Inoltre è un prodotto open-source, che consente a chiunque di contribuire al suo miglioramento. Due funzionalità di Rational Rose e ArgoUML verranno prese in particolare considerazione:

- la possibilità di pubblicare su Web la documentazione prodotta
- la possibilità di salvare le informazioni relative ai diagrammi UML nel formato XMI [Paragrafo 3.4]

Un altro tool basato su UML è la Telelogic Tau UML Suite [42].

3.5.1 Rational Rose

Rational Rose [43], prodotto dalla Rational Software Corporation, è senza dubbio il case-tool commerciale più popolare e più utilizzato. È basato sul linguaggio UML, infatti, in origine fu concepito dagli stessi autori di UML come il primo strumento per la modellazione visuale di diagrammi UML. Tuttavia occorre sottolineare che Rational Rose, al contrario di ArgoUML, non implementa il metamodello UML esattamente come è specificato, ma utilizza una propria rappresentazione interna dei modelli. La causa principale è da attribuire alla struttura stessa del metamodello UML e della DTD di XMI, che non definisce un modo per salvare le informazioni grafiche relative ad un modello. Rational Rose risolve questo problema estendendo la DTD di XMI, mediante l'introduzione del supporto per salvare informazioni grafiche come il posizionamento di un elemento di un diagramma, il tipo di carattere e i colori utilizzati etc. ArgoUML, invece, risolve il problema disaccoppiando le informazioni grafiche relative ai diagrammi UML da quelle concernenti il modello. Le prime vengono salvate nel formato PGML [Paragrafo 1.4], per le ultime viene utilizzato XMI. Adottando questa soluzione, è possibile utilizzare lo standard XMI esattamente come è specificato, senza doverne estendere la DTD.

Questo rende effettivo l'utilizzo di XMI per lo scopo per il quale è stato concepito: lo scambio di informazioni tra applicazioni differenti. Rational Rose offre una vasta gamma di funzionalità, alcune delle quali molto avanzate. Riportiamo un elenco di quelle principali:

- Supporto per le notazioni Booch '93 e OMT, oltre ad UML
- Generazione di codice C++, Java e Ada
- Strumenti di reverse engineering per ottenere diagrammi delle classi da codice C++, Java o Ada
- Strumenti avanzati per la gestione del layout, che consentono di ottimizzare la disposizione degli elementi di un diagramma
- Strumenti per il controllo delle versioni e lo sviluppo di gruppo
- Supporto per CORBA/IDL
- Supporto per Enterprise Java Beans (EJB) e template per lo sviluppo di Servlet e pagine JSP
- Possibilità di pubblicare su Web tutta la documentazione relativa ad un progetto

Una funzionalità recentemente introdotta all'interno di Rational Rose riguarda la pubblicazione su Web di tutti i documenti prodotti nel corso dello sviluppo di un sistema. L'interfaccia ipertestuale si presenta suddivisa in tre regioni, realizzate mediante tre frame HTML. Questo approccio ha il vantaggio di utilizzare esclusivamente il linguaggio HTML. Tuttavia riteniamo che presenti diversi svantaggi, legati soprattutto al fatto che le immagini sono documenti statici, che possono essere solo parzialmente attivate in virtù della mappatura associata ad esse. Esiste un altro tool, Elmuth [68] (acronimo di "HyperTextual UML Environment", leggendo da destra a sinistra), che ha l'obiettivo di realizzare un ambiente ipertestuale in cui è possibile visualizzare diagrammi UML, analizzarne il contenuto, verificare vincoli di consistenza e generare codice. Elmuth, contrariamente a Rational Rose, sfrutta tutta la flessibilità del linguaggio Java per consentire di visualizzare documenti "attivi", che possono essere animati o attivati in qualsiasi modo, andando oltre la semplice mappatura delle immagini. Si tratta di un obiettivo ambizioso, realizzato solo in parte, poiché lo sviluppo completo dei requisiti proposti avrebbe richiesto tempi notevoli.

3.5.2 ArgoUML

ArgoUML [44] è un case-tool open-source in corso di sviluppo presso l'Università della California, il cui scopo è fornire un valido supporto all'analisi e alla progettazione di sistemi software object-oriented. ArgoUML è basato direttamente sulla specifica UML, inoltre è l'unico case-tool che implementa il metamodello UML esattamente come è specificato. Come è noto, le attività di analisi e di progettazione di un sistema software sono tra le più complesse e difficili. La maggior parte dei case-tool commerciali fornisce strumenti che riducono al minimo il lavoro manuale necessario a completare queste attività e a trasformare un progetto in codice. L'aiuto fornito dai case-tool più popolari, solitamente, consiste nell'offrire il supporto per le notazioni più utilizzate, per la generazione di codice, per il controllo delle versioni e altro ancora. Tuttavia, solitamente questi tool non forniscono un supporto per un determinato processo di sviluppo, quindi non offrono a chi sviluppa software un aiuto nel prendere decisioni progettuali. Solitamente presentano una pagina bianca iniziale e consentono all'utente di accedere da subito a tutte le funzioni disponibili, lasciandogli il compito di completare le varie fasi del processo di sviluppo. Al contrario, ArgoUML, pur non disponendo di tutte le funzionalità dei principali case-tool commerciali, si distingue per le sue caratteristiche innovative nel guidare gli sviluppatori attraverso le varie fasi di un processo di sviluppo iterativo che si ispira al Rational Unified Process [45]. Anche utilizzando Rational Rose è possibile applicare il Rational Unified Process. Dal punto di vista materiale, esso si presenta come una guida on-line che illustra in maniera dettagliata i ruoli, le attività, gli elaborati che occorre produrre, con l'aiuto di linee guida, esempi etc. Poiché non sono disponibili strumenti specifici per integrare Rational Rose con il Rational Unified Process, non viene imposto nessun vincolo agli sviluppatori che intendono adottare questo processo di sviluppo utilizzando Rational Rose. Al contrario, ArgoUML impedisce di completare alcune attività se altre non sono state portate a termine. Occorre tuttavia notare che, in alcuni casi, questo può risultare piuttosto restrittivo e limitante. ArgoUML si differenzia dalla maggior parte dei case-tool tradizionali per alcune importanti caratteristiche che possono essere riassunte in quattro punti:

- Applicazione di principi derivanti dalla psicologia cognitiva:
  ArgoUML mette in pratica studi teorici nel campo della psicologia cognitiva, allo scopo di fornire strumenti innovativi per aumentare la produttività e supportare al meglio il processo di sviluppo del software nelle fasi di analisi e progettazione object-oriented. Come detto, ArgoUML si propone di guidare lo sviluppatore attraverso le varie fasi di sviluppo di un processo simile al Rational Unified Process.

- Impiego esclusivo di tecnologie standard:
  ArgoUML utilizza esclusivamente tecnologie standard basate su XML per mantenere le informazioni relative ai diagrammi UML: XMI, PGML e SVG [Paragrafo 1.4]. Ne deriva l'indubbio vantaggio di avere interfacce comuni che favoriscono lo scambio di informazioni tra applicazioni differenti.

- 100% Pure Java:
  ArgoUML è scritto interamente in Java 1.2. Il motivo di questa scelta è dovuto in primo luogo alla portabilità del linguaggio.

- Open source:
  ArgoUML è un prodotto open source. Tutti possono ottenere una copia gratuita del codice sorgente, modificarlo e sottoporre gli aggiornamenti apportati al giudizio della comunità degli sviluppatori di ArgoUML.

3.6 OCL

Un diagramma UML, come un class-diagram, tipicamente non permette di esprimere tutti gli aspetti rilevanti di una specifica. C'è la necessità di descrivere vincoli addizionali riguardo gli oggetti del modello. Questi vincoli sono spesso descritti in linguaggio naturale, sotto forma di note, ma questo conduce inevitabilmente ad ambiguità. D'altra parte l'utilizzo dei cosiddetti linguaggi formali ha lo svantaggio che essi sono accessibili da chi possiede un certo background matematico, ma non dai comuni modellatori di sistemi. L'OCL (Object Constraint Language), sviluppato da IBM, ha cercato di risolvere questo gap. Infatti esso si presenta come un linguaggio formale per la specifica di vincoli e, più in generale, di espressioni precise non ambigue, caratterizzato da una grammatica semplice da leggere e da scrivere, favorendo così la leggibilità da parte dei modellatori. OCL è un linguaggio basato esclusivamente su espressioni, che possono essere verificate e valutate da appositi parser. Queste espressioni, che vengono valutate istantaneamente, non hanno effetti collaterali. Quando un'espressione OCL viene valutata ritorna semplicemente un valore e questo non può cambiare lo stato del modello, anche se un'espressione OCL può essere usata per specificare un cambiamento di stato (per esempio, in una post-condition). Dobbiamo ricordare che OCL non è un linguaggio di programmazione, non si può scrivere flusso di controllo, invocare processi o attivare operazioni non di query.

Poiché OCL in primo luogo è un linguaggio di modellazione non tutto è direttamente eseguibile. OCL è un linguaggio "tipato", ogni espressione OCL ha un tipo. Per essere ben formata, un'espressione OCL deve rispettare le regole di conformità tra tipi stabilite dal linguaggio. Ad esempio, un intero non può essere confrontato con una stringa. Ogni Classifier definito in un modello UML rappresenta un tipo OCL. OCL include anche dei tipi predefiniti supplementari. I vincoli di OCL, tipicamente, sono utilizzati:

- Per specificare invarianti per classi e tipi in un modello di classi
- Per specificare invarianti di tipo per stereotipi
- Per descrivere pre/post-condizioni di operazioni e metodi
- Per descrivere guardie
- Come linguaggio di navigazione
- Per specificare vincoli su operazioni

OCL è utilizzato nella specifica di UML per esprimere alcune regole di correttezza (*well-formedness rules*). Solitamente una regola di correttezza include alcune metaclassi del metamodello UML. Mediante un'espressione OCL vengono specificate le proprietà invarianti per quelle metaclassi e vengono definite operazioni addizionali usate nelle regole *well-formedness*. Nella specifica di UML alcune regole di correttezza sono espresse in linguaggio naturale. Per quelle più complesse, che possono risultare ambigue, viene utilizzato OCL. Per mostrare un esempio di utilizzo del linguaggio OCL, riportiamo alcune espressioni, tratte dalla specifica di UML, che esprimono vincoli di correttezza relativi alle associazioni tra classi. L'espressione allConnections ritorna l'insieme di tutti gli estremi di un'associazione.

- Gli estremi di un'associazione (identificati nel metamodello dalla metaclasse AssociationEnd) devono avere un nome univoco all'interno dell'associazione:

  self.allConnections->forAll(r1, r2 | r1.name = r2.name implies r1 = r2)

- Solo un estremo di un'associazione può essere un'aggregazione o una composizione:

  self.allConnections->select(aggregation <#none)->size <= 1

- Se un'associazione ha tre o più estremi, allora nessuno di essi può essere un'aggregazione o una composizione:

```
self.allConnections->size => 3 implies
self.allConnections->forall(aggregation = #none)
```

3.6.1 Sintassi

Descriviamo qui la sintassi del linguaggio OCL [38]. In Appendice 2 si riporta la grammatica del linguaggio OCL, descritta usando la sintassi EBNF.

Relazioni con il metamodello UML

Specifica del contesto UML

Il contesto di un'espressione OCL in un modello UML può essere specificato con una dichiarazione di contesto all'inizio di un'espressione OCL, introdotta dalla keyword *context*.

Self

Ogni espressione OCL è scritta nel contesto di un'istanza di un tipo specifico. La keyword *self* è usata proprio per riferirsi all'istanza contestuale. Ad esempio, se il contesto è Company, *self* si riferisce a un'istanza di Company.

Invarianti

L'espressione OCL può essere parte di un invariante, che è un vincolo spereotipato come un <<invariant>>. Se l'invariante è riferito a un Classifier (tipo), l'espressione OCL invariante deve essere vera per tutte le istanze di quel tipo in ogni momento. Le espressioni invarianti devono dare risultati booleani.

Esempio:

```
context Company inv:
        self.numberOfEmployees > 50
```

dove:
la keyword *context* introduce il contesto per l'espressione OCL, Company,
self è un'istanza di tipo Company,

inv dichiara che il vincolo è <<invariant>>.

In alternativa a self, si può scegliere un nome differente. Gli invarianti, opzionalmente, possono avere un nome. Nel metamodello UML questo nome è un attributo della metaclasse Constraint.

Pre/post-condizioni

L'espressione OCL può essere parte di una pre-condizione o di una post-condizione, che sono vincoli spereotipati come <<precondition>> e <<postcondition>>, associati con un'operazione o un metodo.

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
        pre : param1 > ...
        post: result = ...
```

dove:
pre e *post* denotano gli stereotipi, rispettivamente pre-condizioni e post-condizioni, del vincolo,
param1, param2 etc… sono i nomi dei parametri,
result denota il risultato dell'operazione, se ce n'è uno.

Ad esempio possiamo scrivere:

```
context Person::income(d : Date) : Integer
        post: result = 5000
```

Le pre/post-condizioni, opzionalmente, possono avere dei nomi. Nel metamodello UML questi nomi sono attributi della metaclasse Constraint.

Package di contesto

Per specificare esplicitamente a quale package i vincoli invarianti o di pre/post-condizioni appartengono, questi vincoli vengono racchiusi tra gli statement *package* and *endpackage*:

```
package Package::SubPackage
context X inv:
        ... some invariant ...
context X::operationName(...)
```

pre: ... some precondition ...

endpackage

Espressioni generali

Ogni espressione OCL può essere usata come il valore per un attributo della metaclasse UML Expression o uno dei suoi sottotipi.

Tipi

Ogni espressione OCL è scritta nel contesto di un modello UML. Tutti i Classifier del modello UML sono tipi nelle espressioni OCL attribuite al modello.

Tipi predefiniti

In OCL esiste un numero di tipi predefiniti e sempre utilizzabili dal modellatore, come Real, Integer, Boolean, String, etc. Essi sono indipendenti da ogni object-model e fanno parte della definizione di OCL. OCL definisce una serie di operazioni sui tipi predefiniti.

Tipi di base

- Real
- Integer
- Boolean
- String

a cui si aggiungono:

- OclExpression: ogni espressione OCL stessa è un oggetto nel contesto di OCL. Il tipo dell'espressione è OclExpression. Questo tipo e le sue proprietà sono usati per definire la semantica delle proprietà che hanno un'espressione come uno dei loro parametri (select, collect, forAll, etc.). L'istanza di OclExpression è chiamata *expression*.

- OclType: tutti i tipi definiti in un modello UML, o predefiniti in OCL, hanno un tipo. Questo tipo è un'istanza del tipo OCL chiamato OclType. L'istanza di OclType è chiamata *type*.

- OclAny: è il supertipo base di tutti i tipi nel nodello. I tipi predefiniti OCL Collection non sono sottotipi di OclAny. Le proprietà di OclAny sono disponibili su ogni oggetto in tutte le espressioni OCL. Tutte le classi in un modello UML ereditano tutte le proprietà definite su OclAny. Per evitare confitti di nome tra proprietà nel modello e proprietà ereditate da OclAny, tutti i nomi delle proprietà di OclType cominciano con “ocl”. Si può usare l’operazione oclAsType() per riferirsi esplicitamente alle proprietà OclAny. L’istanza di OclAny è chiamata *object*.

- Enumeration (tipi enumerati): Il tipo OCL Enumeration rappresenta le enumerazioni definite in un modello UML. Le enumerazioni in UML sono DataType e hanno un nome come ogni altro Classifier. Un’enumerazione definisce un numero di letterali che sono i valori possibili dell’enumerazione.

  Esempio:

  context Person inv:
  sex = Sex::male

  dove Sex è il nome di un DataType con valori “male” o “female”.

Tipi Collection

- Collection: è il supertipo astratto di tutti i tipi collezione in OCL.
- Set: insieme di elementi senza duplicati
- Bag: collezione dove sono permessi duplicati, gli elementi non hanno un ordine definito
- Sequenze: collezione dove gli elementi sono ordinati, un elemento può far parte della sequenza più di una volta

  Essi possiedono un gran numero di operazioni predefinite su di essi. Una proprietà di una collezione è accessibile utilizzando il formalismo “->”, seguito dal nome della proprietà.

Espressioni let e vincoli <<definition>>

A volte un'espressione è usata più di una volta in un vincolo. L'espressione *let* permette di definire un attributo o un'operazione che può essere usato nel vincolo. Un'espressione let può essere inclusa in un invariante o in una pre/post-condizione ed è conosciuta solo nel vincolo specifico.

