Designing Ontological Agents in the Semantic Web

Tesi di Laurea di: Luca Abeti
Relatore: Chiar.mo Prof. Paolo Ciancarini

27 Marzo 2003

III Sessione
Anno Accademico 2001—2002

Keywords: Ontologies, Semantic Web, UML, Agents, ACL
Ringraziamenti

Ringrazio innanzitutto il mio relatore Paolo Ciancarini, per l’attenzione prestata al mio lavoro e per le opportunità da lui offertemi. Un grazie particolare a Valentina Presutti per il suo supporto morale ed intellettuale e per il tempo che mi ha dedicato in questi mesi. Grazie anche a Fabio Vitali per i semplici e costruttivi consigli e per avermi indicato la strada giusta al momento giusto.

Infine ringrazio la mia famiglia - papà, mamma e Sara - per tutto quello che ha fatto per me e per avermi sostenuto e supportato in questi anni.

Un pensiero scontato a chi sempre e per sempre rende lieto il mio lavoro, Elisabetta.
Dedicato a Giovanni e a tutto quello che ha fatto nel mondo.
Un giorno le macchine riusciranno a risolvere tutti i problemi, ma mai nessuna di esse potrà porne uno.

Albert Einstein
Contents

1 INTRODUCTION 9
  1.1 The Semantic Web ................................. 9
    1.1.1 An Overview ................................ 9
    1.1.2 Machine Readable to Machine Understandable ... 10
    1.1.3 The Semantic Web Architecture ................ 12
  1.2 Ontologies on the Semantic Web .................... 14
    1.2.1 The Term Ontology ............................ 14
    1.2.2 Semantic Web Ontologies ...................... 15
    1.2.3 The Power of Ontologies ...................... 16
  1.3 Software Agents .................................. 16
    1.3.1 What is an Agent? ............................ 16
    1.3.2 Agents Classification ........................ 17
    1.3.3 A personal idea of agent .................... 20
  1.4 Thesis Overview ................................ 21

2 TECHNOLOGIES 23
  2.1 The eXtensible Markup Language (XML) ............ 23
    2.1.1 XML Namespaces .............................. 24
    2.1.2 XML Path Language (XPath) ................... 24
    2.1.3 XML Stylesheet Language (XSL) ............... 26
    2.1.4 Other XML Languages .......................... 28
  2.2 Resource Description Framework (RDF) ............. 28
    2.2.1 RDF Serialisation Syntax ..................... 30
  2.3 RDF Vocabulary Description Language (RDF Schema) . 33
  2.4 DAML+OIL and OWL ................................ 35
  2.5 The Unified Modelling Language (UML) ............. 37
    2.5.1 UML Notation ................................ 39
    2.5.2 Static Structure Diagrams .................... 40
    2.5.3 Use Cases .................................... 41
    2.5.4 The Object Constraint Language (OCL) ......... 42
  2.6 UML for Ontology Modelling ........................ 46
2.7 UML Data Binding (UDB) ........................................... 47
  2.7.1 XMI ............................................................... 47
  2.7.2 Java ............................................................. 48
  2.7.3 Processing Knowledge with UML ......................... 49
  2.7.4 UML to RDF schema ........................................ 50
  2.7.5 Generated Java classes .................................... 51
  2.7.6 Limitation and future extensions ......................... 52
2.8 Agent Communication Languages with Ontologies in UML ... 52
2.9 Summary .................................................................. 61

3 METHODOLOGIES ................................................. 63
  3.1 Development of Semantic Web Applications .................. 63
    3.1.1 The Object Oriented Paradigm (OOP) ..................... 64
    3.1.2 The OO Hypermedia Design Method (OOHDM) .......... 65
    3.1.3 The Ontology-Driven Design for Semantic Web Applica-
           tions (ODD) .................................................... 67
  3.2 Agent Oriented Software Engineering .......................... 68
    3.2.1 Agents as a paradigm ...................................... 70
    3.2.2 AO methodologies overview ............................... 72
    3.2.3 The Gaia Methodology .................................... 74
  3.3 Designing Agents with Ontologies ............................. 76
  3.4 Summary .................................................................. 77

4 UML TO DAML MAPPING AND AN APPLICATION SCENARIO .... 79
  4.1 Improving the UDB with DAML Mapping ....................... 79
    4.1.1 Reviewing the UDB architecture: UML for Knowledge
           Representation (UKR) ........................................ 80
    4.1.2 UML to DAML Mapping and Related Issues ............ 82
  4.2 Simple UML to DAML stylesheet ............................... 85
    4.2.1 Modifying uml-to-rdfs.xsl ................................ 86
  4.3 Application Scenarios with VESA: An Ontology-Agent for UniBo
         Exams conversion ............................................. 88
    4.3.1 Architecture of VESA ..................................... 89
    4.3.2 Analysis and Design ....................................... 91
  4.4 The Mind and the Ontologies ................................... 98
  4.5 Conclusions and Future Works ................................. 100

Bibliography ................................................................ 101

Appendix ..................................................................... 107
CONTENTS

A An application of the Jena API 109
B xmi-to-daml output 113
Chapter 1

INTRODUCTION

1.1 The Semantic Web

1.1.1 An Overview

In 1989 a Ginevra CERN’s research team was commissioned to develop a mechanism for the distribution of issues, notes and opinion among the center’s researchers. Tim Berners-Lee, Robert Cailliau et al., identified in Internet, HyperText and SGML the key elements for this application which was presented in 1991 as the World Wide Web. From this first prototype to this day, in only one decade, the Internet and World Wide Web technologies have been an expansion without precedents in the history of technology.

The basic idea of the Web is an information space through which people can communicate and share their knowledge. It is realized as a on screen presentation system for multimedia documents with the use of hypermedia links to allow for navigation. The system is based on some simple concepts: a browser which is the viewer of the hypertextual documents; a server through which it is possible to access the local resources (it uses two protocols: TCP and IP); a standardized mechanism to identify and specify resources on the net called the Uniform Resource Identifier (URI); some communication protocol among which the most important is the HyperText Transport Protocol (HTTP) that permits access to the documents and finally a language based on SGML, the HyperText Markup Language (HTML), to create hypertext documents and link them together.

Although HTML has some tags to add semantics information in the documents, they are not much used, and in any case, the semantics they provide is very limited. To cover this lack, the World Wide Web consortium (W3C) [45] (an organization of organizations founded in 1994 by Berners-Lee to maintain control over WWW’s evolution) has developed a new tecnology
CHAPTER 1. INTRODUCTION

standard: the Extensible Markup Language (XML), it is the foundation of the Semantic Web and a focus of this thesis.

1.1.2 Machine Readable to Machine Understandable

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." 1

The World Wide Web was originally built for human consumption, though everything on it is machine-readable. But it should be useful not only for human-human communication; also machines would be able to participate and help. To achieve this tasks we need a machine-understandable Web.

It is very hard to automate anything on the current Web, and because of the volume of information this Web contains, it is not possible to manage it manually. Leaving aside the artificial intelligence problem of training machines to behave like people, the Semantic Web approach instead develops languages for expressing information in a machine processable form.

The solution proposed is to use metadata to describe the data contained on the Web. Metadata is "data about data" and the distinction between "data" and "metadata" is not an absolute one (many times the same resource will be interpreted in both ways simultaneously).

Hence we present another definition taken from W3C [45] Web site:

**Definition**: The Semantic Web is the abstract representation of data on the World Wide Web, based on the RDF standards and other standards to be defined. It is being developed by the W3C, in collaboration with a large number of researchers and industrial partners.

Semantic comes from the Greek words for sign, signify, and significant, and today "means of" or "relating to meaning", often in language. The Semantic Web is seen as an extension of the current Web. It brings to the Web the idea of having data defined and linked in a way that it can be used for more effective discovery, automation, integration, and reuse across various applications.

The challenge of the Semantic Web, therefore, is to provide a language that expresses both data and rules for reasoning about the data and that allows rules from any existing knowledge-representation system to be exported

---

1.1. THE SEMANTIC WEB

Two important technologies for developing the Semantic Web are already in place: the eXtensible Markup Language (XML) and the Resource Description Framework (RDF). XML lets everyone create their own tags, briefly, XML allows users to add arbitrary structure to their documents but says nothing about what the structures mean. In fact meaning is expressed by RDF, which encodes it in sets of triples, each triple being rather like the subject, verb and object of an elementary sentence.

The triples of RDF form webs of information about related things. Because RDF uses URIs to encode this information in a document, the URIs ensure that concepts are not just words in a document but are tied to a unique definition that everyone can find on the Web.

Another fault of the current web (and web technologies) is that as human language thrives when using the same term to mean somewhat different things, automation does not.

A solution to this problem is provided by the third basic component of the Semantic Web; collections of information called ontologies. A typical ontology set for the Web has a taxonomy and a set of inference rules. The taxonomy defines classes of objects and relations among them. Inference rules supply further power.

Really the computer does not truly "understand" any of this information, but it can in this manner manipulate the terms much more effectively in ways that are useful and meaningful to the human user, besides ontology pages, solutions to terminology (and other) problems begin to emerge.

The real power of the Semantic Web will be to create an environment where software agents, roaming from page to page, can readily carry out sophisticated tasks for users. The effectiveness of such software agents will increase exponentially as more machine-readable Web content and automated services (including other agents) become available. Even agents that were not expressly designed to work together, can transfer data among themselves, when the data come with semantics. As we see in 2.8 the consumer and producer agents can reach a shared understanding by exchanging ontologies, which provide the vocabulary needed for discussion. This crux is the basis of the work presented in this thesis.

Lastly the Semantic Web will break out of the virtual realm and extend into our physical world [7]. URIs can point to anything, including physical entities, which means we can use the RDF language to describe devices such as cell phones and TVs. Such devices can advertise their functionality, much like agents or web services, and realize with simple semantic descriptions features of the so called "home automation".
1.1.3 The Semantic Web Architecture

The Web was designed as an information space; for the human and machine consumption but most information on the Web is designed only for human-human communication. The Semantic Web address to a Web in which machine reasoning will be omnipresent and highly powerful. We can define the Semantic Web as a *web of data* (i.e. like a global database). An outlined architecture of the Semantic Web (not yet a reality) is done by Tim Berners Lee in [5, 46, 6] and in many other issues available on line. In Figure 1.1 we show such architecture.

![The Semantic Web “layer cake” as presented by Tim Berners Lee](image)

At the basis of the entire Semantic Web there are Unicode (an evolution of ANSI/ASCII character set) and Universal Resource Identifier (URI) [15]. URIs can be viewed as resource static mapping that creates the navigable ”space” (thus they are identifiers not recipes). The second layer is composed by the eXtensible Markup Language (XML), XML Namespace and XML Schema which constitute the foundation of the languages used in the Web. XML for instance provide a basic format for structured documents, with no particular semantics.

The RDF and RDF Schema layer include: a basic assertion model, based on a common model of great generality: the Resource Description Framework (RDF), and a schema layer which allows one to declare the existence of new properties and classes and to create range and domains for properties. The RDF being general is very simple, and being simple there is nothing much you
1.1. THE SEMANTIC WEB

can do with the model itself without layering many things on top. Anyway the basic model contains the concept of assertion and quotation, though it will be needed later, applications at this level (using RDF only for metadata) are very numerous (e.g. Dubblin Core, intellectual property right, privacy information). RDF documents at this level are not very potent, we further need a schema layer to the existence of new property and also express a little more, typically we want to constrain the type of object it can apply to. This is allowed by another language the RDF Vocabulary Description Language (RDF Schema) designed to be a simple data typing model for RDF.

Next we have the ontology layer, which is widely discussed in this thesis. It originates with need for more meta information, such as: transitive property, unique, unambiguous, cardinality, etc. and gives a huge extra usage for extra functionality. There are many ontology languages on the web: the Ontology Interchange Language (OIL), the Simple HTML Ontology Extension (SHOE) [32], the DARPA Markup Language (DAML) [2] are more popular. Most interesting is DAML+OIL [27] provided by the W3C, recently renamed as Ontology Web Language (OWL) [12]. DAML+OIL is an ontology and inference language based upon RDF. It takes RDF Schema a step further, by giving us more in depth properties and classes, and allowing more expressiveness than RDF Schema, such as: inverses, unambiguous properties, unique properties, lists, restrictions, and so on. With this more powerful schema concepts, a sort of inference can be done using DAML+OIL or OWL. The principle of "inference" is quite a simple one: being able to derive new data, from data that you already know. For example querying is a form of inference (in a mathematical sense qua being able to infer some search results from a mass of data). Unfortunately, great levels of inference can only be provided using "First Order Predicate Logic" (FOPL) languages, and DAML is not entirely a FOPL.

Inference is one of the driving principles of the Semantic Web, because it will allow us to create Semantic Web applications quite easily. It is considered the key for allowing important agent’s behavior on the Web.

The next layer, is the logic layer. It will be necessary to construct a powerful logical language for making inferences. We need a way of writing logic into documents to allow such things as: rules the deduction of one type of document from a document of another type; the checking of a document against a set of rules of self-consistency; and the resolution of a query by conversion from unknown terms into known terms . In other words this layer attempts to turn a limited declarative language into a Turing-complete logical language, with inference and functions. One can see this language as being a universal language to unify all data systems just as HTML was a language to unify all human documentation systems. Many engines with
inference capabilities exist but there is not one standard engine. There is still a great amount of uncertainty about which logical language to choose for the Semantic Web.

The last two layers are proof and trust. There is very little literature written about this layer since it will become very important in the future. A proof language is a form of RDF which allows one agent to send to another an assertion, together with the inference path to that assertion. Applications uses a generic validation engine with very case-specific tools for producing proofs according to whatever social rules have been devised for the case. One could see this engine being based on the logical layer, or being based on a less expressive rules layer.

Once we have a proof language, digital signatures turn what was a web of reason into a web of trust. Digital signatures are simply little bits of code that can be used to verify that a certain entity wrote a certain document. This is a familiar technology used in PGP to encrypt and sign messages. Here simply this technology is applied to the RDF world. Digital signatures are developed in parallel with the stages above. As more expressive logical languages become available we must define new primitives which describe digital signature, feeding the language itself into a trust engine (an interface engine that has a little digital signature checker built into it).

1.2 Ontologies on the Semantic Web

1.2.1 The Term Ontology

There are a number of vocabulary terms which the Computer Science community uses and sometimes abuses. These terms become even more confusing when used in different research contexts. One example of such a term is ontology. It has a long history in philosophy, in which it refers to the subject of existence. The Artificial-Intelligence literature contains many definitions of ontology; many of these contradict one another. In the context of knowledge sharing the most notable definition is given by Gruber [33]:

"An ontology is a specification of a conceptualization"

That is, an ontology is a description (like a formal specification of a program) of the concepts and relationship that can exist for an agent or a community of agents. This view is a key idea for the subject treated in Chapter 3. In designing ontologies for the purpose of enabling knowledge sharing and reuse, an ontology is a specification used for making ontological commitments, that
1.2. ONTOLOGIES ON THE SEMANTIC WEB

is, an agreement to use a vocabulary in a consistent (but not complete) way, with respect to the theory specified by the ontology. The body of formally represented knowledge, is based on a conceptualization: the objects, concepts and other entities in some area of interest, and relationships that hold among them. A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose [33]. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. We use common ontologies to describe ontological commitments for a set of agents, so that they can communicate about domain of discourse without operating on a globally shared theory.

1.2.2 Semantic Web Ontologies

The word ontology is currently being used also in the Semantic Web circles. The meaning in this context is quite similar to those given for knowledge sharing but obviously targeted for this matter. Hendler in [34] define ontologies as

"A set of knowledge terms, including the vocabulary, the semantic interconnections and some simple rules of inference and logic, for some particular topic."

In this manner we can express ontology inference and simple logics for the vocabulary terms, while more complex logics and inference systems are considered separate from the ontology according to Tim Berners-Lee’s view of the Semantic Web shown in fig.1.1. Essentially here is needed the presence of an acknowledged standard ontology language. It was provided by a number of researchers supported by the US Defense Advanced Research Projects Agency (DARPA) [2] which released the DARPA Agent Markup Language (DAML) [27] discussed in section 2.4.

