UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA
Facoltà di Ingegneria – Sede di Modena
Corso di Laurea in Ingegneria Informatica

Query Processing by Semantic Reformulation

Elaborazione di Interrogazioni tramite l'uso di Corrispondenze Semantiche

Relatore
Chiar.ma Prof.ssa Sonia Bergamaschi

Tesi di Laurea di
Raffaele Capezzera

Correlatore
Chiar.mo Ing. Maurizio Vincini

Anno Accademico 2002 - 2003

Contents

List of Figures

List of Tables

Preface

Introduction

This thesis endevours unifying contributions towards some still unsolved issues for the development of both the *SEWASIE* system and *autonomous semantic Web services*. The former is a European project (name: SEmantic Webs and AgentS in Integrated Economies, number: IST-2001-34825) involving academic and commercial partnership. The latter is a project undertaken at Carnegie Mellon University (Pittsburgh, PA) as part of the DAML program (under the US Air Force Research Laboratory, contract F30601-00-2-0592) as well as the Interoperability program (under ONR, contract N00014-02-1-0499).
The premises of this thesis were born of personal research activity at Intelligent Software Agents Laboratory in the Robotics Institute at Carnegie Mellon University.

Trusting the Socratic teaching that original achievements cannot arise from else than critical analysis of past knowledge, the backbone of this thesis is an accurate examination of relevant papers in the fields of *Intelligent Integration systems*, *agent technology*, *Description Logics*, *ontology languages*.
Furthermore, this thesis attempts to be as self-contained as possible w.r.t. faced problems and proposed solutions.
Herein, it is presented the plan of the work chapter by chapter, accompanied by pointer to the basic reference readings.

I Mediator-Based Systems

It is introduced the problem of query answerability by an Information Source. In the context of mediator-based systems, the Capabilities-Based Rewriting and the Query Expressibility Decision problems are consequently derived. Source capabilities descriptions are then pointed out to be necessary to deal with the mentioned concerns. From this standpoint, sample mediator-based systems (such as MOMIS, TSIMMIS, Information Manifold, SIMS, and Carnot/InfoSleuth) are briefly reviewed.
(Basic references: [1], [14], [47], [57], [98])

II Middle-Agents

As a prerequisite to deal with middle-agents, an introductory definition of agents is firstly presented. Generalizing the problem of query answerability to the fulfilment of an information request in Multi-Agents System, middle-agents are classified w.r.t. their approach to the connection problem. A functionality-based and a privacy-based taxonomy are reported. In addition, interaction protocols for the most interesting type of middle-agents (matchmaker and broker) are devised. After all, some performance parameters for brokered and matchmaker-based organizations are shortly describeed.
(Basic references: [25], [105])

III SEWASIE Brokering Agent

The SEWASIE system architecture is illustrated at a high level of abstration and at a finer detail. In particular, main components such as SEWASIE Information Node, Query Agent, and Brokering Agent are described. Recalling the initial concern with query answerability, query management is formilized within SEWASIE and it is speficically examined how the Brokering Agent is addressed to query management. Some related open issues are ultimately stressed out.
(Basic references: [86], [87])

IV DAML-S Broker

Autonomous semantic Web services are envisaged as a bridge between Semantic Web and Web service technologies. DAML-S is then emphasized as a capability description language for Web services. Along the same direction is the mention of a specific algorithm to match Web services capabilities in order to satisfy a service request. Within this perspective, the DAML-S Broker is introduced as a semantic autonomous Web service as well as a broker middle-agent. Some related open issues are ultimately remarked.
(Basic references: [69])

V Common Threads

As a natural prosecution of the foregoing chapters, the current one starts with an analysis of the open issues common to both the SEWASIE Brokering Agent and the DAML-S Broker. Subsequently, they are treated in deeper details with a separate examination. In particular, it is primarily devised an interaction protocol for the SEWASIE Brokering Agent and it is proposed DQL (DAML Query Language) as the query language for the DAML-S Broker. In the end, it is rendered a common perception of Query Processing divided into three subtasks. Namely, Query Abstraction, Semantic Matching, and Query Rewriting. The thesis is mainly focused on Query Abstraction and Query Rewriting. Keeping them both as a whole, Query Reformulation is defined.

VI Description Logics, OWL, and ODL$_{I3}$

On the way towards a comprehensive discussion of Query Reformulation, some ontology languages relevant to DAML-S Broker and SEWASIE Brokering Agent are introduced. They are OWL and ODL$_{I3}$, respectively. A special attention is addressed to their reasoning attitude. Hence, some hints to Description Logics are given as a substantial ground for the reasoning problems supported by OWL and ODL$_{I3}$.
(Basic references: [14], [46])

VII Concept Rewriting

Keeping on the backgrounding journey to Query Reformulation, the focus is now on terminology transparency to achieve semantic interoperability. The matter of rewriting concepts using terminologies is consequently formalized, and its application to Query Processing is reviewed w.r.t. to two Intelligent Integration systems (OBSERVER and SIMS).
(Basic references: [3], [7], [61])

VIII Rationale for Query Reformulation

This is the core chapter of the whole thesis. Here, the applicatory principles of Query Reformulation are formalized. Such a formalization is figured out into two distinct parts:

Query Abstraction and Query Rewriting. In turn, Query Abstraction articulates in Instance Abstraction and Variable Abstraction, whereas Query Rewriting comprises Concept Transformation and Instance Rewriting.
The last part of the chapter is dedicated to the theoretical explanation of a semantic similarity heuristics -in some extent exploiting the WordNet lexical database- which enhances the operative algorithms for the implementation of Query Abstraction and Query Rewriting.

IX Reasoning Tools

The last preable leading to the final implementation regards the reasoning tools which nowadays are potentially exploitable. Thus, main characteristics of ODB-Tools, JTP, DAMLJessKB, FaCT, and RACER have been sketched.
(Basic references. [43], [12], [31], [53], [39])

X Implementation

At last, it is illustrated the current implementation of the earlier explained theoretical work. The main coded modules are analyzed. At the same time, the applied algorithms are outilined. A sample case study ends up this chapter.

XI Conclusions and Future Work

Future research comes as an ideal continuation of the conclusive remarks. So far, Query Reformulation covers queries constructed as binary predicate triples only. In addition, Semantic Matching and, consequently, Query Reformulation marginally take into account the query predicate. Therefore, preeminent goals of future work will be dealing with queries constructed as conjunction of binary predicate triples, and tackling predicate interpretation. Semantic Matching should accordingly revised.
From this standpoint, investigation will then move across extraction, interpretation, and reformulation of query terms to derive any significant parameter for Semantic Matching w.r.t. the DAML-S paradigm.
At the implementation level, the exploitation of ODB-Tools as an alternative reasoner (currently, the used one is RACER), a DQL interface for the DAML-S Broker, and some further heuristics for optimization of the operative algorithms might be developed.

Acknowledgements

Professor Sonia Bergamaschi, for having given and backed the great opportunity to spend three months doing research at Carnegie Mellon University.
For the accurate revision of the work that has followed.

Research Professor Katia Sycara, for having advised my activity in the Intelligent Software Agents Laboratory.
Massimo Paolucci, for the very interesting discussions which inspired such a research activity.

My parents, for having taught me that whenever *you* learn it's *you* to grow wiser, nobody else.
For this thesis is the last step of a long path and they have been the ultimate support to get up to this point.

With its unique importance.
Like each person underlying this uniqueness. Behind and beyond this thesis.

I wish to thank them all.

Raffaele

In un ufficio di laboratorio,
un'unica finestra su un laboratorio di robot
Fuori, dal corridoio e dalla porta
cielo senza nuvole,
abbraccio di sole su prati verdi…
Alienazione irreversibile?
O necessaria ricerca per consentirci di vivere
"meglio"
quel sole, quel prato, quel cielo?

Abstract

Query Processing in mediator-based systems primarily deals with query answerability. Referring to the integrated information sources, content and query capabilities description are then essential to successfully answer a query. Moving to Multi-Agent Systems, a query must be compiled into an information request to be fulfilled by some information provider.

Both the SEWASIE Brokering Agent (BA) and the DAML-S Broker (DSB) are middle-agents involved in query management and share some open issues. In particular, they must embody matching functionalities to discover an information provider capable to answer a given information request. Using DAML-S as a capability description language let the DSB behave as a semantic Web service, thus achieving a Semantic Matching between an information requester and relevant information providers. Within this framework, DQL (DAML Query Language) has been advised as the query language for the DSB.

Query Processing requires a substantial Query Reformulation process to enable Semantic Matching. Such a Semantic Reformulation is split into Query Abstraction and Query Rewriting. W.r.t. a common reference ontology both Query Abstraction and Query Rewriting have been formalized and implemented. The motivating rationale is to abstract instance or variable terms of the initial query into concepts representing service capabilities. Once a matching provider for such an abstracted query has been found, concept rewriting must be performed in order to supply the provider with the expected parameters. Intra-ontology relationships are exploited to accomplish such a transformation. Moreover, semantic similarity (WordNet aided) heuristics enhances the whole multi-phase procedure.

Casting the DAML-S paradigm to the SEWASIE Brokering Agent might allow to keep the same strategy for Query Processing even for such a broker. Nevertheless, the subtask of Query Reformulation could be equally applied to any situation in which a query needs to be abstracted and rewritten in order to be successfully answered.

A Description Logics based ontology language (i.e. OWL DL) supports the whole process. In spite of this, a proved exportability from/to ODL_{I3} permits to use it too for the reference ontology.

Keywords

Query Reformulation, *Semantic Matching*, *Broker*, *SEWASIE*, *DAML-S*.

I Mediator-Based Systems

1. Source Capabilities Descriptions

The primary question which opens and motivates our discussion is the following:

1. *How can a broker determine whether a query can be successfully answered by an Information Source that advertises with the broker itself?*

At this very initial point, such a question shows up more than a cryptic term. No defintion has been supplied neather for the subject of our concern (i.e. the "broker") nor for the action joining query and broker (i.e. "advertising"), yet.
In order to commence our analysis, here the broker can be regarded as a generic entity which interleaves between a generic user asking a query for some information and some generic sources of information potentially able to provide the queried information (i.e. Information Sources, ISs).
In the field of Intelligent Information Integration (I3, see [I^3 Reference, 1995]), the question above has stimulated a major part of research about *mediator-based* systems.
Among the user and the ISs , these integration system denote the presence of two kinds of actors: *mediator* and *wrapper*. As represented in figure 1, each wrapper is positioned just above an IS, while mediators are half the way among wrappers and users.

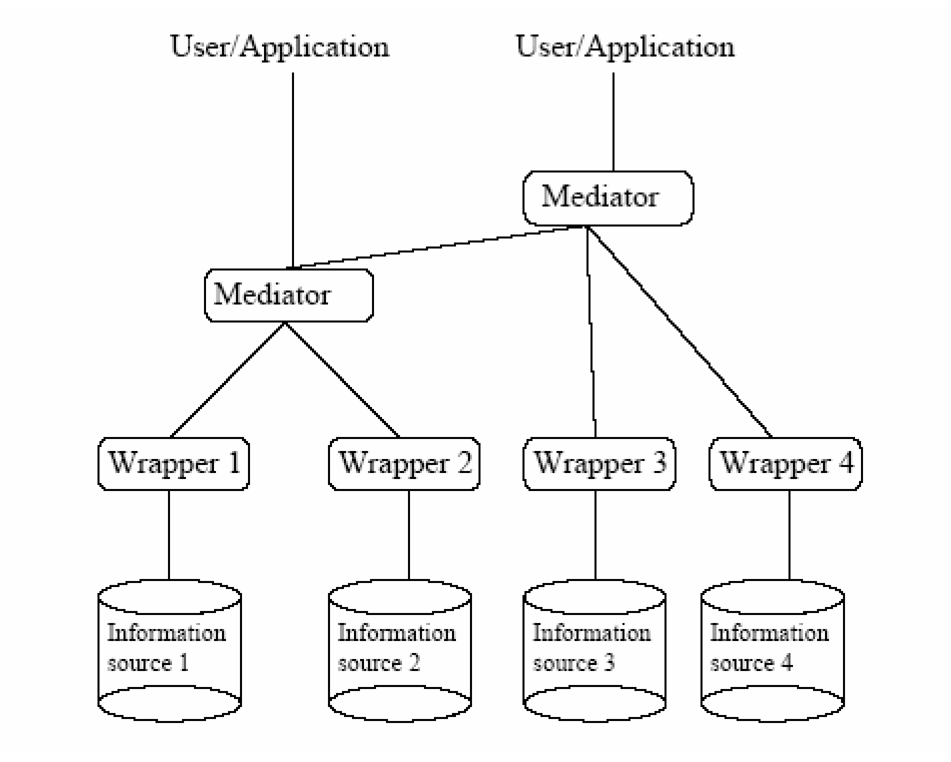


Figure 1 General architecture of a mediator-based system

This structure reflects the respective tasks of wrappers and mediators. The former are essentially translators each in charge of a specific IS, in that every IS can potentially have its own data model and the correspondent data language and query language. Therefore every wrapper is aimed to both export (i.e. to make visible and understandable) the content of the underlying IS to the mediators level, and to translate the queries coming down from the mediators into queries or commands directly supported by the IS. Mediators are instead aimed to interleave between the information-base of the system and the user. As ISs can be widely distributed and internally heterogeneous, mediators provide an integrated view of the wrapped ISs. Furthermore mediators are responsible for query decomposition and planning. Queries coming down from the user level have to be mapped (decomposed) in a suitable way for the wrappers and an execution plan has to be developed.