```
context Person inv:
        let income : Integer = self.job.salary->sum()
        let hasTitle(t : String) : Boolean = self.job->exists(title = t) in
        if isUnemployed then
                self.income < 100
        else
                self.income >= 100 and self.hasTitle('manager')
        endif
```

Per riusare variabili/operazioni let si può usare un vincolo con stereotipo <<definition>> (indicato con la keyword *def*), nel quale sono definite le variabili/operazioni let. Il vincolo <<definition>> deve essere attribuito a un Classifier e può contenere solo definizioni let. Tutte le variabili e le operazioni definite nel vincolo <<definition>> sono conosciute nello stesso contesto di ogni proprietà del Classifier. Queste variabili e operazioni sono pseudo-attributi e pseudo-operazioni del Classifier e sono usate in un'espressione OCL allo stesso modo degli attributi e delle operazioni:

```
context Person def:
        let income : Integer = self.job.salary->sum()
        let hasTitle(t : String) : Boolean = self.job->exists(title = t)
```

Conformità di tipi

OCL è un linguaggio "tipato" ed i tipi di base sono organizzati in una gerarchia che determina delle regole di conformità tra i differenti tipi. Ad esempio, un intero non può essere confrontato con una stringa. Un'espressione OCL che rispetta queste regole è valida, altrimenti non è valida e contiene un errore di conformità. Un tipo type1 è conforme a un tipo type2 quando un'istanza di type1 può essere sostituita in ogni posto dove è prevista un'istanza di type2. Le regole di conformità pei i tipi nei class-diagram sono semplici:

- Ogni tipo è conforme a ogni suo sottotipo
- La conformità è transitiva

Re-typing (Casting)

In certe circostanze si può voler usare una proprietà di un oggetto che è stato definita su un sottotipo del tipo dell'oggetto. Poiché questa proprietà non è definita sul tipo in questione, il risultato sarebbe un errore di conformità di tipi. Quando è sicuro che il vero tipo dell'oggetto è il sottotipo, si può "ri-tipare" l'oggetto con l'operazione oclAsType (OclType).

Regole di precedenza

Esiste un ordine di precedenza tra le operazioni, che può essere modificato dall'uso di parentesi:

Keyword

Le keyword in OCL sono parole "riservate", cioè che non possono essere usate dappertutto in un'espressione OCL, come il nome di un package, un tipo o una proprietà. *Context*, *inv*, *pre*, *post*, *implies* etc. sono esempi di keyword.

Commenti

I commenti sono preceduti dal simbolo "--".

Valori indefiniti

Quando un'espressione OCL viene valutata, c'è la possibilità che una o più query nell'espressione siano indefinite. In questo caso l'espressione completa è indefinita. Gli operatori booleani costituiscono due eccezioni (or, and).

Oggetti e proprietà

Le espressioni OCL possono riferirsi a Classifier, come tipi, classi, interfacce, associazioni (che si comportano come tipi) e DataType. Anche tutti gli attributi, gli estremi di associazione, i metodi e le operazioni senza effetti collaterali che sono definiti su questi tipi etc. possono essere utilizzati. In un class-model, un'operazione o un metodo è definito senza effetti collaterali se l'attributo isQuery delle operazioni è true. Consideriamo proprietà:

- un attributo

- un estremo di associazione
- un'operazione con isQuery true
- un metodo con isQuery true

Proprietà

Il valore di una proprietà su un oggetto definito in un class-diagram si indica con:

context AType inv:
    self.property

dove *self.property* è il valore della proprietà *property* sull'oggetto riferito da self.

Proprietà: attributi

Consideriamo:

context Person inv:
    self.age > 0

Self.age è il valore dell'attributo age sulla particolare istanza di Person identificata da self. Il tipo di questa espressione è il tipo dell'attributo age, un integer. Usando attributi, e operazioni definite sui tipi di valore base, si possono esprimere computazioni sul class-model.

Proprietà: operazioni

Le operazioni in OCL possono avere parametri. L'operazione stessa può essere definita da un vincolo di post-condizione. L'oggetto che un'operazione ritorna è indicato da *result*, di tipo integer nell'esempio seguente:

context Person::income (d: Date) : Integer
    post: result = age * 1000

Proprietà: estremi di associazione e navigazione

Partendo da un oggetto specifico, si può navigare un'associazione sul class-diagram per riferirsi ad altri oggetti e alle loro proprietà. Per fare questo, si naviga l'associazione usando l'estremo di associazione opposto:

object.rolename

Il valore di questa espressione è l'insieme degli oggetti dall'altra parte dell'associazione *rolename*.

Il risultato della navigazione può essere:

- un oggetto: se la molteplicità dell'associazione è al massimo 1
- un set di oggetti: se la molteplicità dell'associazione è >1
- una sequenza: se l'associazione possiede il vincolo {ordered}

Se un rolename è mancante a uno degli estremi di associazione, viene usato come rolename il nome del tipo all'estremo di associazione.

Proprietà predefinite su tutti gli oggetti

Ci sono molte proprietà, predefinite in OCL, che si applicano a tutti gli oggetti:

- oclIsTypeOf (t : OclType) : Boolean
  Ritorna vero se il tipo di self e t sono lo stesso.
- oclIsKindOf (t : OclType) : Boolean
  Determina se t è il tipo diretto o uno dei supertipi si un oggetto.
- oclInState (s : OclState) : Boolean
  Ritorna vero se l'oggetto è nello stato s.
- oclIsNew () : Boolean
  Ritorna vero se, usato in una post-condizione, l'oggetto è creato durante l'esecuzione dell'operazione, e non prima.
- oclAsType(t : OclType) : instance of OclType

AllInstances

In OCL è anche possibile definire proprietà sulle classi/tipi stesse, e non solo su istanze di classe. Ad esempio, AllInstances è un'espressione, predefinita in OCL, che ritorna l'insieme di

tutte le istanze del tipo su cui è definita esistenti al momento della valutazione dell'espressione. Se vogliamo essere sicuri che tutte le istanze di Person abbiano nomi unici, scriveremo:

context Person inv:

Person.allInstances->forAll(p1, p2 | p 1 <> p2 implies p1.name <> p2.name)

3.6.2 Il Package OCL Standard

Ogni modello UML che usa vincoli OCL contiene un package standard predefinito chiamato "UML_OCL". Questo package è usato di default in tutti gli altri package per valutare espressioni OCL e contiene tutti i tipi predefiniti OCL e le loro caratteristiche. Il package OCL ha come prerequisiti i package Core e Common Behavior, mentre il package Data Types ne è influenzato. Per estendere i tipi OCL predefiniti, un modellatore può definire un package separato. Il package standard può essere importato, e ogni tipo OCL può essere esteso con nuove caratteristiche. Riassumiamo sinteticamente in figura la struttura del package OCL [46]:

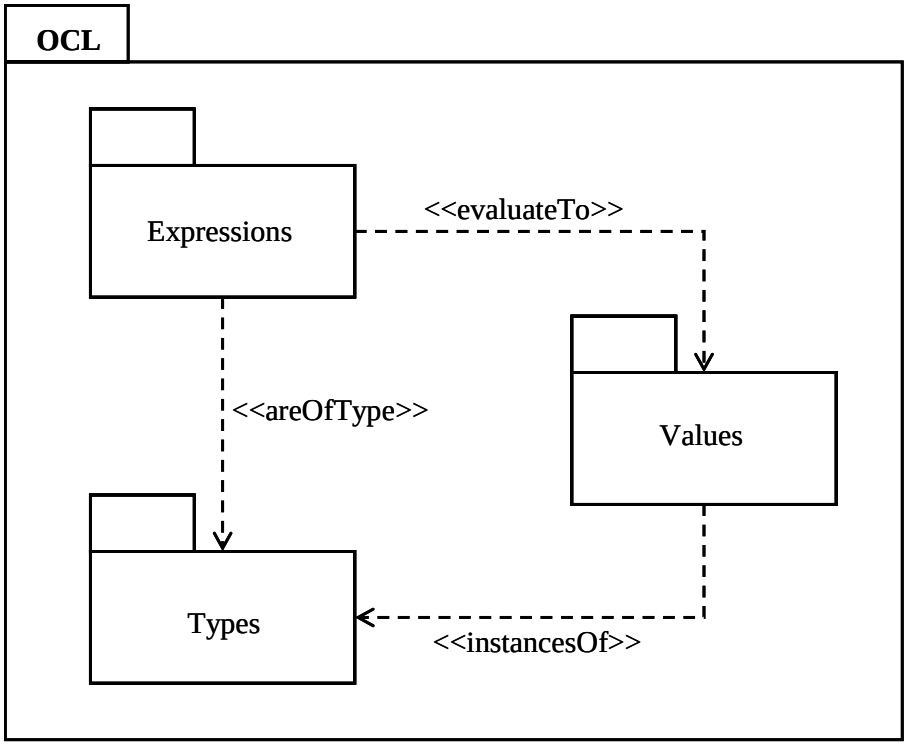

Figura 16 - Package OCL: Struttura

4. UML per lo sviluppo di Ontologie

Le ontologie, nell'ambito del Semantic Web, stanno diventando sempre più importanti, poiché costituiscono le fondamenta, dal punto di vista semantico, per tecnologie che si stanno rapidamente estendendo, come gli agenti software, l'e-commerce e la manipolazione della conoscenza, come già visto nei Capitoli 1 e 2. Un aspetto molto importante per la creazione del Semantic Web consiste nel rendere possibili agli utenti, che non sono esperti di logica, la creazione di ontologie e la realizzazione di contenuti Web che siano leggibili dalle macchine. Per far fronte a queste esigenze il World Wide Web Consortium (W3C) [8] ha progettato una serie di linguaggi che, nel corso del tempo, sono diventati degli standard, e che consentono proprio la strutturazione semantica delle informazioni sul Web e la condivisione dei significati dei concetti tra agenti mobili [Paragrafo 1.4]. L'UML, invece, è stato originariamente disegnato per una comunicazione *human-to-human*, al fine di costruire modelli grafici di sistemi object-oriented, come descritto in dettaglio nel Capitolo 3. In UML le informazioni vengono modellate con class-diagram e vincoli OCL, che possono essere utilizzati anche per sviluppare ed esprimere graficamente complesse ontologie. Quello che si cerca di proporre è un'alternativa a formalismi popolari, come quelli proposti dal W3C, per la rappresentazione di ontologie, costituita dall'uso di un sottoinsieme di formalismi UML associati ad espressioni OCL, un potente strumento per l'espressione di vincoli [Paragrafo 3.6]. La questione che viene posta è se l'UML possa essere visto, oltre che come ormai consolidato elemento di analisi preliminare del software, anche come un efficace linguaggio per rappresentare ontologie, e quindi possa essere inserito nel gruppo dei linguaggi di rappresentazione della conoscenza (Linguaggi KR), insieme ai tradizionali RDF(S), DAML+OIL etc. Un gruppo di ricercatori, tra i quali Kenneth Baclawski, sta cercando di rimuovere le critiche riguardo l'adattabilità dell'UML nella rappresentazione di modelli formali come le ontologie. Queste ricerche sostengono che l'UML è una notazione eccellente nello sviluppo e nel mantenimento di ontologie per i sistemi *agent-based* del Semantic Web. Si sta così cercando di formalizzare ulteriormente l'UML per renderlo sempre più *machine-processable*, in modo che i suoi modelli possano essere usati anche in fase di compilazione e run-time, e non solo come notazione grafica nella comunicazione *human-to-human*, come è stato finora.

4.1 Analisi comparata tra i Linguaggi KR tradizionali e l'UML

Il Web, come già discusso nel paragrafo 1.2, può essere inteso come una base di conoscenza, una fonte di informazioni facilmente reperibili, descritte con un linguaggio di rappresentazione della conoscenza, su cui si possono effettuare interrogazioni. La rappresentazione della conoscenza è uno dei campi dell'Intelligenza Artificiale [3] e si occupa di gestire il modo in cui un programma modella la realtà del mondo in cui è inserito. La conoscenza su una realtà, infatti, per poter essere manipolata deve venire strutturata. Il linguaggio utilizzato per descrivere questa realtà deve essere conciso e non ambiguo. Esso deve indicare, con una sintassi formale e comprensibile alle macchine, come le informazioni sono definite e il tipo di operazioni che sono supportate e deve essere indipendente dal contesto, cioè deve poter raffigurare la conoscenza di qualunque tipo di sistema. I Linguaggi KR sono nati proprio a questo scopo, per descrivere in modo formale le ontologie, cioè la struttura semantica delle informazioni che modellano un dominio di applicazione. L'UML, d'altra parte, è un linguaggio di modellazione universale per dati che andranno a costituire database relativi a un qualunque settore di utilizzo del software. Non ci sorprende, quindi come i Linguaggi KR e l'UML, pur essendo nati con scopi diversi, abbiano molti obiettivi in comune, al di là delle differenze tra i due approcci. Le ragioni per cui l'UML può essere considerato una notazione adatta per la modellazione concettuale di ontologie sono essenzialmente legate alla sua enorme diffusione, alla relativa immediatezza di utilizzo dei suoi strumenti e all'efficienza dei tool basati su di esso, Rational Rose e ArgoUML [Paragrafo 3.5]. L'UML, infatti, è una notazione grafica, standard dell'OMG, basata su anni di esperienza nel campo dell'Ingegneria del Software, largamente adottata nel campo industriale e insegnata in tantissimi corsi universitari. Le tecniche attuali di sviluppo di ontologie, invece, sono basate su rappresentazioni della conoscenza, come KIF [7] e SHIQ [65], che non sono conosciute al di fuori della specifica comunità di ricerca dell'Intelligenza Artificiale, ed i rispettivi Linguaggi KR non hanno ancora nessuna forma grafica standard, essendo linguaggi relativamente recenti. I sistemi *agent-based* industriali, inoltre, spesso necessitano di interagire con sistemi di impresa che possiedono modelli UML già esistenti. I case-tool che si fondano su UML, come Rational Rose o ArgoUML, sono più accessibili degli ontology-tool correnti, come Ontolingua [47] e Protege [48], che richiedono una certa esperienza nel campo della rappresentazione della conoscenza. Bisogna precisare che, esattamente come il Web tradizionale si è sviluppato senza controlli e senza regole, è difficile immaginare che il Semantic Web si evolva come un insieme ordinato di ontologie, costruite da esperti di Intelligenza Artificiale. Soprattutto nei primi tempi, esso si estenderà con un gran numero di ontologie di dimensioni ridotte, che fanno riferimento

l'una all'altra (per riutilizzare o modificare termini comuni) e saranno sviluppate dagli utenti, analogamente a quanto avviene per i contenuti del Web attualmente creati. L'UML possiede dei meccanismi standard per definire estensioni per contesti di applicazione specifici, come gli stereotipi [Paragrafo 3.2.3], ed è provvisto di package e di altri meccanismi di modularità che mancano nella maggior parte dei Linguaggi KR. Un altro aspetto in cui i linguaggi di rappresentazione della conoscenza tradizionali sono carenti è quello che riguarda la possibilità di esprimere processi e comportamenti dinamici. L'UML, invece, oltre ai diagrammi di struttura statica, che vengono utilizzati durante la fase di progettazione di un sistema per descriverne l'architettura (class-diagram e object-diagram), possiede anche dei formalismi appositi per mostrare la sequenza di messaggi scambiati tra diversi oggetti (sequence-diagram e collaboration-diagram), per descrivere gli stati in cui può trovarsi un certo oggetto e i cambiamenti di stato che subisce al verificarsi di determinati eventi (state-chart) e per visualizzare attività che si svolgono in parallelo e possono sincronizzarsi in qualche modo (activity-diagram) [Paragrafo 3.2.2]. Queste potenzialità dell'UML si sposano bene con la concezione del Web di un mondo "aperto" e dinamico, dove le pagine crescono a dismisura ed ognuna contiene una parte infinitesima di quello che può essere raccolto da un agente software, dove i dati disponibili cambiano continuamente e le informazioni possono diventare, di volta in volta, inconsistenti, inaffidabili o non disponibili. L'UML, d'altra parte, non ha ancora una semantica formale, mentre i formalismi come le Description Logics hanno una semantica ben definita [Paragrafo 1.2]. La semantica dell'UML è definita da un metamodello, definito in termini di UML stesso [Paragrafo 3.3], e vincoli addizionali espressi in OCL [Paragrafo 3.6]. Lo sviluppo di una semantica formale per l'UML è un'area di ricerca molto attiva. L'UML, infatti, è un linguaggio molto esteso, con la presenza di molta ridondanza, e la grande varietà di tipi di modelli offerti aumenta la difficoltà nel definirne una semantica davvero uniforme e standardizzata. Il gruppo di ricerca pUML (precise UML) [62] sta cercando di formalizzare un cuore di costrutti di modellazione UML, dai quali gli altri elementi di linguaggio possano essere derivati [Paragrafo 3.3.2]. Come già discusso nel Paragrafo 3.5.1, dedicato al tool Rational Rose, l'UML ha un altro limite, quello di non essere ancora efficacemente "Web-enabled", cioè di non possedere formalismi ottimizzati per essere processato sul Web. La modellazione di ontologie in UML è object-oriented e, attualmente, non possiede una sintassi lineare, RSF(S), DAML+OIL etc., invece, sono basati su frame, la cui sintassi è lineare. Sia l'UML che i linguaggi ontologici sono legati al formalismo XML, ma DAML+OIL, definito in termini di RDF, ha una rappresentazione standard XML, l'OMG, invece, adotta XMI [Paragrafo 3.4], un linguaggio basato su XML, come modello per trasferire dati. La visione di interoperabilità semantica tra gli approcci dei Linguaggi KR e dell'UML è simile, ma il modo di interpretare i concetti è differente. Il problema principale che si riscontra nell'uso dell'UML per mappare i formalismi di rappresentazione di ontologie tipici del W3C consiste nella presenza di significative

incompatibilità semantiche tra l'UML e questi linguaggi di modellazione, descritte in dettaglio nel Paragrafo 4.3, dove vengono anche presentate delle soluzioni di mapping che risolvano le incongruenze riscontrate. In particolare, nell'UML non c'è il concetto primitivo di *Associazione*, le associazioni, cioè, possono esistere solo nel contesto di due o più classi. Nei linguaggi come RDF(S) o DAML+OIL, invece, le proprietà (*Property*) sono elementi primitivi che possono essere definiti in un'ontologia senza riferimenti a classi particolari, sono cioè "first-class objects". Se, ad esempio, un'ontologia RDF(S) o DAML+OIL afferma che: "La compagnia compra un veicolo" e "La persona compra un cane", "compra" è la stessa proprietà. In UML, invece, queste sono due associazioni "compra" distinte. Uno degli obiettivi più importanti del Semantic Web è quello di incorporare regole logiche nelle ontologie. In questo ambito va fatto ancora molto e uno degli obiettivi del progetto DAML, ad esempio, è quello di produrre un linguaggio migliorato, DAML-L (DAML-Logic), che supporti le regole logiche. UML, d'altra parte, include già un potente meccanismo per esprimere regole logiche, l'OCL, che può essere definito essenzialmente come una variante della logica dei predicati di primo ordine usata per scrivere vincoli su strutture ad oggetti. Possiamo affermare che DAML+OIL è il linguaggio di rappresentazione base (analogo ai diagrammi UML), mentre DAML-L fornisce asserzioni logiche (analogamente a OCL per UML). Sintatticamente, OCL è sufficientemente espressivo per rappresentare ogni regola di logica del primo ordine che un disegnatore di ontologie può voler specificare. D'altro canto, semanticamente, OCL manca di una specificazione formale. In quest'ambito si sta cercando di dotare anche OCL di una sovrastruttura semantica [63]. La sintassi object-oriented di OCL, inoltre, è diversa da ogni linguaggio logico comunemente usato e tentare di scrivere regole in OCL può essere molto complicato per chi non ne ha molta esperienza.

4.2. Mapping tra linguaggi di modellazione

Affrontiamo il problema di mappare differenti linguaggi di modellazione. Un mapping può preservare la semantica o meno, se la preserva si parla di equivalenza semantica. Il significato preciso di questa nozione dipende dal linguaggio, ma usualmente ha questa definizione: "Due modelli M1 e M2 sono semanticamente equivalenti se esiste una corrispondenza 1-1 tra le istanze di M1 e le istanze di M2 che preserva le relazioni tra le istanze". Equivalenza semantica tra due modelli significa che i modelli differiscono uno dall'altro per caratteristiche non essenziali, come rinominazioni, riordinazioni o ridondanza. Chiamiamo f la funzione di mapping tra due linguaggi di modellazione L1 e L2, che preserva l'equivalenza semantica. "#M" è la

"size" del modello M, cioè della sua serializzazione. Se M1 e M2 sono modelli semanticamente equivalenti in L1, allora f(M1) e f(M2) sono semanticamente equivalenti. Un mapping può essere definito sull'intero linguaggio di modellazione o parzialmente (*partial mapping*). Nel caso del mapping tra UML e DAML+OIL, descritto nel paragrafo successivo, il mapping è parziale, cioè è definito solo su quegli aspetti dell'UML necessari per esprimere ontologie DAML+OIL. Un mapping può essere *one-way* o *two-way*. In questo secondo caso preserva la semantica in entrambe le direzioni. E' questo il caso del mapping tra UML e DAML+OIL. Un two-way mapping da L1 a L2 è una coppia di funzioni di mapping (f1,f2), con f1 tra L1 e L2, e f2 tra L2 e L1, tale che, se f1 è definita su M, allora f2 è definita su f1(M), e viceversa per f2 e f1. Un mapping two-way è *stabile* se per ogni M su cui è definita f1, f1(f2(f1(M)))=f1(M), e lo stesso per i modelli di L2. Anche se la stabilità di un mapping è desiderabile, non è strettamente necessaria. Un mapping two-way generalmente non è *inverso*, lo è solo quando i linguaggi sono davvero molto simili. E' richiesto, invece, che un mapping two-way sia limitato (*bounded*). In questo caso, per ogni M su cui è definita f1, la sequenza #f1(M), #f2(f1(M)), #f1(f2(f1(M))), etc. è limitata [52].

4.3 Mapping UML-DAML+OIL

Le affinità e le differenze tra UML e DAML+OIL sono state attentamente analizzate da K. Baclawski. Per conciliare una delle più importanti incongruenze, quella relativa al concetto di *Property*, viene proposta una modesta estensione all'infrastruttura UML. L'UML viene inoltre arricchito con degli stereotipi, specializzazioni di costrutti di modellazione UML [Paragrafo 3.2.3.3], che corrispondono a elementi della sintassi DAML+OIL.

Concetto di Property:

Come già accennato, in DAML+OIL, come in tutti i linguaggi di modellazione di ontologie tradizionali, OIL, RDF(S) etc., le proprietà sono elementi primitivi, cioè possono essere definiti in un'ontologia senza riferimenti a classi, e sono solo unidirezionali. In UML, invece, non c'è il concetto primitivo di Associazione. Le associazioni, infatti, possono esistere solo nel contesto di due o più classi, cioè di due o più estremi di associazione. In figura 17 è rappresentata un'associazione ("mother") di cui Person costituisce la classe domain (o souce) e Woman la classe range (o target). La proprietà "mother", in DAML+OIL, esiste indipendentemente dalle due classi, ma non in UML, dove l'associazione "mother" è dipendente dall'esistenza delle due

classi agli estremi di associazione. Per gestire questa incongruenza è stata proposta un'estensione al metamodello UML [Paragrafo 4.3.1]. Un'associazione UML ha una sola classe dominio e una sola classe range, le proprietà DAML+OIL, invece, possono avere più di una classe domain, diretta conseguenza del fatto che una proprietà DAML+OIL è associata a molte classi. Ciò aumenta la flessibilità e la possibilità di riuso, ma si scontra anche con il concetto di modularità tipico dell'UML. Il concetto di Property in DAML+OIL riassume in se sia il concetto di Associazione [Figura 17] che di Attributo dell'UML [Figura 18]. Le proprietà, infatti, assumono valori che possono essere oggetti o semplici letterali. Gli oggetti, inoltre, non devono essere necessariamente nominati esplicitamente, possono anche essere anonimi. Nell'approccio del W3C non c'è una chiara architettura e questo può condurre a situazioni ambigue (ad esempio, una classe può diventare istanza di un'altra classe). L'UML risolve questo aspetto con una netta distinzione tra classi e istanze.

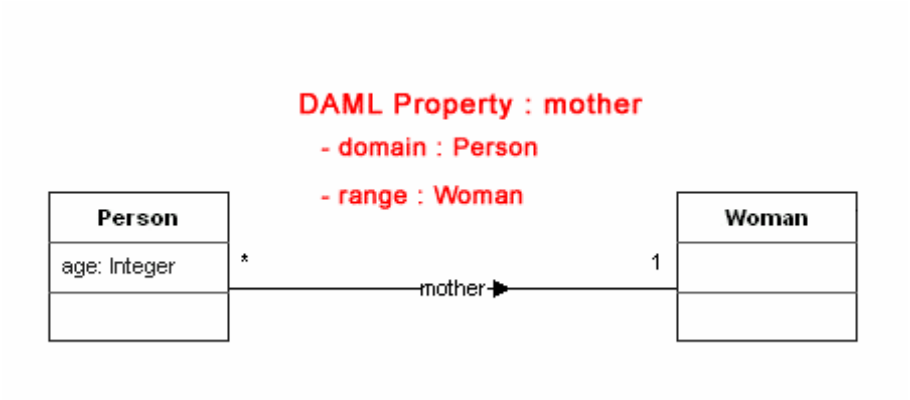

Figura 77 - Associazione-Property

In DAML+OIL possiamo esprimere l'associazione in figura 17 con questa Restriction [Paragrafo 1.4.5.3] sulla classe Person:

```
<daml:Class rdf:ID="Person">
        <daml:label> Person </daml:label>
        <daml:subClassOf>
                <daml:Restriction>
                        <daml:onProperty rdf:resource="#mother"/>
                        <daml:toClass rdf:resource="#Woman"/>
                </daml:Restriction>
```

```
        </daml:subClassOf>
</daml:Class>
<daml:Class rdf:ID="Woman">
        <daml:label> Woman </daml:label>
</daml:Class>
<daml:Property rdf:ID="mother"/>
```

Figura 18 - Attributo-Property

Le subProperty DAML+OIL sono mappate con la dipendenza stereotipata <<subPropertyOf>> tra 2 associazioni. In figura 19 la proprietà “father” è un raffinamento della proprietà “parent”:

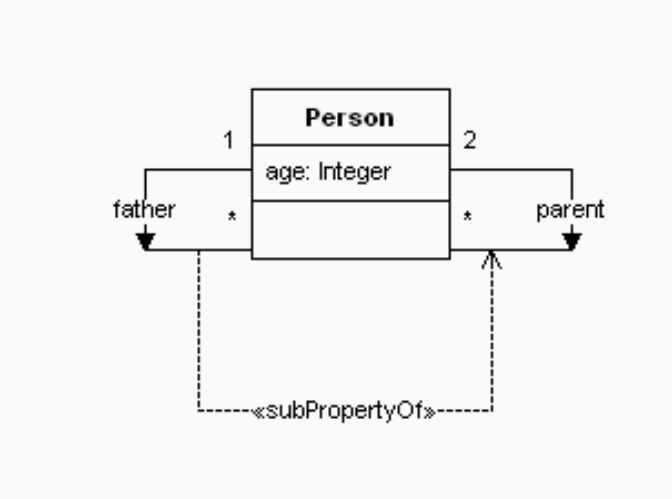

Figura 19 - subProperty

Cardinalità:

I vincoli di molteplicità UML sugli estremi di associazione corrispondono abbastanza fedelmente ai vincoli di cardinalità DAML+OIL. Come si vede in figura 20, un'associazione UML con molteplicità 1 sarà mappata con uno specifico valore di cardinalità nella corrispondente PropertyRestriction. Un'associazione con molteplicità costituita da un range di valori sarà mappata invece con un minimo e un massimo di cardinalità nella corrispondente PropertyRestriction [Paragrafo 1.4.5.3]. Le uniche incompatibilità derivano dal fatto che le proprietà DAML+OIL sono primitive e unidirezionali. La prima caratteristica comporta che, mentre in DAML+OIL si possono specificare i vincoli di cardinalità per ogni domain di una proprietà tutti in una volta, in UML bisogna indicarli separatamente per ciascun estremo di associazione. D'altra parte, mentre UML permette di indicare i vincoli di molteplicità su tutti gli estremi di un'associazione, in DAML+OIL bisogna introdurre una proprietà inversa per indicare un vincolo di cardinalità sul dominio di una proprietà.

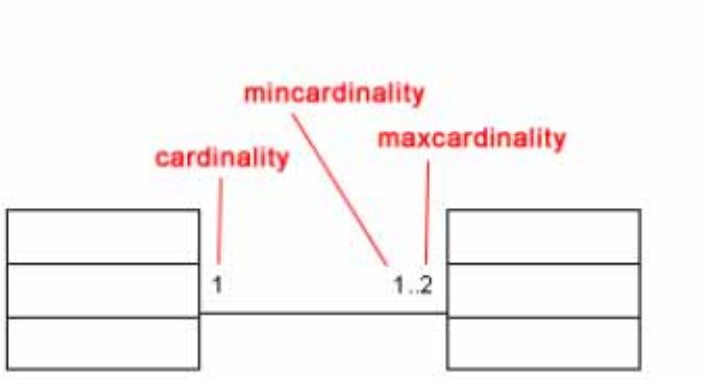

Figura 20 - Cardinalità

Unambiguous e Unique Property:

Una UnambiguousProperty è definita come una relazione che, dato un certo elemento target, viene sempre generata dallo stesso elemento source. Il contrario avviene in una UniqueProperty, come già descritto nel paragrafo 1.4.5.4. Entrambe queste proprietà costituiscono una restrizione di cardinalità. Nel primo caso il valore della proprietà identifica univocamente l'oggetto, nel secondo l'oggetto identifica univocamente il valore della sua proprietà.

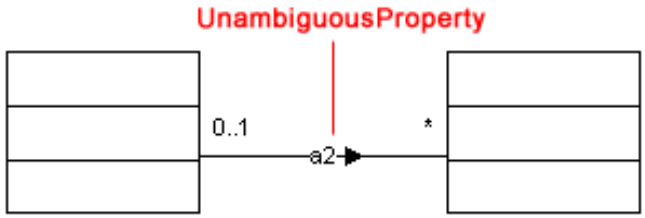

Figura 21 - UnambiguousProperty

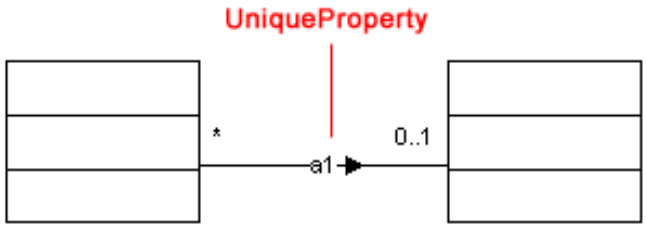

Figura 22 - UniqueProperty

Semantica di vincoli:

Un vincolo UML restringe la possibile istanzializzazione per un dato modello. In DAML+OIL e negli altri linguaggi di rappresentazione della conoscenza, invece, i vincoli sono assiomi dai quali si può ricavare inferenza logica. Ad esempio, supponiamo che in un class-diagram UML ci siano le classi Studente e Dipartimento e un'associazione "specializzazione" che indica il dipartimento nel quale uno studente si sta specializzando. Supponiamo che ci sia un vincolo di cardinalità su "specializzazione" che vincoli uno studente a specializzarsi al massimo in un dipartimento. Ora consideriamo il caso che uno studente si stia specializzando in Informatica e Chimica. In UML questa è una violazione ai vincoli di cardinalità. In DAML+OIL si può invece concludere che Informatica e Chimica sono lo stesso dipartimento.

Riassumiamo il mapping UML-DAML+OIL, che può essere considerato un'estensione al mapping UML-RDF(S):

UML	**DAML+OIL**
Package stereotipato <<Ontology>>	*ontology*
tagged value "versionInfo" su un package	*versionInfo*
<<import>>: stereotipo dipendenza tra package <<ontology>>	*imports*
Associazione binaria/unidirezionale [Fig. 17] Attributo [Fig. 18]	*ObjectProperty* *DataTypeProperty*
Entrambi hanno la nozione di classe con istanze Classe Difficile da rappresentare Relazioni di generalizzazione	 *class* Come sets (disjoint, union) Gerarchia
Nome della classe	*label*
note	*comment*
tagged value "seeAlso"	*seeAlso*
tagged value "isDefinedBy"	*isDefinedBy*
<<instanceOf>>	*type* Nel contesto dell'annotazione
Tipo di modelElement	*type* Nel contesto dell'ontologia
Tipo primitivo di attributo: String, Boolean, Integer, Real…	*Literal*
Valore dell'attributo	Value
Initial value di un attributo	default
Specializzazione/generalizzazione	*subClassOf*
Classe source/domain di un'associazione [Fig. 17] o Classe contenente l'attributo [Fig. 18]	*domain*: classe che dichiara la proprietà *subClassOf* PropertyRestriction
Classe target/range di un'associazione [Fig. 17] o Tipo di attributo [Fig. 18]	*range*: insieme dei valori validi per la proprietà: moltplicità=1 -> oggetto moltplicità>1 -> Bag

	{ordered} -> Sequence *toClass* on PropertyRestriction
Molteplicità [Fig. 20]	Cardinalità
Molteplicità n	*cardinality*=”n”
Molteplicità m…n	*minCardinality*=”m” *maxCardinality*=”n”
0...*	Manca la cardinalità
source di un’associazione con molteplicità 0…1 o 1 [Fig. 21]	*UnambiguousProperty*
target di un’associazione con molteplicità 0…1 o 1 [Fig. 22]	*UniqueProperty*
Dipendenza stereotipata <<subPropertyOf>> tra 2 associazioni [Fig. 19] Semplice gerarchia	*subPropertyOf*
Generalizzazione tra classi stereotipate <<DAMLproperty>> Gerarchia complessa	*subPropertyOf*
Dipendenza stereotipata <<equivalentTo>>	*equivalentTo*
Dipendenza stereotipata <<sameClassAs>> tra 2 classi	*sameClassAs*
Dipendenza stereotipata <<samePropertyAs>> tra 2 associazioni	*samePropertyAs*
dipendenza stereotipata <<inverseOf>> tra 2 associazioni *	*inverseOf*
Stereotipo <<TransitiveProperty>> su un’associazione	*TransitiveProperty*
DataType	DataType
L’UML possiede associazioni ordinate ({ordered}), che definiscono implicitamente una lista nel contesto dell’associazione OCL prevede i tipi Collection Liste e associazioni ordinate sono comunque difficili da mappare.	RDF possiede le nozioni di *Bag*, *Seq* e *Alternative* DAML le ha sostituite con la nozione di *List*

Non sono presenti in UML	RDF(S) e DAML+OIL possiedono classi “universal”: *Resource* *Literal* *Thing*
Visibilità: *Public* (+) *Protected* (#) *Private* (-)	Visibilità: *Public*

4.3.1 Estensione UML

Per conciliare le incompatibilità tra UML e DAM+OIL per quanto riguarda il concetto di *Property*, è stata proposta una modesta estensione al metamodello UML, che prevede di arricchire la specificazione MOF [Paragrafo 3.3] con due nuovi elementi di modellazione: le nozioni di *Property* e *Restriction*. Se si volesse tradurre due associazioni UML con lo stesso nome con la stessa proprietà DAML+OIL si violerebbe il fatto che una proprietà RDF(S) ha solo una classe range, vincolo comunque superato da DAML+OIL. Se si volesse tradurre ogni associazione con una diversa proprietà DAML+OIL si verrebbero a creare molti domain per le proprietà DAML+OIL, che non sarebbero più esprimibili in UML. L’UML ammette associazioni non binarie, mentre in DAM+OIL le proprietà possono solo essere binarie. Quindi un’associazione non binaria in UML non può comunque essere rappresentata con una singola proprietà DAML+OIL, ma va reificata con una classe DAML+OIL con tante proprietà a seconda della cardinalità dell’associazione. Si potrebbe quindi pensare di reificare in ogni caso. Nell’esempio proposto in figura 17 si potrebbe quindi trasformare l’associazione “mother” in una classe Mother. Questo, però, comporterebbe un disegno più complesso, le unità che costituiscono la reificazione non sarebbero più esplicitamente legate l’una all’altra e, inoltre, il mapping automatico risultante, secondo la definizione del paragrafo precedente, sarebbe illimitato (*unbounded*). Consideriamo, infatti, il caso di un mapping UML-DAML+OIL che mappi ogni associazione UML in una classe DAML+OIL, ed ogni estremo di associazione UML in una proprietà DAML+OIL, certamente necessario per classi di associazione e associazioni non binarie. Un’associazione binaria, in questo modo, viene mappata con una classe e due proprietà (f1(M)), che verrebbero mappate indietro in una classe e due associazioni (f2(f1(M))), a loro volta mappate in tre classi e quattro proprietà (f1(f2(f1(M)))), e così via, con un risultato che è evidentemente illimitato. Si vuole invece proporre un mapping UML-DAML+OIL che sia

limitato. Viene così proposta l'estensione all'infrastruttura UML di figura 23. Una *Property* qui non viene intesa come gruppo di associazioni, ma come un'aggregazione di estremi di associazione provenienti da differenti associazioni. La *Property*, in questo caso, può esistere senza essere associata a nessuna classe, ciò è ottenuto imponendo la cardinalità 0…* sull'aggregazione. Ogni estremo di associazione, la cui nozione non ha bisogno di essere modificata, può essere descritto da al più una proprietà (cardinalità 0...1). *Property* è una specializzazione di *Classifier*, quindi un'associazione può essere una specializzazione di un'altra, in accordo con RDF(S), che ammette che una proprietà sia una sottoproprietà di un'altra. Vanno aggiunti vincoli OCL per assicurarsi di non avere specializzazioni senza significato, come proprietà che sono specializzazioni di associazioni, e la semantica della specializzazione di proprietà va specificata con molta attenzione. In più, si può specificare che i Classifier *Property* possono essere solo specializzazioni o generalizzazioni di altri Classifier *Property*. Una *Property*, come in DAML+OIL, può essere vincolata da zero o più *Restriction* (cardinalità 0…*). Le istanze di una restrizione sono gli oggetti che soddisfano una condizione su una o più proprietà (cardinalità 1...*) associate con la restrizione (*onProperty*). Anche le *Restriction* sono Classifier ed ognuna è messa in relazione con almeno una classe (cardinalità 1…*) dall'associazione *toClass*. Analogamente ai Classifier *Property*, anche i Classifier *Restriction* possono essere solo specializzazioni o generalizzazioni di altri Classifier *Restriction*. Il fatto che nessuna *Property* può avere più di uno degli *AssociationEnd* di un'associazione può essere espresso con vincoli OCL.