Once a universally agreed ontology language exists, it is expected for the Semantic Web to have a great number of small ontological components largely connected with pointers to each other and developed by web users, in much the same way that web content is currently created. In contrast to the vision of a few large, complex and consistent ontologies, shared by great numbers of users, such the AI researcher’s wish. However, to concretely realize this ability a phenomenon similar to the early days of the web must occur. It is necessary to make it possible for a number of different users to create the machine-readable web content without being logic experts. Moreover not only can pages be created with links to numerous ontologies, but also
the ontologies themselves can include links between them, to reuse terms. Anyway, also in the context of Semantic Web (as in the AI context) the real power of ontologies is in sharing, and the more people use common terms, the better it is.

1.2.3 The Power of Ontologies

We build agents that commit to ontologies, and we design ontologies so we can share knowledge with and among these agents. This is one of the most common use of ontologies to describe ontological commitments for a set of agents, so that they can communicate about a domain of discourse without necessarily operating on a globally shared theory. This idea is based on the Knowledge Level prospective, which is a description of the knowledge of an agent that is independent of the Symbol-Level representation internally used by the agent. Practically, a common ontology defines the vocabulary with which queries and assertions are exchanged among agents. In this thesis we made a hypothesis to combine the power of ontology sharing for agents with the expressive power of Semantic Web languages, that using Cranefield’s UML Data Binding (section 2.8) can drive to a new architecture for Agent Communications Languages (ACL). This hypothesis is better argued in 3.3.

1.3 Software Agents

In the following sections agents are introduced. This exposition is intentionally not exhaustive, it is aimed to give the general idea of agents and agents’ features, without regard to important formal and architectural aspects. Other characteristics related to agents and in particular to Agent Oriented Software Engineering can be found after in section 3.2. However to give a complete panorama of all software agent features is beyond the purpose of this thesis and many aspects have been left out.

1.3.1 What is an Agent?

"We have as much chance of agreeing on a consensus definition for the word "agent" as AI researchers have of arriving at one for "intelligence". " ².

There is a general difficulty in defining precisely what agents are. From the advent of software agents, there has been much discussion about what such agents are and how they differ from common programs. So we can say

²Carl Hewitt at the thirteenth international workshop on distributed AI.
that there is not an universally accepted (or valid) formal definition of agent, and probably there will never exists.

Agent is a term used in many heterogeneous research and development fields. Workers involved in agent research have offered a variety of definitions, each hoping to explicate his or her use of the word "agent" and grew directly out of the set of examples of agent the definer had in mind. A rational way to find a meeting point for this definition, has been to compare these explanations and search for a more general definition of agent. Franklin in [29] do exactly this and bring out this definition:

"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in a pursuit of its own agenda and so as to effect what it senses in the future.”

What is highlighted in such a definition is the autonomy and that software agent "live" in its environment. So a more simple and recent definition is given by Michael Wooldridge in [48]:

"An agent is a computer system capable of autonomous action in some environment”

That seems more friendly.

Great attention is posed also on the definition of boundary between a simple program and an agent. For example a payroll program sense the world via its input and act on it via its output, but is not an agent because its output would not normally effect what is sense later. It runs once and go into a coma, waiting to be called again [29]. Thus another important point is that all software agents are programs, but not all programs are agents.

However I think that the utility of these definitions is quite insignificant. The multiplicity of roles that agents can play necessitate further restrictions that must be added to define more particular classes of agent. So, the real question is not define general requirements for agents, but to find a common agent classification that makes clear what may be considered an agent and what does not. We find more jumble from this viewpoint because a classification is much more subjective than a definition.

1.3.2 Agents Classification

As result of lack in a general definition of agent, some researcher had to invent specialized synonymous of this term. This brought to a vast collection of adjectives which precede the world ‘agent’: search agents, report
agents, mobile agents, navigation agents, role-playing agents, management agents, search and retrieval agents, domain-specific agents and many others. We prefer thinking the existence of agents in a *multi-dimensional space* [37] (meaning that we express some characteristics of agents which may appear in heterogeneous agent systems) rather than report a specific agent taxonomy. Firstly, agents may be classified by their mobility (i.e. by their ability to move around some network). This yields the classes of static or mobile agents. Secondly, they may be classed as either deliberative or reactive. Deliberative agents possess an internal symbolic, reasoning model and they engage in planning and negotiation in order to achieve coordination with other agents. Reactive agents on the contrary do not have any internal, symbolic models of their environment, and they act using a stimulus/response type of behaviour by responding to the present state of the environment in which they are embedded. Thirdly, agents may be classified along several ideal and primary attributes which agents should exhibit. A minimal list of those is: autonomy, learning and cooperation

- Autonomy refers to the principle that agents can operate on their own without the need for human guidance, and have some kind of control over their action and internal state. [50].

- Cooperation (with other agents) is the raison d’être for having multiple agents in the first place, in contrast to having just one. In order to cooperate, agents need to possess a social ability, i.e. the ability to interact with other agents, and possibly humans, via some communication language [50].

- Lastly, agents would have to learn as they react and/or interact with their external environment. Learning in this way may also take the form of increased performance over time.

There are other attributes of agents which may be considered secondary to those already mentioned. For example, an agent could be versatile (i.e. it has many goals or it engages in a variety of tasks); some researchers are also attributing emotional attitudes to agents (i.e. agent get ’fed up’), agents are also imbued with mentalistic attitudes or notions such as Beliefs, Desires and Intentions - BDI agents -.

We use these characteristics to derive other types of agents and we can generate in this way a typology which join or map many existing agent types and classifications. What follow shows an overview of some relevant agent types:
1.3. SOFTWARE AGENTS

Collaborative Agents

They are agents that emphasise autonomy and cooperation (with other agents) in order to perform their tasks. They have to *negotiate* in order to reach mutually acceptable agreements on some matters. Key general characteristics of these agents include autonomy (with pro-activness i.e. "take the initiative"), social ability and responsivness.

Interface Agents

They emphasize autonomy and learning in order to perform tasks for their owners. Pettie Maes, one proponent of this class of agents and one of the pioneers of agent research, points out that the key metaphor for interface agents is that of a *personal assistant* who collaborates with the user in the same environment. The objective of interface agents research (as Maes sees it) is to work towards Alan Kay’s dream of having indirectly managed human-computer interfaces [36].

Mobile Agents

Mobile agents are computational software processes capable of roaming wide area networks (WANs) such as the WWW. Features of this agents are: interacting with foreign hosts, gathering information on behalf of its owner and coming 'back home' having performed the duties set by its user. These duties may range from a flight reservation to managing a telecommunications network. However, as we define in 1.3.1 mobility is neither a necessary nor sufficient condition for agenthood. Mobile agents are agents because they are autonomous and they cooperate (albeit differently to collaborative agents). The public perception of agents is almost synonymous with mobile agents.

Internet Agents

Internet or information agents have to help us manage the explosive growth of information we are experiencing currently, and which we will continue to experience henceforth. Information agents perform the role of managing, manipulating or collating information from many distributed sources. They have varying characteristics: they may be static or mobile; they be non-cooperative or social; and they may or may not learn.
Reactive Agents

Reactive agents are a special category of agents which do not possess internal, symbolic models of their environments. They act/respond in a stimulus-response manner to the actual state of the environment in which they are embedded.

Hybrid Agents

Since each item of the exposed typology has its own strengths and deficiencies, one way for maximise the strengths and minimise the deficiencies is to adopt a hybrid approach. Hybrid agents refer to those who attempt to combine two or more agent philosophies within a singular agent or agent architecture.

1.3.3 A personal idea of agent

With regard to the definitions and the taxonomy exposed above for agents, it is clear that presently there is not in the scientific community a universally agreed and complete formal definition of what an agent is. Specially reading [29] I mean that every agent researcher has his intuition of what an agent is, and the difference between a software agent and a common program. However, when these people try to give a formal definition or a classification of agents, it becomes naturally limited to her or his specific research scope. Only the Wooldridge definition [50, 52, 51, 48], detailed in 3.2.1, seems to be general and valid enough for a large computer science camps of research.

Since agents as envisioned by Tim Berners Lee, and all people involved in agent studies, will become in a relatively distant future an everyday reality, my own opinion is that it would be proper, in addition to the other, a non-formal definition that does not explain what an agent is or what an agent does, but instead generally defines what is the natural idea of an agent. I mean a definition that easily expresses the intuition of agent, as it is done for other computer science entities: software, programs, classes, internet, web.

This is my personal, general, non-formal, from anyone understandable definition of agent:

"An agent is a technology which simulates human behavior and the functionalities of the human brain to carry out its tasks."

It is less AI-like than one could think. It is naive enough to include a very large formal definition of the research community and actual existents soft-
ware agents. It is so simple that it can be understood also by a child. May be just another definition to the hundred yet given and that will be give in.

1.4 Thesis Overview

In this chapter the three main concepts at the basis of our work have been presented i.e. Semantic Web, Ontologies and Agents.

Chapter 2 will investigate some significant technologies which involve the study of the concepts presented in this chapter. The eXtensible Markup Language (XML) is presented together with some fundamental XML family languages, such as XML Namespaces, XML Path Language (XPath), XML Stylesheet Language (XSL). Particular emphasis is given to the Resource Description Language (RDF), RDF Vocabulary Description Language (RDF Schema) and the DARPA Agent Markup Language + Ontology Interchange Language (DAML+OIL) due to their relevance in the matters treated. An overview of the Unified Modelling Language (UML) is also exposed given its importance for model ontologies in the UML Data Binding (UDB) technology. Hence, the UDB procedure to transform UML diagrams in RDF schema and Java classes is given together with its extension for the abstract syntaxes of Agent Communication Languages (ACLs).

In chapter 3 the attention is moved from the technologies to the methodologies. Some methodologies for the development of web applications are exposed, among which the Ontology-Driven Design for Semantic Web applications (ODD). Also an overview of the Agent Oriented Software Engineering (AOSE) is given and the Gaia methodology is presented. The chapter ends with an application hypothesis of the ODD to the development of agents.

Chapter 4 describes an improvement of the UDB architecture and an application scenario. The UDB architecture is revisited allowing DAML+OIL mapping and improving benefits for agent utilization. This new architecture is called UML Knowledge Representation (UKR). The UML to DAML mapping is analyzed and an early uml-to-daml stylesheet for UKR is produced. An application scenario is finally presented with the Validation Exams Software Agent (VESA) which using ontologies and UKR will be capable of comparing and converting exams. A draft analysis and design for VESA is exposed as an application of the Ontology Driven method for agents (not "formally" defined) hypothesized in chapter 3. The last two sections of chapter 4 give: a general idea which may guide the development of ontologies with agents in the semantic web, and conclusions with related future works for the topics exposed throughout the thesis.
2.1 The eXtensible Markup Language (XML)

The Extensible Markup Language (XML) is a meta-language of markup that allows exchange and use of structured documents over the internet. The design of XML was motivated by two observations: the inflexibility of HTML and the complexity of SGML [14].

Indeed XML is a subset of SGML (that would have served the same purpose as XML) defined to avoid some complexity of SGML, which makes it unusable. The main feature of SGML is the utilization of Document Type Definition (DTD), that allows to define grammars for elements written in a document. Since HTML implements only one particular document model, the XML has been defined making it possible to use documents of application-specific document types, which can be created, distributed and interpreted in an XML environment. There are many benefits associated with the use of XML, here a list of the most important [47]:

- **Self describing documents**: XML allows for the ability to include DTDs inside the document. This means that a single XML document can contain the rules used to compose the document together with the actual content.

- **Browseable document structure**: An XML document can be easily analyzed by XML software, even if it does not contain a DTD (it is said to be a well-formed document). Consequently, even documents which are not self describing can still be interpreted and browsed.

- **Platform-Independence**: XML is an open standard and can be implemented by anyone interested in XML applications.
Data Interchange: One common issue of data interchange is the difference of structural aspects of the data to be exchanged. We can think of XML as an intermediate syntax to express easily complex structured data.

Document Publishing: XML is perfect to express structured documents, independently from their final destination, thus the same document can be transformed for the web, cellular phone, e-book etc.

Figure 2.1 show a little DTD with a valid document, i.e. conforms to the structure defined by the DTD itself.

With XML there have been developed a set of associated languages (called XML-family languages) : for links, namespaces, stylesheet and information description, exposed in the next sections.

2.1.1 XML Namespaces

XML namespaces provide a simple method for qualifying element and attribute names used in Extensible Markup Language documents, by associating them with namespaces identified by URI references. It is a W3C standard [16] motivated by the envision that a single application of Extensible Markup Language (XML) may contain elements and attributes that are defined and used by multiple software modules. If a markup vocabulary exists which is well-understood and for which there is useful software available, it is better to re-use this markup rather than re-invent it.

An XML namespace is a collection of names, identified by a URI reference, it is declared using a family of reserved attributes. Such an attribute’s name must either be xmlns (default name) or have xmlns: as a prefix. The attribute’s value, a URI reference, is the namespace name identifying the namespace. It should have the characteristics of uniqueness and persistence because it identifies on the Internet the namespace we are using.

Figure 2.2 present an example of namespaces usage.

2.1.2 XML Path Language (XPath)

XPath is a language for addressing parts of an XML document. XPath operates on the abstract, logical structure (known as the data model) of an XML document, rather than its surface syntax, and is designed to be embedded in a host language such as XSLT 2.0. The basic building block of XPath is the expression that returns one of the following types of objects: boolean, string, number or set of nodes. The most important expression
2.1. THE EXTENSIBLE MARKUP LANGUAGE (XML)

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

<!ELEMENT TESI (FRONTESPIZIO, RINGRAZIAMENTO?,
  CAPITULO+)>
<!ELEMENT FRONTESPIZIO (TITOLO, AUTORE+, RELATORE+)>  
<!ELEMENT TITOLO (#PCDATA)> 
<!ELEMENT AUTORE (#PCDATA)> 
<!ELEMENT RELATORE (#PCDATA)> 
<!ATTLIST RELATORE
  titolo (Chiar.m. Prof. | Prof.) #IMPLIED
  materia CDATA #REQUIRED >

<!ELEMENT RINGRAZIAMENTO (#PCDATA)> 
<!ELEMENT CAPITULO (#PCDATA | NOME | SEZIONE)*> 
<!ELEMENT NOME (#PCDATA)> 
<!ELEMENT SEZIONE (#PCDATA, SOTTOSEZIONE?)+> 
<!ELEMENT SOTTOSEZIONE (#PCDATA)> 

<TESI>
  <FRONTESPIZIO>
    <TITOLO> NOME e ontologia nel Semantic Web </TITOLO>
    <AUTORE> Luca Abeti </AUTORE>
    <RELATORE titolo="Chiar.m. Prof. materia="Ing. del software">
      Paolo Ciancarini
    </RELATORE>
  </FRONTESPIZIO>

  <CAPITULO>
    <NOME>Introduzione</NOME>
    Capitolo introduttivo

    <SEZIONE>
      In questa sezione introduce XML
    </SEZIONE>

    <SEZIONE>
      In questa sezione c'è una introduzione ai linguaggi della famiglia XML
    </SEZIONE>

    <SOTTOSEZIONE>
      Parliamo di XPath e delle sue relazioni con XML ad esempio...
    </SOTTOSEZIONE>

  </SEZIONE>

  <CAPITULO>
  </CAPITULO>

</TESI>
```

Figure 2.1: An example of XML DTD with a valid XML file
Figure 2.2: An example of namespace usage

type is *Path Expression* that can be used to locate nodes within a tree. Path expressions may be relative or absolute (beginning with "/") and consist of a series of one or more steps, separated by "/". These steps called *Step Expression* are composed of three parts:

- **Axis**: which defines the "direction of movement" for the step starting by the *context node* (es. child, parent, self)

- **Node Test**: which specifies the node kind and/or name of the nodes to be selected by the step (es. para, text())

- **Predicate**: which consists of an expression, called a predicate expression, enclosed in square brackets.

An example of an XPath expression that returns all the "section" that descend from a "chapter" may be in the form of:

```
/child::doc/child::chapter/descendant::section
```

### 2.1.3 XML Stylesheet Language (XSL)

XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying
2.1. THE EXTENSIBLE MARKUP LANGUAGE (XML)

formatting. XSL is far more important for XML than Cascading Style Sheet (CSS) for HTML, because HTML specifies semantics and standard formatting for all elements (modified using CSS), while XML documents do not contain any formatting information, unless specified by a style sheet. Consequently, without any stylesheet information, XML documents can only be displayed in a structure-oriented way, for example representing the document tree. A transformation expressed in XSLT is called a stylesheet because, in the case when XSLT is transforming into the XSL formatting vocabulary (XSL-FO), the transformation has the functions of a stylesheet.

XSLT is a language for transforming XML documents into other XML documents. A transformation expressed in XSLT describes rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree, and a template is instantiated to create part of the result tree. The structure of the result tree can be completely different from the structure of the source tree; elements from the source tree can be filtered and reordered, while arbitrary structure can be added.

Thus an XSLT stylesheet contains a set of template rules. A template rule has two parts: a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree. This allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

XSLT makes use of the expression language defined by XPath for selecting elements to process, to realize conditional processing and for generating text.

A stylesheet is represented by a `<xsl:stylesheet` element in an XML document, with a version attribute indicating the version of XSLT that the stylesheet requires:

```xml
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="P">
  <H2><xsl:apply-templates/></H2>
</xsl:template>

</xsl:stylesheet>
```

This simple stylesheet transforms all the element P in H2, and may be used for example, on a HTML document (qua exist also a DTD XML for HTML [11]). XSLT can be useful in many environments in which XML is used, we will see in section 2.7 an interesting application of stylesheet to produce XML documents and Java Classes.
2.1.4 Other XML Languages

There are many other XML based languages, and others will appear in the future to meet different needs of the Semantic Web. I have focused on the most important (Standard or Recommendation of the W3C [16] [18] [17]) omitting XPointer, XLink, XBase, XSL-FO because their discussion goes beyond the scope of this thesis. In the next sections I elaborate on two other XML languages that constitute the basis for our discussion; RDF and OWL.

2.2 Resource Description Framework (RDF)

All the XML-family languages are aimed to generate information not only for human consumption but also usable for automated applications, as exposed in section 1.1.2. There is not information in an HTML document that denotes what topic the document treats or the source of the information. Metadata allows the authors of documents to specify such type of information.

Metadata are "data about data" (for example, a library catalog that describes publications) and specifically in this context "data describing Web resources". The distinction between "data" and "metadata" is not an absolute one; it is a distinction created primarily by a particular application, and many times the same resource will be interpreted in both ways simultaneously.

Resource Description Framework (RDF) is a foundation for processing metadata. It provides interoperability between applications that exchange machine-understandable information on the Web.

The foundation of RDF is a model for representing named properties and property values mentioned RDF data model, it is a syntax-neutral way of representing RDF expressions. Two RDF expressions are equivalent if and only if their data model representations are the same.

The basic data model consists of three concepts:

- **Resources**: Any entity imaginable may be a resource described by RDF expressions. An entire Web page; a part of a Web page (e.g. a specific HTML or XML element); an entire Web site, are all resources. Moreover an object may also be considered a resource that is not directly accessible via the Web (e.g. a car). Resources are always named by URIs [15], and the extensibility of URIs allows the introduction of identifiers for anything.

- **Properties**: Any aspect, characteristic, attribute, or relation used to describe a resource is a property. Each property has a specific meaning, permitted values and its relationship with other properties.


- **Statements:** The Resource (subject) together with a named property (predicate) and the value of that property for that resource(object) is an RDF statement. The object of a statement can be another resource (a URI) or it can be a literal (a simple string).

The main goal of RDF is to define a mechanism for describing resources that makes no assumptions about a particular application domain, nor defines the semantics of any language. RDF properties may be thought of as attributes of resources and in this sense correspond to traditional attribute-value pairs. We can also represent relationships between resources, and think an RDF model as an entity-relationship diagram. In an object-oriented design view, resources correspond to objects and properties correspond to instance variables.

**Examples**

Consider this simple sentence:

*Luca Abeti is the creator of the resource http://www.cs.unibo.it/ abeti*

we can recognize the following data model:

- Resource: http://www.cs.unibo.it/ abeti
- Property: Creator
- Statement:

<table>
<thead>
<tr>
<th>Subject</th>
<th><a href="http://www.cs.unibo.it/">http://www.cs.unibo.it/</a> abeti</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td>Creator</td>
</tr>
<tr>
<td>Object</td>
<td>“Luca Abeti”</td>
</tr>
</tbody>
</table>

Figure 2.3: A pictorial representation of an RDF statement
As shown in figure 2.3 we can also represent RDF statements pictorially using directed labeled graphs. Where nodes (ovals) represent resources and arcs represent named properties. Literals will be drawn as rectangles.

A little more complex sentence may be:

The individual referred to ID 101115 named Luca Abeti is the owner of the car with ID AP121091 model Volkswagen Beetle black colored.

For briefness we expose only the visual representation of this sentence in fig. 2.4.

![Figure 2.4: More complex RDF statements representations](image.png)

### 2.2.1 RDF Serialisation Syntax

With the RDF data model we have a manner for representing metadata, but for our purpose a concrete syntax for encoding and transporting this metadata is also needed. The syntax used by the W3C in the definition of RDF [13] uses XML.

A concrete syntax is essentially required for the purposes of creating and exchanging metadata, so RDF also needs the XML namespace facility to precisely associate each property with the schema (see section 2.3) that defines the property. The formal namespace name of the RDF specification is: http://www.w3.org/1999/02/22-rdf-syntax-ns\#.
2.2. RESOURCE DESCRIPTION FRAMEWORK (RDF)

Basic syntax

Commonly several properties of a resource will be given together, it is possible to group multiple statements for the same resource into a unique Description element. The Description element names, in an about attribute, the resource to which each of the statements apply and supply an identifier ID attribute to indicate the resource do not yet exist. The value assigned to the about or ID attribute is called the referent, and is respectively the URI associated to the resource or an identifier unique in the document.

Within a Description element we may have several properties related to the resource specified in the about attribute. The value of a property can be another resource or a literal element. The resource attribute is used to specify other resources identified by URIs.

Property names must be associated with a schema. This can be done by qualifying the element names with a namespace prefix to unambiguously connect the property definition with the corresponding RDF schema or by declaring a default namespace as stated in 2.1.1.

We have shown an examples of this syntax in fig.2.3:

```xml
<?xml version="1.0"?>
<RDF
 xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:l="http://description.org/schema/">
  <Description about="http://www.cs.unibo.it/~abeti">
    <l:Creator>Luca Abeti</l:Creator>
  </Description>
</RDF>
```

Containers

As is common to refer to a set of properties of the same resource, so frequently it is necessary to collect resources. RDF containers are used to hold such lists of resources or literals. There are three types of container objects:

- Bag: An unordered list of resources or literals.
- Sequence: An ordered list of resources or literals.
- Alternative: A list of resources or literals that represent alternatives for the (single) value of a property.

In order to represent a collection of resources, RDF uses an additional resource that is an instance of a collection. This resource must be declared to be an instance of one of the container object types defined above using the property type.

Here is an example that uses bag container for the statement:

*The members of Abeti’s family are Luca, Gianni, Angela, and Sara.*

```xml
<rdf:RDF>
  <rdf:Description about="http://family.com/it/ap/abeti">
    <f:members>
      <rdf:Bag>
      </rdf:Bag>
    </f:members>
  </rdf:Description>
</rdf:RDF>
```

**Reification**

RDF can be used for making statements about other RDF statements; these are higher-order statements. This process is formally called reification in Knowledge Representation terminology.

To make a statement about another statement, we must attach properties to the original statement. This is made adding a type property to the original subject-predicate-object model of the statement. The value of the type property describes the type of the new resource and is also used more generally to declare the type of any resource as shown for the containers. For example, let me consider the sentence:

*Elisabetta says that Luca is the creator of the resource http://lucaabeti.com*

we can represent this assertion in RDF with reification:
2.3 RDF VOCABULARY DESCRIPTION LANGUAGE (RDF SCHEMA)

The Resource Description Framework (RDF) is a general-purpose language for representing information in the Web, but does not support more sophisticated RDF vocabulary description. RDF defines a simple model for describing relationships among resources in terms of named properties and values. RDF Schema add the ability to say certain things about certain kinds of resources.

RDF Schema does not constrain the document structures, but allows information for understanding the document itself. The Schema is specified in terms of the basic RDF information model as a set of RDF resources to be used to describe characteristics of other RDF resources or properties.

We present an overview of the basic vocabulary of RDF:

- **rdfs:Resource**: All things described by RDF are called resources, and are members of this class.

- **rdfs:Literal**: This class represents the class of literal values such as strings and integers. Property values such as textual strings are examples of RDF literals.

- **rdfs:Property**: Represents those resources that are RDF properties. Is subclass of rdfs:Resource.
CHAPTER 2. TECHNOLOGIES

Figure 2.5: Class Hierarchy for the RDF Schema

- **rdfs:Class**: This corresponds to the generic concept of a type or category of resource (i.e. Class) of object oriented programming. Defining a new class, the rdf:type property must be rdfs:Class.

- **rdfs:subClassOf**: This property represents a specialization relationship between classes of resource. The rdfs:subClassOf property is transitive.

- **rdfs:subPropertyOf**: The property rdfs:subPropertyOf is an instance of rdf:Property that is used to specify that one property is a specialization of another. Sub-property hierarchies can be used to express hierarchies of range and domain constraints.

- **rdfs:range**: Is an instance of rdf:Property that is used to indicate the class or classes that the values of a property will be members of. The value of an rdfs:range property is always a Class and is only applied to properties.

- **rdfs:domain**: Is an instance of rdf:Property that is used to indicate the class or classes that will have as members any resource that has the indicated property.

In figure 2.5 relations among these concepts are depicted.
2.4 DAML+OIL and OWL

The DARPA Agent Markup Language (DAML) + Ontology Interchange Language (OIL) is a semantic markup language for web resources. DAML+OIL was born to combine features of other Semantic Web ontology language such as OIL and SHOE (Simple HTML ontology Extension) [32] for extend RDF and RDF Schema with richer modelling primitives. DAML+OIL’s ontologies (or knowledge bases) are a set of RDF triples specified in a DAML+OIL vocabulary. They are structured in: an optional and repeatable headers followed by zero or more class element, property elements and instances.

Headers permit to assert inside the RDF document "this is an ontology", adding further information like version and references to other DAML+OIL ontologies.

An example of a header may be the following:

```xml
<daml:Ontology rdf:about=" ">
  <daml:versionInfo> 0.23 Beta</daml:versionInfo>
  <rdfs:comment> An example ontology </rdfs:comment>
</daml:Ontology>
```

Class element, `daml:Class` is used to define object classes and is a subclass of `rdfs:Class`. It may contain many elements among which `rdfs:subClassOf`, `daml:disjointWith`, `daml:equivalentTo` are the most significative. `daml:Class` also permit property restriction and boolean combination of class expressions (e.g. class and other class expressions).

DAML+OIL provide primitives for defining properties, extending the ones available in RDF, such as `daml:samePropertyAs` or `daml:inverseOf` elements. Moreover it is possible to assert within the property additional information about it, as transitivity, uniqueness and non ambiguity.

One of the key features of any web-based ontology language we find also in DAML+OIL, is that statements about entities (classes and properties) can be distributed among different locations. This is possible because an assertion about an element has the same logic status as the assertion made within the `rdf:ID` attribute in the element.

Figure 2.6 shows a short example of a DAML+OIL ontology.

Instances of both classes and property (i.e. object and pairs) are written in RDF and RDF Schema syntax.

As a revision of DAML+OIL, the W3C Web Ontology Working Group are designing OWL Web Ontology Language [12]. In fact a number of problems have been discovered in the design of DAML+OIL, mostly related to the
\begin{verbatim}
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:daml="http://www.w3.org/2001/12/daml+owl#"
>
  <daml:ontology rdf:about="#">
    
    <!-- A simple ontology in DAML+OIL -->

    <daml:versionInfo sid: daml:example ver 0.2/>

    <rdfs:comment>
      A simple ontology in DAML+OIL
    </rdfs:comment>

    <daml:Class rdf:ID='Person'>
      <rdfs:label>Person</rdfs:label>
      <rdfs:comment>Class of person</rdfs:comment>
    </daml:Class>

    <daml:Class rdf:ID='Male'>
      <rdfs:subClassOf rdf:resource='#Person' />
      <daml.disjointWith rdf:resource='#Female' />
    </daml:Class>

    <daml:Class rdf:ID='Female'>
      <rdfs:subClassOf rdf:resource='#Person' />
      <daml.disjointWith rdf:resource='#Male' />
    </daml:Class>

  </daml:ontology>

</rdf:RDF>
\end{verbatim}

Figure 2.6: A DAML+OIL representation of an ontology
changes in the newest versions of RDF. However there are not substantial changes from DAML+OIL, and the abstract syntax of OWL can easily be viewed as an abstract syntax for DAML+OIL. The most important changes are:

- the removal of qualified number restrictions
- the ability to directly state that properties can be symmetric
- the removal of some unusual DAML+OIL constructs.

OWL is a set of three, increasingly complex languages.

1. **OWL Lite** defined with the intention of creating a simple language that will satisfy users primarily needing an ontology language.

2. **OWL DL** includes the complete OWL vocabulary, interpreted under a number of simple constraints.

3. **OWL Full** includes the complete OWL vocabulary, interpreted more broadly than in OWL DL, with the freedom provided by RDF.

Because OWL is not yet largely utilized and supported by many tools, in this thesis DAML+OIL will be referred to the Ontology Language for the Semantic Web.

## 2.5 The Unified Modelling Language (UML)

The Unified Modelling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems. The UML represents a collection of the best engineering practices that have proven successful in the modelling of large and complex systems. It focuses on a standard modelling language, not a standard process, thus UML must be applied in the context of different development processes. UML provide users with a ready-to-use, expressive visual modelling language to develop and exchange meaningful models with extensibility mechanisms to extend the core concepts.

UML is structured in a four-layer metamodel architecture. The meaning of each layer is shown in the Table below:
### Layer Description Example

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>meta-metamodel</strong></td>
<td>The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.</td>
<td>MetaClass, MetaAttribute, MetaOperation</td>
</tr>
<tr>
<td><strong>metamodel</strong></td>
<td>An instance of a metamodel. Defines the language for specifying a model.</td>
<td>Class, Attribute, Operation, Component</td>
</tr>
<tr>
<td><strong>model</strong></td>
<td>An instance of a metamodel. Defines a language to describe an information domain.</td>
<td>StockShare, askPrice, sellLimitOrder, StockQuoteServer</td>
</tr>
<tr>
<td><strong>user objects</strong></td>
<td>An instance of a model. Defines a specific selllimitorder, information domain.</td>
<td>SoftwareShare98789, 654.56, StockQuoteSvr32123</td>
</tr>
</tbody>
</table>

The meta-metamodeling layer forms the foundation for the metamodeling architecture. The primary responsibility of this layer is to define the language for specifying a metamodel.

A metamodel is an instance of a meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models. Metamodels are typically more elaborate than the meta-metamodels that describe them, especially when they define dynamic semantics.

A model is an instance of a metamodel. The primary responsibility of the model layer is to define a language that describes an information domain.

User objects are an instance of a model. The primary responsibility of the user objects layer is to describe a specific information domain.

UML is structured within the metamodel layer. The UML metamodel defines the complete semantics for representing object models using a subset of UML notation and semantics to specify itself. In this way the UML metamodel bootstraps itself in a manner similar to how a compiler is used to compile itself.

The metamodel is described in a semi-formal manner using three views and both text and graphic presentations. These three views are: Abstract syntax, Well-formedness rules and, Semantics.

Since the metamodel layer is relatively complex it is decomposed into logical packages as shown in the UML class diagram of figure 2.7 taken from [31].
2.5. THE UNIFIED MODELLING LANGUAGE (UML)

2.5.1 UML Notation

UML defines twelve types of diagrams, divided into three categories: Four diagram types represent static application structure (describing the elements in the system and their relationship); five represent different aspects of dynamic behavior; and three represent ways you can organize and manage application modules.

These three domains are called:

- **Structural Diagrams**: that include two static structural diagrams called the *Class Diagram* and *Object Diagram* that show the entity in the system with their static relationship, and two implementation diagrams; *Component Diagram*, and *Deployment Diagram* for the implementation aspects of the model (i.e. source code structure and execution-time configurations).

- **Behavior Diagrams**: composed of the interaction diagrams called *Collaboration Diagram* and *Sequence Diagram*, for model collaboration among objects and rules, and *Statechart Diagram*, *Activity Diagram* very similar to finite state machine. Also the *Use Case Diagram*, used by some methodologies during requirements gathering, belong to this
domain.

- **Model Management Diagrams**: include Packages, Subsystems, and Models. They are used essentially to describe the logical organization of models. It corresponds to the model management view in fig. 2.7.

The most important module is the structural classification, because its static diagrams (Class and Object) are the foundation of every UML model. So we focus our attention largely on these diagrams. In the following sections there is a discussion of each of these relevant diagrams.

### 2.5.2 Static Structure Diagrams

**Class Diagram**

Class diagrams show the static structure of the model, in particular, the things that exist (such as classes and types), their internal structure, and their relationships to other things. A class diagram is a collection of (static) declarative model elements, such as classes, interfaces, and their relationships, connected as a graph to each other and to their contents.

We start discussing classes from their relationships (association and generalization), and their contents (attributes and operations). In a class diagram, classes are represented by boxes with three parts separated by horizontal lines: the name of the class, the attributes of the class (specified by their name, type and visibility) and the operations of the class (specified by name, argument list, return type and visibility). Three types of relationships may be used between classes:

- **association**, is the most general type of relationship and is represented by solid lines between two classes with optionally named ends, or roles.

- **generalisation**, represented by lines with large hollow arrow heads pointing to the super class. We can add constrains between brackets as the predefined *incomplete, complete, disjoint, overlapping*.

- **aggregation**, an association with a diamond at the aggregate end of the link. It represent a relation of whole-part between the instances of the classes.

- **composite aggregation**, a stronger type of aggregation notated by a solid black diamond which implies the same life-cycle by the instances of the whole and parts.
The ends of association and aggregation relationships may be annotated with multiplicity indicators giving a range of numbers. This denotes how many instances of the class at one end of the relationship can be associated with each instance of the class at the other end. Also, a small barbed arrow head may be used to specify that an association or aggregation relationship may only be navigated in one direction.

Rectangles with dog-eared corners are notes, a common use of notes is to attach OCL constraints to classes and associations.

Note that a "class" diagram may also contain interfaces, packages, relationships, and even instances, such as objects and links; in effect a better name would be "static structural diagram" but for convenience reasons OMG [30] maintain this name.

Class diagrams may be further organized into packages either with their underlying models or as separate packages that build upon the underlying model packages. Classes declared in the class diagrams are used in most other UML diagrams.

Object Diagram

A static object diagram is an instance of a particular class diagram; it shows a snapshot of the detailed state of a system at a point in time with a graph of instances, including objects and data values.

It depicts objects and links between objects-instances consistently with the relationships that hold between the linked objects’ classes. The class of each object included in the diagram, must be specified and the object may optionally be named. The values of the object’s attributes must be shown.

UML itself does not define a standard set of primitive types for attribute and operation declarations; however, the Object Constraint Language (OCL) does.

The use of object diagrams is fairly limited, mainly to show examples of data structures. We show an example of a class diagram in figure 2.8 with a related object diagram (figure 2.9).

2.5.3 Use Cases

Use case diagrams show actors and use cases together with their relationships. A use case is a kind of classifier representing a coherent unit of functionality provided by a system, a subsystem, or a class. An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates.
The standard icon for an actor is a "stick man" figure with the name of the actor below. A use case diagram is a graph of: actors, a set of use cases (represented as ellipses), possibly some interfaces, and the relationships between these elements. Use cases may optionally be enclosed by a rectangle that represents the boundary of the containing system.

There are several standard relationships among use cases or between actors and use cases. The only relationship between actors and use cases is the Association represented as a solid line between these two elements. With associations instances of actors and instances of use cases communicate with each other. In order to describe situations where an instance of a use case may be extended by some behavior specified by another use case, an Extend relationship between these two elements can be used. Similarly, an Include relationship indicates that an instance of a use case will also contain the behavior as specified by another use case. Also, Generalization from use case and actors are permitted.

2.5.4 The Object Constraint Language (OCL)

The Object Constraint Language (OCL) is a formal language used to express constraints. Users of the Unified Modelling Language and other languages can use OCL to specify constraints and other expressions attached to their models.
2.5. THE UNIFIED MODELLING LANGUAGE (UML)

Figure 2.9: A UML object diagram representing a piece of family relations in "the bold and the beautiful" soap opera
A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is commonly a need to describe additional constraints about the objects in a model. Such constraints are often described in natural language which will always result in ambiguities.

In order to write unambiguous constraints, there are a lot of formal languages available. The disadvantage of this traditional formal languages is that they are hard to use for persons without a strong mathematical background. OCL has been developed to fill this gap. It is a formal language that remains easy to read and write.

The Object Constraint Language is a pure expression language so the OCL expressions are evaluated and they do not have side effects; i.e., the evaluation of expressions cannot alter the state of the corresponding executing system. When an OCL expression is evaluated, it simply returns a value and cannot change anything in the model.

Introducing constraints in the UML model can be useful for a number of different purposes:

- To specify invariants on classes
- To describe pre- and post conditions on Operations and Methods
- To specify types in the class model
- To specify constraints on operations

Figure 2.10: A use case representing a marriage
2.5. THE UNIFIED MODELLING LANGUAGE (UML)

- To describe more accurately the behavior of elements in the model

**OCL notation**

All the expressions of OCL for a UML model must be within a so-called context declaration at the beginning of an OCL expression. The context declaration of the constraints is denoted by the keyword **context**.

An expression can be defined as:

- an invariant (**inv**): a property on an element which must remain true for all instances of the element at any time.

- a precondition (**pre**): a condition which must be true before the execution of a specific piece of the system.

- a postcondition (**post**): a condition which must be true after the execution of a piece of system.

Contexts for invariants are classes (or types), while contexts for pre- and postconditions are operations.

Moreover OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type conformance rules of the language. OCL define four basic type: Boolean, Integer, Real and String. Other types may be derived from the model since all classifiers from the UML model are types in the OCL expressions. OCL defines a number of operations on the predefined types with implicit casting mechanisms. All these types are considered subtypes of the predefined type OclType.

OCL also provides a mechanism to define and use type Collections. The Collection type defines a large number of predefined operations to enable the OCL expression author (the modeler) to manipulate collections. There are three different collection types: **Set**, **Sequence**, and **Bag**. A Set is the mathematical set. It does not contain duplicate elements. A Bag is like a set, which may contain duplicates. A Sequence is like a Bag in which the elements are ordered. Both Bags and Sets have no order defined on them.

All constraints (expressed in OCL, natural language or other formalisms) are represented in UML as a text string hold in bracket. Often constraints are inserted in note elements. As shown in the class diagram of 2.8.
2.6 UML for Ontology Modelling

Recently ontologies are becoming increasingly important in many research areas. They provide the main semantic foundation for many of the most critical technologies. Despite this growing interest, nowadays techniques for ontology development are essentially based on the traditions of AI knowledge representation research. The most common formalism used to represent ontologies are the Knowledge Interchange Format (KIF), and KL-ONE. KIF has a Lisp-like syntax for expressing sentence of the first order predicate logic (it provide extension for represent metaknowledge) and though is highly expressive, is very laborious to use. Thus Standford University Knowledge Sharing Laboratory have to done an editing tool, Ontolingua, to use KIF ontologies at a higher level. KL-ONE is based on "structural inheritance networks": networks containing descriptions of named concepts with generalization/specialization links between them, and description logics as mechanisms to introduce and specify concepts.

Stephen Cranefield in [21] proposes the use of a subset of Unified Modelling Language (UML) together with its associated Object Constraint Language (OCL) as an alternative formalism to express ontologies. He supports this choice with three observations:

- UML has a very large user community and is supported by widely-adopted CASE tools. Users will be more likely to be familiar with this notation than KIF or description logics.

- Unlike description logic formalisms, there is a standard graphical representation for models expressed in UML that allow users of distributed information systems to browse an ontology and discover concepts that can appear in their queries.

- The Object Constraint Language (OCL) is powerful and allows the expression of constraints that cannot be described using description logic.

This ontology representation language, model an ontology as a static model. It consists of a UML class diagram (containing OCL constraints) where classes do not have operations together with an object diagram.

In chapter 4 this method is used to represent the used ontology. As this approach may be, in my own opinion, an important progress in the representation languages and sharing mechanism for ontology, there is still a lot lacking due to the research status of the related items. There is a great need for a formal semantics for UML (OMG gives only an informal
definition of the semantics expressed in English). Further research is also needed to clarify what types of inference would be desirable and possible to support for ontologies designed in UML. Cranefield said that UML cannot be always considered an alternative in all situations; but presently for a system where the required type of reasoning about ontologies can be restricted to answer specific questions (that must be however identified) UML is a stronger candidate. The expectation is that the sort of reasoning required for a domain as a distributed information system could be performed using only class and object diagrams and using OCL constraint as an extra specification of the system behavior. Moreover, the meta-metamodel of UML allows to define several ontology representation language (never expressed in UML) to model various domains.

2.7 UML Data Binding (UDB)

We see in section 2.6 the benefits of representing ontologies through UML. However, although there is a standard XML-based format for exchanging models, there is not a similar technology for exchanging instances of a UML model. This fact translated in our domain means that we shall exchange ontologies but not domain knowledge. Stephen Cranefield [19, 20, 21] proposes a solution to this issue using RDF, XSLT and Java classes with a technology called UML Data Binding (UDB) exposed in this section. We retain it proper to give a prior summary presentation of two key technologies of UDB: XML-based Metadata Interchange (XMI) and the Java programming language.

2.7.1 XMI

The XML-based Metadata Interchange (XMI) is a language developed by the Object Management Group (OMG) [30] to support the interchange of any kind of metadata, including both model and metamodel information. Specifically, XMI is intended to help programmers, using the Unified Modelling Language (UML) with different languages and development tools, to exchange their data models with each other using XML. XMI is an interchange format for metadata that is defined in terms of the Meta Object Facility (MOF) standard (i.e. using the MOF Model meta-metamodel used in the typical OMG metamodel architecture as is in the UML metamodel layers shown in 2.5).

The two major components of XMI specification are:
1. The *XML DTD Production Rules* for producing XML Document Type Definitions (DTDs) for XMI encoded metadata. XMI DTDs serve as syntax specifications for XMI documents, and allow generic XML tools to be used to compose and validate XMI documents.

2. The *XML Document Production Rules* for encoding metadata into an XML compatible format. The production rules can be applied in reverse to decode XMI documents and reconstruct the metadata.

The DTD Production rules provide a way for creating a DTD from an MOF-based metamodel. The DTD defined by these rules may be used to validate the XML text, created following the XML Document Production Rules. As we can define the UML metamodel in terms of MOF Model we can generate using XMI’s XML DTD Production Rules a DTD for the UML metamodel, and use UML metadata in XML format using the XML Document Production Rules. Thus the main benefit of using XMI is to identify a standard XML DTDs that allows the exchange of UML (and MOF) information. XMI will also enable the automatic generation of XML DTDs for each meta information model. XMI’s extensibility allows DTD generation to cover additional domains such as data warehousing, component-based development, web metadata and obviously ontology too.

### 2.7.2 Java

The term Java means many things: an island, a type of coffee, an object oriented programming language, the specification of a virtual machine, a set of standard libraries, a tool to enhance Web pages in enabled browser... we could continue. Though all these aspects are very interesting, for our purpose we detail Java only as an object oriented programming language.

Java is being developed by Sun Microsystems since 1991, it was initially designed as a sort of OS for toaster. The Java team started with C++ but they soon realized that such a language did not meet their requirements for a small, compact and easily portable runtime system. So they developed a new language environment. The five main characteristics of Java language may be reassumed in:

1. **Java is Object Oriented**: Codes and data are inside objects, Java syntax is not too different from C++ but it is easier. There is no support for multiple inheritance.

2. **Java is compiled**: A source file in "*.java" extension is compiled to a bytecode file ("*.class") targeted to a Virtual Machine.
3. *Java is interpreted*: The bytecode is hardware independent and can be converted to the host native machine language.

4. *Java is concurrent and reactive*: It is possible to catch exceptions. Objects can be internally concurrent (threads).

5. *Java is pointerless*: No pointer arithmetic, no direct access to computer memory, garbage collection.

Initially the interest about Java was due to its support for the World Wide Web with the "applets" (chunk of Java code embedded in HTML documents). Very soon the combination of Java and WWW exposed the power of a new computing model based on mobile code and data. However concurrency and control problems together with some security issues braked the run.

Elaborating the syntax of Java goes beyond the aim of this exposure. Anyhow Java is a basic technology for many ontology and agent application tools as evidenced by the UDB implementation.

### 2.7.3 Processing Knowledge with UML

XMI is a suitable format for the serialisation of ontologies expressed as UML class diagrams. However, XMI does not provide a good solution for serialising object diagrams represents knowledge. Encoding object diagrams with XMI we obtain a serialisation with one element for each different component of the diagram. It would be much more convenient to have an encoding that is specialised to the ontology (i.e. class diagram) used, so that the object diagram could be serialised in a form containing elements corresponding to individual objects.

Figure 2.11, taken from [20], describes the implemented technology of Cranefield’s UDB. First, using a CASE tool for UML (e.g. Argo/UML or Rational Rose) the ontology is graphically designed and saved using the standard XML-based format XMI. Then a pair of XSLT stylesheet receiving as input the XMI representation of the ontology produce:

- A set of Java classes and interfaces corresponding to the ontology
- A representation of the ontology using RDF with the modelling concepts of RDF Schema.

With the generated Java classes an application could represent knowledge about objects in the domain as in memory data structures. The generated "schema" in RDF define domain-specific concepts that an application can reference when serialising this knowledge using RDF/XML.
CHAPTER 2. TECHNOLOGIES

2.7.4 UML to RDF schema

We see in 2.2 that RDF is a simple resource-property-value model for expressing metadata about resources on the Web, and RDF Schema a set of predefined resources and relationship between them that permit to define meta-metamodels. "Domain shemas" (i.e. ontologies) can then be expressed as sets of RDF triples using meta-classes and properties of RDF Schema. The XMI to RDFS stylesheet produce this RDF schema (using the XML encoding) from the XMI document representing the ontology.

In this mapping there are some issues the stylesheet may prevent. Firstly generating an RDF schema that corresponds to the object oriented model, the RDF properties are first-class objects not defined within the context of a particular class. Which means there may be conflict range declarations if the same property is used to represent a field in two different classes. The solution adopted was simply to prefix each property name with the name of the class, that has some disadvantage in presence of inheritance (field-property are represented with different prefixes) that however concern only the human-readability of the document that there is not the purpose of this step. Another issue is represented by the lack in RDF Schema of mechanism to parametrise a collection type by the class of elements it may contain. So Cranefield introduces a non standard property `rdfsx:collectionElementType`
2.7. UML DATA BINDING (UDB)

to allow this description. The rdfsx namespace name is

http://nzdis.ontago.ac.nz/0_1/rdf-schema-x#.

2.7.5 Generated Java classes

The mapping between UML class diagrams and Java is mostly straightforward: classes and interfaces map to their equivalents in Java; attributes and associations map to Java fields. It uses a Java list or set, depending on the multiplicity and the presence or absence of an "ordered" constraint on the association. Also OCL primitive type: Boolean, Integer, Real and String are mapped to the corresponding Java class types (instead of the Java primitive types). Possible operations, not yet supported, are declared in the corresponding Java classes and interfaces as empty method bodies. The visibility of attributes and association ends specified in the UML model is respected (though they are all public). Furthermore this stylesheet provides a field-names checking against Java reserved words, and correct eventually these terms, affixing an underscore character to avoid Java compilation errors.

The set of Java classes generated facilitate the process of knowledge communication in a object-oriented representation form. This allows Java applications to instantiate instances of the domain concepts. In order to provide mechanism for marshal and unmarshal object-oriented information as RDF/XML documents, the generated classes include marshalling and unmarshalling methods (used along with the additional utility classes MarshalHelper and UnmarshalHelper of the marshalling framework). The aim of the marshalling code is to allow a Java application to maintain an internal representation of object-oriented knowledge (as in-memory structures) and to easily read and write parts of this knowledge to and from a format suitable for transmission or publication on the Web. As object-oriented data structures consist of a network of interlinked objects, knowledge represented propositionally are self-contained statements of knowledge. When serialising knowledge, an application may wish to include only some of the information it knows about a domain, i.e. it needs a way to determine which related object serialize along with particular object of interest. To allow this selectivity, a method of MarshalHelper (marshalObject ) takes a collection of objects as an argument that are all serialised togheter with any link between them. Links to any objects outside this collection will not be serialised. The output of this stylesheet must be postprocessed to produce separate Java source file for each class and interface.
2.7.6 Limitation and future extensions

There are some limitations in the current versions of UDBs’ stylesheets. N-ary associations, qualified associations and feature related to the class scope are not supported. There is also no mapping of UML namespace to Java package. Other approaches to ontology modelling also allow varied forms of inference to be performed on ontologies and on knowledge so expressed. Instead UML is defined in terms of meta model with meaning of elements described in plain English so it does not currently have this facility. As we discussed in 2.6 a sort of reasoning could be performed using class and object diagrams alone, but the stronger candidate for this approach is the Object Constraint Language. OCL is a powerful mechanism for expressing inference rules (it is essentially a variant of first-order logic with an OO syntax) but this expressiveness also mean that reasoning about OCL expressions will be undecidable. However, Cranefield [20] addresses the research in this way supposing a ”macro language” on top of OCL comprising predicates which ensure reasoning over these expressions is tractable.

2.8 Agent Communication Languages with Ontologies in UML

Related to the UML Data Binding technology and the use of UML as an ontology modelling language, the really important matter for this dissertation regards the use of UML as support to the agent communication.

Ontologies play an important role in defining the terminology that agents use in the exchange of knowledge-level messages and therefore the choice of an ontology representation language is a significant issue when designing multi-agent system. We have seen in 2.6 the traditional approach for ontology modelling: KIF based on first-order predicate logic and KL-ONE that use description logics as formalization of frame-based knowledge; and the benefits of use UML to model ontologies. However, in distributed agent systems, beside the ontology, agents need a concrete knowledge exchange language to be able to communicate with each other. Traditionally software agent communication languages (ACLs) as KQML and the FIPA ACL [28] have been based on the exchange of information represented as ”sentences” in a logic based content language such as respectively the Knowledge Interchange Language (KIF) and FIPA’s Semantic Language (SL). The agent communication language has an outer layer that specifies information needed for routing the message, understanding the context and parsing the content of the message, as well as the type of communicative act (e.g. ’inform’ or ’request’) repre-
2.8. AGENT COMMUNICATION LANGUAGES WITH ONTOLOGIES IN UML

Presented by the message and the language in which the content is expressed. KIF, SL and other logic-based content languages use a declarative representation of knowledge, i.e. based on a static collection of assertion and facts, without details about the use to which the knowledge might be put.

While declarative approaches for exchange information are canonical practices in agent-based systems, are not the only possibility. Object-oriented data structures can be viewed as encoding of UML object diagrams, which can be considered to be to declarative representations of knowledge. We can incorporate in this manner object-oriented knowledge into agent-based information systems. If two agents represent domain knowledge as object diagrams, it would be most convenient for them to communicate information to each other in this form as well. Details of how agent messages may be extended with object-oriented content, that is, how it is possible to include object-diagrams within message content, are given in [22].

If an agent developer is using an object-oriented model of the problem domain and is also using an agent development toolkit based on an object-oriented language, it may be most convenient to encode domain knowledge using objects and include these objects within messages to play the role of propositions.

Given a set of ontologies described using UML class diagrams, knowledge about the domains described in these ontologies can be expressed as instances of these classes in the ontologies. This knowledge can therefore be formalized as a UML object diagram (this diagram may be most convenient for agents rather than using knowledge base containing separate facts).

There is in fact in the agent-based community a general trend away from defining languages in terms of linear string based syntax, in favor of “abstract syntax” representations, which can be mapped onto various alternative concrete syntaxes. Using an UML-based abstract syntax for an ACL and his associated content language [24], the distinction between an ontology and an abstract syntax (the set of definition of concept that can be expressed in a language) becomes blurred when either three different types of run-time object are modelling using the same formalism. Cranefield in [26] summarizes this approach as follows. "An ontology in UML constitutes an abstract syntax for the domain of discourse."