From this perspective, our initial concern is vitally sensible for both mediators and wrappers. In fact, we can reformulate question 1 in analogous questions for mediators and wrappers:

2. *How can a mediator determine whether a query can be successfully answered by the wrapped Information Sources below it?*
3. *How can a wrapper determine whether a query can be successfully answered by the underlying Information Source?*

Even though the prior aim remains the answerability of a query, such an issue takes slightly different connotations for a mediator and a wrapper.

For a mediator, resolving the answerability of a query means resolving the *Capabilities-Based Rewriting* (CBR) problem (see [Papakonstantinou et al., 1996]). Practically, a mediator needs to know the *query capabilities* (i.e. the set of supported queries) of the wrapped sources so that the query mapping can be successfully accomplished. In other words, the global query posed by the user must be rewritten into local queries according to the query capabilities of the underlying sources.

For a wrapper, our question is an alternative formulation of the *Query Expressibility Decision* (QED) problem (see [Vassalos and Papakonstantinou, 1997]). A wrapper needs to know the *query capabilities* of the underlying IS so that the translation of the local query from the mediator into a query supported by the IS can be successfully accomplished.

So, in mediator-based systems, we notice that the answerability of a query is strictly related to the description of the information sources' query capabilities.

Once a query has been properly rewritten (CBR) and translated (QED) the considered IS can understand it and parse its content in order to find an answer. Therefore, at this point we also need the description of information sources' *content capabilities*, so that an answer to the query can be eventually found.

Figure 2 shows how source capabilities are known along the components of a mediator-based system.

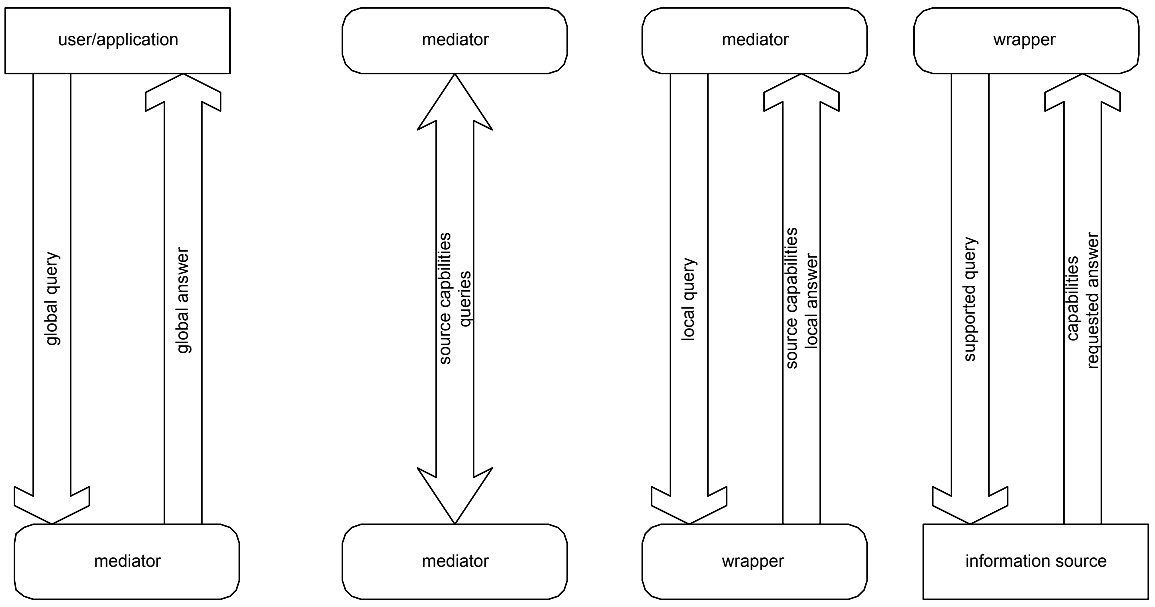


Figure 2. Information flows among the components of a mediator-based system.

In mediator-based systems *query capabilities* descriptions and *content capabilities* descriptions about the ISs are tightly interconnected. Algorithms developed to cope with the answerability of a query firstly depend on those descriptions. Above all, those descriptions rely on the integration model followed by the system.
The current state of the art shows two distinct directions to build up an integration system:

- *Global As View* (GAV);
- *Local As View* (LAV).

In the GAV approach, the global model of the integration system is inducted as a view over the local information sources. In the essence, the integrated view of the system is a *mapping* where each element in that global view corresponds to one or more elements in the local sources. The formulation of such a mapping can be achieved in several fashions. Some examples are *mapping rules* (see [Bergamaschi et al. 2001] *articulation axioms* (see [Huhns et al, 1993]).
By the contrary, in the LAV approach, the models of local ISs are deducted as a view over the global model of the integration system. Now, we have a mapping where each element in the local sources corresponds to one or more elements in the global model and the global model can be a superset of the involved sources.
The formulation of the global model is then another distinguishing feature among various systems. For instance, we can basically characterize the global model as a proper *ontology* (see chapter 7 for a definition) or as a conceptual model not necessary organized as an ontology.
A further feature that also affects sources' query/content capabilities descriptions is the infrastructure where the integration system is placed. It might happen to deal with a mediator-based integration system nested within a Multi-Agents infrastructure (MAS, see chapter 2 for a proper definition), or we might simply deal with an integration system constituted as self-standing, monolithic system.
In the sequel of this first chapter, we will review the attempts made by five mediator-based systems to specifically face the answerability of a query. Those systems are taken as paradigmatic instances to highlight the different influence of the just hinted characteristics on the suggested solutions for the answerability of a query. Peculiar responses to questions 2 and 3 will be the backbone of such a brief analysis.

2. Sample Mediator-Based Systems

2.1. MOMIS

Characteristics

- MOMIS (Mediator environment for Multiple Information Sources, [Bergamaschi et al., 2001]) was conceived as a pool of tools, to provide an integrated acciesss ot heterogeneous information stored in tradiational databases (e.g., relational, object-oriented) or file systems, as well as in semistructured data sources. It was developed as a joint collaboration between the University of Modena and Reggio Emilia, CSITE-CNR, and the University of Milano and Brescia, within the MURST "Interdata" and the D2I Italian research projects.
- MOMIS implements the wrapper/mediator I^3 architecture ([I^3 Reference, 1995]) on a GAV approach. A Global Virtual View (GVV) of the integrated sources is constructed as a common ontology.
- The data model is ODM_{I3}, while the data representation language is ODL_{I3}, and the query language over the GVV is OQL_{I3}.
- Gathered tools comprise the cluster generator ARTEMIS ([Castano et al., 2000]) which performs affinity-based classification of local classes for the synthesis of global classes and the lexicon ontology WordNet.
- A designedly developed module is ODB-Tools, a Description Logic (specifically $\mathcal{OLCD}$, [Beneventano et al., 1998]) based engine, which performs schema validation for the generation of a Common Thesaurus (of inter-source terminological relationships) and semantic query optimization.

Achievements

- Combination of reasoning capabilities of Description Logics with affinity-based clustering techniques. It allows both the validation of inter-source knowledge used for the integration and the identification of candidates to integration.
- Interactive exploitation of WordNet (see [Miller et al., 1990]) combined with subsequent affinity analysis.
- Capability of explicitly introducing kinds of knowledge as integrity constraints, and extensional relationships.
- Possibility to check the global consistency of implicit and designer-provided knowledge.
- *Content capabilities* descriptions do not need to be explicitely declared since queries will be posed against the global integrated classes, which represent a kind of global content description of the wrapped ISs. Thereby, the system is able to automatically identify the relevant sources to answer a query.

Limitations

- At the moment the module devoted to query management within the mediator component does not support query optimization and answer composition functionalities based on definition of extensional axioms and integrity constraints defined on global classes (i.e. the CBR problem), even if these problems have been faced at the theoretical level.

- Hence, no language is provided to describe query and content capabilities of the wrapped Information Sources.

2.2. TSIMMIS

Characteristics

- TSIMMIS (The Stanford IBM Manager of Multiple Information Sources) was presented by Stanford DB Group in 1994 (see [Chawathe et al., 1994]).
- It is a mediator-based integration system for heterogeneous sources conceived on a GAV approach.
- The integrated view of the system and, more generally, all the data conveyed across the system, are represented in Object-Exchange Model (OEM, [Papakonstantinou et al., 1995]).
- Mediator Specification Language (MFL, [Papakonstantinou et al., 1996, bis]) serves as both the view definition language and the query language.
- The source contents and querying capabilities are instead described in Wrapper Specification Language (WSL, [Vassalos, 1996]).

Achievements

Most of the results relevant for our concern come from the theoretical research developed in parallel to TSIMMIS -e.g. these results are the rationale for the definition of the wrappers ([Hammer et al., 1997]). Mainly they were published first on ([Vassalos and Papakonstantinou, 1997]) and then enriched and improved on ([Vassalos and Papakonstantinou, 2000). We can list them as follows.

- Introduction of p-Datalog as a language for descriptions of *query capabilities* of information sources.
- Development of an algorithm to decide whether a query is described by a p-Datalog description, hence a working solution for the QED problem.
- Development of an algorithm to decide whether a query can be answered by combining supported queries, hence an implemented solution for the CBR problem. In turn, this is equivalent to the problem of answering the user query using an infinite set of views described by a p-Datalog program (for an interesting survey, see [Halevy, 2001]).
- Proof that p-Datalog cannot express existing (i.e. corresponding to *real* sources) recursive sets of conjunctive queries.
- Consequent extension of RQDL (Relational Query Description Language, originally defined in [Papakonstantinou et al., 1996], such that it can describe the set of all conjunctive queries.
- Development of the QED algorithm for RQDL.
- Development of the CBR algorithm for RQDL.
- Implementation of an algorithm that takes as input RQDL descriptions of queries supported by the sources and outputs an RQDL description of all queries supported by the system itself.

Limitations

- The results above are proved only for sources that support conjunctive queries. However there is a planned intention to extend the work also to non-conjunctive queries, i.e. queries involving aggregates and negation.

- All the implemented algorithms require as an input the hard-coded descriptions of the query capabilities supported by the sources.
  This drawback can be figured out looking back at the GAV approach followed by TSIMMIS. Even though this approach allows a straightforward query planning, it shows its bad face when the information sources of the system need to be updated. Then it turns out that lots of elements of the integrated view might be modified. Equally, lots of query capabilities descriptions have to be modified if one or more information sources are added, deleted, or updated.
- Connecting to TSIMMIS users can specify queries in TSIMMIS query language only. However the algorithms mentioned above apply for user queries formulated in any common language.

In TSIMMIS, WSL is used in place of RQDL to supply the system with templates describing *query capabilities* of the sources. Mediators will use those templates to rewrite the user query -formulated in MSL- in a comprehensible manner for the sources. Each wrapper will use those templates and the provided source schema to translate an incoming query -formulated in MSL- in a query directly supported by the underlying source. Essentially, while WSL provides templates to describe source querying capabilities, source schemas contribute to fill up those templates with the respective contents. The wrappers will then be able to create *content capabilities* descriptions from WSL templates. The mediator will specifically use these descriptions to rewrite user MSL queries into MSL subqueries, so that each wrapper will be able to translate the incoming MSL subqueries into queries or commands supported by the underlying sources.
In the end, it must be noticed that templates have to be manually provided to the system.

2.3. Information Manifold

Characteristics

- Information Manifold is a global information system built at AT&T Bell Laboratories in the middle of the last decade (see [Levy et al., 1996]).
- The system is thought on a LAV approach.
- The used data model is called World View.
- CLASSIC knowledge representation language (see [Borgida et al., 1989]) is used to express the World View as a combination of a relational model with an object-oriented model [Levy et al., 1995].
- The World View can accept queries posed in any query language proposed for object-relational databases.

Achievements

- Contents of the sources are described as queries over a set of relations and classes of the global model (World View) –sources' *contents capabilities* are so declared.
- Capabilities records over the data model are used to describe which queries a source can answer about its contents –sources' *query capabilities* are so declared.
- Implementation of an algorithm that uses source descriptions to create query access plans to several sources to answer a query.

Limitations

- Source descriptions are created by hand and then inserted.
- The knowledge representation language doesn't support recursive queries –in that case the necessary view rewriting is undecidable.
- The knowledge representation language -that is the language used to describe source capabilities, too- is less expressive than p-Datalog (see TSIMMIS).
- Since the semantic correctness of a query rewriting is based on maximally contained rewritings (for a survey, check [Halevy, 2001]), the system may return an incomplete answer without being able to signal it.

2.4. SIMS

Characteristics

- SIMS -Services and Information Management for Decision Systems- was introduced by the Information Agents Research Group (University of South California) in 1993 (see [Arens et al., 1993]). Nowadays it is exploited in a reshaped Web-based version called Ariadne (reference at [Knoblock et al., 1998]).
- It is a mediator-based integration system for heterogeneous information sources based on a LAV approach. The system is specialized to a single "application domain".
- An automatic elaboration of a set of axioms confers also a GAV perspective to the system.
- The domain (global) model is specified in a subset of Loom (see [MacGregor, 1990])
- Queries are also expressed in the domain model language (see above).