```
allConnections: Set (AssociationEnd);
allPropConnections:Set (Property);
self. allConnections->intersection(self. allPropConnections:Set(T)):Set(T);
size(#T)<=1
```

dova è stato utilizzato il formalismo *AllConnections*, già visto nel paragrafo 3.6, ed è stata introdotta l'espressione *allPropConnections*, l'insieme di tutti i Classifier *Property* di un'associazione.

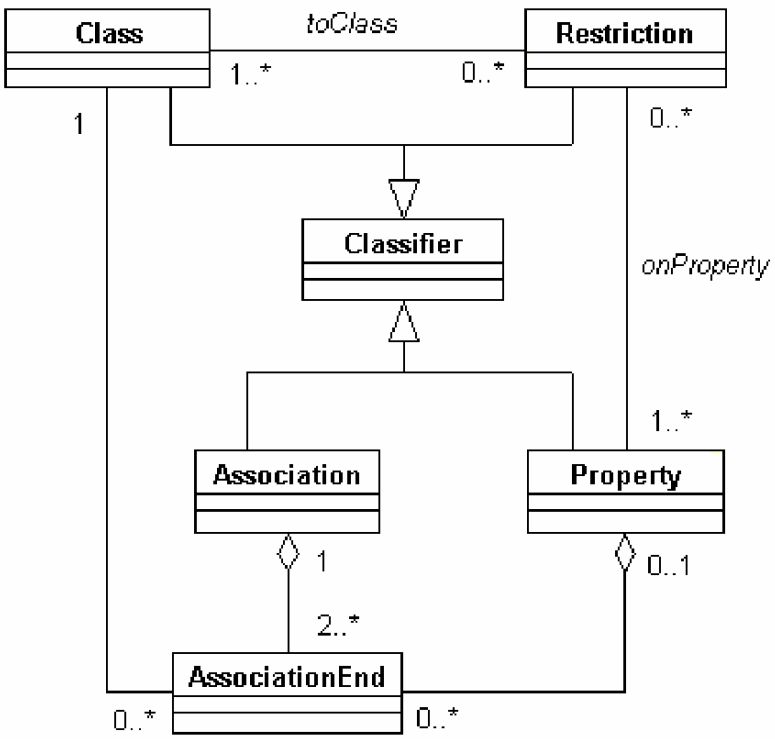

Figura 23 - Estensione UML

4.4 Il Linguaggio UML per le ontologie: Progetti in corso

Ultimamente, molti progetti di ricerca e iniziative commerciali stanno cercando di applicare l'UML nel campo della rappresentazione di ontologie, non solo come front-end grafico per altri linguaggi di rappresentazione della conoscenza (ad esempio DAML+OIL), ma anche direttamente. L'UML è stato applicato inoltre a vari work correlati con le ontologie, come la mappatura e la verifica di consistenza di ontologie. Il progetto UBOT (UML Based Ontology Tool-set), descritto nel paragrafo seguente, è stato sviluppato dalla Lockheed Martin Corporation e sta cercando di utilizzare UML come front-end per visualizzare e editare ontologie. L'approccio è quello di K. Baclawski [Paragrafo 4.3], cioè prevede di estendere l'UML, definendo un prototipo UML per DAML+OIL che mappi le specifiche UML in quelle DAML+OIL. UBOT utilizza tool basati sull'UML (Telelogic Tau UML Suite, Rational Rose,

ArgoUML) per generare un formato XMI, che viene tradotto in DAML+OIL. Il progetto UBOT ha sperimentato anche metodi formali per verificare la consistenza delle ontologie DAML+OIL. A questo scopo è stato sviluppato ConsVISor, un tool che prende in input ontologie UML o DAML+OIL e che, ragionando sulla semantica, identifica dichiarazioni contraddittorie nelle ontologie. Questo tool è utile specialmente nel Semantic Web, dove ci sono ontologie che ne importano altre e ontologie che sono sviluppate dall'estensione di ontologie già esistenti. Stephen Cranefied e Martin Purvis [53] hanno studiato l'uso dei class-diagram UML per rappresentare ontologie e l'uso di object-diagram UML per rappresentare istanze. Cranefield sta promuovendo lo sviluppo di ontologie usando UML e studiando il problema del mapping UML-RDF [Paragrafo 4.4.2]. Ha progettato anche un tool, chiamato "UML-Data-Binding" [56], che prevede la generazione di classi Java e di un RDF-Schema da classi codificate in formato XMI. In questo caso vengono utilizzate le strutture UML già esistenti per descrivere ontologie e l'UML è usato direttamente, non come una sintassi grafica per un altro linguaggio di rappresentazione della conoscenza. Il progetto CODIP (Component for Ontology Driver Information Push) [57] sta lavorando su un tool chiamato DUET (DAML-UML Enhanced Tool), basato su Rational Rose, che studia un profilo UML per DAML. F. Bergenti e A. Poggi [58] hanno proposto un approccio basato sull'uso dell'UML per modellare vari aspetti dei sistemi multi-agente. D. Skogan [59] ha definito un meccanismo per generare una DTD XML da un class-diagram UML e lo ha implementato come uno script per Rational Rose, usato per lo scambio di informazioni geografiche. Il mapping però è definito solo su un sottoinsieme di UML e molti aspetti utili dei class-diagram, incluse le relazioni di generalizzazione, non sono supportati. Il sito Web www.interdataworking.com propone vari sistemi per convertire tra loro diversi formati di dati. Uno, in particolare, può essere usato per produrre uno schema RDF da un documento XMI, sebbene non venga fornita nessuna informazione sul mapping, ne su quante primitive UML siano supportate. Lo schema risultante è definito utilizzando un misto di proprietà e (meta)classi da RDF-Schema e dalla rappresentazione RDF del metamodello UML di S. Melnik [60]. Il tool Xpetal [61] converte i modelli, nel formato di output "petal" del tool di modellazione Rational Rose, in una rappresentazione RDF. Non vengono però forniti dettagli sul mapping tra UML e RDF-Schema, ne su quali aspetti dell'UML vengano supportati. Altri due progetti intendono produrre DTD e XML-Schema da modelli espressi in linguaggi di modellazione per ontologie, rispettivamente Frame Logic e OIL. Il secondo progetto, in particolare, afferma che la nozione di ereditarietà di tipo in XML-Schema non corrisponde bene all'ereditarietà nei modelli object-oriented, corrispondenza che è stata la ragione della scelta di RDF nel progetto di Cranefield qui descritto [54].

4.4.1 UBOT

UBOT (UML Based Ontology Tool-set) [49] è un progetto sviluppato dalla Lockheed Martin Corporation e fa parte del progetto DAML [19]. L'obiettivo è quello di creare un tool-set che supporti:

- Tecniche di modellazione grafica, creazione, estensione e verifica formale di ontologie DAML+OIL
- Processo di estrazione di testo dal linguaggio naturale
- Annotazione DAML+OIL di risorse di informazioni per agenti mobili

UBOT è pensato per quegli utenti che hanno una minima conoscenza riguardo la rappresentazione della conoscenza e la teoria degli agenti mobili. E' fondato, infatti, sull'utilizzo di UML, una notazione grafica largamente conosciuta e adottata, basata su anni di esperienza nel campo dell'analisi del software. Il team UBOT è così strutturato:

- Lockheed Martin Management & Data Systems: architettura, sviluppo e integrazione
- Versatile Information Systems (Northeastern University): verifica formale di UML
- Lockheed Martin Advanced Technology Center: test di DAML+OIL e UBOT
- Kestrel Institute: metodi formali automatici

L'idea è quella di utilizzare UML come front-end per visualizzare e editare ontologie e generare DAML+OIL. L'approccio scelto è quello di K. Baclawski [Paragrafo 4.3], che ha definito un prototipo UML che mappa le specifiche UML in quelle DAML+OIL. UBOT utilizza tool UML per editare e generare un formato XMI, che viene poi tradotto in DAML+OIL. Una nuova ontologia viene modellata graficamente (o viene estesa una già esistente) con una GUI (Grafical User Interface) UML. Possibili GUI UML sono Rational Rose 2000, ArgoUML o Telelogic Tau UML Suite, di cui si è già parlato nel paragrafo 3.5. Un altro obiettivo è quello relativo allo sviluppo di tool che eseguano il controllo di consistenza della conoscenza rappresentata nelle ontologie ed è affidato al progetto VIS (Versatile Information System). A questo scopo è stato sviluppato ConsVISor, tool che prende in input ontologie UML o DAML+OIL e che, ragionando sulla semantica, identifica dichiarazioni contraddittorie nelle ontologie [50]. Questo tool è utile specialmente nel Semantic Web dove ci sono ontologie che ne importano altre e ontologie che sono sviluppate dall'estensione di ontologie già esistenti. Il team del progetto VIS

è costituito da M. Kokar, K. Baclawski, J. Smith e J. Letkowski. Per supportare questa fase di controllo di consistenza si cerca di sfruttare l'uso di WordNet, un database lessicale sviluppato dall'Università di Princeton [32] e utilizzato anche nel progetto MOMIS [Paragrafo 2.3.1]. In WordNet le parole sono raggruppate in set di sinonimi, in tassonomie, che sono facilmente identificabili. Un altro aspetto del progetto UBOT è costituito da AeroDAML [72, 73], un tool di mark-up della conoscenza che applica le tecniche di estrazione di informazioni dal linguaggio naturale, NPL (Natural Language Processing), per generare automaticamente mark-up DAML+OIL dalle pagine Web. L'uso di questo tool prevede semplicemente in entrata un URI e in uscita l'annotazione DAML+OIL per la pagina Web considerata. AeroDAML consiste di un sistema di estrazione delle informazioni chiamato AeroText e di componenti per la generazione di DAML+OIL. AeroText è un sistema di estrazione di dati (text-mining) ad alte prestazioni, fornisce tool grafici avanzati ed è dotato di un'architettuta molto versatile per supportare una grande varietà di analisi del testo. I componenti che devono generare il codice DAML+OIL traducono i risultati dell'estrazione in un modello a triple RDF che utilizza la sintassi DAML. Questo è realizzato riferendosi a un'ontologia di default che è direttamente correlata alla base di conoscenza linguistica usata nel processo di estrazione. Nel passo conclusivo il modello RDF è serializzato per produrre la notazione DAML+OIL risultante. Il livello più basso dell'ontologia di default AeroDAML è basato sulla base di conoscenza comune di AeroText, mentre il livello più alto è basato sulla gerarchia di nomi di WordNet. AeroDAML genera un mark-up che consiste in parole (entità), legate alle ontologie come istanze di classi, e relazioni, legate alle ontologie come istanze di proprietà. Estendere il numero di parole e relazioni annotabili e legarli in un'ontologia semanticamente più ricca renderebbe AeroDAML ancora più flessibile e professionale.

Viene qui schematizzata l'architettuta UBOT:

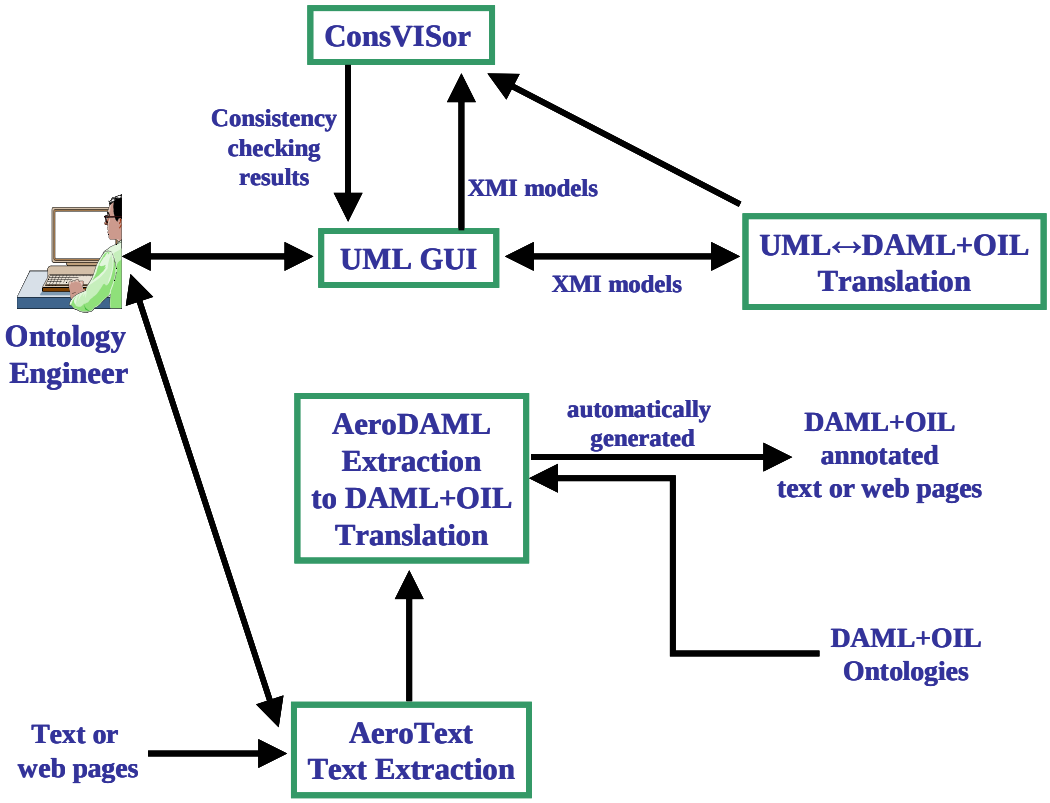

Figura 24 - UBOT: architettura

4.4.2 La proposta di S. Cranefield

Viene qui presentata una tecnologia, proposta da Stephen Cranefield, ricercatore presso l'Università di Otago [54], per supportare l'uso dell'UML al fine di rappresentare ontologie e conoscenza di un dominio ("ontology" e "domain knowledge") nell'ambito del Semantic Web. L'applicazione dell'UML al Sementic Web qui proposta è basata sulle seguenti tre affermazioni:

- I class-diagram UML assicurano una capacità di modellazione statica che si addice bene alle ontologie rappresentative
- Gli object-diagram UML possono essere interpretati come rappresentazioni dichiarative di conoscenza
- Se un'applicazione nel Semantic Web è stata costruita usando tecnologia object-oriented, sarà vantaggioso usare lo stesso paradigma per modellare ontologie e conoscenza

Nella tecnologia presentata, la modellazione di concetti in un dominio (l'ontologia) è realizzata attraverso un linguaggio di modellazione espressivo e standardizzato, l'UML [Capitolo 3], il paradigma di modellazione object-oriented dalla comunità di "software engineering", che possiede formati grafici, una vasta utilizzazione, tool commerciali di alto livello (Rational Rose e ArgoUML [Paragrafo 3.5]), e un linguaggio molto espressivo di espressione di vincoli, l'OCL [Paragrafo 3.6]. Nonostante l'UML sia stato sviluppato per supportare le fasi di analisi e di progettazione nell'ambito dell'Ingegneria del Software, esso ha cominciato ad essere utilizzato anche per altri problemi di modellazione, come abbiamo già visto precedentemente. Per permettere l'uso di rappresentazioni UML di ontologie e di conoscenza servono formati standard per pubblicare sul Web le ontologie e la conoscenza riguardanti gli oggetti del dominio, in modo che possano essere trasmesse tra gli agenti mobili. Servono anche librerie software per aiutare gli scrittori di applicazioni a tradurre e comporre le informazioni serializzate. Per i diagrammi di classe UML si utilizza il formato XMI [Paragrafo 3.4]. XMI, infatti, definisce uno standard per pubblicare modelli UML che è supportato da tool come Rational Rose e ArgoUML, come già descritto più in dettaglio nel paragrafo 3.5. Ci sono inoltre librerie di classi Java, già esistenti o in fase di sviluppo, che forniscono un'interfaccia alle applicazioni per accedere a questa informazione. Non esiste però una tecnologia simile per aiutare le applicazioni Java a costruire, serializzare e leggere codifiche object-oriented di conoscenza che sono state concettualizzate come object-diagram UML. Ciò che viene richiesto è un modo per generare da un'ontologia un formato di codifica specifico per la conoscenza degli oggetti di un certo dominio e una API (Application Programmer Interface) per permettere la creazione, l'importazione e l'esportazione

di quella conoscenza sul Web. Come si vede in figura 25, un operatore disegna graficamente un'ontologia usando un case-tool basato sull'UML, come Rational Rose o ArgoUML, e la salva in formato XMI (con Unisys XMI). L'ontologia viene modellata come un class-diagram UML. I vincoli che non possono essere espressi solamente con gli elementi di modellazione UML vengono espressi in OCL. La rappresentazione dell'ontologia in XMI costituisce l'input per due tipi di mapping, definiti e implementati per produrre, rispettivamente, uno un set di classi e interfacce Java corrispondenti a quelle dell'ontologia, l'altro una rappresentazione dell'ontologia in RDF, usando i concetti di modellazione definiti in RDF-Schema. La generazione di classi Java e dell'RDF-Schema da classi codificate in formato XMI è realizzata da un tool chiamato "UML-Data-Binding" (Cranefield, 2001) [55], che utilizza una coppia di fogli di stile XSLT [Paragrafo 1.4.2.1] nella creazione dei mapping. Le classi Java generate permettono ad un'applicazione di rappresentare conoscenza sugli oggetti nel dominio come strutture dati in memoria, sotto forma di un object-diagram, realizzato come una rete di istanze delle classi generate. Lo schema in RDF generato definisce concetti specifici del dominio a cui un'applicazione può riferirsi quando serializza questa conoscenza usando RDF (nella sua codifica XML) [Paragrafo 1.4.3]. La composizione e scomposizione di reti di oggetti da e verso documenti RDF/XML è svolta da due classi Java appositamente dedicate. Bisogna notare che lo schema RDF generato non contiene tutte le informazioni del modello UML originario. Se un'applicazione vuole accedere all'informazione ontologica nella sua completezza deve usare il documento XMI originale con l'aiuto di una della API Java disponibili che supportano l'elaborazione di modelli UML. Lo scopo di RDF-Schema è quello di definire le risorse RDF corrispondenti a tutte le classi, interfacce, attributi, e associazioni dell'ontologia in modo tale da supportare la serializzazione RDF dell'informazione riguardante le istanze.

Possiamo schematizzare la tecnologia di rappresentazione della conoscenza qui proposta con la seguente figura:

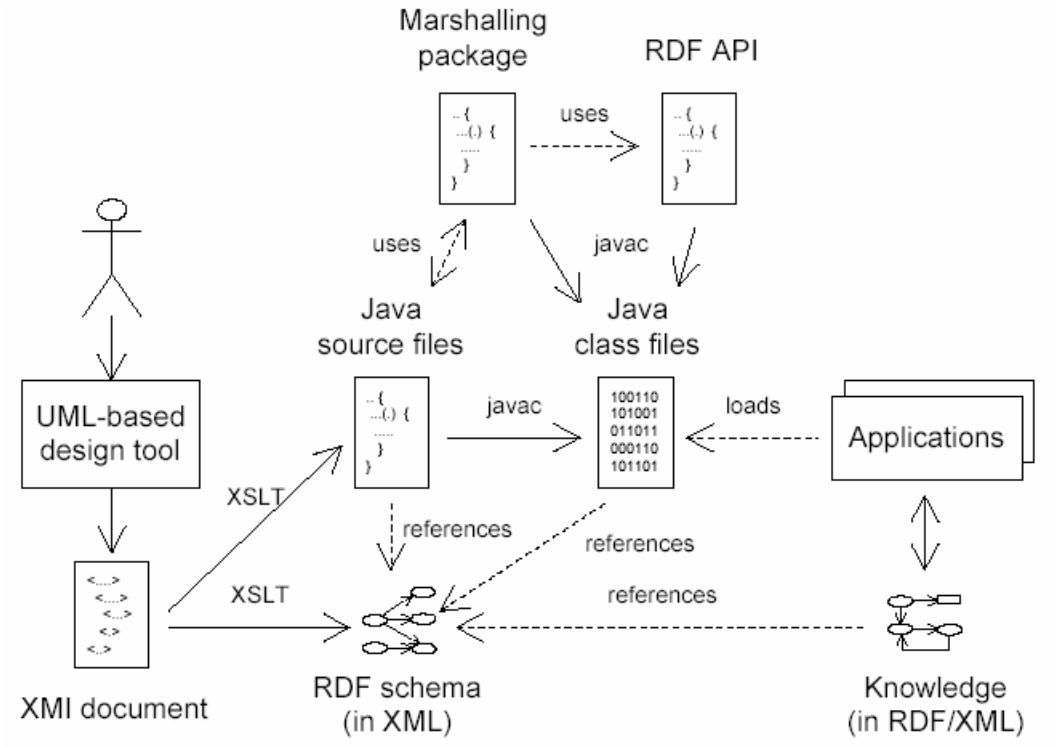

Figura 25 - Tecnologia proposta da Cranefield

In dettaglio, i componenti software utilizzati sono:

- xmi_to_field_model.xsl: stylesheet XSLT che trasforma il file XMI, ottenuto dal modello UML, in una semplice rappresentazione XML corrispondente ai package, alle classi, alle interfacce e ai class-field del modello UML

- field_model_to_java.xsl: un'altro stylesheet XSLT che, dalla rappresentazione XMI del modello UML, genera codice sorgente Java in un formato a file singolo

- redirect.xsl: lo stylesheet che, per produrre file di codice sorgente Java separati per ogni classe e interfaccia, trasforma il codice sorgente Java nel formato a file singolo in file Java separati

- field_model_to_RDF.xsl: stylesheet che trasforma la rappresentazione XMI del modello UML in una rappresentazione dell'ontologia RDF-Schema

- java_type_mapping.xsl: per mappare i tipi OCL in tipi Java

- rdf_type_mapping.xsl: per mappare i tipi primitivi OCL in Literal RDF-Schema

- java_reserved_words.xml: per il checking di field_model_to_java.xsl rispetto a una lista di keyword Java

- field_model.dtd: per la validazione dell'output di xmi_to_field_model.xsl

Il problema da affrontare è quello di tradurre un modello UML, di tipo object-oriented e con differenti tipi di strutture, come classi, attributi, associazioni, associazioni di classe, in un modello RDF, a frame, dove le classi hanno solo proprietà. Le affinità e le incongruenze tra questi due approcci sono già state analizzate nel paragrafo 4.3, dove è stata presentata la soluzione proposta da K. Baclawski. Nel progetto di Cranefield tutti i nomi di classi, interfacce, attributi e associazioni vengono rappresentati con un URI (Uniform Resource Identifier) e ogni nome di proprietà è anticipato da un prefisso rappresentante il nome della classe, in modo che siano uniche per ciascuna classe (ClassURI.FieldName). In presenza di ereditarietà i campi di una classe possono essere rappresentati da proprietà con prefissi differenti, alcuni specificanti la classe stessa, altri una classe genitore. In caso di estremi di associazione senza nomi nel modello UML è necessario generare nomi di default per questi campi. Questo viene risolto dalle regole di navigazione in OCL [Paragrafo 3.6]. Inoltre, è stata costruita un'estensione a RDF-Schema con la proprietà non standard *collectionElementType*, per ovviare al fatto che RDF-Schema non ha un meccanismo per parametrizzare un tipo collezione, come rdf:Bag, dalla classe degli elementi che può contenere. Siccome RDF non prevede un meccanismo per limitare il tipo di elementi in un Bag o in una Sequenze, è stata introdotta questa nuova proprietà:

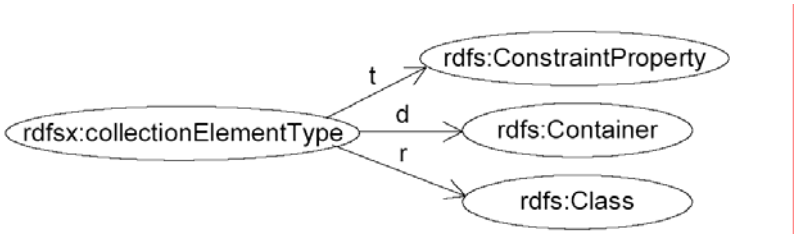

Figura 26 - Estensione a RDF-Schema

dove:

t = rdf:type
d = rdfs:domain
r = rdfs:range
rdfs = http://www.w3.org/2000/01/rdf-schema#
rdfsx = http://nzdis.otago.ac.nz/0_1/rdf-schema-x#

Esistono alcune limitazioni con le correnti implementazioni dei mapping XMI-Java e XMI-schema RDF. I fogli di stile attualmente trattano elementi basilari dei class-diagram, come classi di associazioni, vincoli {ordered} su estremi di associazioni e associazioni one-way. D'altra parte, associazioni n-rie, associazioni qualificate e tipi enumerazione non sono supportati. Non esiste inoltre mapping tra Namespace UML e Package Java. Gli estremi di associazione sono rappresentati da campi con un singolo valore o con un Bag/Sequence di valori in base alla loro molteplicità. Le classi Java generate distinguono tra campi opzionali e obbligatori, ma intervalli di molteplicità che sono specificati nel dettaglio non possono essere rappresentati precisamente. Ci sono altri aspetti avanzati dei class-diagram che non sono trattati, ma questi sono considerati di secondaria importanza per la modellazione di ontologie.

4.4.2.1 Esempio

Nel progetto di Cranefield, come già detto, l'ontologia viene modellata come un class-diagram UML, usando Rational Rose, mentre una parte della conoscenza viene codificata come un object-diagram UML. I vincoli che non possono essere espressi solamente con gli elementi di modellazione UML vengono espressi in OCL:

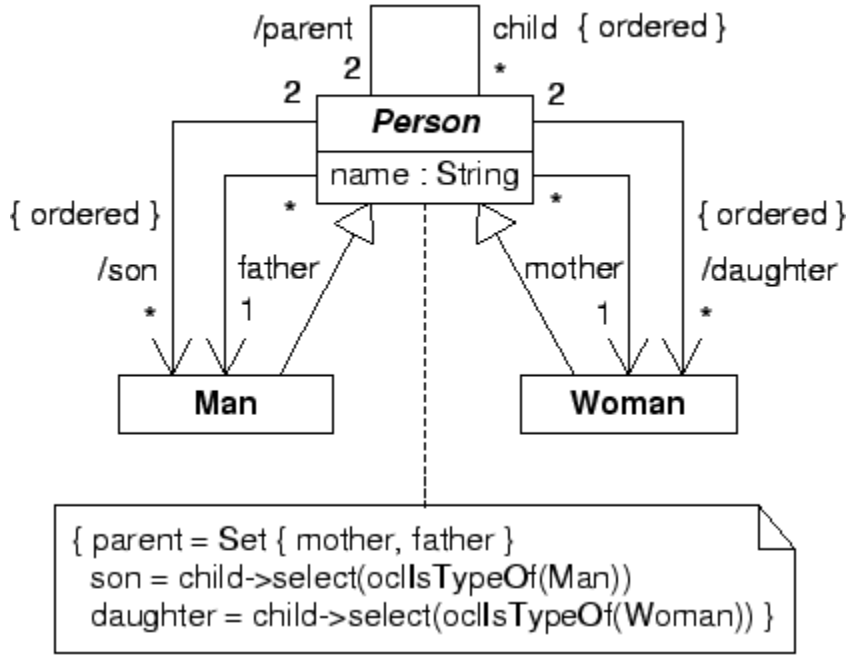

Figura 27 - Esempio: ontologia Family

I rettangoli identificano le classi. In alto c'è il nome della classe (in italico se la classe è astratta). Sotto il nome della classe, separatamente, vengono elencati gli attributi. Le linee tra le classi rappresentano le relazioni di associazione. Sugli estremi di associazione ci sono etichette riportanti i rolenane e i numeri che indicano la cardinalità, cioè quanti oggetti di quella classe possono essere in associazione con le istanze della classe all'altro estremo dell'associazione. Una freccia aperta indica che l'associazione può essere attraversata solo in una direzione, la classe a un estremo ha informazioni sull'associazione, mentre quella all'altro estremo no. Un vincolo {ordered} indica che c'è un ordine definito sull'insieme di oggetti a quell'estremo di associazione. In termini implementativi, questo significa che l'estremo di associazione è rappresentato da una struttura dati Sequence all'interno della classe all'altro estremo, invece che da un insieme (Bag). Una linea con una freccia chiusa rappresenta una generalizzazione, la punta della freccia è rivolta verso la classe più generale. Il nome di ogni elemento di modellazione può essere preceduto da uno slash ("/"), che indica che l'elemento è derivato, cioè può essere ottenuto da altri elementi nel modello, sebbene Rational Rose non supporti questo concetto. Il rettangolo contiene vincoli sulla classe Person espressi usando OCL, che può essere usato in associazione con l'UML per vincolare il modello qualora gli elementi strutturali dell'UML da

soli non siano sufficienti a esprimere le specifiche desiderate. In questo caso i vincoli specificano come *parent, son* e *daughter* sono in relazione con *mother, father* e *child*.

Lo schema RDF corrispondente al class-diagram "Family" è il seguente (la codifica RDF/XML per questo schema è riportata in Appendice 3):

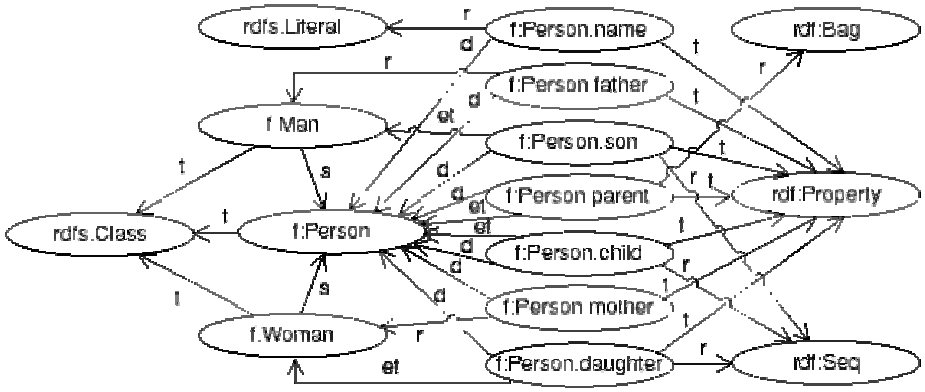

Figura 88 - RDF-Schema dell'ontologia Family

dove:

s = rdfs:subClassOf
et = rdfsx:collectionElementType
rdf = http://www.w3.org/1999/02/22-rdf-syntax-ns#
f = *any new namespace chosen for this schema*

Mappando un class-diagram in uno schema RDF l'obiettivo non è esprimere tutti i dettagli del modello, ma facilitare la serializzazione delle istanze del modello. Lo schema qui generato, ad esempio, non esprime l'informazione che una persona ha esattamente due genitori o che il nome di una persona è espresso come una stringa. Questo richiederebbe estensioni ad RDF-Schema e non è richiesto in quanto la conoscenza di queste relazioni è presente nel codice generato per la classe Java Person. Se un'applicazione vuole accedere alla descrizione completa dell'ontologia può accedere direttamente alla descrizione XMI usando una API per XMI. Anche i vincoli OCL sono qui ignorati.

4.4.2.2 OCL

Ricerche recenti hanno mostrato come vincoli di integrità UML possono essere espresse come trasformazioni di grafo del metamodello UML [62]. La figura 29 mostra come può essere generata nuova conoscenza in forma di object-diagram combinando conoscenze esistenti sull'ontologia. In questo esempio un agente ha comunicato a un altro che c'è un oggetto della classe Man con "Kim" come valore del suo attributo name. L'altro agente sa che c'è un oggetto Person con nome Kim e che questo oggetto è il figlio di un oggetto Person con nome Bob. L'ontologia per questo dominio afferma che Man è una specializzazione di Person e include due vincoli OCL. Uno definisce la regola derivata (indicata con "/") son (un son è un figlio che è un man), l'altro stabilisce che l'attributo name identifica univocamente oggetti della classe Person. Passando per vari livelli di inferenza l'agente può concludere che due oggetti con name Kim sono lo stesso oggetto e perciò Kim è un son. Implementare questo stile di deduzione in tool UML costituisce un altro spunto per ricerche future.

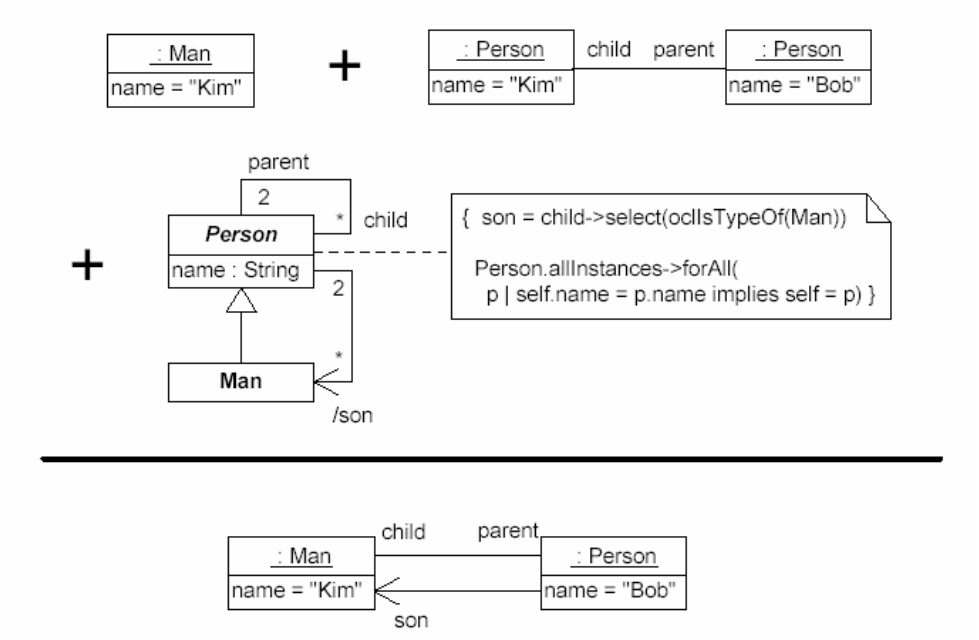

Figura 29 - Esempio di inferenza su conoscenza UML

Conclusioni e sviluppi futuri

In questa tesi è stato presentata l'architettura del "Semantic Web", il nuovo modo di concepire il Web proposto da Tim Berners-Lee [Capitolo 1], e sono stati brevemente illustrati i linguaggi che ne permettono l'implementazione, soffermandosi in particolare sugli standard proposti dal W3C: XML e RDF(S). E' stato poi descritto il linguaggio ontologico DAML+OIL, nato dalla fusione di due linguaggi per la definizione di ontologie, DAML e OIL. Quello che si è cercato di proporre in questa tesi è un'alternativa a questi popolari formalismi per la rappresentazione di ontologie, costituita dall'uso combinato di formalismi UML, il linguaggio di modellazione object-oriented standard dell'OMG, e di espressioni OCL, un potente strumento per l'espressione di vincoli. La questione che viene posta è se l'UML possa essere visto, oltre che come ormai consolidato e diffuso elemento di analisi preliminare del software, anche come un efficace linguaggio per rappresentare ontologie, e quindi possa essere inserito nella lista dei linguaggi di rappresentazione della conoscenza, insieme ai tradizionali RDF(S), DAML+OIL etc. Alcuni progetti di ricerca cercano di dimostrare la fattibilità di questo approccio. Il progetto UBOT, ad esempio, propone un accoppiamento di UML e linguaggi di ontologie, infatti, utilizza tool UML (Rational Rose o ArgoUML) per editare e generare un formato XMI, che viene poi tradotto in DAML+OIL. Il progetto UBOT ha sperimentato anche metodi formali per verificare la consistenza delle ontologie DAML+OIL. A questo scopo è stato sviluppato ConsVISor, un tool che prende in input ontologie UML o DAML+OIL e che, ragionando sulla semantica, identifica dichiarazioni contraddittorie nelle ontologie. Ai fini del mapping tra le specifiche UML e le specifiche DAML+OIL, K. Baclawski ha studiato un prototipo UML che ne prevede l'estensione con un insieme di stereotipi corrispondenti a classi e proprietà in DAML+OIL. Viene inoltre proposta un'estensione al metamodello UML per renderlo compatibile con le proprietà DAML+OIL. La proposta di S. Cranefield, invece, prevede che, da un'ontologia rappresentata mediante UML, vengano generati automaticamente un RDF-Schema corrispondente e un insieme di classi Java per facilitare l'uso di object-diagram come strutture di rappresentazione della conoscenza. Un interessante sviluppo futuro è costituito dall'utilizzo concreto di questi tool, applicandoli ad esempi pratici di ontologie espresse in UML, e dal confronto delle loro prestazioni con quelle dei traduttori del sistema MOMIS già implementati. Bisogna ricordare, infatti, che i linguaggi standard di rappresentazione della conoscenza sono stati studiati, nell'ambito del progetto SEWASIE, in relazione con il linguaggio di mediazione $ODLI_3$ utilizzato dal sistema MOMIS. Sono stati così creati traduttori per ontologie da $ODLI_3$ a XML-Schema, RDF(S), DAML+OIL, OWL, etc. e viceversa, in modo da permettere l'implementazione dei rispettivi wrapper. Un altro esperimento interessante riguarda il confronto tra il tool di controllo della consistenza di schemi di UBOT, ConsVISor, e ODB-Tools. Anche

mettere a confronto il numero e il tipo di vincoli esprimibili da uno schema UML, integrato con OCL, con quelli esprimibili da $ODLI_3$, nato esplicitamente per rispondere alle problematiche di integrazione di informazioni da fonti eterogenee, è utile per capire quale delle due metodologie abbia più potenzialità al fine di descrivere sistemi reali. Nel tentativo di mappare le specifiche di un linguaggio object-oriented, come l'UML, nelle specifiche dei linguaggi di rappresentazione della conoscenza, si sono riscontrate varie incongruenze, che si ripercuotono nei progetti che prevedono di tradurre l'output di tool basati sull'UML (Rational Rose o ArgoUML) in questi linguaggi. I tool UBOT e UML-Data-Binding, ad esempio, sono ancora in fase di sviluppo e prevedono, come già detto, estensioni al metamodello UML, limitazioni di vario genere etc. Inoltre, né l'UML né il linguaggio di espressioni di vincoli ad esso associato, l'OCL, possiedono una semantica formale, anche se esistono molte ricerche attive in questo ambito, come il progetto pUML (preciseUML). D'altra parte, bisogna ricordare altri aspetti molto importanti che rendono invece l'UML una notazione adatta per la modellazione concettuale di ontologie. Esattamente come il Web tradizionale si è sviluppato senza controlli e senza regole, è difficile immaginare che il Semantic Web si evolva come un insieme ordinato di ontologie, costruite da esperti di Intelligenza Artificiale. Soprattutto nei primi tempi, esso si estenderà con un gran numero di ontologie di dimensioni ridotte, che fanno riferimento l'una all'altra e saranno sviluppate dagli utenti, analogamente a quanto avviene per i contenuti del Web attualmente creati. Sotto questo aspetto l'UML si rivela una notazione eccellente, in quanto è un linguaggio grafico, standard, universale, largamente diffuso e associato a tool di immediato utilizzo, come Rational Rose e ArgoUML [Capitolo3]. Le tecniche attuali di sviluppo di ontologie, invece, sono basate su rappresentazioni della conoscenza che non sono conosciute al di fuori della specifica comunità di ricerca dell'Intelligenza Artificiale, ed i rispettivi Linguaggi KR non hanno ancora nessuna forma grafica standard. Ricordiamo inoltre che l'introduzione di contenuti semantici nel Web dovrà cercare di mantenere le caratteristiche peculiari del Web, come la larga quantità di informazioni disponibili e la decentralizzazione delle informazioni, ma contemporaneamente dovrà consentire una ricerca più mirata e precisa e uno scambio di informazioni tra gli agenti software, che le trovano nel Web e le elaborano in base al loro significato. Come già visto nel Paragrafo 4.1, le potenzialità dell'UML, provvisto di package, meccanismi di modularità e formalismi per esprimere la sincronizzazione di processi e le dipendenze tra attività, si sposano bene con la modularità e dinamicità di processi che il Semantic Web richiede, aspetto riguardo cui i Linguaggi KR sono inefficienti. A questo proposito, va sottolineato che, nel mapping tra UML e DAML+OIL presentato nel paragrafo 4.3, sono stati analizzati solo i class-diagram, cioè i componenti statici della modellazione UML, per mostrare la relazione diretta tra i loro elementi e quelli di un'ontologia, come i concetti di classe, gerarchia e attributo. L'UML, però, come già visto, è un linguaggio che, nella sua totalità, mette a disposizione formalismi molto più ricchi e complessi. Scoprire come essi possano essere

ulteriormente sfruttati per modellare vari aspetti del Semantic Web, come la gestione delle interazioni tra gli agenti software e l'organizzazione dei servizi Web offerti, è un altro ambito interessante per ricerche future.

Appendice

1. Sintassi $ODLI_3$

Si riporta la sintassi BNF (Backus Nauer Form) del linguaggio $ODLI_3$, comprendente sia gli elementi sintattici del linguaggio standard ODL che le estensioni:

```
<OdlSpecification>      ::=  <Definition> .
                        |    <Definition> ;
                        |    <Definition> ; <OdlSpecification>
<Definition>            ::=  <TypeDcl>
                        |    <ConstDcl>
                        |    <ExceptDcl>
                        |    <Interface>
                        |    <RuleDcl>
                        |    <ExtRuleDcl>
                        |    <ThesaurusRelation>
                        |    <Module>
                        |    <Source>
                        |    <WnAnnotation>
                        |    <error>
<TypeDcl>               ::=  typedef <TypeDeclarator>
                        |    <ConstrTypeSpec>
<TypeDeclarator>        ::=  <TypeSpec> <Declarators>
<Declarators>           ::=  <Declarator>
                        |    <Declarators> , <Declarator>
<Declarator>            ::=  <Identifier>
                        |    <Identifier> <ArraySizeList>
<ArraySizeList>         ::=  <FixedArraySize>
                        |    <ArraySizeList> <FixedArraySize>
<TypeSpec>              ::=  <SimpleTypeSpec>
                        |    <ConstrTypeSpec>
<SimpleTypeSpec>        ::=  <BaseTypeSpec>
                        |    <TemplateTypeSpec>
                        |    <Identifier>
<BaseTypeSpec>          ::=  <FloatingPtType>
                        |    <IntegerType>
                        |    <CharType>
                        |    <BooleanType>
                        |    <OctetType>
                        |    <RangeType>
                        |    <DateType>
                        |    <AnyType>
<FloatingPtType>        ::=  float
                        |    double
<DateType>              ::=  <DATE>
                        |    <TIMESTAMP>
<IntegerType>           ::=  <LongIntType>
```

| | **short**
| | **unsigned long**
| | **unsigned short**
<LongIntType> ::= **integer**
| | **int**
| | **long**
<CharType> ::= **char**
<BooleanType> ::= **boolean**
<OctetType> ::= **octet**
<RangeType> ::= **range** { <RangeSpecifier> }
<RangeSpecifier> ::= <SignedIntegerLiteral> , <SignedIntegerLiteral>
| | <SignedIntegerLiteral> , + **infinite**
| | **- infinite** , <SignedIntegerLiteral>
<AnyType> ::= **any**
<TemplateTypeSpec> ::= <ArrayType>
| | <StringType>
| | <CollectionType>
<ArrayType> ::= **array** < <SimpleTypeSpec> , <IntegerLiteral> >
<$$1> ::=
<ArrayType> ::= **array** < <SimpleTypeSpec> > <$$1>
sequence < <SimpleTypeSpec> , <IntegerLiteral> >
<StringType> ::= **string** [<IntegerLiteral>]
| | **string**
<CollectionType> ::= <AttrCollectionSpecifier> < <SimpleTypeSpec> >
<AttrCollection ::= **set**
Specifier> | | **list**
| | **bag**
<ConstrTypeSpec> ::= <StructType>
| | <UnionType>
| | <EnumType>
<StructType> ::= **struct** <Identifier> { <MemberList> }
<MemberList> ::= <Member>
| | <MemberList> <Member>
<Member> ::= <TypeSpec> <Declarators> ;
<UnionType> ::= **union** <Identifier>
switch (<SwitchTypeSpec>) { <SwitchBody> }
<SwitchTypeSpec> ::= <IntegerType>
| | <CharType>
| | <BooleanType>
| | <EnumType>
| | <ScopedName>
| | <RangeType>
<SwitchBody> ::= <Case>
| | <Case> <SwitchBody>
<Case> ::= <CaseLabelList> <ElementSpec> ;
<CaseLabelList> ::= <CaseLabel>
| | <CaseLabel> <CaseLabelList>
<CaseLabel> ::= **case** <ConstExp> :
| | **default** :
<ElementSpec> ::= <TypeSpec> <Declarator>
<EnumType> ::= **enum** <Identifier> { <EnumeratorList> }
<EnumeratorList> ::= <Enumerator>
| | <EnumeratorList> , <Enumerator>
<Enumerator> ::= <Identifier>
<ConstExp> ::= <OrExpr>