The result is a basis-framework for the development of agent system that combine object-oriented information representation with agent messaging protocols.

There are three types of objects that may exists in an agent system at run-time: domain objects, knowledge objects and message objects; and three respective models: ontologies, content languages and agent communication languages. Figure 2.12 shows how this object and models are related and
CHAPTER 2. TECHNOLOGIES

Figure 2.12: A meta-modelling perspective on agent messaging

how can be expressed in the same modelling framework.

This is a meta-modelling view of an agent system. The bottom layer (Level 0) shows the concrete entities that exist in and around an executing agent. The Level 1 above depicts the models that define the properties of the Level 0 related instances. Thus, a domain object is an instance of a concept in an ontology, a knowledge object is an instance of (i.e. expression in) a content language and similarly a message object is an instance of an ACL model. On top, Level 2 shows the meta-model used to describe the Level 1 models: the UML meta-model (see section 2.5) with some suitable adjustments.

With this viewpoint, it can be seen that the notion of ontology is strongly related to the notions of content and agent communication language abstract syntaxes (they appear at the same level of the meta-model), and this allows us to refer to all these models as "ontologies".

To allow this meta-model agent system to operate, it is necessary to have an abstract syntax representation for Agent Communication Languages and their associated content languages.

With UDB, exposed above, we have a way of mapping from a high-level abstract specification of language structures to specific computer language bindings (that can be plugged into an agent platform). UML class diagrams describing an ACL and a content language can be automatically mapped to a set of Java classes representing the concepts in the languages, and a RDF schema corresponding to the language. With marshal and unmarshal methods for Java classes for representing messages to and from RDF as seen in 2.7.5.

In fig. 2.13 and fig.2.14 taken from [24] Cranefield depicts with UML class diagrams a subset of an ACL and a content language (based on FIPA
2.8. AGENT COMMUNICATION LANGUAGES WITH ONTOLOGIES IN UML

Figure 2.13: A UML class diagram for an Agent Communication Language

Figure 2.14: A UML class diagram for a fragment of a Content Language
ACL and FIPA SL specifications). The ACL model defines interface types for every concept required by the communicative acts supported by the ACL.

The communicative acts are a list of standard possible acts that specifies the ACL; in the case of FIPA ACL these are the concept of a proposition, an action description and a definite description. All these are modelled in figure 2.13 as interface i.e. with no operations. It is just declared that the concept exist. It can then be declared that a content language include representation for one or more of these concepts also using UML, and relationship between classes in the content language model and interfaces in the ACL model. As is made in fig. 2.14 with lollipop symbols. With this formalization of the relationship between an ACL and a content language, is explained how a well-formed message can be constructed by nesting a content language expression within an ACL wrapper.

We can for example express the following FIPA ACL message as an object-diagram:

"Gianni is the father of Luca and is an employee by profession"

expressed in FIPA ACL as:

\[
\begin{align*}
\text{(inform :sender agentsend} \\
\text{:receiver agentrec} \\
\text{:ontology (sequence (Family Professions People))} \\
\text{:language FIPA-SL} \\
\text{:content ((= (iota ?x (and (parent Luca ?x) (profession ?x employee))) Gianni))}
\end{align*}
\]

Without dwelling in the FIPA ACL syntax that seem intuitive enough, 2.15 presents the UML object diagram corresponding to this message.

However in this example we simply show how knowledge is modelled with object diagrams using the abstract syntax representation for the ACL and content language. Here we use ontologies in the inform message without saying how exactly they are linked to the objects. Being able to use all the models and objects of the metamodel described above it is our aim. Shown in figure 2.16 is an example of this kind of message using the ontology shown in fig. 2.17.

"The man Gianni is father of Luca"

The FIPA ACL expression is roughly:
(inform
  :sender AgentA.bologna
  :receiver AgentB.bologna
  :ontology FamilyOntology
  :language FIPA-SL2
  :content (= (iota ?y (exists ?x (and (man ?x) and (name ?x "Luca") (father ?x ?y)))) Gianni)
)

The diagram in 2.17 shows a message object $m$, of a type (InformRef) that is specialized for informing an agent (the receiver) that the object corresponding to a given description (the object of type OODefDescriptor) has a given reference (the object linked to $m$ with ref role, i.e. Gianni).

Note that concepts in the Family ontology only appear indirectly in this message using "reference by name" attached to an object descriptor at the "classifier" end of a link ("classifier" is the UML name for the type of an object). It is important to avoid confusion between objects being represented, and their descriptors. The first are the objects subject of the inter-agent communication (may be real word objects, internet resources and so on) the second are references to this objects or object diagrams which must be used for example from other agents. To allow this, a UML extension of data type was necessary. UML assumes that objects have an identity that is implicitly
Figure 2.6: An agent message expressed as a UML object diagram with references to an Ontology
2.8. AGENT COMMUNICATION LANGUAGES WITH ONTOLOGIES IN UML

![Figure 2.17: An ontology describing family relationships](image)

provided by the underlying implementation infrastructure and which do not need to be included as an attribute of the corresponding class in the model. It makes no commitment about the nature of this identity or the way in which links between objects are implemented. Hence also for proper object (referred as "objects with identity") an extension is necessary.

![Figure 2.18: A metamodelling view of agent systems and models](image)

The meta-model detailed above can now be shown in its recent extension [23] in 2.18. Here the relationship between ontologies and content language expressions are refined. The result is the production of ontology-specific con-
tent languages. The inclusion of domain-specific objects within messages can be considered as representing an object-oriented encoding of values, propositions and definite descriptions. The model in 2.19 is the CL package that depicts abstract generic content language concepts. It is built on the requirement of FIPA ACL semantics and its communicative acts library. Given an ontology, a specialized content language can be generated either as simple application-specific representation or as an extension of an existing general-purpose content language.

Figure 2.19: The Content Language (CL) package: generic content language concepts

In 2.20 a simple explicative example of an ontology-specific content language is presented. The generalization relationship between the SpecialisedSL package and the general SL one indicates that all concepts in SL are included in SpecialisedSL. A valuetype declared in the ontology maps to an equivalent declaration corresponding to the generated language, while a class with the <<resource>> stereotype maps to a structurally identical class that implements the CL::Proposition interface. That means that objects of City sort, can be used as descriptions of corresponding domain entities (i.e. describe their attribute values and links). Also a descriptor class is generated to allow the use of the class by an object-oriented structure which can so use it as the query part of a definite descriptor. Therefore this class implements the interface CL::NonGroundTerm because a DefDescription (a propositional expression where the bound variable represent the object) comprises two non-ground terms, linked by common free variable: a query expression and a
template for the result. For classes without stereotypes, there are only a corresponding proposition class, without the definite descriptor class (because we do not know if the object has or not identity). In this case to return a reference of the object we simply refer to an instance of the object.

A fundamental principle of agent communication languages is that a message should be able to be understood without knowledge of the agent that sent it. This is where ontologies are crucial: the content of a message must be expressed according to an ontology that define the semantics of that content. For the purpose of representing various aspects of the world in information system, the object-oriented model have many benefits. The possibility to use an easy modelling language such as UML make this think more straightforward and powerful. Finally the presence of UDB, XMI, Java and other Semantic Web technologies close the circle presenting an important structure for knowledge interchange and representation for software agent systems.

2.9 Summary

In this chapter technologies for ontologies and the Semantic Web have been investigated. Particular attention is posed on RDF and UML because their understanding is fundamental for UDB’s processes. For our purposes all the matters related to the ontologies and the Semantic Web are useful only if applied to agents. Moreover ontologies and semantic web shall not exist if there are no agents which use them. The vision of Cranefield for ontologies is to join semantic web and agents, and UDB in this sense is the architecture that actually links these two worlds. However for real progress of the technolo-
gies here exposed we cannot leave out of consideration the methodological aspect. Methodologies are fundamentals for refining technics, and there are not, nowadays, methods to design agents with ontologies. The purpose of the next chapter is to outline how such methodology may be developed.
Chapter 3

METHODOLOGIES

3.1 Development of Semantic Web Applications

In only one decade the World Wide Web (WWW) has drastically changed the availability of electronically accessible information. Currently the WWW contains more than 3 billion static documents, which are accessed by over 300 million users [8]. These reasons induced many commercial and non-commercial organizations to join this virtual space, and more and more people to take part in the development of so called web applications. Instinctively a great number of software engineering researchers begin to develop Web design methods to capture web application essences, through software engineering models. A loose definition of what a web application is, may be the one given by Conallen [10]: "A Web application is a Web system (Web server, HTTP, browser) in which user input (navigation and data input) effects the state of the business". Further a more pragmatic characterization can be found in the remark that the WWW is based on the hypertext paradigm, inasmuch as it is composed of pages which can be linked to each other through links (using URLs) [10]. In this manner a web application is thought of as a particular kind of hypermedia application. Thus many methods for developing web applications were developed that investigate different approaches. The most useful technics are those based on the Object Oriented model, such as the Object-Oriented Hypermedia Design Method (OOHDM) [42] next exposed. Together these methodologies, some specific extensions of UML have been proposed, as the Conallen one’s [10]. Note that also OOHDM is based on the UML model. With the awareness of the problems related to the management of the WWW data, and the introduction of the technologies (and opportunities) of the Semantic Web, a new development outlook in software
engineering for web applications has been presented. In the next sections I briefly show the Object Oriented Paradigm (OOP), detail on the OOHMD methodology, and a recent development process based on ontology design: the Ontology-Driven design for Semantic Web application (ODD) [9].

3.1.1 The Object Oriented Paradigm (OOP)

The Object Oriented Paradigm is applied in a wide range of Computer Science matters. There are systems based on objects, object oriented languages, object oriented data bases, operative systems, methodologies and hardware all inspired on this paradigm. All the object oriented languages (such as Java) allow the abstraction of Class which include the definition of attributes (fields) and operations (methods) in the same wrapper. Classes define object types allowing the creation of instances, i.e. concrete objects.

The object model is based on a few simple but powerful concepts:

- **Abstraction**: is used for control the complexity of a system using simple concepts. It is realized filtering unessential details of the element being described denoting only its essential characteristic.

- **Encapsulation**: also referred as "information hiding" is the process of hide all the internal detail to facilitate the abstraction. The abstraction work only if the internal complexity is invisible.

- **Inheritance**: is just a hierarchy of abstractions through which existing representation are used to define new (more specialized) representations. May be also denoted as an "is a" hierarchy.

- **Polymorphism**: this implies that a specification may have many implementations. Whit polymorphism a new representations is defined as variations of an existing one.

these are principles that we could find in any object oriented system to exploit all the advantage of this model.

The object model affect all the phases of the software life cycle. So exist an Object-Oriented Analysis (OOA) and an Object-Oriented Design (OOD) methods. With the OOA we describe what the system does; studying a domain, and defining objects and classes for him. An OOA method contain at least:

- models for entity and relationship: where classes and attribute are stated together with class relations. Example diagrams for this information are class diagram or object diagrams.
3.1. DEVELOPMENT OF SEMANTIC WEB APPLICATIONS

- models for the structure: where data and functionality are considered in an internal view, i.e. the static structure is modelled and the output are rated.

- models for the behavior: which represent the dynamic aspect of the system. Besides also the interaction and collaboration feature among the element described in the other models are modelled.

With the analysis we obtain a description of the problem domain. The OOD goal is to describe how the system works to address the identified requirements during the analysis phase. The design phase can be decomposed in two parts: an architectural design phase and a detailed design phase. The first gives a global vision of the system and optimizes the reusability and extensibility, the second details the description of all the classes, giving more important information useful in the implementation phase.

3.1.2 The OO Hypermedia Design Method (OOHDM)

The Object-Oriented Hypermedia Design Method is a model based approach for building hypermedia applications [42]. It has been extensively used to design different kinds of applications such as: Web sites, information systems, multimedia presentations, etc. It comprises four different activities namely conceptual design, navigational design, abstract interface design and implementation. Figure 3.1 taken from [43] summarize the entire OOHDM design process.

Conceptual Design

During this step a model of the application domain is built, using well known object-oriented modeling principles and primitives. The main concern during this step is to capture the domain semantics as "neutrally" as possible, with little or no concern for the types of users and tasks. The conceptual model may not reflect the fact that the application will be implemented in the WWW environment, since the key application model will be built during navigational design. The product of this step is a class and instance schema built out of Sub-Systems, Classes and Relationship. The instance schema is used to describe exceptional objects and it is intended to avoid class explosion.
CHAPTER 3. METHODOLOGIES

Figure 3.1: Summary of the OOHDM Method

Navigational Design

In OOHDM, an application is seen as a navigational view over the conceptual model. The objects (items) the user navigates are not the conceptual objects, but other kind of object that are ”built” (through a view mechanism) from one or more conceptual objects. Different navigational models may be built from the same conceptual schema, so for each user profile we can define a different navigational structure of object and relationship, according to the task this kind of user must perform. The navigable objects of a hypermedia application are defined by a navigational class schema. Nodes are defined as views of the classes defined in the conceptual model, and links are as views of the relationship.

Abstract Interface Design

Once the navigational structure has been defined, we must specify which interface objects the user will perceive and how the interface will behave. We define an abstract interface model where is described the way in which different navigational objects will look like, which interface objects will activate navigation and which interface transformations will take place.
3.1. DEVELOPMENT OF SEMANTIC WEB APPLICATIONS

Implementation

To obtain a running implementation, the designer has to map conceptual, navigational and abstract interface objects onto the particular runtime environment being targeted. When the target implementation environment is not fully object-oriented, we have to map the model objects into concrete objects available in the chosen implementation environment. The advantage of having an object-oriented abstract interface specification allow simplifying the interface in implementations in different platforms.

There are many advantages to using structural object-oriented models; they provide high level abstraction and composition mechanism (e.g. classification, aggregation, etc.) with a well defined semantics. Furthermore, the relations between the models seem very straightforward and this allows for representing and controlling different views of the web application modeled.

For these reasons the OOHDM has been chosen as a basis for the Ontology-Driven design method for semantic web application exposed in the next section.

3.1.3 The Ontology-Driven Design for Semantic Web Applications (ODD)

In [9] Valentina Presutti defines a process for developing Semantic Web Applications using the ontologies as a basis for an incremental and prototyped-based model inspired on OOHDM. A Semantic Web Application is defined as a Web application where all pages are annotated with semantics information. The Ontology Driven Design (ODD) is based on the use of UML as modelling language and UML Data Binding (UDB) for diagrams serialization as described in section 2.7. As in OOHDM a set of object-oriented models describing particular design concerns are developed. The method comprises the following activities:

- Ontology Design
- Conceptual Design
- Navigational Design
- Object Model
- Interface Definition
- Implementation
Relations among these steps are depicted in 3.2. As Navigational Design and Interface Definition with Implementation are very similar to the one in OOHDM, we focus our attention mainly on the remaining activity.

In the **Ontology Design** an ontology model is defined. A domain specific ontology schema is created in UML in terms of Sub-Systems, Classes and Relationships. With this model all typical concepts of the domain being considered are described, together with its relationship and organization structure. The ontology model is very similar to the **Conceptual Model** that may be thought of as the same level of abstraction. These two steps are only conceptually separated but practically they are reduced to one single step. The same mapping mechanism described in OOHDM is implemented in the **Navigational Design**, so classes are mapped to nodes and relationships to links among nodes. In the **Object Model** a collection of UML object diagrams is contained. These diagrams describe an abstract representation of the knowledge information contained in a specific Semantic Web Application that we are developing. It is not properly a step of the process because it evolves during the application life-cycle. Finally the **Interface Definition** and **Implementation** steps has exactly the same meaning of those defined in OOHDM.

I retain that Presutti’s approach is very useful for this work because it is taken as a basis for the agent described in 4.3 and for the proposed methodology at the end of this chapter.

### 3.2 Agent Oriented Software Engineering

Since the 1980s, software agents and multi-agent systems have received a great deal of attention and, as a result, the industry is beginning to get interested in using this technology to develop its products. This area is one of the most active of research and development in computing generally. As a great work was done in the past two decades in development of agent theories, languages, architectures and concrete applications; research in Agent-Oriented Software Engineering is only a recent contribute to the agent technology. The development of methodologies for agent engineering have the purpose to enable inexpensive development and maintenance of agent-based software. In addition, the software should be flexible, easy-to-use, scalable and of high quality [1]. Thus, agent-oriented (AO) methodologies have to assist an agent-based application, in all the phases of the life cycle, including its management. In other words quite similar to the research issues of other branches of software engineering. The next sections present a survey of agent-oriented methodologies starting with the idea of agent as a software
Figure 3.2: The ontology-driven method for Semantic Web Applications
engineering paradigm to a hypothesis: the application of the ontology driven approach to agent design.

### 3.2.1 Agents as a paradigm

To introduce agents as software engineering paradigms, we have first to agree to a better formal definition of agents than the explicative one given in 1.3.1. We retain that Wooldridge’s [50, 52] definition is the most useful for our purposes. He identifies a classification scheme based on two general usages of the term ”agent”: a weak, and a stronger notion of agency [50]. The weak notion is the most general, and is used to give the essential properties of an agent (i.e. to define it). Therefore an agent is a system that enjoys the following properties:

- **autonomy**: agents operate without the direct intervention of humans or others, they encapsulate some state and make decisions about what to do based on this state;

- **reactivity**: agents perceive the environment in which they are situated (internet, the physical world and so on), and are able to respond in a timely fashion to changes that occur in it;

- **pro-activness**: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by ”taking the initiative”.

- **social ability**: agents interact with other agents (and possibly humans) through ACLs, and typically have the ability to engage in social activities (such as cooperative problem solving or negotiation) in order to achieve their goals.

The basic idea that inspires the Wooldridge definition is that a UNIX-like software process, where the properties listed above can be enucleated. This weak notion of agency is also that used in the agent-based software engineering.

The stronger notion is the most malleable. Here the weak notion is preserved, and in addition other specific characteristics can be introduced for a specific research fields. We can move around mobility, veracity, benevolence, rationality and so on, similarly as we do in the taxonomy given in section 1.3.2.

An important question is why agents and multi-agent systems are seen as a new direction in software engineering [52]. There are several reasons
but the most important is that agents can be considered a powerful software-engineering paradigm (as is for objects). This is due to the natural metaphor we may find in the agent idea. So, just as many domains can be conceived of consisting of a number of interacting but essentially passive objects, so many others can be conceived as interacting, active agents with social ability. There are hence many domains where agents abstraction can bring great benefits in the modelling phases.

This however, does not mean that agents will substitute objects in the future. That is one of the most common pitfalls of agent oriented software engineering [49]. Agent oriented programming can be seen as an extension of object-oriented programming, that can be seen as a successor of structured programming again. If we try to model every entity in a system as an agent (that is conceptually more clean than other approaches) the result is only a terribly not efficient system. It is the same thing to model in OO programming all data types (integer, boolean, etc.) as objects, it is ”pure” OO, and ensures consistent treatment of data, but it is too inefficient to be implemented. So we can think of agents as the natural progress of programming: from sub-routines to procedure & functions, to abstract data type, to object and finally (for the moment) to agents [48]. Therefore agents represent the next advance in abstraction, and may be used by software developers to better understand, model and develop important system domains (firstly complex distributed systems).

An important subject concerns the difference between agents and objects. While there are obvious similarities, also significant differences can be identified.

- As the main characteristic of agents is their autonomy, the first difference is in the degree to which agents and objects are autonomous. An object has autonomy (control) over its state (principle of encapsulation), but it does not have control over its behavior. If an object has a public method, other objects can invoke it whenever they wish, i.e. the object has no control over whether or not that method is executed. Instead, agents do not invoke methods, rather they request action to be performed, i.e. the other agents may say no. The locus of control of the decision to execute or not an action is different in agent and object systems. In object the decision lies with the object that invokes the method, in agent the decision lies with the agent that receive the request.

---

1Wooldridge and Jenning have identifies the main pitfalls that await the agent system developer. This is the pitfall 7.1 You see agents everywhere
• The second distinction regards the notion of flexible behavior. The standard object model does not support reactive, pro-active or social behaviour.

• Both paradigms use message passing to communicate with each other, but while message-passing for objects is just method invocation, agents distinguish different types of messages (for example support high-level interaction using ACLs) and model these messages frequently as speech acts making use complex protocols to negotiate.

• Finally objects have successfully been used as abstractions for passive entities in the real-world. Objects are assumed to be quiescent for most of the time and become active only when there is a method invocation. While agents have their own thread of control, and carry it out updating their internal state, observing the environment, executing sporadic action, that is, remaining continually active.

Though many object-oriented languages (e.g. Java), attempt to improve their features approaching to the agent paradigm, they do not capture the full idea of agent as an autonomous entity.

3.2.2 AO methodologies overview

If agents are to realise their potential as software engineering paradigm, then it is necessary to develop software engineering techniques that are specifically tailored to them [51]. We have seen in the previous section the fundamental mismatch between the concepts used by object-oriented development and those in the agent-oriented view. These differences are obviously reflected in the engineering methodologies used to model these systems. Thus, extant approaches fail to adequately capture an agent’s flexible, autonomous problem-solving behaviour and the richness and complexity of the interaction and organizational structures of the agent systems.

Despite this, to avoid building a methodology from scratch, researchers on agent-oriented methodologies have followed the approach of extending existing methodologies to include relevant aspects of the agents. These extensions can be broadly divided into two groups:

• those inspired by object-oriented development

• those that adapt knowledge engineering or other techniques.

The predominant approach for agent system is the first. There are several reasons that justify extending existing object-oriented methodologies:
1. Firstly the similarities between the object-oriented and the agent-oriented paradigm.

2. Another possible advantage comes from the common usage of object-oriented languages to implement agent-systems (e.g. Java is widely used for this purpose).

3. Obvious advantages may derive from the popularity of object-oriented methodologies (and UML in particular), especially in the industry where this paradigm has been successfully tested.

4. The three common views of the object-oriented system (static, functional and dynamic) described in 3.1.1 are also interesting for agent analysis.

5. Some other interesting techniques of OOP can also be used (e.g. use cases for identify agents and roles).

Having stated these benefits there are also several disadvantages and pitfalls that must be taken into consideration using an agent-oriented methodology derived from the OOP. One of these is the kind of decomposition that object-oriented design encourages. Agents are more coarse-grained than objects, i.e. an agent system implemented using an object-oriented programming language will typically contain a great number of objects but will contain far fewer agents. Again, many aspects of agent systems (pro-activity, social ability, reactivity) are not captured by the OOP and thus by object-oriented methodologies.

Knowledge engineering methodologies also have some advantages. They are a good basis for multi agent systems modelling as they provide techniques for modeling agent knowledge. Anyway, these methodologies are not as extendable as the object-oriented ones and they induce to develop too centralized systems disappointing social aspects.

Also distinction for agent-oriented methodologies can be done for high-level methodologies, that describe all the process to model and develop an agent-based system, and specific design methodologies that focus only on some aspects, such as improving coordination, cooperation or communication.

Despite the large amount of methodologies for AOSE, in this thesis we will treat only a high-level, object-oriented-based AO methodology: Wooldridge's et al Gaia [51, 52].
3.2.3 The Gaia Methodology

The Gaia\textsuperscript{2} methodology has been specifically tailored to the analysis and design of agent based systems \[51\]. It is inspired by the FUSION object-oriented method of which some aspects have been maintained. Gaia is appropriate for the development of large scale real-world applications. It deals with both the macro (societal) level and the micro (agent) level aspects of design and is neutral with respect to both the target domain and the agent architecture.

In applying Gaia, the analyst moves from abstract to increasingly concrete concepts, it allows an analyst to go from the statement of requirements to a design that is sufficiently detailed that it can be implemented directly. It provides an agent-specific set of concepts through which a software engineer can understand and model a complex system. Gaia associates the concept of building an agent-based system to a process of organizational design. This definition summarize the advantage of this methodology over the others one, that commonly fail to represent the autonomy and problem solving nature of agents and do not give ways for efficient model interaction.

In figure 3.3 taken from \[51\] the gaia model is sown. The analysis stage is divided into roles model and interactions model. The objective of this step is to develop an understanding of the system and its structure (without reference to the implementation detail). The role model permits one to identify roles in the system. The idea of role is the key factor in developing a system as a society. Note that a role (as in a real organization) will be instantiated with actual elements. A role consists of four attributes: responsibilities, permissions, activities, and protocols. Responsibilities are of two types: liveness properties - something good will happen - , and safety properties - nothing bad happens - , and determine the role’s functionality. Permissions represents what the role is allowed to do, in particular which information resources it is allowed to access. In other words it represents the ”right” associated to a role. The activities are computations that a role performs without interacting with other roles. Finally protocols represent the specific patterns of interaction, i.e. the way that it can interact with other roles.

In the role model the key role are identified. Here we can view the role as an abstract description of an entity’s expected function i.e. a role as an office with resources and responsibilities associated with it. The interaction model serves to capture and represent the dependencies and relationships between the various roles. The model consists of a set of protocol definitions, one for each type of inter-role interaction. So a pattern of interaction has been created between the various roles to define any particular sequence

\textsuperscript{2}The name Gaia comes from the James Lovelock’s hypothesis that all the earth’s organisms act together to regulate the earth’s environment.
3.2. AGENT ORIENTED SOFTWARE ENGINEERING