Achievements

From the point of view of an efficient query processing (i.e. query optimisation problem), SIMS tackles both the arising issues. These consist in firstly selecting a set of sources that can be used to answer a query, and secondly in generating a cost-effective query access plan that specifies the order of retrieval and manipulations on the data. Clearly our focus is on the former aspect. Here we list the interesting results mainly published first on [Arens et al., 1996, bis] and then revised and improved on [Ambite et al., 2001].

- Pre-compiled domain axioms describe combinations of sources that may be relevant for an incoming query, if possible. Hence, sources' *content capabilities* descriptions are so provided.
- Exact source definitions -created through mapping onto the domain model- support complete answers to queries.
- When complete answers are not possible, the system can determine if and in what way the answer is incomplete. To actually perform this prediction the system exploits an efficient organization -properly said "lattice"- of the domain axioms.
- There is no need to provide *query capabilities* descriptions for the sources because the user is assisted to formulate query in the domain-level language, only. The system itself is then responsible to direct queries to the sources in an understandable way.

Limitations

- As is has been just remarked, SIMS accepts queries formulated only in its own domain-level language.
- Exact source definitions (see above) don't allow the system to represent a source as an arbitrary join over the classes of the domain model. Thus, there is the admitted -and noticed to the user- risk of partial information representation.
- Because of potentially high complexity, to show experimental results sources and query-language are assumed to deal only with positive views and positive queries with order constrains (see [Duschka and Genesereth, 1997]).

2.5. Carnot and InfoSleuth

Characteristics

- Carnot (for a detailed explanation, refer to [Huhns et al., 1993]) is an integration system -primary thought for heterogeneous databases such as enterprise information models- developed at Microelectronics and Computer Technology Corporation (MCC) from 1990 to 1994. The MAS infrastructure InfoSleuth started at MCC in 1994 has its actual roots in the Carnot project (refer to [Bayardo et al., 1997]).
- Carnot is structured in a LAV framework.
- In Carnot, the adopted global schema -or context- is usually the existing Cyc knowledge base (reference at [Lenat and Guha, 1990]). This schema is also referred to as the common ontology of the system. Sometimes the common ontology can also be expressed and accessed through Carnot's own knowledge representation tool called KRBL (Knowledge Representation Base Language, see [Woelk et al., 1992]).
- Hence, in Carnot, semantics are usually expressed either in Cyc's logic language, that is GCL (once more, see [Woelk et al., 1992]), or in KRBL.
- In Carnot, a set of articulation axioms captures bidirectional mappings between local sources and global context. Thus a source-independent maintenance of the system is guaranteed. In a sense, this reminds to a GAV perception of the system.
- In InfoSleuth, Cyc's local contexts are adjusted and integrated into multiple ontologies belonging to diverse user groups.
- InfoSleuth's ontologies can be represented in multiple ways (e.g. in KIF, [Genesereth and Fikes, 1992], or in LDL, [Zaniolo, 1991]).
- In InfoSleuth queries are internally formulated in SQL or KIF.
- In InfoSleuth, specialized broker agents match information needs (specified in terms of some ontology) with currently available resources. This mechanism is called "Information Brokerage".

Achievements

In Carnot, articulation axioms formally specify the semantic relationships between the global model and each local model (resource). That is the way to make local sources' *content capabilities* descriptions available at the integrated view level.

- In InfoSleuth, query processing is accomplished by proper agents, which look up the common ontological models known to the system (see "Information Brokerage" among the characteristics of InfoSleuth). Therefore, ontologies supply the sources' *content capabilities* descriptions required to answer a query.

- Furthermore, in InfoSleuth, *query capabilities* descriptions for the information sources have no reason to exist, since queries have to be posed in terms of the ontologies known to the system (see below).

Limitations

- In Carnot, to access the resources through the global view queries need first to be translated into GCL and then into different resource management languages.
- In Carnot, user interaction is required to develop articulation axioms (the semantic backbone of the system).
- In InfoSleuth, the user is compelled to select an available ontology that corresponds to the interested domain, if possible. The user will then be guided to formulate his/her query in terms of the selected ontology.

2.6. Comparative Summary

Our motivating questions, namely 2 and 3, are exhaustively answered only by the work related to TSIMMIS and by the work regarding Information Manifold. The remaining systems (MOMIS, SIMS, and Carnot/InfoSleuth) bypass some of our concerns, in that they directly let the user formulate queries over the provided domain models or ontologies. We should then investigate the tools that allow the user such a formulation. In practice, we should analyse the procedures that implicitly force the user to choose a query over a limited range of available alternatives. Obviously, such an investigation is not feasible.
However, we can thoroughly compare the different strategies to answer a query, which is granted to belong to the set of understandable queries for the ISs.
We can ultimately summarize that, while TSIMMIS and Information Manifold explicitly resort to both *query capabilities* and *content capabilities* descriptions for the ISs, MOMIS, SIMS and Carnot/InfoSleuth implicitly infer *query capabilities* and explicitly describe *content capabilities* (in terms of their global models).
The following table synthesizes some relevant features highlighted for each system.

	GAV	LAV	Ontology	MAS Infrstructure	Explicit Query Capabilities	Explicit Content Capabilities	Algorithms to infer how and where a query can be answered
MOMIS	♦	-	♦	-	-	♦	♦
TSIMMIS	♦	-	-	-	♦	♦	♦ (w.r.t. given templates)
Information Manifold	-	♦		-	♦	♦	♦
SIMS	♦	♦	♦	-	-	♦	♦
Carnot	♦	♦	♦	-	-	♦	♦
InfoSleuth	♦	♦	♦	♦	-	♦	♦

Table 1. Mediator-based systems and source capabilities descriptions.

II Middle-Agents

1. Agents

After a comprehensive review of the definitions for *agent* available in literature, [Franklin and Graesser, 1996] provides the following statement:
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 pursuit of its own agenda and so as to effect what it senses in the future.
Furthermore, the behaviour of an agent can be qualified in greater details by a host of properties. An agent can be:

- autonomous, if it exercises control over its own actions;
- reactive (or intelligent w.r.t. sensing and acting), if it responds in a timely fashion to changes in the environment;
- goal-oriented (in other words pro-active or purposeful), if its intelligence does not simply imply acting in response to the environment;
- temporally continuous, if it embodies a continuously running process;
- communicative (or socially able), if ti communicates with other agents, even including people;
- able to learn (or adaptive), if it changes its behaviour on the basis of its previous experience;
- mobile, if its able to transport itself from one machine to another;
- flexible, if its actions are not scripted;
- characterized by a believable "personality" and emotional state.

Every agent, by the definition above, satisfies the first four properties.
Possible subclassification schemes may also be via control structures on the agent itself, via environments where an agent acts (e.g., database, file system. network, Internet), via language (in which an agent is written), or via concrete application of the agent.
For the purposes of our discussion, we are interested in agents dealing with information sources, namely information agents. In [SIG] the following definition is traced:
An *information agent* is a computational software entity (i.e. an intelligent agent) that may access one or multiple, distributed, and heterogeneous information sources available, and pro-actively acquires, mediates, and maintains relevant information on behalf of its user(s) or other agents preferably just-in-time.
So, our information agents are at least autonomous, communicative, mobile agents.

2. Middle-Agents

2.1. The Connection Problem

The environment where our information agents are settled is an open Multi-Agent System

(MAS). As pointed out in [Jennings et al., 1998], a MAS can be defined as a loosely coupled network of problem solvers (in our context embodied by information agents) that work together to solve problems that are beyond the individual capabilities or knowledge of each problem solver (some examples of MAS are RETSINA and MIKS, respectively described by [Sycara et al., 2001] and [Gelati et al., 2001]).
Referring to the taxonomy presented in [Wong and Sycara, 2000], in an open MAS there are two types of agents: *end-agents* and *middle-agents*. End-agents need or can offer services. Middle-agents exist to enable interactions among end-agents. End-agents act as *providers* when they offer services; and act as *requesters* when they need them. It goes without saying that an agent can act both as a provider and as a requester in a system. Strictly speaking, middle-agents offer services too. Those services include locating end-agents for each other, and intermediating their transactions and dispute resolutions.
A crucial problem that arises is the *connection problem* (see [Davis and Smith, 1983]), i.e. finding the other agents who might have the information or other capabilities an agent needs. The connection problem modelling originates some classification schemes for middle agents. In the following we move thorough two main categorizations for middle-gents respectively devised in [Wong and Sycara, 2000] and [Decker et al., 1997].

2.2. Functionality-Based Taxonomy

In a capability-based coordination, providers' functionality (i.e. what they can actually provide) and requesters' needs are given by specifications called *capabilities*. When offering a service, providers sometimes accompany capabilities with *service parameters*, which specify ulterior conditions for the service suitability. On the other side, requests specifications can be accompanied by *preferences*, which act as counterparts of service parameters.
The mentioned taxonomy was peculiarly envisaged for middle-agents. It basically builds on activity protocol specifications for middle-agents. Those specifications are described by answers to the following questions:

1. Who sends information to middle-agents?
   Either providers push information to middle-agents and requesters pull information from middle-agents, or vice-versa.
2. How much information is sent to middle-agents?
   Either merely capabilities (requests), or capabilities (requests) plus service parameters (preferences).
3. What happens to the information middle-agents receive?
   Either it is broadcast or it is kept in a local database.
4. How is the content of the database (see foregoing question) used?
   Either it is browsed or it is queried.
5. How much information is specified in a query (see preceding question) to middle-agents?
   Again, either only capabilities (requests) or capabilities and service parameters (requests and preferences) can be provided.
6. Does the middle-agent intermediate transactions?
   Obviously, either yes or no.

Using valid combinations of values along these six dimensions, twenty-eight different types of middle agents can be obtained. For instance, a *matchmaker* is defined as a middle agent that allows providers to advertise their capabilities (and service parameters), and requesters to send requests. In response to a request, a matchmaker returns the contact information of appropriate service providers. The requester agent then chooses a service provider and interacts without the matchmaker's intermediation. By the contrary, a middle agent essentially characterized by its interleaving position between requester and provider is the *facilitator* (or *broker*). At first, provider agents advertise their capabilities with the facilitator and the facilitator keeps the advertised capabilities in its database. In response to a request for a service made by a requester agent, the facilitator selects one of the appropriate providers for that particular service and delegates the service request to it. The alerted provider does the service and return the result to the facilitator, which eventually forwards the result to the requester.

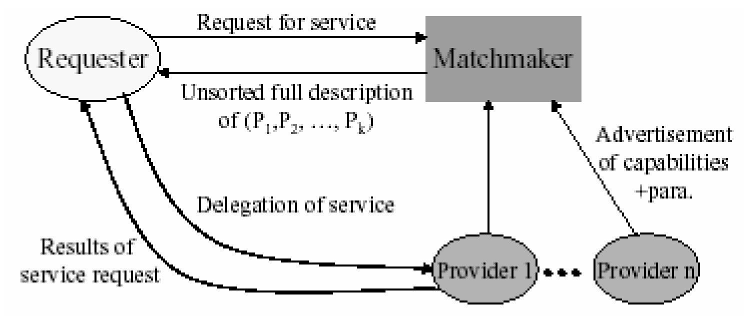

Request for service
Requester
Matchmaker
Unsorted full description of (P1,P2, ..., Pk)
Advertisement of capabilities +para.
Delegation of service
Results of service request
Provider 1
Provider n

Figure 3. Block diagram for the matchmaker's simplified interaction protocol.

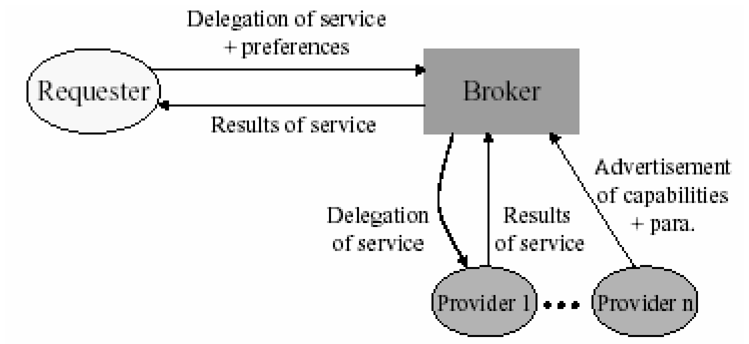


Figure 4. Block diagram for the facilitator's (broker's) simplified interaction protocol.