```
<OrExpr>                ::=  <XOrExpr>
                        |    <OrExpr> | <XOrExpr>
<XOrExpr>               ::=  <AndExpr>
                        |    <XOrExpr> ^ <AndExpr>
<AndExpr>               ::=  <ShiftExpr>
                        |    <AndExpr> & <ShiftExpr>
<ShiftExpr>             ::=  <AddExpr>
                        |    <ShiftExpr> >> <AddExpr>
                        |    <ShiftExpr> << <AddExpr>
<AddExpr>               ::=  <MultExpr>
                        |    <AddExpr> + <MultExpr>
                        |    <AddExpr> - <MultExpr>
<MultExpr>              ::=  <UnaryExpr>
                        |    <MultExpr> * <UnaryExpr>
                        |    <MultExpr> / <UnaryExpr>
                        |    <MultExpr> % <UnaryExpr>
<UnaryExpr>             ::=  <Signes> <PrimaryExpr>
                        |    <PrimaryExpr>
<PrimaryExpr>           ::=  <Identifier>
                        |    <IntegerLiteral>
                        |    <FloatingPtLiteral>
                        |    ( <OrExpr> )
                        |    ( <error> )
<ConstDcl>              ::=  const <StringType> <Identifier> = <StringLiteral>
                        |    const <CharType> <Identifier> = <CharacterLiteral>
                        |    const <IntegerType> <Identifier> = <ConstExp>
                        |    const <FloatingPtType> <Identifier> = <ConstExp>
<SignedIntegerLiteral>  ::=  <IntegerLiteral>
                        |    <Signes> <IntegerLiteral>
<SignedFloating         ::=  <Signes> <FloatingPtLiteral>
PtLiteral>              |    <FloatingPtLiteral>
<Signes>                ::=  -
                        |    +
<ExceptDcl>             ::=  exception <Identifier> { <OptMemberList> }
<OptMemberList>         ::=
                        |    <MemberList>
<ScopedName>            ::=  <Identifier>
                        |    :: <Identifier>
                        |    <ScopedName> :: <Identifier>
<Interface>             ::=  <InterfaceDcl>
<IntView>               ::=  interface <InterfaceName>
                        |    view <InterfaceName>
<InterfaceName>         ::=  <Identifier>
<InterfaceDcl>          ::=  <IntView> : <InheritanceSpec>
                             <OptTypePropertyList> <OptPersistenceDcl>
                             <SingleInterfaceBody>
                        |    <IntView> : <InheritanceSpec>
                             <OptTypePropertyList> <OptPersistenceDcl>
                             <SingleInterfaceBody> <InterfaceBodyUnionList>
                        |    <IntView> <OptTypePropertyList> <OptPersistenceDcl>
                             <SingleInterfaceBody>
                        |    <IntView> <OptTypePropertyList> <OptPersistenceDcl>
                             <SingleInterfaceBody> <InterfaceBodyUnionList>
<InheritanceSpec>       ::=  <Identifier>
                        |    <InheritanceSpec> , <Identifier>
```

<OptTypePropertyList> ::=
| (<OptSourceSpec> <OptExtentSpec> <OptKeySpec>
<OptCandKeySpec> <OptForKeySpec>)
<OptSourceSpec> ::=
| **source** <SourceType> <Identifier>
<SourceType> ::= **relational**
| **nfrelational**
| **object**
| **file**
| **semistructured**
<OptExtentSpec> ::=
| **extent** <ListExtent>
<ListExtent> ::= <Identifier>
| <ListExtent> , <Identifier>
<OptKeySpec> ::=
| **key** <Key>
<OptCandKeySpec> ::=
| <CandKeySpecList>
<CandKeySpecList> ::= <CandKeySpec>
| <CandKeySpecList> <CandKeySpec>
<CandKeySpec> ::= **candidate_key** <Identifier> <Key>
<OptForKeySpec> ::=
| <ForKeySpecList>
<ForKeySpecList> ::= <ForKeySpec>
| <ForKeySpecList> <ForKeySpec>
<ForKeySpec> ::= **foreign_key** (<ForeignKeyList>)
references <Identifier> <OptRefKeyList>
<OptRefKeyList> ::=
| (<ForeignKeyList>)
<ForeignKeyList> ::= <Identifier>
| <ForeignKeyList> , <Identifier>
<Key> ::= (<PropertyList>)
<PropertyList> ::= <PropertyName>
| <PropertyList> , <PropertyName>
<PropertyName> ::= <Identifier>
<OptPersistenceDcl> ::=
| **persistent**
| **transient**
<InterfaceBody ::= <InterfaceBodyUnion>
UnionList> | <InterfaceBodyUnionList> <InterfaceBodyUnion>
<InterfaceBodyUnion> ::= **union** <Identifier> <SingleInterfaceBody>
<SingleInterfaceBody> ::= { <InterfaceBody> }
<InterfaceBody> ::=
| <Export> ;
| <Export> ; <InterfaceBody>
<Export> ::= <TypeDcl>
| <ConstDcl>
| <ExceptDcl>
| <AttrDcl>
| <RelDcl>
| <OpDcl>
<AttrDcl> ::= <OptReadonly> **attribute** <DomainType> <AttributeName>
<OptFixedArraySize> <OptMappingRuleDcl>
<OptReadonly> ::=
| **readonly**