Figure 3.3: Relationship between Gaia’s models

of execution steps. Iterating the definition of these first two models, an elaborated analysis is carried out for the design phases.

The Gaia design process involves the realization of three models: the agent model, the service model and the acquaintance model. With the agent model agent types are identified together with agent instances that will realize these agent types at run-time. An agent type can be identified by a set of closely related agent roles. Agent instances that will appear in a system are annotated on the agent type with instance quantifiers (n, n...m, *) in a similar manner as objects in UML.

The aim of the service model is to identify the services associated with each agent role together with their main properties (inputs, outputs, pre- and post-conditions). Note that the Gaia services model does not prescribe how the documented services must to be implemented.

Finally the acquaintance model simply define the communication links that exist between agent types. They do not define what or when messages are sent, they simply indicate that communication exists. This model is no more than a graph with nodes corresponding to agent types and arcs to connect this.
3.3 Designing Agents with Ontologies

The aim of this section is to issue an hypothesis of extension of the ontology-driven approach exposed above to the development of internet agents.

The Ontology-Driven Design for the development of Semantic Web Applications can be usefully used for hypermedia applications. It has the same power in this sense as the OOHDM on which it is developed. Despite this methodology contemplates the presence of "applications" that operate on the produced object model, the engineering process has nothing to say about the development of this application. As the most probable software that will interact with semantic web applications are agents [9], it may be interesting to include an agent development process with these methodologies.

We have seen extensively the interaction between ontologies and abstract syntax for ACL (see section 2.8). An ontology-driven AOSE, based on Presutti's intuition of ontologies as the basis for application design, capable of addressing powerful software (i.e. agent) engineering methodologies will allow the integration of the "description" (as an ontology) of the domain where agents work, and the description for both the abstract and the concrete syntax of the agent communication language used. Supporting these facts with the UML Data Binding Technology, we can manage the knowledge both as in memory data structure (with Java classes) and in RDF notation (with the generated RDF schemas).

All these matters are very useful in improving reactivity and pro-activeness properties giving a flexible way to adapt to the environmental changes and improving social ability exploiting the real knowledge in the world, given from the the Semantic Web rather than other specific ACL's content languages. In other words this is autonomy.

Furthermore there are many active researchers in the development of UML extensions for agents (such as AUML of Odell et all [4, 39]) and recently in the development of methods to introduce abstract description of agent interaction protocol (as ontologies) in the protocol itself [25]. That will open new outlooks for a further extension of this work permitting the introduction of knowledge for dynamic aspects in such new methodology. With hopes a standard formal semantics for UML and the development of the remaining Semantic Web layers with related improvement of the UDB technology in the near future make the hypothesis of an ontology-driven agent development process more credible.

Finally I want to close by asking a question: And if ontology were the next programming paradigm?
3.4 Summary

This chapter gives an overview of the methodologies used in the development of semantic web applications and agents. Related to web applications, based on the Object-Oriented Paradigm we investigate the Object-Oriented Hypermedia Design Method on which the Ontology Driven Design is based. Also a general introduction to the agent oriented methodologies is given and in particular we explore Wooldridge’s Gaia. The aim of this presentation is to give a basis for a hypothetical Ontology Driven Agent Method based on the ODD and an agent oriented methodology. Thinking of the great number of agents which in future will be on the internet, and their few typologies (one or two hundred agent types) Gaia seems to be a stronger candidate for this merge. Since a formal definition of the methodology is not given in this chapter a rough application of this thesis is carried out in the next chapter, where we try to apply the method to the design of a case study.
Chapter 4

UML TO DAML MAPPING AND AN APPLICATION SCENARIO

4.1 Improving the UDB with DAML Mapping

This section exposes the natural enhancing of UDB with the support of DAML+OIL mapping together with issues due to the differences between UML and DAML models. In 2.7 the procedure used by Cranefield to implement RDF schemas from UML is shown. Another similar tool is being developed by the UML Based Ontology Toolset (UBOT) project’s team. The UBOT project is part of the DARPA Agent Markup Language (DAML) Program with the purpose of allowing effective machine-readability of annotated information on the web. Thus it is significant also to detail the UBOT architecture to give a complete idea of how this mapping system shall work.

In figure 4.1 taken from [44] the UBOT steps are summarized: an ontology engineer would graphically model a new ontology or extend an existing one with a UML CASE tool (e.g. Rationan Rose [41]). The model will be exported from the UML GUI in XMI format to the UML Formalization component, which translates the model for consistency checking in Specware. Therefore the engineer will correct the model based on checking results. The model will then be exported to the UML DAML translation component. This is a very high-level architecture, especially in the definition of the DAML translation component, and gives full autonomy in the implementation of this mapping, remarking only the obvious need of XMI. The UDB idea can successfully be applied to UBOT schema, resulting in a new
We now expose a new proposal for an architecture based on the UDB and UBOT structures, called UML for Knowledge Representation (UKR). UML Data Binding is based essentially on two stylesheets: \texttt{xmi-to-rdfs.xsl} and \texttt{xmi-to-java.xsl}. Since using XSLT stylesheets to translate an XMI structure in RDFS is powerful and is the most natural manner to deal with transformations through XML documents, this solution is not so good and general for Java Classes and methods. In addition assuming a direct change of an imported DAML ontology by the human user (thought the UML GUI), as is done by UBOT, is not the goal of semantic web agents (the real ”users” of ontologies), and is only a secondary aspect. Our observations are:

- Who models the ontology is an ontology engineer (as in UBOT), but may also be, in the future, (more generally and for smaller systems) a simple web master or an expert home user as is presently for HTML. Hence the necessity of some general UML/DAML CASE tools is expected, but it is not rational for these reasons to do a manual exploration and extension of ontologies (i.e. using UML GUIs). The utility of ontologies in this manner is restricted to supporting only page an-
4.1. IMPROVING THE UDB WITH DAML MAPPING

notation, but we argue at length that this is only a means and not the
final goal of DAML.

- Stated that representing knowledge is the purpose of UKR, who must
use and evolve this knowledge are software agents rather than people
(see 3.3). People should only intervene to support these agents though
this is only a secondary aspect.

- Agents need easy ways to retrieve and read the ontologies: for the
purposes of sharing, comparing and evolving their knowledge. So it will
be more likely for them to directly use the DAML+OIL schema together
with an XML API rather than be based on specific classes produced
in the xmi-to-java.xsl stylesheet. In general it is more correct to
represent the agent’s in-memory class schema starting with the effective
DAML+OIL ontology rather than directly from the XMI. Because it
may drive in inconsistencies (also in a future scenario) and especially
because the real XML ontology representation language is DAML+OIL
and not XMI. Hence it is expected that agents will be more familiar
with this language rather than others for its ontologies (beside XMI is
much more complex than DAML as regards human readability).

- With these conclusions a two-way mapping from UML to DAML and
vice versa is not a necessary condition for the realization of UKR. Cer-
tainly it would be a fundamental facility for developers, as is the on-
tology consistency check, but presently we prefer not to address these
problems.

The elaborated process is shown in figure 4.2. At this state of development
our attention concerns mainly the XSLT that translates XMI in DAML+OIL
annotation. It is essential that this mapping is carried out accurately because
all the other transformations depend on it. A significant work in this sense
is being done by the UBOT project’s team [3], in the next section this
issue and other mapping aspects are detailed. The remaining steps regard
the translation in object oriented classes of the ontology and the related
marshalled and unmarshalled methods. They can be thought of as being
provided by general API for specific programming languages in which agents
are developed. There are many projects in this sense, in many of the most
well known languages. This feature made the UKR architecture general
enough to be useful in different agent development environments. We make
a hypothesis of UKR-extension framework of such APIs where it is necessary
to completely allow the marshalling framework capabilities.
Chapter 4. UML to DAML Mapping and Application Scenario

Figure 4.2: The UML Knowledge Representation (UKR) Architecture

Referring to the Java Programming Language, the Jena Java API for RDF [38] (which support DAML+OIL as well) developed by Hewlett Packard Laboratories seem to be a good API to introduce in a Java-UKR architecture. Simple application examples of Jena can be found in appendix A.

4.1.2 UML to DAML Mapping and Related Issues

There are various similarities and differences between UML and DAML, that must be taken in consideration to develop a coherent mapping between these two languages. An exhaustive study of the most problematic issues can be found in [3]. Expressing knowledge in a machine-readable form requires that it be represented as data, for this reason many concepts of knowledge representation languages can be found in data language and vice versa. One of the fundamental differences between data languages and KR languages is the distinction of meta-levels [35]. In KR languages as DAML, Class is an entity whose instances are classes (while class entity is an instance of itself). That means KR languages incorporate not only modeling capabilities, but also include meta-modeling, meta-meta-modeling and so on. On the other hand, KR languages do not have packages and modularity capability. Otherwise UML and DAML have many characteristics in common. Based on these mapping can be done (Figure 4.3 shows relations between concepts).
The most problematic difference is in the DAML concept of \textit{property}. The incompatibility resides essentially in the fact that DAML notion of property is a first-class modeling element (e.g. it can exist independently of any class), while UML associations are not (they are defined in terms of association ends). Baclawski et all in [3] with a modest extension of the UML metamodel allows a consistent mapping between the two models.

In fig. 4.3 both an high level mapping of concepts and the elaborated version of this view are shown. Class mapping is quite similar to the one made by UDB; and also the \texttt{instance-of} concept can be easily realized, with types for the relation and properties for attributes. Mapping between multiplicity and cardinality is realized adding a cardinality value for the property restriction if there is a single value in the association end, or using minimum and maximum cardinality if the association end contain range values. Unique and Unambiguous Properties are used for relations which have 1 (or 0..1) to many (*) multiplicity. Other important dependencies expressed in DAML are modeled with stereotypes: TransitiveProperty, equivalentTo, sameClassAs etc.

As these similarities between UML and DAML seem to make mapping quite easy to realize, there are a number of issues that render this work less straightforward. The UBOT research team gives a description of these incompatibilities, summarized here:

- **Containers**: DAML uses the notion of \textit{List} to represent containers (replacing RDF \textit{Bag}, \textit{Seq} and \textit{Alt}). Since UML supports lists with OCL, they have different interfaces and it is hard to allow an automated mapping.

- **Universal Classes**: The universal classes used by DAML and RDFS (i.e. \textit{Literal}, \textit{Resource} and \textit{Thing}) do not have a counterpart in UML.

- **Property**: In addition to the differences described above, other issues such as the number of domain classes, non binary associations, multiplicity constraints representation and the decomposition of DAML description for classes and properties make this concept one of the most problematic for the entire mapping.

- **Namespaces**: UML and DAML use a very different kind of method to link constructs between them. The mapping must take into consideration these facts to ensure the representation and distinction of names.

- **Constraints**: In DAML constraints are specified as subclasses of a restriction class, in UML usually in OCL. Besides problems in cardinality
Figure 4.3: Mapping between UML and DAML concept (taken from [3])
constraints for property also the semantics of these representations are different. In UML constraints restrict the possible instance for a model element, in DAML constraints are used to support logical inference. This may lead to erroneous deductions.

A way to get over most of these obstacles is to use a modest extension of UML elaborated by the UBOT team and exposed in fig. 4.4.

4.2 Simple UML to DAML stylesheet

This section describes how a first step in the realization of UKR has been done. We start from the uml-to-rdfs and uml-to-html stylesheets also used in UDB and try to improve them to introduce a first DAML+OIL notation. Since we have seen the many problems related to the mapping, the aim of
this section is not to give a complete realization of this stylesheet (because it will require at least a UML extension) but to give an idea of how concretely such stylesheets work. In any case the uml-to-daml stylesheet will constitute the core of the future stylesheet used by UKR, which is expected to be a uml-to-owl mapping rather to DAML+OIL. A lot of work should be however carried out by the UBOT project, the OMG, the W3C and other research in this field before a complete UML to DAML (or OWL) mapping can be correctly realized and used by applications.

4.2.1 Modifying uml-to-rdfs.xsl

There are many variable factors to take into consideration for the development of this stylesheet. The first is UML. UML model is specified currently in version 1.4, old specifications exist and new ones will appear in the future, with new features, extensions and improvements. Also XMI changes, it changes to support new features and adjusts itself to the UML versions. Moreover for each different specification of XMI, there are usually different UML DTDs (sometimes also unofficial) which permit serializing models. Also DAML (better OWL) is continuously updated because it is not a W3C Recommendation yet. This scenario is rather discouraging for an application that must use all these matters together. We however made our stylesheet based on the Cranefield one, and continue to use XMI 1.0 for UML 1.3. Fragments of this stylesheet are shown in figure 4.6. Related concepts are described in table 4.2.1.

In figure 4.5 we show a Class "Person" with an attribute "name" represented in XMI; figure 4.6 shows a fragment of xmi-to-daml which processes this class, the resulting DAML class is depicted in figure 4.7. In appendix B the full daml markup for the class diagram in fig. 2.17 is given.

<table>
<thead>
<tr>
<th>Class Model</th>
<th>daml:Ontology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>daml:Class</td>
</tr>
<tr>
<td>Attribute</td>
<td>daml:DatatypeProperty (or daml:ObjectProperty)</td>
</tr>
<tr>
<td>Binary Association</td>
<td>daml:ObjectProperty</td>
</tr>
<tr>
<td>Generalization</td>
<td>rdfs:subClassOf</td>
</tr>
</tbody>
</table>

Table 4.1: Mapping between UML and DAML realized by xmi-to-daml.xsl
4.2. SIMPLE UML TO DAML STYLESHEET

Figure 4.5: An XMI serialization of the class person shown in fig. 2.17

<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/>
<owl:Class rdf:about=""/"}

Figure 4.6: A fragment of the developed xmi-to-daml stylesheet
All matters exposed above in this thesis have raison d’être only if applied to a concrete application domain. We are developing an agent called Validation Exams Software Agent (VESA) that exploits ontology and annotated resources of a university domain to carry out the transferring of student exams between different universities or different studies. The ontology used for this work is a specific extension of the one presented in [9] called UniBo ontology. This is a significant scenario that highlights the interoperability aspect of ontologies. A student asks for a transfer from his current university to the University of Bologna. Obviously the ontology of the former university has a non-empty intersection with the UniBo ontology. This fact is due to patterns of thought discussed shortly in 4.4. The agent has to recognize what exams can be validated and what cannot, and produces a coherent response to the user query. All this process is based on ontology sharing to represent the university domain and more specifically the exam and course organization.
4.3. APPLICATION SCENARIOS WITH VESA: AN ONTOLOGY-AGENT FOR UNIBO EXAMS CONVERSION

4.3.1 Architecture of VESA

The figure 4.8 shows the general architecture on which VESA is developed. It take two fundamental inputs: one or more ontology references (which indicate where ontology concepts are described) and a set of semantic data (that constitute the input to be processed). In this way the agent can understand that information and work on it. The output of VESA is the response of the validation expressed in XML with semantic annotation (here indicated in OWL regarding a future realization).

The environment where VESA lives and carries out its tasks is depicted in figure 4.9. A user (i.e. another agent, a web service or a human person) instructs the agent to validate an exam, then the agent decides to accept or not the required task. Once accepted, VESA searches for any precedences in its knowledge that may be useful for the validation. If no precedences are found the referred ontology is analyzed and compared with the UniBo one. If who supplies the referred agent is another agent, ontology sharing scenarios may be realizable with new metabolized knowledge from both the agents. When ontology consistence is found the student’s study plan can be understood and explored. The VESA agent is being implemented with the Java programming language in parallel with the UKR and ODD progress.
Figure 4.9: VESA Environment
4.3.2 Analysis and Design

We have briefly exposed in 3.3 the hypothesis for an Agent Oriented Software Engineering methodology based on the Ontology Driven Design. Giving a complete description of the method is not the purpose of this thesis, here we argue this idea testing it in the VESA case study.

Agent and Ontology design methodology in our point of view must be put beside. Thinking of a method to put beside the Ontology Driven Design our choice has fallen on Wooldridge’s Gaia. This choice is supported by the similarities between Gaia models and the ODD ones. In particular Ontology & Conceptual Design of ODD and the Role & Interaction Model of the Gaia analysis are at the same level of abstraction. In fact with our conclusions it would be more correct to refer to Ontology/Conceptual Design as Ontology/Conceptual analysis because the nature of what we are modeling; i.e. the ontology as an abstract view of the system (in this sense Gaia analysis gives the same abstraction for the agents). Similarities can also be found between Gaia design stage (agent model, services model and acquaintance model) and navigational design. We can outline the new methodology in this manner making a distinction between a high-level analysis and design of the agent and the ontologies and a low-level design and implementation. In the high-level step an ontology schema is developed for the domain of discourse, for the ACL and the content language following the ODD method indications. Also roles and their interactions are identified and represented in the related Gaia models. At this point we have a high level representation of the system (like a ”meta representation” but it is incorrect to define it in such terms). Then the design scenarios can be modeled starting from this point, giving a more detailed representation of the system with agents, their interactions, universes of discourse and concrete syntaxes represented in the low-level phase. In the implementation context all these concepts are translated in concrete objects which represent the ontology and realize the agent behavior.

We have said that the engineering of VESA is proceeding in parallel with the evolution of the exposed technologies and this methodology. Thus exploring further the related item can bring to a substantial revision on the steps presented above. However a first draft of the high-level VESA analysis can also be given.

Some use cases are depicted from the requirements statement. In figure 4.10 the general use case is shown. VESA can play three fundamental operations: Exam validation, Ontology update, Equivalence rules update.

Figure 4.11 represents the Exam validation use case where the agent determines what exams may be validated and what may not. The user
(a generalization of roles) send a request to VESA which evaluates whether to accept it or not. If the request is accepted based on the ontology and the study plan to be validated the agent returns an output to the user. Once the operation ends, VESA checks if new exam equivalence has been produced by the elaboration and eventually stores it for future queries (we may suppose it represents this knowledge in a XML/RDF database available on the web). Equivalence rules update is shown in figure 4.12. VESA decides to carry out this operation both if it finds changes in the UniBo ontology schema and if it retains proper review equivalences based on past validations. Ontology update (Figure 4.13) is a reactive operation whereby the agent inspects known ontologies to explore new changes and eventually update its ontology schema. This operation involves some equivalence rule updates.

The designed ontology is a specific extension of the UniBo ontology described by Presutti [9]. Figure 4.14 shows the package involved in this definition: the General package that contains general concepts to be used; and the UniBo Package (an extension of General) detailed in figure 4.15.

For the VESA domain, the packages we had to improve were the Teaching and the General packages as we show in 4.16 and 4.17.

In the Teaching package a Faculty is an aggregation of Courses where different Matters can be treated which are Topics aggregations. Zero or more Exams can be related to each Matter, i.e. many students can take...
4.3. APPLICATION SCENARIOS WITH VESA: AN ONTOLOGY-AGENT FOR UNIBO EXAMS CONVERSION

Figure 4.11: Exam Validation Use Case

Figure 4.12: Equivalence Rules Update
Figure 4.13: Ontology Update Use Case

Figure 4.14: Fundamental packages of the UniBo Ontology
4.3. APPLICATION SCENARIOS WITH VESA: AN ONTOLOGY-AGENT FOR UNIBO EXAMS CONVERSION

Figure 4.15: Detail of the UniBo Package

The same exam and each student can take it more than once. However a student can insert an exam only if the exam is accepted; and for each matter there is at least one accepted exam in the study plan (all expressed in the OCL constraint). Matters instances are related with the <<equivalent to>> stereotype, used for represent equivalence in the DAML mapping. The main criteria with which exams are evaluated is based on attributes typed with EffortMeasure. The definition of this class can be found in the General Package.

The General Package presents the definition of some general concepts: address, organization, person, academic year and an important class that is the effort measure. This class, adequately sub-classed, is based on our general philosophy described in the next section. The basic parameter whereby exams are evaluated is the universal concept of working, related to the concept of effort and time. In figure 4.17 time is modeled using concepts of the Time Model described in the OMG Specification [40] that require a future extension of the UML meta-model. In this manner we can bring exams measure to a more general natural concept: time. Thus this is one dynamic part of the ontology which the agent can change, adapting itself to the new environment scenarios or improving its knowledge with new exam measures encountered. Obviously it is desirable to have a general ontological definition of time, effort and work, shared by all agents. This point is the subject matter of the next section.

Also a Gaia roles model for VESA has been drafted. We have identified four roles: Validation Role, Ontology Role, Gateway Role and Equivalence
Figure 4.16: Improved Teaching Package
4.3. APPLICATION SCENARIOS WITH VESA: AN ONTOLOGY-AGENT FOR UNIBO EXAMS CONVERSION

Figure 4.17: Improved General Package
Role Schema: Ontology Role

Description: This role manages and maintains the ontology. It guarantees the consistence and the attainability of the associated URI.

Protocols and Activities:

Permissions: Generate ontologySchema
Generate ontologyURI

Responsibilities: Liveness:
ontologyRole = (URIverify.[URIchange]—ontologyUpdate) ω
ontologyUpdate=(requestChance.ontologyChange)

Safety:
ontologySchema is consistent
ontologyURI is found

Table 4.2: Ontology Role

Role Schema: Gateway Role

Description: This role manages the matches with different ontologies. It also recognizes new ontologies or explores for changes in the known ones.

Protocols and Activities:

Permissions: Read supplied ontologySchema
Read supplied ontologyURI

Responsibilities: Liveness:
gatewayRole = (readOntology.checkDifferences.[requestChanges])ω

Safety:
true

Table 4.3: Gateway Role

Role similar to the use case exposed above. In the following tables the role model is shown.

4.4 The Mind and the Ontologies

What about ontologies as patterns of thought?

Patterns, models, ontologies, schemes, diagrams, are all instruments used by the human to represent and manage knowledge. From the undertaking of building the pyramids in 2000 B.C. to the one of building complex software systems in 2000 A.D. humans have the necessity to depict complex structures.

All the evolution of our species due to technology is marked by the transfer of ideas from the mind to the real world. And in this transfer, patterns of
### Equivalence Role

**Description:** This role manages and maintains exams equivalences.

**Protocols and Activities:**

**Permissions:** 
- Generate examEquivalence
- Read ontologySchema ontologyURI

**Responsibilities:**
- **Liveness:** 
  \[
  \text{equivalenceRole} = (\text{examValidation}.[\text{addEquivalence}] — \text{examValidation})
  \]
- **Safety:** 
  \[
  \text{examEquivalence is consistent}
  \]

<table>
<thead>
<tr>
<th>Role</th>
<th>Permissions</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalence Role</td>
<td>Generate examEquivalence, Read ontologySchema ontologyURI</td>
<td>Liveness: equivalenceRole = (examValidation.addEquivalence — examValidation)</td>
</tr>
</tbody>
</table>

Table 4.4: Equivalence Role

### Validation Role

**Description:** This role validates what exams to admit and which to reject.

**Protocols and Activities:**

**Permissions:** 
- Read examEquivalence
- Read ontologySchema
- Read ontologyURI

**Responsibilities:**
- **Liveness:** 
  \[
  \text{validationRole} = (\text{readExam}.\text{readOntology}.\text{checkConsistence}.
  \text{requestValidation}.\text{makeResponse})
  \]
  \[
  \text{makeResponse} = (\text{accept} — \text{reject})
  \]
- **Safety:** 
  \[
  \text{true}
  \]

<table>
<thead>
<tr>
<th>Role</th>
<th>Permissions</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Role</td>
<td>Read examEquivalence, Read ontologySchema, Read ontologyURI</td>
<td>Liveness: validationRole = (readExam.readOntology.checkConsistence.requestValidation.makeResponse) makeResponse=(accept—reject)</td>
</tr>
</tbody>
</table>

Table 4.5: Validation Role
thought act as infrastructure to the correct correlation of what we conceive with how it can work in the real world. For instance as long as there are cars on roads they shall have four wheels; a dictionary shall always have a known order, a light shall be forever turned on or turned off. I mean, there are universal constants of thought bound to the physical world, the culture, the biologic nature of the human brain and to history that lead to interesting meeting points, for knowledge and its representation. We could also refer to these constants as meta-knowledge or (for KR languages not structured in meta-level) as universal ontological concepts. We have stated previously our idea of ontology (which splits the scientific world). Rather than assuming a unique or few upper-ontologies we assume the existence of thousands ontologies distributed and interlinked in the web. Though a unique ontology could be the best solution for practical goals of the semantic web, it could not be the best way to promote the enhancement of an agent’s reasoning power. Rather, an ontology of key concepts (less that one hundred items) might bring good results in many research fields and the end of anarchy in knowledge representation.

We give an example of the utility of such representation in section 4.3.2 where one of these general concepts (working in hours) is used to facilitate reactivity and pro-activity, in other words autonomy. Other key ontological concepts as receiver, sender, message, organization, time, yes, no, number may be, in the philosophy of our work and in the Cranefield’s intuition (for abstract syntax, ontologies, ACLs and interaction protocols) the missing step for a correct and ”human” interaction among software components, that is among agents. Agents as software, and why not as paradigm.

What Artificial Intelligence wants is to make machines capable of reasoning; presently we are still investigating how to represent knowledge and how to realize basic inferences. Inserting in this phases within ontologies, with the ontologies, patterns of thought (i.e. patterns of conceptualization) like the human one and using these as a basis for future work is a fundamental point that could take us to the general goal of having an agent with its own head.

4.5 Conclusions and Future Works

The Semantic Web idea has been created by the W3C specifically to allow agents to use information in the World Wide Web. It is expected that in the future almost all human data and information will be accessible through the web. In this scenario Agent Oriented Software Engineering will became more and more important in complex software systems and the study of these
methodologies is an important field in the future of software engineering. There are many aspects not addressed in this thesis, and the problem is much more complex than it may appear. Though many optimistic researchers think that "intelligent" agents could be a reality in a few years, our vision is more pessimistic. Agents, as we describe this entity have a long way to go before becoming a part of our lives. It is more correct to talk about decades rather years for concrete results. The solution exposed in this thesis of agents helped by ontologies seems to be a general accepted tendency expressed by many authoritative agent researchers. But in addition to the needed growth of the technologies and methodologies here exposed also development of other agent aspects may be considered. Interaction protocols, ACLs, methods for coordination and negotiation, formal analysis of agents are only some of the matters that we have not discussed at length but which have a great importance in the agent development.

We have investigated many arguments in this thesis, and we have put forth some hypotheses. In particular the UML Knowledge Representation architecture may be a good basis for further reasoning, and a starting point for the development of VESA. Also the Ontology Driven Agent method is an idea that could be refined and improved given the importance that ontologies have for agents. The vision of ontologies as a paradigm is an intuition on which to think. The discussion is more complex on the relation among ontologies and patterns of thoughts, but important conclusions may be derived from this idea, primarily a concrete solution of many issues related to the absence of agent systems of a few ontological core concepts. Other aspects as ontologies consistence checking will be a fundamental subject given that ontologies will become so important as in our vision.

Lastly, the work presented here is strongly related to the advancement in many other research fields. Specifications for the new layers of the semantic web, changes to the UML metamodel, AI knowledge research and many other matters will bring substantial changes to the contents of this thesis. In any case, the principal idea of ontologies as a fundamental part of future technologies is a consolidated fact, this leads us to believe that ontologies will be the key factor for the new agent & WWW era.
Bibliography


Appendix A

An application of the Jena API

Java Source File that manages some simple RDF Statements