2.3. Privacy-Based Taxonomy

Besides capability-based coordination, the connection problem can be solved taking into account the privacy policy. To this regard, the space of solutions to the connection problems is depicted in [Decker et al., 1997]. Within that framework, *capabilities* are again defined as (meta) knowledge about what types of requests can be serviced by a provider, whereas *preferences* are (meta) knowledge about what types of information have utility for a requester. Accordingly, a specific *request* is an instance of agent's preferences, while a specific *reply* or action in service of a request is an instance of an agent's capabilities. An

advertisement is a capability specification such that the agent creating the advertisement is committed to servicing any request that satisfies the advertisement's constraints. Symmetrically, a *want-ad* is a preference specification by a requester who is committed to accepting any reply that meets the constraints in the preference specifications.
We notice that an evident dialectic looms for the service transaction at different level of specification. Above all stands the duality between preferences and capabilities. Subsequently, it emerges the contraposition between request and reply, more plainly represented by the necessary matching between want-ad and advertisement.
From the privacy point of view, preference information can be initially be kept private at the requester (provider), be revealed to some middle-agents, or be known by the provider itself. The same three possibilities exist for capability information. The overall combination of these states leads to nine general middle-agent roles in information-gathering organizations. In complete analogy with the capability-based analysis, here again our attention is on the middle-agents called *matchmaker* (*yellow pages*) and *broker*. The former stores capability advertisements that can then be queried by requesters; the requesters choose and contact any provider they wish directly. The latter understands both the preferences and capabilities, and routes both requests and replies appropriately.
The table 2 in clarifies the comprehensive taxonomy of middle-agents w.r.t. the policy for initial privacy rights.

	Capabilities initially known by		
Preferences initially known by	provider	provider, middle-agent	provider, middle-agent, requester
requester	-	*front-agent*	*matchmaker (yellow pages)*
requester, middle-agent	*anonymizer*	*broker*	*recommender*
requester, middle-agent, provider	*blackboard*	*introducer (bodyguard)*	*arbitrator*

Table 2. Privacy-based taxonomy for middle-agents.

Despite of the quoted classifications, there is still a kind of tricky disagreement among agent researchers all over the world, specially concerning with the meaning of the term *broker*. Most of the researchers adopt the term *broker* for what we named so far as *middle-agent*. We will hold the given semantics for the term *broker*, though being aware that a *broker* may alternatively be referred to as *facilitator* or *mediator*.
We will now step further along our discussion with a detailed representation of the interactions taking place in matchmaker-based systems and brokered systems.

2.4. Interaction Protocols

We will take advantage of UML (refer to [Stevens and Pooley, 2000]) sequence diagrams to plainly show the interaction protocols. For the sake of simplicity, though without a loss of precision, we will hereafter use the term *request* rather than *want-ad*. A request for a service will be compiled (generated) to communicate to a Web service some need of information expressed by a *query*. So, in a sequential workflow, it is assumed that::

1. a query for some information is instantiated;
2. the corresponding request for a service to retrieve the needed information is compiled;
3. a matching engine will look for an advertisement of a service able to satisfy the compiled request.

Futhermore, we assume the initial query as a single-shot query, herein meaning

that the required service can be successfully accomplished with a single-step transaction. From the sequential workflow perspective, that implies the service provider to reply just once to a service request.
Otherwise, removing the single-shot query hypothesis might imply a multi-step interaction between requester and provider for the service exchange to be eventually fulfilled.

2.4.1. Matchmaker

As it is illustrated by the sequence diagram in figure 5, the requester-matchmaker-provider interaction protocol can be devised as follows:

1. During the design phase, the requester stored the matchmaker's advertisement. Thereafter, the requester knows the matchmaking services provided by the matchmaker and its location.
2. When provider enters the system, it advertises its service capabilities to the matchmaker. The matchmaker consequently stores the exposed capabilities.
3. As the requester needs some information to solve a problem, it instantiates (compiles) a query to get the required information.
4. The corresponding request for a service is then compiled by the requester.
5. That request is sent to the matchmaker. So, the matchmaker seeks for an adequate service advertisement for the received request. An advertisement successfully matching the service request is then sent back to the requester.
6. The requester parses the found advertisement to determine whether it needs some extra-parameters in addition to those specified by the initial query.
7. A new query is then instantiated, thereby making sure that all the parameters to perform the advertised service are provided.
8. The compiled query must be converted into a new service request to be sent straight to the selected service provider.
9. Upon receiving the service request, the provider elaborates a reply , that way executing the required service. The computed reply is forwarded back to the requester.
10. The requester compiles the received reply into an interpretatble format.

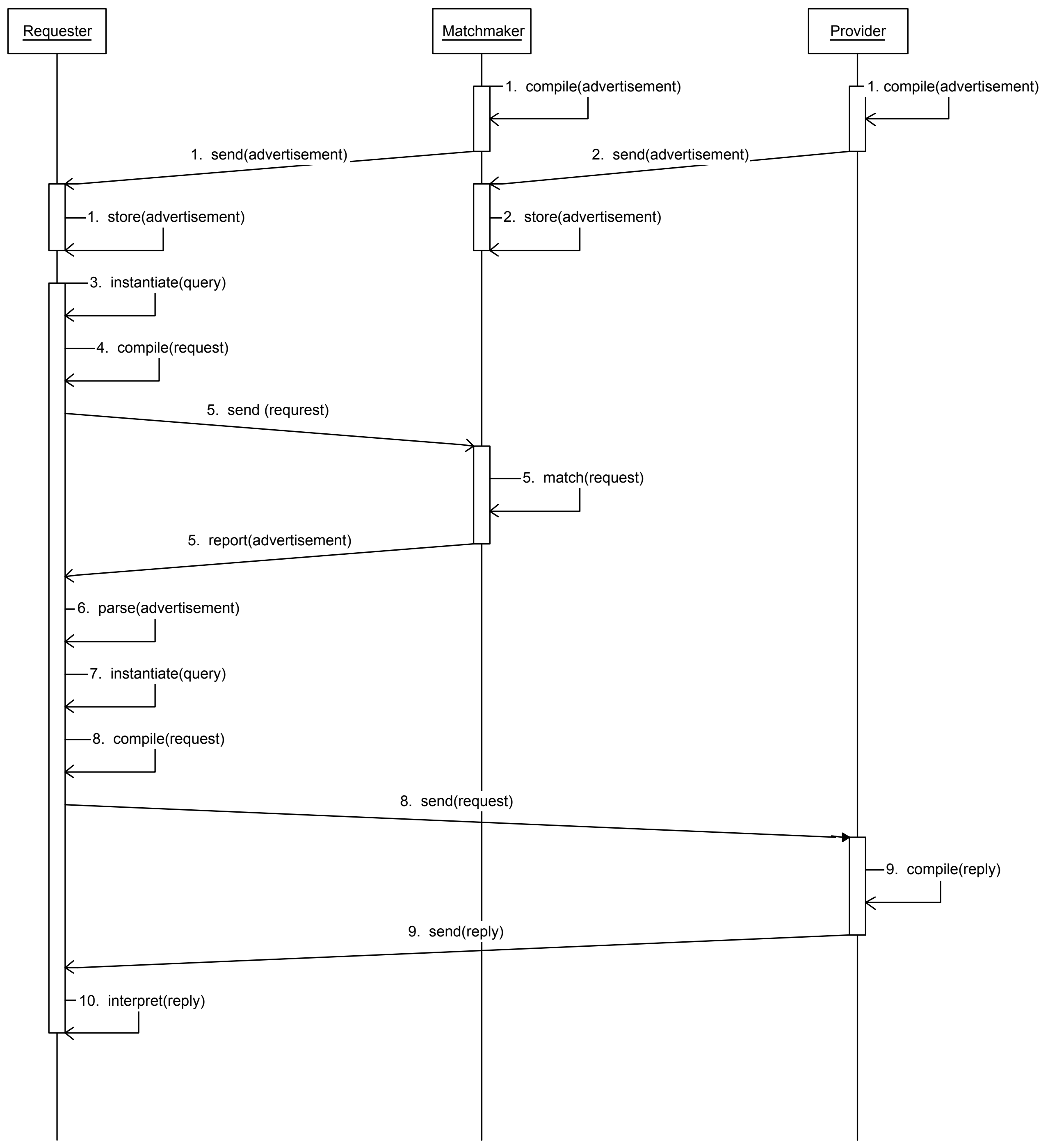


Figure 5. Sequence diagram for the matchmaker interaction protocol.

2.4.2. Broker

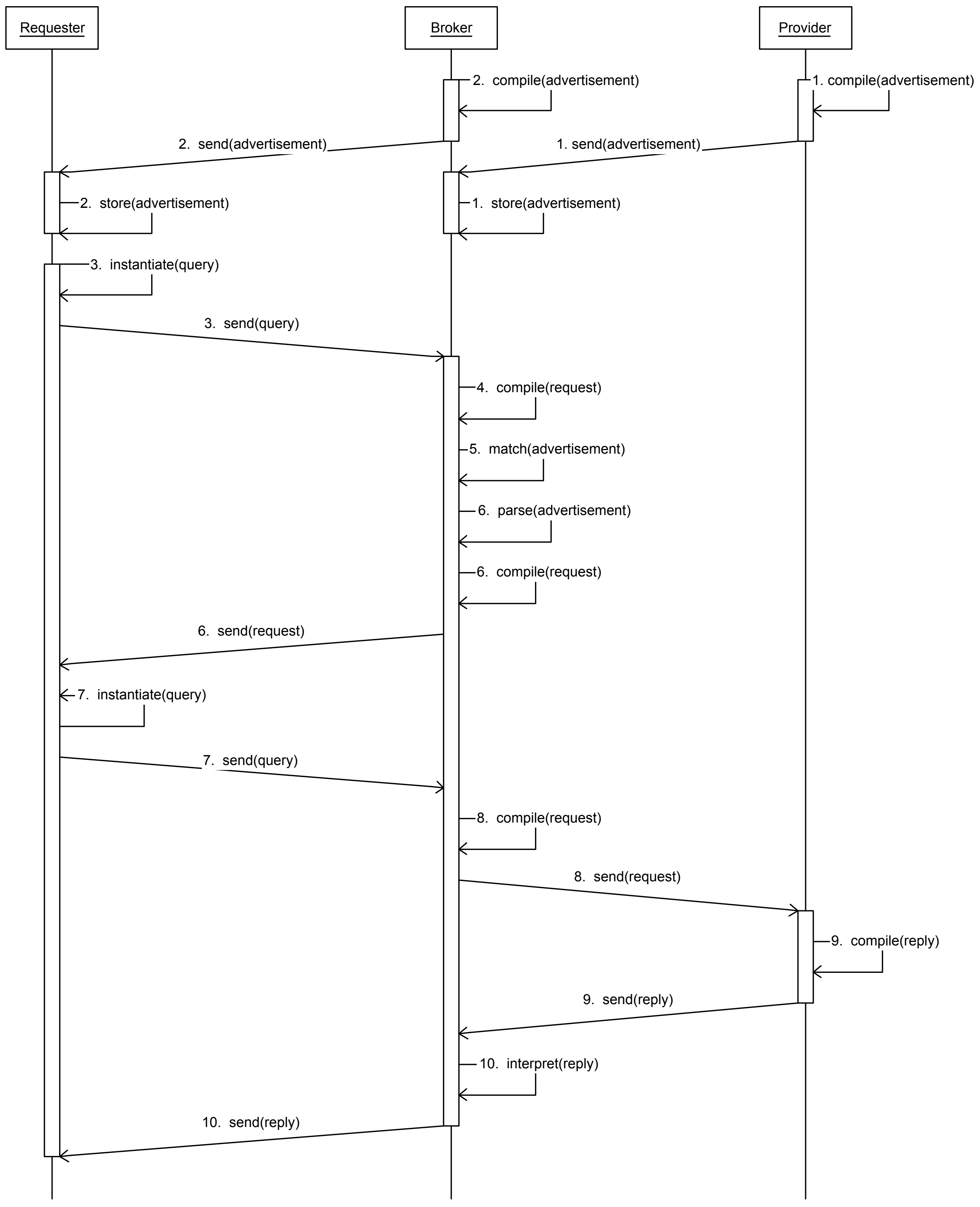


Figure 6. Sequence diagram for the broker interaction protocol.

After the sequence diagram in figure 6, we can equally present the requester-broker-provider interaction protocol through the following list of activities:

1. During the design phase, the requester stored the broker's advertisement. Thereafter, the requester knows the brokering services provided by the broker and its location.
2. When the provider enters the system, it advertises its service capabilities to the broker. The broker consequently stores the exposed capabilities.
3. As the requester needs some information to solve a problem, it instantiates (compiles) a query to get the required information. That query is addressed straight to the broker.
4. Upon receiving the requester's query, a corresponding request for a service is compiled by the broker.
5. Then, the broker seeks for an adequate service advertisement for the generated request.
6. An advertisement successfully matching the service request is parsed to determine whether it needs some extra-parameters in addition to those specified by the initial requester's query. A proper request for those potential extra-parameters is then compiled by the broker and forwarded to the requester.
7. So, a new query is instantiated by the requester, thereby making sure that all the parameters to perform the advertised service are supplied. The just created query is then sent to the broker.
8. The broker must again convert the received query into a new service request to be sent directly to the selected service provider.
9. Upon receiving the service request, the provider elaborates a reply, thereby executing the required service. The computed reply is forwarded back to the broker.
10. Once the reply has reached the broker, it is compiled into an interpretable format for the requester. Finally, the broker can return an intelligible service reply to the requester.