```
<AttributeName>          ::=  <Identifier>
                         |    <Identifier> ?
<DomainType>             ::=  <SimpleTypeSpec>
                         |    <StructType>
                         |    <EnumType>
<OptFixedArraySize>      ::=
                         |    <FixedArraySize>
<FixedArraySize>         ::=  [ <IntegerLiteral> ]
<OptMappingRuleDcl>      ::=
                         |    <MappingRuleDcl>
<MappingRuleDcl>         ::=  mapping rule <MapRuleList>
<MapRuleList>            ::=  <MapRule>
                         |    <MapRuleList> , <MapRule>
<MapRule>                ::=  <LocalAttributeName>
                         |    <DefaultValue>
                         |    <MapAndExpression>
                         |    <MapUnionExpression>
<LocalAttributeName>     ::=  <LocalClassName> . <Identifier>
<LocalClassName>         ::=  <Identifier> . <Identifier>
<DefaultValue>           ::=  <Identifier> . <Identifier> = <StringLiteral>
<MapAndExpression>       ::=  ( <MapAndList> and <LocalAttributeName> )
<MapAndList>             ::=  <LocalAttributeName>
                         |    <MapAndList> and <LocalAttributeName>
<MapUnionExpression>     ::=  ( <MapUnionList> union <LocalAttributeName>
                              on <Identifier> )
<MapUnionList>           ::=  <LocalAttributeName>
                         |    <MapUnionList> union <LocalAttributeName>
<RelDcl>                 ::=  relationship <TargetOfPath> <Identifier> inverse
                              <InverseTraversalPath> <OptOrderBy>
<TargetOfPath>           ::=  <Identifier>
                         |    <RelCollectionType> < <Identifier> >
<RelCollectionType>      ::=  set
                         |    list
<InverseTraversalPath>   ::=  <Identifier> :: <Identifier>
<OptOrderBy>             ::=
                         |    { order_by <ScopedNameList> }
<ScopedNameList>         ::=  <ScopedName>
                         |    <ScopedNameList> , <ScopedName>
<OpDcl>                  ::=  <OptOneway> <OperTypeSpec> <Identifier>
                              <ParameterDcls>
                              <OptRaisesExpr> <OptContextExpr>
<OptOneway>              ::=
                         |    oneway
<OperTypeSpec>           ::=  <SimpleTypeSpec>
                         |    void
<ParameterDcls>          ::=  ( <ParamDclList> )
                         |    ( )
<ParamDclList>           ::=  <ParamDcl>
                         |    <ParamDclList> , <ParamDcl>
<ParamDcl>               ::=  <ParamAttribute> <SimpleTypeSpec> <Declarator>
<ParamAttribute>         ::=
                         |    in
                         |    out
                         |    inout
<OptRaisesExpr>          ::=
```

| **raises** (<ScopedNameList>)

<OptContextExpr> ::=

| **context** (<StringLiteralList>)

<StringLiterall>st> ::= <StringLiteral>

| <StringLiteralList> , <StringLiteral>

<ExtRuleDcl> ::= **extrule** <Identifier> <ExtRuleSpec>

<ExtRuleSpec> ::= <ForAll> <Identifier> **in** <ExtRuleType>

<ExtRuleType> ::= <ExtRuleIsa>

| <ExtRuleBottom>

<ExtRuleBottom> ::= (<LocalClassName> **and** <LocalClassName>)
then <Identifier> **in bottom**

<ExtRuleIsa> ::= <LocalClassName> **then** <Identifier> **in** <LocalClassName>

<RuleDcl> ::= **rule** <Identifier> <RuleSpec>

<RuleSpec> ::= <ForAll> <Identifier> **in** <Identifier> :
<RuleBodyList> **then** <RuleBodyList>

| { **case of** <Identifier> : <CaseList> }

<ForAll> ::= **for all**

| **forall**

<RuleBodyList> ::= (<RuleBodyList>)

| <RuleBody>

| <RuleBodyList> **and** <RuleBody>

<$$2> ::=

<RuleBody> ::= <DottedName> <RuleConstOp> <OptRuleCast>
<LiteralVALUE> <$$2> <DottedName> <RuleConstOp>
<OptRuleCast> <DottedName>

| <DottedName> **in** <DottedName>

| <ForAll> <Identifier> **in** <DottedName> : <RuleBodyList>

| **exists** <Identifier> **in** <DottedName> : <RuleBodyList>

| <DottedName> = <SimpleTypeSpec> <Identifier>
(<DottedLiteralList>)

<DottedLiteralList> ::= <DottedLiteral>

| <DottedLiteralList> , <DottedLiteral>

<DottedLiteral> ::= <OptSimpleTypeSpec> <DottedName>

| <OptSimpleTypeSpec> <LiteralValue>

<OptSimpleTypeSpec> ::=

| <SimpleTypeSpec>

<RuleConstOp> ::= =

| >=

| <=

| <

| >

<LiteralValue> ::= <SignedFloatingPtLiteral>

| <SignedIntegerLiteral>

| <CharacterLiteral>

| <StringLiteral>

<DottedName> ::= <Identifier>

| <DottedName> . <Identifier>

<OptRuleCast> ::=

| (<SimpleTypeSpec>)

<CaseList> ::= <CaseSpec>

| <CaseList> <CaseSpec>

<CaseSpec> ::= <LiteralValue> : <DottedName>

<ThesaurusRelation> ::= <LocalAttributeName> <ThesRelType>
<LocalAttributeName>
<ThesRelOptProducer> <ThesRelOptValidated>

| | <LocalClassName> <ThesRelType> <LocalClassName> <ThesRelOptProducer> <ThesRelOptValidated>
| | <LocalClassName> <ThesRelType> <LocalAttributeName> <ThesRelOptProducer> <ThesRelOptValidated>
| | <LocalAttributeName> <ThesRelType> <LocalClassName> <ThesRelOptProducer> <ThesRelOptValidated>

<ThesRelType> ::= **syn**
| **bt**
| **nt**
| **rt**

<ThesRelOptProducer> ::=
| <IntegerLiteral>

<ThesRelOptValidated> ::=
| **true**
| **false**

<Module> ::= **module** <Identifier> { <OdlSpecification> }

<Source> ::= **source** <SourceType> <Identifier> <SourceOdli3Params> { <OdlSpecification> }

<SourceOdli3Params> ::=
| (<SourceOdli3ParamsLines>)

<SourceOdli3ParamsLines> ::= <Identifier> = <StringLiteral>
| <SourceOdli3ParamsLines> , <SourceOdli3ParamsLines>

<WnAnnotation> ::= <WNANNOTATION> <DottedName> <LEMMAVALUE> = <StringLiteral> , <LEMMASYNTACTICCATEGORY> = <IntegerLiteral> , <LEMMASENSENUMBER> = <IntegerLiteral>

2. Grammatica OCL

Si riporta la grammatica del linguaggio OCL, descritta usando la sintassi EBNF (Extended BNF):

```
oclFile                 :=  ( "package "packageName
                            oclExpressions
                            "endpackage" )+
packageName             :=  pathName
oclExpressions          :=  ( constraint )*
constraint              :=  contextDeclaration
                            ( ( "def" name? ":" letExpression* ) |
                            ( stereotype name? ":" oclExpression ) )+
contextDeclaration      :=  "context"
                            ( operationContext | classifierContext )
classifierContext       :=  ( name ":" name ) |
                            name
operationContext        :=  name "::" operationName
                            "(" formalParameterList ")"
                            ( ":" returnType )?
stereotype              :=  ( "pre" | "post" | "inv" )
operationName           :=  name | "=" | "+" | "-" | "<" | "<=" | ">=" | ">" | "/" | "*" |
                            "<>" | "implies" | "not" | "or" | "xor" | "and"
formalParameterList     :=  ( name ":" typeSpecifier
                            ( "," name ":" typeSpecifier )* )?
typeSpecifier           :=  simpleTypeSpecifier |
                            collectionType
collectionType          :=  collectionKind
                            "(" simpleTypeSpecifier ")"
oclExpression           :=  ( letExpression* "in" )? expression
returnType              :=  typeSpecifier
expression              :=  logicalExpression
letExpression           :=  "let" name
                            ( "(" formalParameterList ")" )?
                            ( ":" typeSpecifier )?
                            "=" expression
ifExpression            :=  "if" expression
                            "then" expression
                            "else" expression
                            "endif"
logicalExpression       :=  relationalExpression
                            ( logicalOperator
                            relationalExpression )*
relationalExpression    :=  additiveExpression
                            ( relationalOperator
                            additiveExpression )?
additiveExpression      :=  multiplicativeExpression
                            ( addOperator
                            multiplicativeExpression )*
```

```
multiplicativeExpression    :=  unaryExpression
                                ( multiplyOperator
                                unaryExpression )*
unaryExpression             :=  ( unaryOperator
                                postfixExpression ) |
                                postfixExpression
postfixExpression           :=  primaryExpression
                                ( ("." | "->")propertyCall )*
primaryExpression           :=  literalCollection |
                                literal |
                                propertyCall |
                                "(" expression ")" |
                                ifExpression
propertyCallParameters      :=  "(" ( declarator )?
                                ( actualParameterList )? ")"
literal                     :=  string |
                                number |
                                enumLiteral
enumLiteral                 :=  name "::" name ( "::" name )*
simpleTypeSpecifier         :=  pathName
literalCollection           :=  collectionKind
                                "{"
                                ( collectionItem
                                ( "," collectionItem )* )?
                                "}"
collectionItem              :=  expression ( ".." expression )?
propertyCall                :=  pathName
                                ( timeExpression )?
                                ( qualifiers )?
                                ( propertyCallParameters )?
qualifiers                  :=  "[" actualParameterList "]"
declarator                  :=  name ( "," name )*
                                ( ":" simpleTypeSpecifier )?
                                ( ";" name ":" typeSpecifier "=" expression )?
                                "|"
pathName                    :=  name ( "::" name )*
timeExpression              :=  "@" "pre"
actualParameterList         :=  expression ( "," expression )*
logicalOperator             :=  "and" | "or" | "xor" | "implies"
collectionKind              :=  "Set" | "Bag" | "Sequence" | "Collection"
relationalOperator          :=  "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator                 :=  "+" | "-"
multiplyOperator            :=  "*" | "/"
unaryOperator               :=  "-" | "not"
typeName                    :=  charForNameTop charForName*
name                        :=  charForNameTop charForName*
charForNameTop              :=  /* Characters except inhibitedChar and ["0"-"9"]; the
                                available characters shall be determined by the tool
                                implementers ultimately.*/
charForName                 :=  /* Characters except inhibitedChar; the available
                                characters shall be determined by the tool implementers
                                ultimately.*/
inhibitedChar               :=  " " | "\"" | "#" | "\'" | "(" | ")" |
                                "*" | "+" | "," | "-" | "." | "/" |
                                ":" | ";" | "<" | "=" | ">" | "@" |
```