```java
import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
import com.hp.hpl.mesa.rdf.jena.model.*;
import com.hp.hpl.mesa.rdf.jena.vocabulary.*;
import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.File;

public class Jenardfexemp extends Object
{
    static String URIperson = "http://people.com/italy/101115";
    static String givenNome = "Luca";
    static String familyNome = "Abeti";
    static String fullName = givenNome + " " + familyNome;

    public static void main(String args[]) {
        try {
            //Creo il modello RDF di base dall’interfaccia Model
```
APPENDIX A. AN APPLICATION OF THE JENA API

// implementata come Codelmem
Model modello = new ModelMem();

//Creo la risorsa con l'URIperson
Resource luca = modello.createResource(URIperson);

//Aggiungo una proprietà alla risorsa
luca.addProperty(VCARD.FN, fullName);

//Aggiungo altre proprietà alla risorsa
luca.addProperty(VCARD.N, modello.createResource()
    .addProperty(VCARD.Given, givenNome)
    .addProperty(VCARD.Family, familyNome));

//Ora mostro gli statement generati
StmtIterator iteratore = modello.listStatements();

while (iteratore.hasNext()){
    //Prendo il prossimo statement e lo tengo in stat
    Statement stat = iteratore.next();

    //Smembro lo statement in delle variabili locali apposite
    Resource soggetto = stat.getSubject();
    Property predicato = stat.getPredicate();
    RDFNode oggetto = stat.getObject();
    //Non so ancora se l’oggetto è un Litteral o
    //un’altra Resource. RDFNode è un tipo per entrambi

    //con il metodo toString stampo le parti dell’RDF come stringhe

    System.out.print("Soggetto :" + soggetto.toString());
    System.out.print(" " + "Predicato :" + predicato.toString());
    //Quindi faccio la differenza per l’oggetto
    if (oggetto instanceof Resource) {
        System.out.print(" " + "Oggetto :" + oggetto.toString());
    } else {
        System.out.print(" Oggetto :" + oggetto.toString() + ".");
    }
    System.out.println(" .");
    }
//Ora scrivo sullo Standard Output l'RDF in sintassi XML (RDF/XML)
System.out.println(".");
modello.write(new PrintWriter(System.out));

//Ora lo scrivo su un file
File firstrdf = new File("firstrdf.out");
modello.write(new FileWriter(firstrdf));

} catch(Exception e){
System.out.println("Errore: " + e);
}
}

The relative output

Soggetto :http://people.com/italy/101115
Predicato :http://www.w3.org/2001/vcard-rdf/3.0#N
Oggetto :anon:1fc4bec:f3c2ea9f8c:-7fff .

Soggetto :anon:1fc4bec:f3c2ea9f8c:-7fff
Predicato :http://www.w3.org/2001/vcard-rdf/3.0#Family
Oggetto :

Soggetto :anon:1fc4bec:f3c2ea9f8c:-7fff
Predicato :http://www.w3.org/2001/vcard-rdf/3.0#Given
Oggetto :

Soggetto :http://people.com/italy/101115
Predicato :http://www.w3.org/2001/vcard-rdf/3.0#FN
Oggetto :

<rdf:RDF
APPENDIX A. AN APPLICATION OF THE JENA API