2.4.3. Observations

The study of the reported abstract models underlies three basic observations:

- The same elementary operations were used to describe both the models. Therefore, matchmaker and broker paradigms induce an equivalent software complexity. The remarkable difference between the presented models lies on the particular distribution of the elementary operation, as it emerges from a comparison of the sketched diagrams.
- Step 8 and step 10 for both the sequence diagram extended explanations prescribe translation algorithms. Respectively, at step 8 we may want to pass from a requester's query to a service request understandable for the service provider, whilst at step 10 we may want to turn the service provider's reply into a service reply comprehensible for the requester.
  Concerning with a matchmaker, every requester must then embed special-purposed algorithms for every provider wishing to perform a required service.
  By the contrary, in a brokered system, only the middle-agent must dispose of translation algorithms for every provider which advertised with the broker itself.

- As we implicitly assumed that requester and middle-agent always share a common Knowledge Base, we do not need to resort to any translation algorithm for query/request understanding between requester and middle-agent.

2.5. Efficiency, Robustness, and Adaptivity for Brokered and Matchmaker-Based Organizations

After the argumentation above, in [Decker et al., 1997] some conclusions were drawn about efficiency, robustness, and adaptivity of the main category of middle-agents, namely broker and matchmaker.
We recall that efficiency measures the total elapsed time taken by a requester to satisfy a service objective. Robustness is intended as the basic vulnerability of an information-gathering organization (in particular, a middle-agent-dependant organization), i.e. the sum of the expected costs of each possible failure times the probability of that failure. At last, adaptivity is the ability of an organization to quickly adapt to new preferences or capabilities measured as a function of the distance that the information has to travel, and the costs of keeping that information up-to-date.
Concerning with these parameters, matchmaker-based organizations -with proper caching mechanisms- are highly resistant to total failure. However not simple optimal load balancing is possible: using a matchmaker has slightly higher overhead due to the extra queries. Matchmaker-based organizations indeed offer preference privacy to requesters, and each requester keeps total control over its own decisions, so that adapting to changing preferences is immediate.
On the opposite side, brokered organizations can handle the dynamic entry and exit of agents, and provide easy manipulation of load balancing. Nevertheless, they suffer from the need of agents to have static knowledge of the broker (or the brokers, if the information-gathering organization is interleft by a network of brokers). In addition, a broker is a communication bottleneck, since all requests and replies need to go through it. Worst of all, a brokers form a single point of failure that cannot be mitigated even via local caching.

III SEWASIE Brokering Agent

1. SEWASIE High Level Architecture

The goal of the SEWASIE (i.e., SEmantic Webs and AgentS in Integrated Economies) project is the design and the implementation of an advanced search engine allowing personalized information access via a machine-processable semantics of data in order to provide the basis of structured web-based communication.
Research and development of SEWASIE system (for deeper details refer to [SEWASIE]) are pursued by a European consortium. That consortium comprises the University of Modena (Italy), RWTH of Aachen (Germany), the University of Roma "La Sapienza" (Italy), the "Libera Università" of Bolzano (Italy), IBM Italia S.p.A. (Italy), CNA Servizi Modena s.c.a.r.l. (Italy), Thinking Networks AG (Germany), FGFFV (Germany). The project is coordinated by Professor Sonia Bergamaschi, University of Modena and Reggio Emilia.

Tools and methods have been developed to create/maintain multilingual ontologies, with an inference layer grounded in W3C standards (XML, XMLS, RDF, RDFS, OWL), that are the basis for the advanced search mechanisms and that provide the terminology for the structured communication exchanges.

From an architectural point of view, SEWASIE aims to provide an open and distributed architecture based on intelligent agents (brokers, mediators, and wrappers) facing scalability and flexibility issues while offering one central point of access to the user (the high level architecture of the system is reported in figure 7).

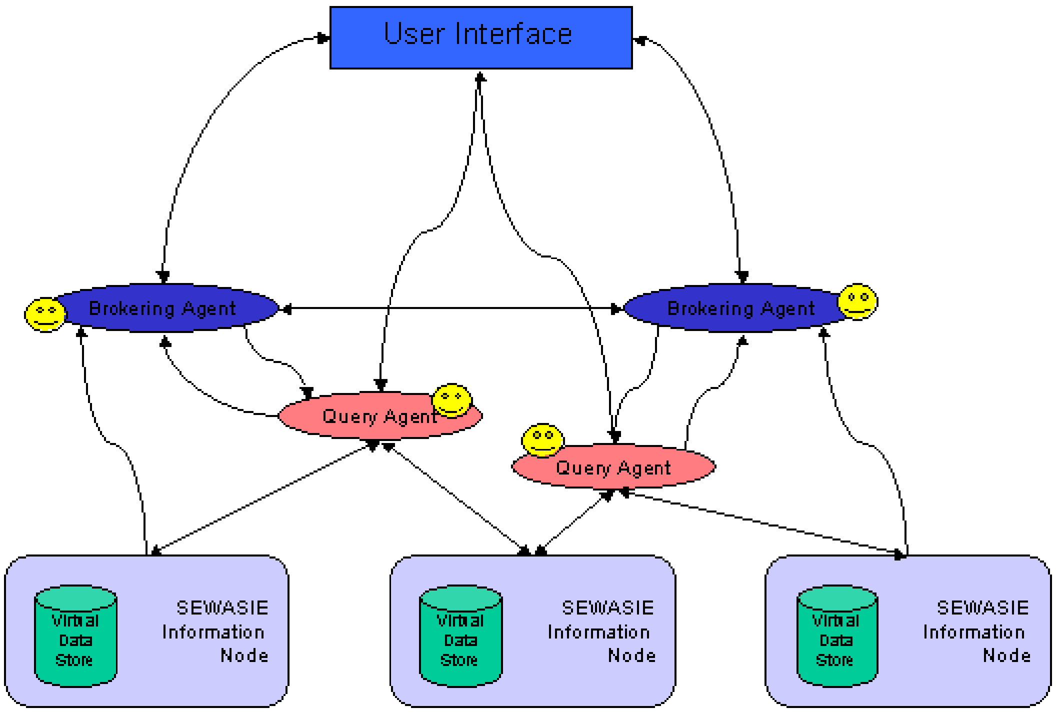


Figure 7. SEWASIE distributed architecture. High Level perspective.

Within the perspective of this paper, the illustration above shows the actors on SEWASIE's stage we are mainly interested in. They are:

- the User Interface (UI);
- the Query Agent (QA);
- the Brokering Agent (BA);
- the SEWASIE Information Node (SINode).

The UI is the group of modules which work together to offer an integrated user interaction with the SEWASIE system.
The SINodes group together the modules, which work to define and maintain a single administrative, or logical node of information presented to the network.
The QAs are the carriers of the user query from the UI to the SINodes, and have the task of solving a query, interacting with the BAs. Starting from a specified SINode, they may access other SINodes, collect partial answers and integrate them.
The BAs are responsible for maintaining a view of the knowledge handled by the SEWASIE network, as well as the information on the specific content of some SINodes which are under direct control (of each BA).
SEWASIE system in fact realizes the dictates of a Peer to Peer (P2P) architecture. A couple of alternative P2P networks can actually be identified within the SEWASIE system:

- BAs network:
  each BA communicates with other peers in order to have information about out-contolled SINodes;
- Inter SINodes network:
  a SINode provides to other SInode the knowledge about the involved Information Sources; it is possible to specify coordination formulas explaining how the data in

one peer must related data in a an acquaintance.

The basic (generic) user query scenario figured by SEWASIE designers concerns a user at a workstation (or handling a handheld computer, or a cellular phone with network connection capabilities), looking for information on a topic (possibly within the context of a broader-scoped task). The user may then issue a request expressed in some “natural” language style to the network. The UI translates the user request into a query, keeping into account the past history and present context of the user, and sends a probe out (the QA) scouting for answers.
The QA connects into the network BAs and queries them for info on the matter of interest. A typical interaction between a QA and a BA may imply that the BA will provide directions to relevant SINodes and information on SINode contents, or reference the QA to other BAs. The QA will then move to such SINodes and query them, or may move on to the other BAs to ask them for directions again.
When the QA eventually receives the SINode answers, it has to integrate them (data reconciliation) in a way that is meaningful to the entity (the user) which issued the original request to the system.
There is another family of scenarios of interest, which involve the creation of a new SINode, and the updating or the cancellation of an existing SINode.

2. SEWASIE Detailed Architecture

As we already hinted, the SEWASIE system aims to realise a virtual network, the SEWASIE Virtual Network (SVN), whose nodes are the SINodes. Figure 8 gives an overview of the SVN.

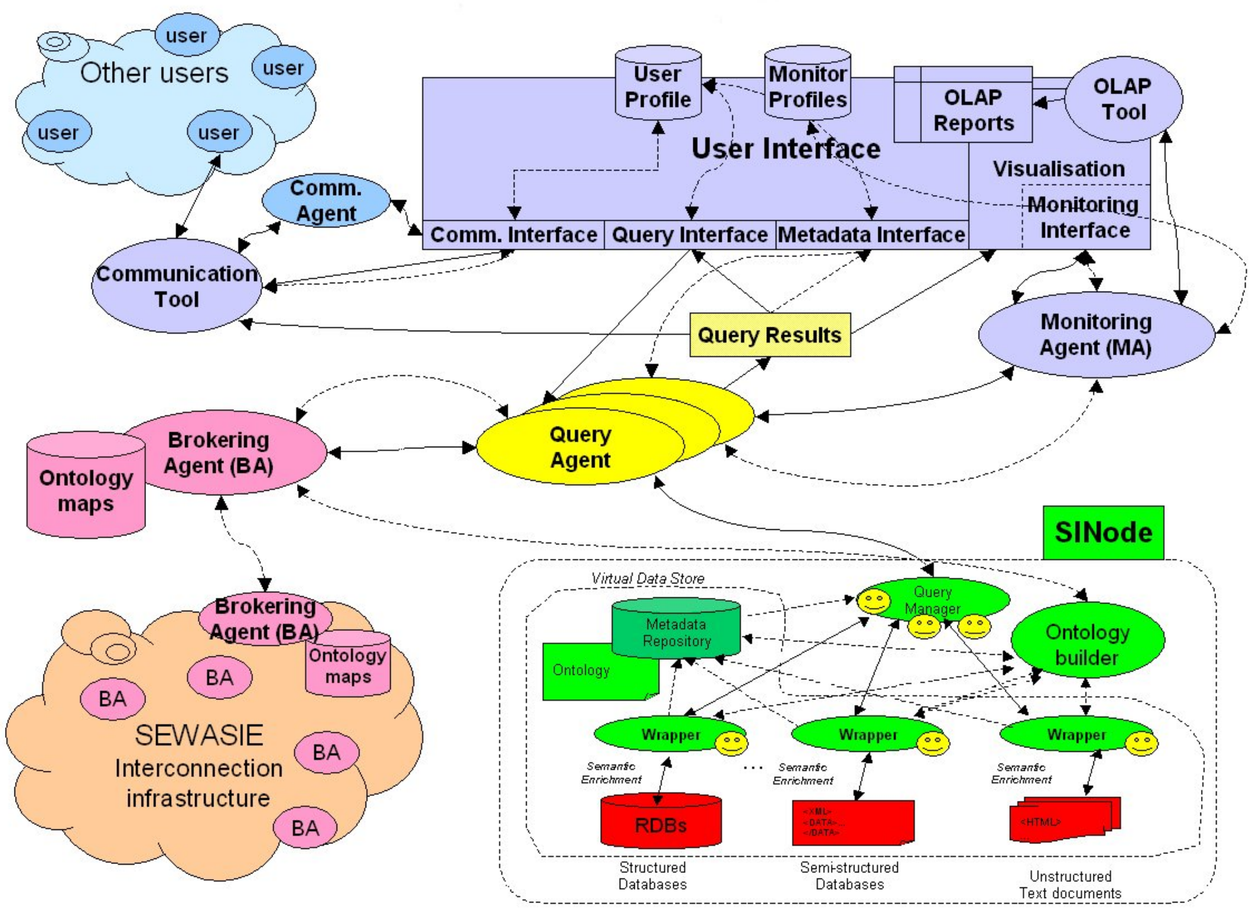


Figure 8. SEWASIE distributed architecture. Detailed perspective of the SVN.

Since we are interested in the SEWASIE's strategy for query answering, we will immediately move our emphasis on the actors which receive the query produced by the UI. They are: the QA, the BA, and the SINode.

2.1. SINode

SINodes (see [SEWASIE D2.1]) are mediator-based systems, each including a Virtual Data Store (VDS), an Ontology Builder (OB), and a Query Manager (QM).

A VDS represents a virtual view of the overall information managed within any SINode and consists of the managed information sources, wrappers, and a metadata repository.
The managed Information Sources (ISs) are heterogeneous collections of structured, semi-structured, or unstructured data, e.g. relational databases, XML/HTML or text documents. A wrapper implements common communication protocols and translates to and from local access languages. There is one wrapper linked to each IS.

According to the metadata provided by the wrappers, the OB performs semantic enrichment processes in order to create and maintain the current ontology which is made up of the Global Virtual integrated View (in short GVV) of the managed sources and the mapping description between the GVV itself and the integrated sources. Ontologies are built on a logical layer based on existing W3C standard.