```
                    "[" | "\\" | "]" | "{" | "|" | "}"
number          :=  ["0"-"9"] (["0"-"9"])*
                    ( "." ["0"-"9"] (["0"-"9"])* )?
                    ( ("e" | "E") ("+" | "-")? ["0"-"9"] (["0"-"9"])* )?
string          :=  """
                    (
                            ( ~["'","\\","\n","\r"] ) |
                            ( "\\"
                                    ( ["n","t","b","r","f","\\","'","\""] |
                                            ["0"-"7"]
                                            ( ["0"-"7"] ( ["0"-"7"] )? )?
                                    )
                            )
                    )*
                    """
```

3. UML-Data-Binding

Il seguente documento XML codifica le triple RDF corrispondenti all'ontologia "Family" descritta in figura 27 [Paragrafo 4.2.2.1]:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [ <!ENTITY f 'http://nzdis.otago.ac.nz/0_1/family#'> ]>
<rdf:RDF xml:lang="en"
        xmlns:f="&f;"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:rdfsx="http://nzdis.otago.ac.nz/0_1/rdf-schema-extensions#">
<rdfs:Class rdf:ID="&f;Person"/>
<rdfs:Class rdf:ID="&f;Man">
        <rdfs:subClassOf rdf:resource="&f;Person"/>
</rdfs:Class>
<rdfs:Class rdf:ID="&f;Woman">
        <rdfs:subClassOf rdf:resource="&f;Person"/>
</rdfs:Class>
<rdf:Property ID="&f;Person.name">
        <rdfs:domain rdf:resource="&f;Person"/>
        <rdfs:range rdf:resource="rdfs:Literal"/>
</rdf:Property>
<rdf:Property ID="&f;Person.parent">
        <rdfs:domain rdf:resource="&f;Person"/>
        <rdfs:range rdf:resource="rdf:Bag"/>
        <rdfsx:containerElementType rdf:resource="&f;Person"/>
</rdf:Property>
<rdf:Property ID="&f;Person.child">
        <rdfs:domain rdf:resource="&f;Person"/>
        <rdfs:range rdf:resource="rdf:Seq"/>
        <rdfsx:containerElementType rdf:resource="&f;Person"/>
</rdf:Property>
<rdf:Property ID="&f;Person.father">
        <rdfs:domain rdf:resource="&f;Person"/>
```

```
        <rdfs:range rdf:resource="&f;Man"/>
</rdf:Property>
<rdf:Property ID="&f;Person.son">
        <rdfs:domain rdf:resource="&f;Person"/>
        <rdfs:range rdf:resource="rdf:Seq"/>
        <rdfsx:containerElementType rdf:resource="&f;Man"/>
</rdf:Property>
<rdf:Property ID="&f;Person.mother">
        <rdfs:domain rdf:resource="&f;Person"/>
        <rdfs:range rdf:resource="&f;Woman"/>
</rdf:Property>
<rdf:Property ID="&f;Person.daughter">
        <rdfs:domain rdf:resource="&f;Person"/>
        <rdfs:range rdf:resource="rdf:Seq"/>
        <rdfsx:containerElementType rdf:resource="&f;Woman"/>
</rdf:Property>
</rdf:RDF>
```

Bibliografia ragionata

[1] Tim Berners-Lee, James A. Hendler, Ora Lassila. *The Semantic Web*. Scientific American, May 2001.
http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html

[2] World Wide Web Consortium.
http://www.w3.org

[3] XML.
http://www.w3.org/XML

[4] RDF.
http://www.w3.org/RDF

[5] DAML, DAML+OIL.
http://www.daml.org
http://www.w3.org/TR/daml+oil-reference

[6] OWL.
http://www.w3.org/2004/OWL

[7] Valentina Tamma. Theoretical foundations of ontologies. *An Ontology Model supporting Multiple Ontologies for Knowledge sharing*. PhD Thesis, University of Liverpool, 2001.

[8] Nicola Guarino.
Formal ontologies and information systems. Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS'98), Trento, Italy, June 1998.

[9] MOMIS.
http://dbgroup.unimo.it/Momis

[10] SEWASIE.
http://www.sewasie.org

[11] UML.
OMG Unified Modeling Language Specification, ver. 1.4, Settembre 2001.

[12] UBOT.
http://ubot.lockheedmartin.com

[13] *UML for Ontology Development*.
http://ubot.lockheedmartin.com/ubot/papers/publication/KER4.doc

[14] *Extending UML to Support Ontology Engineering for the Semantic Web*.
http://ubot.lockheedmartin.com/ubot/papers/publication/UMLOntology.pdf

[15] *UML and the Semantic Web*.
http://www.semanticweb.org/SWWS/program/full/paper1.pdf

Bibliografia

[1] Tim Berners-Lee. *Semantic web roadmap.* Internal note, 1998. World Wide Web Consortium.

[2] Tim Berners-Lee, James A. Hendler, Ora Lassila. *The Semantic Web*. Scientific American, May 2001.
http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html

[3] Artificial Intelligence. Sonia Bergamaschi. *The Disciplines of Artificial Intelligence.* Lucidi delle lezioni per il corso di Rappresentazione della Conoscenza presso l'Università di Modena, Facoltà di Ingegneria Informatica, 2004-2005.

[4] J. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. PWS Publishing, 2000.

[5] Description Logics. Raffaele Capezzera. *Elaborazioni di interrogazioni tramite l'uso di Corrispondenze Semantiche.* Tesi di laurea, Università di Modena, Facoltà di Ingegneria Informatica, 2002-2003.

[6] Franz Baader, Werner Nutt. *Basic Description Logics.* Description Logic Handbook. Cambridge University Press. 2002.

[7] KIF.
http://logic.stanford.edu/kif/kif.html

[8] World Wide Web Consortium.
http://www.w3.org

[9] XML.
http://www.w3.org/XML

[10] XML Schema.
http://www.w3.org/XML/Schema

[11] XSLT.
http://www.w3.org/TR/xslt

[12] XLink.
http://www.w3.org/TR/xlink

[13] SVG.
http://www.w3.org/TR/SVG

[14] XMI.
http://www-306.ibm.com/software/awdtool/library/standards

[15] RDF.
http://www.w3.org/RDF

[16] RDF-Schema.

http://www.w3.org/TR/rdf-schema

[17] OIL.
http://www.ontoknowledge.org/oil/downl/oil-whitepaper.pdf

[18] On-To-Knowledge.
http://www.ontoknowledge.org

[19] DAML, DAML+OIL.
http://www.daml.org
http://www.w3.org/TR/daml+oil-reference

[20] OWL.
http://www.w3.org/2004/OWL

[21] SHOE.
http://www.cs.umd.edu/projects/plus/SHOE

[22] Notation 3
http://www.w3.org/DesignIssues/Notation3.html

[23] G. Wiederhold. *Mediators in the architecture of future information systems*. IEEE Computer, 1992.

[24] Arpa I3 reference architecture.
http://www.isse.gmu.edu/I3_Arch/index.html

[25] Valentina Tamma. Theoretical foundations of ontologies. *An Ontology Model supporting Multiple Ontologies for Knowledge sharing*. PhD Thesis, University of Liverpool, 2001.

[26] Nicola Guarino. *Formal ontologies and information systems.* Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS'98), Trento, Italy, June 1998.

[27] Nicola Guarino. *Semantic matching: Formal ontological distinctions for information organization, extraction, and integration*. Technical report, Summer School on Information Extraction, Frascati, Italy, July 1997.

[28] Nicola Guarino. *Understanding, building, and using ontologies.* A commentary to "Using Explicit Ontologies in KBS Development", by van Heijst, Schreiber, and Wielinga.

[29] MOMIS.
http://dbgroup.unimo.it/Momis

[30] SEWASIE.
http://www.sewasie.org

[31] ODB-Tools.
http://www.dbgroup.unimo.it/ODB-Tools.html

[32] WordNet.
http://www.cogsci.princeton.edu/~wn

[33] ODLI3. S. Bergamaschi, D. Beneventano, M. Vincini, S. Castano. Integrazione di informazione: il linguaggio ODLI3 e la logica descrittiva OLCD. Technical Report TR-R03, Università di Modena e Reggio Emilia, 1998.

[34] ODL. Morgan Kaufmann. *The object database standard*.

[35] OMT. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[36] OOSE. I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley, 1992.

[37] Booch. G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.

[38] UML. *OMG Unified Modeling Language Specification*, ver. 1.4, Settembre 2001.

[39] Rational Software Corporation.
http://www.rational.com

[40] OMG.
http://www.omg.org

[41] MOF. OMG, *Meta Object Facility (MOF) Specification*, ver. 1.4, Aprile 2002.
http://www.omg.org/docs/formal/00-04-03.pdf

[42] Telelogic Tau UML Suite.
http://www.telelogic.com

[43] Rational Rose 2000.
http://www-306.ibm.com/software/rational

[44] ArgoUML
http://argouml.tigris.org

[45] Rational Unified Process.
http://www.rational.com/products/rup/index.jsp

[46] OCL, Metamodello. Mark Richters, Martin Gogolla. *A Metamodel for OCL*. Proc. 2nd Int. Conf. Unified Modeling Language (UML'99).

[47] Ontolingua.
http://www.ksl.stanford.edu/sns.shtml

[48] Protege.
http://protege.stanford.edu/

[49] UBOT.
http://ubot.lockheedmartin.com

[50] ConsVISor.
http://vistology.com

[51] Paul A. Kogut, Lockheed Martin Management & Data Systems
Stephen Cranefield, University of Otago

Lewis Hart, GRC International
Mark Dutra, Sandpiper Software, Inc.
Kenneth Baclawski, Northeastern University
Mieczyslaw K. Kokar, Northeastern University
Jeffrey Smith, Mercury Computer
UML for Ontology Development.
http://ubot.lockheedmartin.com/ubot/papers/publication/KER4.doc

[52] Kenneth Baclawski, Northeastern University
Mieczyslaw K. Kokar, Northeastern University
Paul A. Kogut, Lockheed Martin Management & Data Systems
Lewis Hart, GRC International
Jeffrey Smith, Mercury Computer
William S. Holmes III, Lockheed Martin Management & Data Systems
Jerzy Letkowski, Western New England College
Michael L. Aronson, Lockheed Martin Management & Data Systems
Extending UML to Support Ontology Engineering for the Semantic Web.
http://ubot.lockheedmartin.com/ubot/papers/publication/UMLOntology.pdf

[53] Stephen Cranefield and Martin Purvis, University of Otago.
UML as an ontology modelling language. Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, 1999.
http://secml.otago.ac.nz/agents/Assets/documents/dp9901sc.pdf

[54] Stephen Cranefield, University of Otago. *UML and the Semantic Web*. Proc. of the International Semantic Web Working Symposium, Palo Alto, 2001
http://www.semanticweb.org/SWWS/program/full/paper1.pdf

[55] Stephen Cranefield, University of Otago. *Networked knowledge representation and exchange using UML and RDF*. Journal of Digital Information, 1(8), 2001.
http://jodi.ecs.soton.ac.uk/Articles/v01/i08/Cranefield

[56] UML-Data-Binding. Cranefield, 2001.
http://nzdis.otago.ac.nz/projects

[57] CODIP, DUET
http://codip.grci.com/

[58] F. Bergenti, A. Poggi. *Exploiting UML in the Design of Multi-Agent Systems.* Engineering Societies in the Agents World, Lecture Notes in Computer Science 1972 (Springer), 2000.

[59] D. Skogan. *UML as a schema language for XML based data interchange*. Proceedings of the 2nd International Conference on The Unified Modeling Language (UML'99), 1999.
http://www.ifi.uio.no/~davids/papers/Uml2Xml.pdf.

[60] Sergey Melnik. UML in RDF.
http://www-db.stanford.edu/~melnik/rdf/uml/, 2000.

[61] XPetal tool.
http://sourceforge.net/projects/xmodel, 2001.

[62] pUML.
http://puml.org

[63] M. Cengarle, A. Knapp. *A formal semantics for OCL 1.4*. M.Gogolla, C. Kobryn. UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference Proceedings. Toronto, Canada, Springer (2001).

[64] Paola Bertolotti. *Semantic Web: analisi e utilizzo di strumenti per la sua realizzazione.* Tesi di laurea, Università di Torino, Facoltà di Ingegneria Informatica, 2001-2002.

[65] Manlio Marchica. *La conoscenza di MOMIS: il ruolo di XML-Schema e RDF*. Tesi di laurea, Università di Modena, Facoltà di Ingegneria Informatica, 2001-2002.

[66] Paola Prampolini. *Dallo "Human Oriented" HTML al linguaggio ODLI3 per l'integrazione di informazioni.* Tesi di laurea, Università di Modena, Facoltà di Ingegneria Informatica, 2001-2002.

[67] Gabriele Mezzetti. *Una libreria per l'implementazione del metamodello UML*. Tesi di laurea, Università di Pisa, Facoltà di Ingegneria Informatica, 2002-2003.

[68] Paolo Pinciani. *Un ambiente di Web Publishing per documenti UML*. Tesi di laurea, Università di Bologna, Facoltà di Ingegneria Informatica, 2001-2002.

[69] Mirko Orsini. *Interoperabilità tra ontologie eterogenee: i traduttori OWL-ODLI3*. Tesi di laurea, Università di Modena, Facoltà di Ingegneria Informatica, 2003-2004.

[70] Wrapper XML
http://dbgroup.unimo.it/Momis/prototipo/modules/wrapperXML

[71] Silvana Castano, Alfio Ferrara. Ontological Representation of Heterogeneous Datasources. Integration Knowledge for the Semantic Web. University of Milan.

[72] Paul Kogut. William Holmes. *AeroDAML: Applying Information Extraction to Generate DAML Annotations from Web Pages*.
http://ubot.lockheedmartin.com/ubot/papers/publication/AeroDAML2.pdf

[73] Paul Kogut. Jeff Heflin. *Semantic Web Technologies for Aerospace*.
http://ubot.lockheedmartin.com/ubot/papers/publication/SemanticWebrevised.doc

Ringraziamenti

Desidero ringraziare:

La mia famiglia, che mi ha permesso di arrivare alla fine di questo lungo percorso e di raggiungere un obiettivo così importante;
La Professoressa Sonia Bergamaschi, per la disponibilità e l'ottimismo dimostratomi durante la realizzazione di questa tesi;
Fabio, per il costante appoggio e il continuo incoraggiamento che ha saputo offrirmi in questi anni;
Tutti i miei colleghi di Università, la vostra compagnia e simpatia hanno alleggerito le interminabili lezioni e la difficoltà degli esami. Senza le nostre lunghe "pausette" questo periodo di studi sarebbe stato forse più breve, ma sicuramente mai così divertente. Un grazie di cuore a Morgan, Diablo, Giuly, Giorgio, Silvia, Cinzia, Lode, Sandro, Pivot, Stefano, Alle, Masso, Space, Bonzo, Gervis, Elisa, Mona, Carlo, Raffa, Max S.K., Marzia, Robbie, Matteo, Juve, Ceci, Leo, Giulio e a tutti gli altri che hanno condiviso con me questa "masochistica avventura";
L'Ingegner Daolio Alessandra, una persona davvero speciale con cui condivido un'amicizia vecchia ormai 12 anni, ma portati bene, Umberto, che mi è sempre stato vicino, e tutte le persone che mi hanno tenuto compagnia e fatto sorridere in questi anni, Saretta, Elen, Paolo, Stefy, Ste, Adriana, Pinola, Frigno, Forrest, Katia, Leo, MarcoB, MarcoM, Ilaria, Ari, Vichy, Cesca, Sara, Giorgia e tantissimi altri che non ho citato, ma a cui rivolgo un sincero grazie.

Grazie a tutti voi, che avete reso piacevoli e indimenticabili questi anni,

Patrizia