```xml
<rdf:RDF>
  <rdf:Description rdf:about='http://people.com/italy/101115'>
    <vcard:FN>Luca Abeti</vcard:FN>
    <vcard:N rdf:nodeID='A0'/>
  </rdf:Description>
  <rdf:Description rdf:nodeID='A0'>
    <vcard:Given>Luca</vcard:Given>
    <vcard:Family>Abeti</vcard:Family>
  </rdf:Description>
</rdf:RDF>
```
Appendix B

xmi-to-daml output

An example of the xmi-to-daml output for the family class diagram shown in figure 2.17

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:daml="http://www.w3.org/2001/10/daml+oil#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema" xml:lang="en">
  <daml:Ontology rdf:about="">
    <daml:versionInfo>$Id: xmi-to-daml version 0.0.1beta </daml:versionInfo>
    <rdfs:comment> This DAML Ontology has been automatically generated by the xmi-to-daml stylesheet </rdfs:comment>
  </daml:Ontology>
  <daml:Class rdf:ID="Person"/>
  <daml:DatatypeProperty rdf:ID="Person.name">
    <rdfs:range rdf:resource="rdfs:resource="http://www.w3.org/2000/10/XMLSchema#string"/>
  </daml:DatatypeProperty>
  <daml:Class rdf:about="#Person">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#name"/>
    </daml:Restriction>
```

113
APPENDIX B. XMI-TO-DAML OUTPUT

</daml:Class>

<daml:ObjectProperty ID="Person.parent">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="rdf:Bag"/>
  <rdfsx:containerElementType rdf:resource="Person"/>
</daml:ObjectProperty>

<daml:ObjectProperty ID="Person.child">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="rdf:Seq"/>
  <rdfsx:containerElementType rdf:resource="Person"/>
</daml:ObjectProperty>

<daml:ObjectProperty ID="Person.father">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="Man"/>
</daml:ObjectProperty>

<daml:ObjectProperty ID="Person.son">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="rdf:Seq"/>
  <rdfsx:containerElementType rdf:resource="Man"/>
</daml:ObjectProperty>

<daml:ObjectProperty ID="Person.mother">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="Woman"/>
</daml:ObjectProperty>

<daml:ObjectProperty ID="Person.daughter">
  <rdfs:domain rdf:resource="#Person"/>
  <rdfs:range rdf:resource="rdf:Seq"/>
  <rdfsx:containerElementType rdf:resource="Woman"/>
</daml:ObjectProperty>

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

<daml:Class rdf:ID="Woman"
<rdfs:subClassOf rdf:resource="Person"/>
</daml:Class>

</daml:Ontology>
</rdf:RDF>