The Metadata Repository (MR) holds the ontology and the knowledge required to establish semantic inter-relationships between the SINode itself and the neighbouring ones.
The chosen ontology data model is ODM_{I3}. The associated description language for integration purposes and ontology exposition is ODL_{I3} (check [Bergamaschi et al., 2001] for more details), while the query language is known as OQL_{I3} .
Notice that the adoption of specific languages for intra-node communication does not prevent to put at the SEWASIE network disposal the information managed by SINode in other formats. In particular, investigation is in progress on how to import/export such information expressed in ODL_{I3} from/to W3C standards such as XML, XMLS, RDF, RDFS, and OWL (see [XML Reference, 2000], [XMLS Reference 1, 2001], [XMLS Reference 2, 2001], [RDF Reference, 2003], [RDFS Reference, 2003], and [OWL Reference, 2003], respectively).

At last, the QM is the coordinated set of functions which
1. take an incoming query,
2. define a decomposition of the query w.r.t. the mapping of the GVV of the SINode onto the specific data sources available and relevant for the query,
3. send the queries to the wrappers in charge of the data sources,
4. collect their answers,
5. perform any residual filtering as necessary, and finally
6. deliver whatever is left to the requesting query agent.

The queries dealt by the QM must be written in OQL (see [ODS, 1997]) with some restrictions and extensions resulting in the just mentioned language OQL_{I3}.

2.2. Query Agent

As we said earlier, the QA is the actual carrier of a query from the UI to the system. The term “query” must be intended as a general statement in OQL_{I3} which may be interpreted by the QM within each SINode. Beyond the proper request to be satisfied (see section 3 of this chapter), this query also includes information on the context of the user at the time of the establishment of the query.
The QA is a sort of network query manager and "motion item" of the system, and it should be the only carrier of information among the UI and the remaining actors of the system. Therefore it should be able to do several jobs:

- carrying a query (of course) plus the relevant pieces of the user ontology/profile which may help the BAs qualify the semantics of the query;
- defining the query plan, doing the query rewriting for a specific SINode, and merging the results from several SINodes;
- processing the information given by the BA and identifying the SINodes to be accessed for answering the query, and on which further BA to contact to possible get more answer to the query;
- carrying back the results (both data and metadata);
- moving from location (SINodes or BAs) to location (SINodes or BAs) avoiding repeated visits and deciding when the search should stop.

During this process, the QA is informed by the BAs about which SINodes contain relevant data, so that the QA may access such SINodes, collect partial answers and integrate

them. When the QA is informed that a certain SINode may have relevant answers to the query, it has to pose the right query to such SINode.
With regard to the query plan, we remind that the QA should be able to define it with no global schema defined at any single point in the system, and it should do so based on the knowledge of the BAs.
Moreover, we notice that QAs can be instantiated not only by the users for each request to the system, but also by the system itself.

2.3. Brokering Agent

A metadata brokering agent (so far just "brokering agent", BA) is a "semantic broker" of the SEWASIE system, being responsible for maintaining the metadata about the SEWASIE network.
Referring to the global architecture of SEWASIE, the main task of the BA is to provide metadata about the SINodes to the QA. Furthermore, a BA may be connected to other brokering agents, thereby forming the P2P network we earlier mentioned.
The knowledge within a BA is represented as an ontology. In fact, a BA

- knows about the ontologies which are present in the underlying SINodes,
- has some information about related (to its own) ontologies in other nodes, and
- has generic information about other ontologies.

The depth of the information of the BA becomes more and more shallow with the distance (with respect to some metrics) between the ontologies for which it is "expert" (those of its underlying SINodes) and other ontologies covered within the SEWASIE system. Its information on other (non local) ontologies is incomplete.

According to the source of the knowledge stored by the BA's ontology, and distinguishing the location of the BA within the network, we can identify two classes of BAs:

- *local service* BAs are serving a single SINode or a group of SINodes and are positioned close to the SINodes;
- *pure informant* BAs are collectors of references to several BAs and may specialise in a certain domain, and are possibly positioned anywhere in the network

At an abstract level, the operational behaviour of the BA can be divided into two phases:

1. *Design* phase (at mapping time)
    - The BA receives a local ontology from a SINode and has to map it into its own ontology.
    - The BA receives information about other ontologies from other BAs. This information has also to be mapped into the existing ontology of the BA.
2. *Runtime* phase (at query time)
    - A QA wants to know where it can find the information for a query. The terms of the query have to be matched with the concepts known by the BA. The BA looks up its internal meta information and returns pointers to the SINodes or other BAs which have the requested information.
    - A component of the SEWASIE system extracts the complete ontology managed by the BA.

The design and runtime phases are not strictly separated. Ontologies will be mapped and updated during the whole life cycle of a BA.
At design time, the map of semantic relationships among concepts from SINodes and other BAs relies on terminological relationships. They are created by the BA which, in a

semi-automatic way, analyses the meaning of the concepts in different ontologies and tries to discover terminological relationships among them by exploiting lexicon ontologies such as WordNet. Once the repository has been created, the brokering agent is in charge of its maintenance: changes in the SEWASIE network have to be integrated to make the repository consistent with the new scenario.
At runtime, when a request from a QA is received, the quality of the answer of the BA depends to a large degree on the quality of the semantic relationships that have been created. Incorrect relationships will lead to incorrect to the QA. In similar fashion, incomplete (or missing) relationships will lead to incomplete results.
Automatic creation of the semantic relationships is possible, but the creation of relationships based only on the lexical similarity between terms will lead to incorrect and incomplete results. On the other hand, manual control of the creation is not possible in every case as a BA might handle a large number of SINodes. Thus, updates to the semantic relationships might happen quite frequently and manual control would significantly slow down the process of establishing semantic relationships.

Summing up, we identified two main tasks of a BA:
- creation and maintenance of the semantic relationships, and
- answering requests made by a component of the SEWASIE system, especially the query agent.

These tasks will be fulfilled by two different modules: the *map keeper* and the *playmake*r. Furthermore, these modules will make use of two support modules. The *listener* module will provide an interface to external components and implement the basic communication infrastructure for the brokering agent. The *librarian* module maintains the repository of the brokering agent. The repository will store information about the ontologies, e.g. concepts, mappings, access information about SINodes. It will process the updates generated by the map keeper and answer the requests of the playmaker. Figure 9 contextualizes these modules in the top level architecture of the BA.

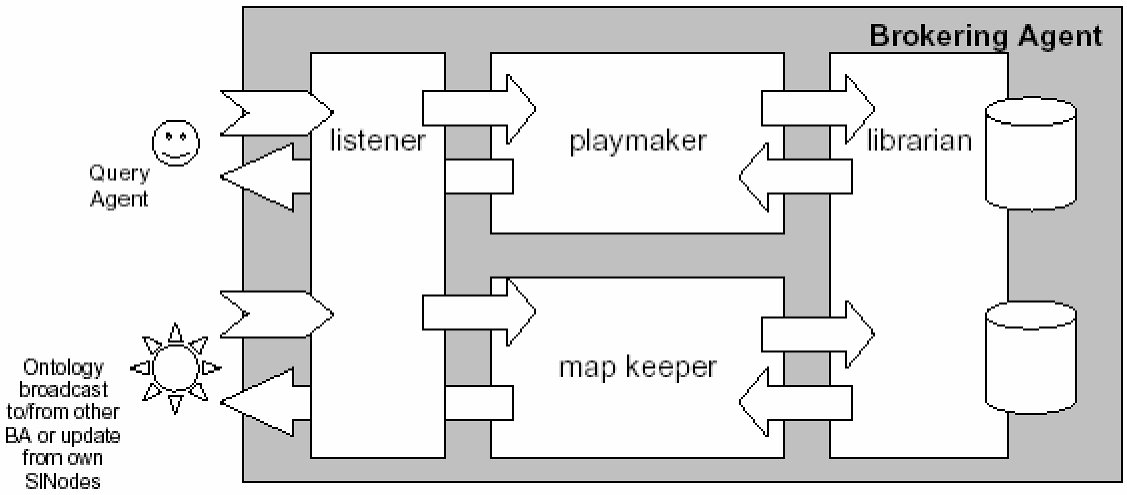

Brokering Agent
listener
playmaker
librarian
map keeper
Query Agent
Ontology broadcast to/from other BA or update from own SINodes

Figure 9. Brokering Agent high level architecture.

3. Query Management

3.1. Query Formalization

In a first place, it urges to remark that the QAs are capable of answering to positive queries which terms come from the ontology exposed to the user interface. Those queries are further restricted to the class of connected acyclic conjunctive queries. Moreover, only unary (classes) and binary (relationships) terms are admitted.
In this framework, a conjunctive query is an expression

$$\{x_1, ..., x_k \text{ s.t. } T_1(y_1), ..., T_n(y_n), R_1(z_1, w_1), ..., R_l(z_l, w_l)\}$$

where the letters where the letters x, y, z, w denote variables or basic data constants (numbers, strings, etc.), T and R unary and binary terms respectively. Variables x_1, ..., x_k are called *distinguishe*d, and represent the values which are going to be returned by the query. Remaining variables connect (join) the terms in the body of the query (i.e. $T_1(y_1), ..., T_n(y_n), R_1(z_1, w_1), ..., R_l(z_l, w_l)$), and are considered as existential quantified. Terms are conjoined (i.e. they all have to be satisfied), and the order in which they appear does not matter.
The body of the query can be considered as a graph in which variables (and constants) are nodes, and binary terms are edges. A query is connected (or acyclic) when for the corresponding graph the same property holds.
The result of a query is the set of k-tuples whose values, substituted to the distinguished variables, make the body of the query satisfied in the integrated view of the SINodes.
For example, a query to retrieve the suppliers selling on the Italian market would be

$$\{x \text{ s.t. } Supplier(x), selling\_on(x, y), Italian\_market(y)\}.$$

3.2. Query Management Formalization

Now, it can be theoretically described query management in the SEWASIE framework (see [SEWASIE D3.2]). From all the above observations about SEWASIE actors, we may argue that it involves different tasks, summarized as follows:

- Given a query Q expressed in terms of an ontology understood by a Brokering Agent B,
    1. Single out the SINodes S_1, ..., S_n managed by B that are relevant for computing the answer to Q, and reformulate Q in terms of n queries Q_1, ..., Q_n, to be posed to the SINodes S_1, ...,S_n, respectively.
    2. Single out the brokering agents B_1, ...,B_m (besides B) that may have links to SINodes containing relevant information for computing the answer to Q, and reformulate Q in terms of m queries T_1, ..., T_m, to be posed to the brokering agents B_1, ...,B_m, respectively.
    3. Reconstruct the answer to Q on the basis of the answers to Q_1, ..., Q_m, T_1, ..., T_m.

  Note that step (2) above is recursive, in the sense that answering a query T_i posed to the brokering agent B_i is done through the very same process we are describing. This means that the overall strategy for query management must deal with the problem of how to stop recursion. Note also that both step (1) and step (2) are carried out by the QA on the basis of the meta-data (mapping) exported by the brokering agent B.
- Given a query posed in terms of the GVV associated to an SINode, retrieve the answer to the query. This task is carried out by the QM of the SINnode of interest,

and is characterized as follows.

1. The first goal of this task is to derive a query plan that is able to correctly access the data sources under the control of that SINode.
2. The second goal of this task is to execute the query, thus computing the corresponding answer.

3.3. Brokering Agent and Query Management

We now restrict our attention on the sub-task of query management that is also the task most frequently executed by a BA. Namely, for a query given by a QA the BA must find the SINodes which have to be accessed to answer the query.
In the following, we will describe the complete process, starting at the query definition in the query interface until the query agent receives the result from the BA and generates a query plan. This process is graphically shown as mixed activity/use case diagram in figure 10.

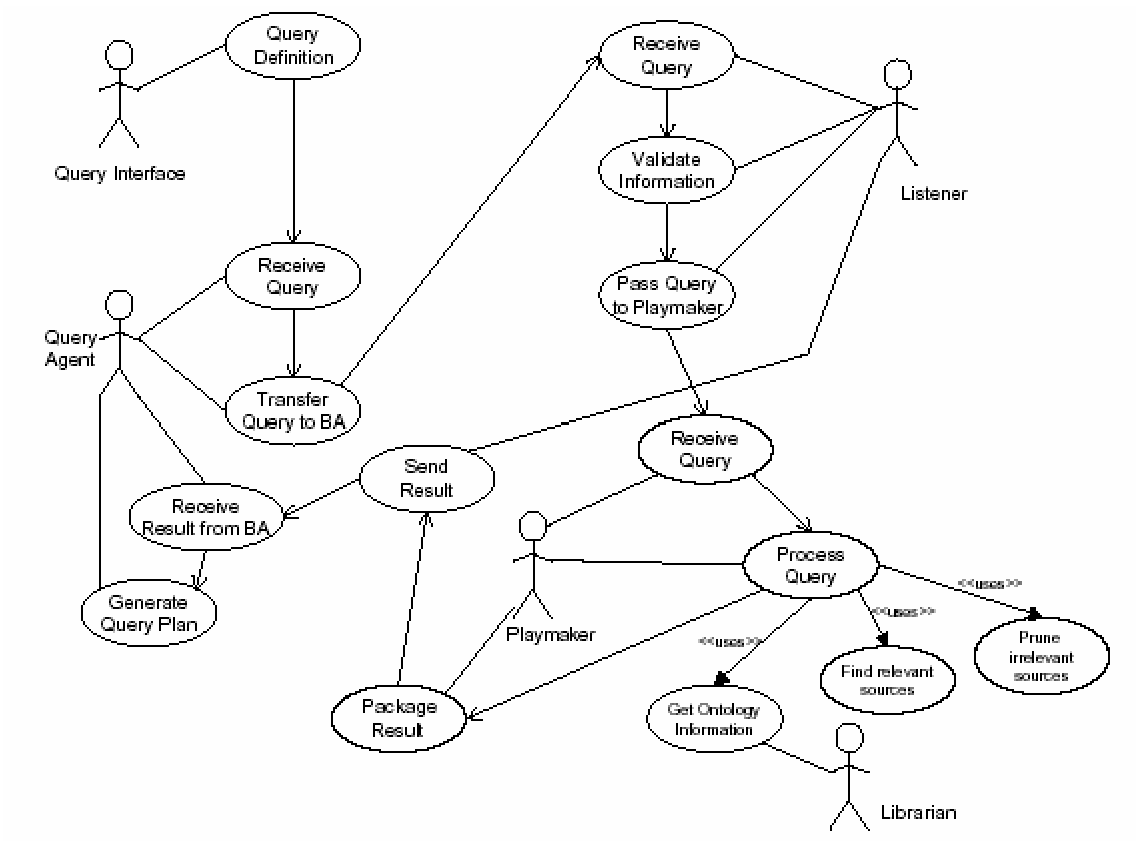


Figure 10. Hybryd activity/use case diagram for query management by QA and BA.

The represented activities can be enumerated as follows:

1. Query definition in the query interface
   The user enters a query in the query interface in natural language. the query interface will generate from the user input a query expressed in a formal language using concepts defined in the user ontology or the ontology of the BA. For this task, it is necessary to match the keywords entered by the user with elements of an ontology, i.e. the query interface resolves the ambiguity of keywords entered by the user. The BA's ontology was priorly exported. The query interface will package the

formal query expression and context information, which might be necessary to answer the query, into a standard format (at the transport layer level) and send it to a QA.

2. Reception of the query by the QA
   Basically, the QA will receive the query and forward it directly to a BA to get information about the SINodes. However, the QA might do some pre-processing of the query.
3. Reception of the query by the BA
   The listener module of a BA will receive the query, verify the user information, and forward the query to the playmaker of the same BA.
4. Processing of the query by the BA
   After the query has been received by the listener, the playmaker in the same BA will first analyse the query and build a list of concepts which are used by the query. Then, for each concept it will search for the concepts which are relevant for the query. For example, if C is a concept of the query and D is a concept of an IS integrated within a controlled SINode, it must check whether $C \subseteq D$ or $C \cap D \neq \emptyset$ holds. If this is the case, the SINode which provides information about D is relevant to answer the query. In addition to the information about SINodes, the playmaker will also check whether elements of ontologies provided by other BAs are relevant to answer the query. If this is the case, a pointer to the BA will also be included in the answer.
5. Prune sources which are not relevant
   The initial step, that is to find the relevant sources, will result in a complete list of sources. However, some sources might not be relevant for the query if we take the exact constraints and the definition of the sources into account. For example, if we are looking for companies that have been founded before 1995, then a source providing information about companies founded after 1998 is not relevant.
6. Packaging the result
   The playmaker will extend the document in the standard format received as input by attaching information about the SINodes (address, supported protocol) and the mappings of the source concepts to the original concepts. This information will be directly attached to a concept of the query, as different concepts in the query might be provided by different sources. In addition, information about other BAs having relevant information will also be attached to the document. The extended formatted document is sent back to the listener, and the listener sends the document straight to the waiting QA.
7. Query plan generation by the QA
   Based on the information received from the BA, the QA will generate a query plan. The QA might have to contact first other BAs which also have information for the query. Furthermore, a cost estimation is done to estimate the response time of the query. If this is too high, the user is asked (by the query interface) to specify more constraints in his/her query. If the cost estimation is feasible, the QA will execute the query.

As a quality consideration notice that, since the playmaker searches for source concepts which have a non-empty intersection with concepts of the query, i.e. $C \cap D \neq \emptyset$, in general, this might lead to incorrect results. In fact, the source concept D might contain more information than requested by the query. For example, if we are looking for an Italian textile company, a source containing information about small and medium sized companies in Europe is also relevant as Italian textile companies might be included in this source, but it contains also many other companies which are not relevant. On the other hand, if we would ignore this source, several companies would be possibly not included in the result. The query agent might be able to add an additional constraint to the query for this source (e.g. SELECT * FROM Companies WHERE country='Italy' and branch='Textile') if this is supported by the source. Thus, from the viewpoint of the BA, it will return all sources which might be relevant. The query agent has to decide how and if a source is queried.

3.4. Open Issues for Brokering Agent Implementation

To successfully perform query management, several types of information must be available by the BA:

- the ontologies of the SINodes with their interleaving mappings (including implicit and explicit relationships);
- ontologies (or parts of them) provided by other BAs;
- the query as formal query expression using terms which are known to the brokering agent, or using terms which are included in the context information of the query (e.g. mappings of the user-defined concepts to concepts known by the brokering agent)
- protocols prescribing the interaction rules with QAs, BAs, and SINodes.

Essentially, those necessities can be reduced to the disposal of:

- an ODL_{I3} ontology by the BA serving as:
    - knowledge base to interprete OQL_{I3} queries posed by the QA to the BA;
    - knowledge base gathering some relevant information from the SINodes underlying the considered BA;
- a matching operator for the incoming queries against the BA ontology;
- a mapping function connecting the BA ontology with the ontologies of the underlying SINodes;
- a transport and communication layer to let agents communicate each other.

Keeping a structural similarity with the global ontology exposed by an SINode, members of the SEWASIE project devised the following components for the BA ontology:

- annotated Global Virtual Views (GVVs) of the underlying SINodes, such that they include the global classes of each SINode GVV, their taxonomic hierarchy, and their attributes with some lexical annotations;
- remote BA GVV references, which are limited to an identificative for the reference to the remote BA, and to some root classes for the taxonomy of the remote BA GVV;
- mappings (i.e. semantic relationships) among classes and attributes from either of the above GVVs.

Concerning with the semantically intensive functions committed to the *mappings* component of the BA ontology, solutions are under investigation by SEWASIE designers. In the sequel, we will rather propose a contribute for the implementation of both the query matching functionalities and the interaction protocol between QAs, BAs, and SINodes.

IV DAML-S Broker

1. Autonomous Semantic Web Services

1.1. Overview

Nowadays the Web is a collection of human-readable pages, though virtually unintelligible to computer programs. Two parallel efforts have emerged in recent years to overcame this difficulties: the Semantic Web (see [Berners-Lee et al., 2001]) and the Web services (refer to [Web Services Architecture, 2003]).
As Semantic Web is providing tools for explicit markup of Web content, it will help create a repository of computer-readable data.
Along the same path, a Web service is defined as a software system designed to support interoperable machine-to-machine interaction over a network.
Agent technologies (see chapter 2) are then tightly connected to Web services development, since a Web service is viewed as an abstract notion that must be implemented by a concrete agent.
Within this framework, work at Intelligent Software Agents Laboratory (Carnegie Mellon University, Pittsburgh, PA) under the direction of Research Professor Katia Sycara is focused on the implementation of Web services as autonomous, goal-oriented agents (again, refer to chapter 2) that select other agents to interact with and that flexibly negotiate their interaction models, acting variously in client-server and peer-to-peer models. The resulting Web services, thus called *autonomous Semantic Web services*, use ontologies and semantically annotated Web pages to automate the fulfilment of tasks and transactions (see [Paolucci and Sycara, 2003]).

Since the purpose of a Web service is to provide some functionality on behalf of its owner (i.e. a person or an organization), Web services architectures can easily cast the requester-provider paradigm (see chapter 2). From Web services perspective, a *provider entity* is the person or organization that provides an appropriate agent to implement a particular service, whilst a *requester entity* is a person or organization that wishes to make use of a provider entity's Web service. Falling in the Web services implementation perspective, we can analogously talk of provider agents and requester agents. Therefore it arises the necessity for a middle-agent (refer to chapter 2) able interleave the service exchange between a provider agent and a requester agent. Reminding the discussed solution of the connection problem through capability-based coordination, the Semantic Web naturally supplies markup languages and related inference mechanisms for respectively representing and reasoning with core Web service concepts. Capability-based service discovery and service interoperation at runtime would be thus supported. As both a formal markup language and an ontology for describing Web services, DAML-S (refer to [DAML-S, 2001]) is the first attempt to effectively exploit Semantic Web for Web services implementation.
From this standpoint, the DAML-S broker designed at Intelligent Software Agent Laboratory exactly addresses the implementation of an autonomous Semantic Web service that can play the role of a broker middle-agent.
In the next sections of this chapter we will undertake a detailed analysis of the DAML-S broker development. We will start with an overview of Web services infrastructure, subsequently heading to the vision of DAML-S as the capability description language for the DAML-S broker Web service.

1.2. Web Service Technologies

Web service architecture involves many layered and interrelated technologies. Figure 11 illustrates some of these technology families.

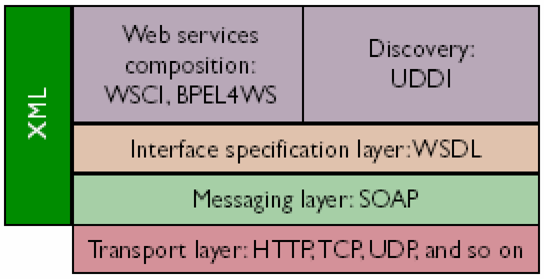

XML
Web services composition: WSCI, BPEL4WS
Discovery: UDDI
Interface specification layer: WSDL
Messaging layer: SOAP
Transport layer: HTTP, TCP, UDP, and so on

Figure 11. Simplified stack diagram of Web service technologies.

The unifying factor is XML (see [XML Reference, 2000]) which cuts across all layers.
SOAP (see [SOAP Reference 1, 2003] and [SOAP Reference 2, 2003]) defines a format for passing messages between Web services.
WSDL (see [WSDL Reference 1, 2003] and [WSDL Reference 2, 2003]) describes each service's interface - how to contact it and how to serialize the information exchanged.
WSCI (see [WSCI Reference, 2003]) and BPEL4WS (see [BPEL4WS Reference, 2003]) provide mechanisms for describing how multiple Web services can be assembled to participate in shared business processes.

UDDI (see [UDDI Reference, 2003]) provides a set of publishing, browsing, and inquiry functionalities for extracting information from a given registry. Service discovery is thus facilitated, in that UDDI registries can be looked up in order to find desired Web services.
At last, at the transport level Web services communicate through well-known protocols such as HTTP, TCP, et cetera.

1.3. DAML-S as a Capability Description Language

Conceived as an ontology, DAML-S uses OWL (see [OWL Reference, 2003]) or DAML+OIL (see [DAML+OIL Reference, 2001]) based constructs to define the concept of a Web service –recently DAML-S has been turned into OWL-S, thus becoming committed to the exclusive usage of OWL constructs.
Parallely conceived as a markup language, DAML-S supports the description of specific Web services that users or other services can discover and invoke by means of standards such as WSDL and SOAP (see previous section).
DAML-S (the reference document is [DAML-S, 2001])employs semantic annotations and ontologies to relate each Web service's description to a description of its operational domain. From the most general point of view, the DAML-S way to represent a Web service can be schematised with a top level ontology (i.e. ["Service" OWL ontology]) which includes four main classes:

- a *service description* (refer to ["Service" OWL ontology]);
- the *Service Profile* (refer to ["Profile" OWL ontology]);
- the *Service Model* (refer to ["Process" OWL ontology]);
- the *Service Grounding* (refer to ["Grounding" OWL ontology]).

Notice that each of those classes is in turn described in greater details by the specific OWL ontology referenced within the parenthesis on the right-hand side.
Keeping the standpoint of the top-level OWL ontology for services, the resulting class diagram shows the very high-level structure of the DAML-S approach to Web services description -with an overloading of the specifications declared by the ontology ["Service" OWL ontology] we have also rendered some attributes of the class ServiceProfile that will better explained later.

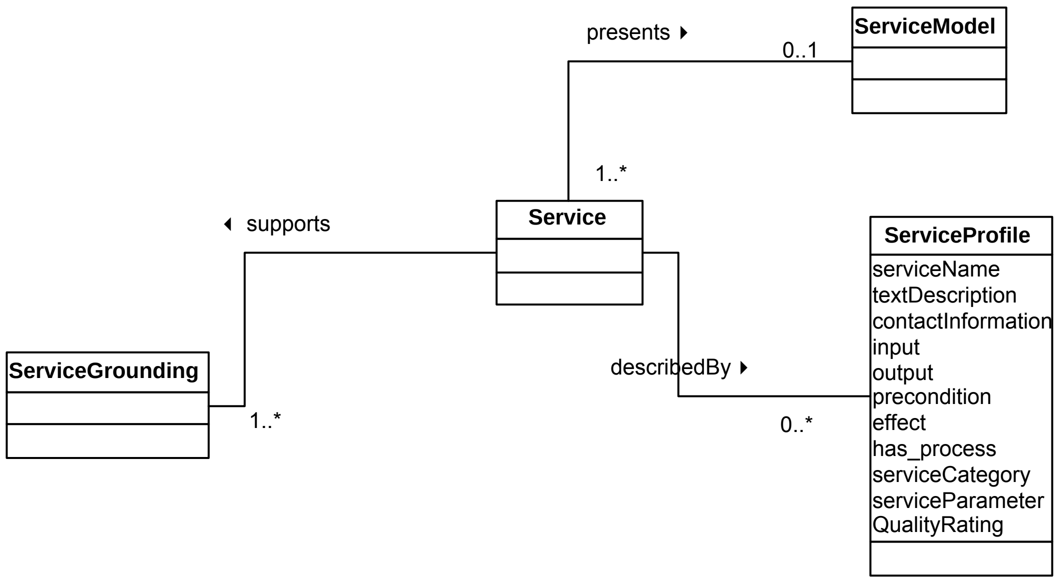


Figure 12. Simplified class diagram for the top-level structure of DAML-S descriptions.

The presented structuring is motivated by the need to provide three essential types of knowledge about a service, each characterized by the question it answers:

- What does the service require of the user(s), or other agents, and provide for them? Hence, the class Service *presents* the class ServiceProfile.
- How does the service work? Hence the class Service is *describedBy* the class ServiceModel.
- How is the service used? Hence, the class Service *supports* the class ServiceGrounding.

In the whole, the hinted structure provides semantic composition (through the ServiceModel), discovery layers (i.e. the ServiceProfile), and a grounding (achieved by the ServiceGrounding) to comprehensively map Web services descriptions to message-transport specifications. Therefore we can augment and refine the Web services infrastructure illustrated in figure 13 as follows:

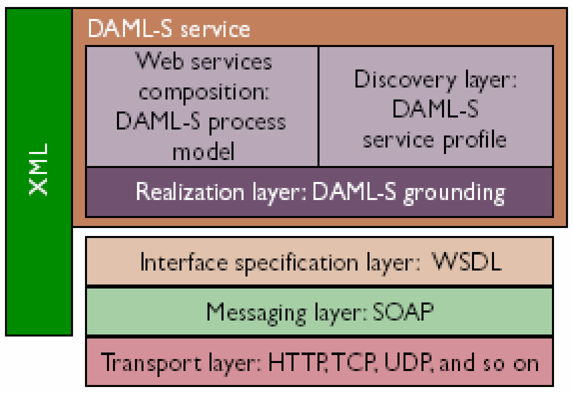

XML
DAML-S service
Web services composition: DAML-S process model
Discovery layer: DAML-S service profile
Realization layer: DAML-S grounding
Interface specification layer: WSDL
Messaging layer: SOAP
Transport layer: HTTP, TCP, UDP, and so on

Figure 13. Web services infrastructure using DAML-S.

Since we are particularly interested in Web services employment as middle-agents, we will mainly concentrate on the DAML-S ontology which best resembles the capability-based model for requester/provider interactions. Namely the Service Profile.
Nevertheless, we will provide just some outlines of the other DAML-S ontologies mentioned earlier. Namely, the Service Model and a Service Grounding.

The Service Model ontology views a Web service as a process. Three main classes characterize the considered process. They are: *ProcessModel*, *Process*, and *ControlConstruct*.
The first one keeps the unique connection with the upper level Service (["Service" OWL ontology]) ontology.
The class *Process* then details the control structure and the data flow of the service.
At last, *ControlConstruct* allows agents to monitor the execution of a service request.
DAML-S processes are defined as an organized collection of specific processes. The taxonomic criterion grounds on some control flow structures adaptable to any arbitrary workflow. A *Process* may consist of :

- Atomic process, which have no sub-processes and provide a grounding (see below) that enables a requester to compile an invocation message and interpret the execution of the relative process as a "black box", or "single step" execution.
- Composite processes, which are compositions of other processes.
- Single processes, which correspond to undecomposable abstract process having a single step execution like atomic processes, but, unlike atomic processes, with no associated grounding. Therefore, single processes are not directly invokable.

An interaction protocol can be derived from the Service Model, in that, by analysing the specified workflow of processes (see the processes classification above), information to send and information to be received are known at any given time during a service transaction. In addition, the Service Model specifies at a finer granularity what is divulged by the Service Profile, thereby making available a detailed characterization of a requested or provided service.

A Service Grounding fully describes how to access the relative service. That is accomplished through a mapping from the atomic processes, defined by the Service Model, to WSDL specifications messages.

At last, a Service Profile basically consists of three types of information:

1. a general description (attributes *serviceName*, *textDescription*, and *contactInformation*);
2. a functionality description (attributes *input*, *output*, *precondition*, and *effect*);
3. a parameter description (attributes *has_process*, *serviceCategory*, *serviceParameter*, and *qualityRating*).

Generally speaking, the Service Profile is used for advertising, registry, discovery, and matchmaking. Conjunctively, once a potential service-using agent has located a service appropriate for its need, the Service Model and a Service Grounding together give enough information for an agent to make use of the selected service.
Within a Service Profile, the functional behaviour (i.e. the functionality description) is clearly the core component. A service is basically viewed as a transformation of a set of inputs into a set of outputs. Additionally, a set of conditions, referred to as preconditions, must be satisfied in order to execute the service. Finally, a set of changes on the Service environment can be experienced as the set of effects caused by the Service execution.
Taking the example of a book selling service, the service function could transform a title of a book and a credit card number (inputs) into a receipt for the sale (output). The

preconditions could be the credit card validity and an associated creditworthy account, while an effect could be the physical possession of the purchased book.
Inputs, Outputs, Preconditions and Effects (hereafter IOPE) are analogous to capability/preferences specifications advertisements/want-ads for middle-agents. Thus, IOPE play an essential role during service matching. In fact, a feasible matching engine will have to make sure that the IOPE expected by the requester match the IOPE supplied by the selected service provider.

2. DAML-S Matching Engine

In [Paolucci et al., 2002] it is presented an algorithm for semantic matching of Web services capabilities based on DAML+OIL (see [DAML+OIL Reference, 2001]) ontologies - a future adoption of OWL ontologies would be straightforward.
Assuming that a request for a service is expressed in the form of a Service Profile and that a service provider advertises its own service capabilities again in the form of a Service Profile, the matching process should perform a semantic combining between the respective IOPE on the requester and the provider side.
The single parameters composing each IOPE specification are expressed w.r.t. to some shared ontologies, hence they are univocally interpretable.
Usage of Service Profile should then overcome syntactic discrepancies and difference in modelling abstractions that could arise between requests and advertisements.
Referring to the shared DAML+OIL ontologies, the matching engine accurately reasons on the available subsumption hierarchies, thus leading to a final coupling between the requested IOPE and the advertised IOPE based on semantics rather than syntax or modelling abstractions.
The main rationale behind the algorithm is that an advertisement matches a request when the service provided by the advertiser can be of some use for the requester. Specifically, an advertisement matches a request when all the outputs of the request are matched by the outputs of the advertisement, and all the inputs of the advertisement are matched by the inputs of the request.
The degree of success depends on the degree of match detected. For instance, if one of the request's output is not matched by any of the advertisement's output the match fails.
The degree of match between two outputs or two inputs depends on the relation between the ontology concepts associated with those inputs and outputs. In other words, the degree of match is determined by the minimal distance between concepts in the taxonomy tree underlying the reference ontology.
Hence, degrees of match are ranked in a discrete scale as follows:

1. *Exact*
   Any requested output (outR) must be equivalent (i.e. being a class comprising of the same set of individuals) to an advertised output (outA). However, even if outR is a sub-class of outA (i.e. all the individuals within the class outR belong also to the class outA) and the provider commits to supply outputs consistent with every immediate subtype of outA then the match is still exact.
2. *Plug-in*
   If the concept outA subsumes the concept outR, thus intending neither the sub-class or the equivalence relationship mentioned above.
3. *Subsumes*
   If outR subsumes outA.

4. *Fail*
   No subsumption relation can be identified neither from advertisement to request nor vice-versa.

Since the exposed rationale is that the requester expect first and foremost that the provider achieves the output requested at the highest degree, input matching is used only as a secondary score to break ties between equally scoring outputs.
Recalling the usage of UDDI registries (see section 1.2) to enable Web services discovery, it is worthy to remark that the just presented matching engine can augment UDDI registries with a layer performing capability-based matching. UDDI in fact provides poor search facilities, in that it allows only keyword-based search of services by respective service names. DAML-S matching engine and UDDI registries can be easily jointed by means of a communication module and a DAML-S/UDDI translation module, as it is illustrated in the figure below. By the way, the resulting architecture can be stated to address the implementation of a matchmaker middle-agent.

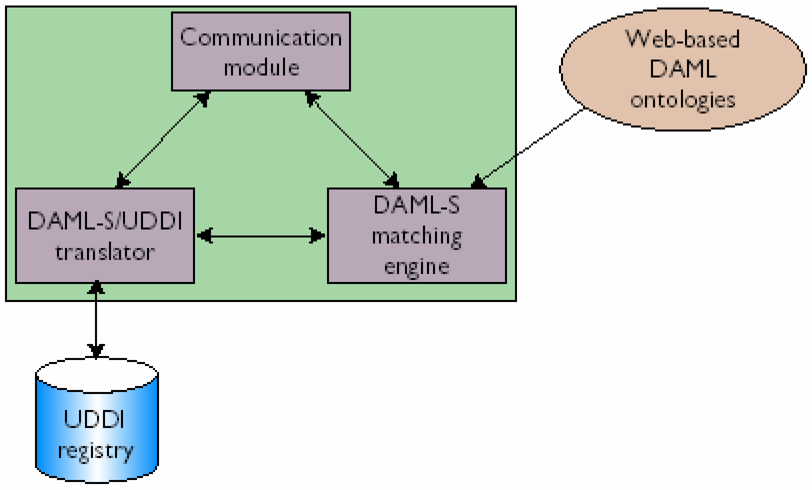


Figure 14. DAML-S/UDDI combined architecture for service matching.

3. DAML-S Broker Implementation

3.1. Specification

In this section we will eventually describe the realization of the DAML-S broker by devising the interaction protocol for a broker middle-agent (see chapter 2) in order to satisfy all the features of an autonomous semantic Web service introduced so far.
Before stepping off with our examination, we remind all the discussed characteristics, such as:

- Web service infrastructure;
- DAML-S as the adopted capability description language;
- matching engine presented in [Paolucci et al., 2002].

On this background, in [Paolucci and Sycara, 2003] we find a comprehensive picture for the architecture of a DAML-S Web service:

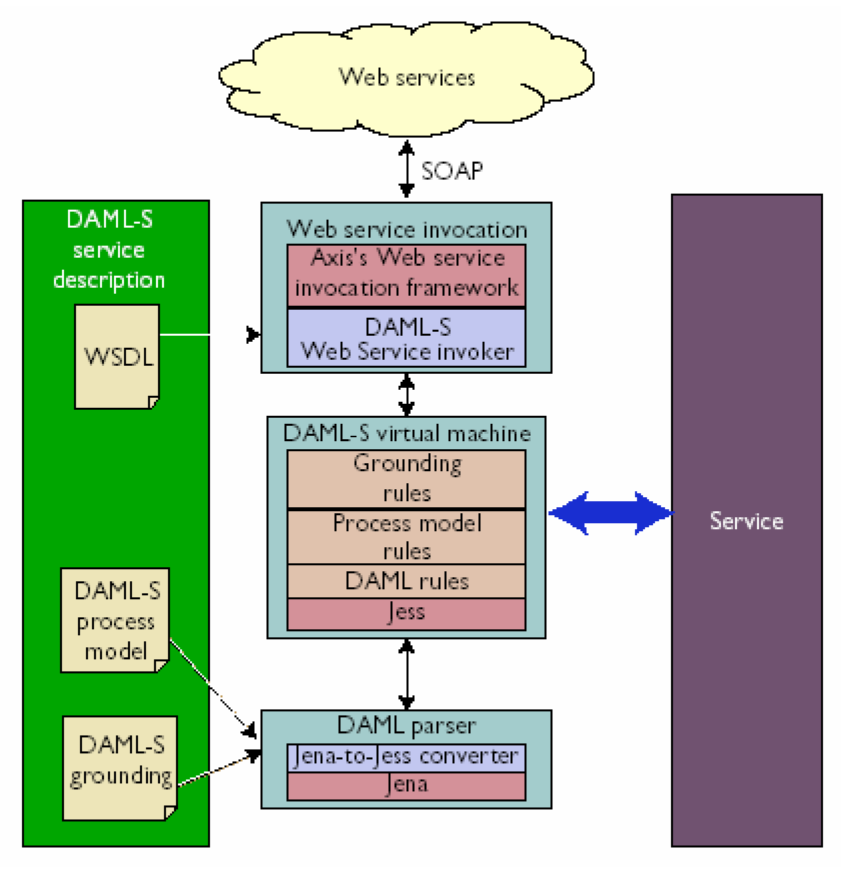


Figure 15. DAML-S Web service.

Besides some special-purpose components -such as Axis (see [Axis]), Jess (see [Friedman-Hill, 1995]), or Jena (see [Jena])-, the figured architecture for autonomous semantic Web services includes a module worthy of some brief ulterior annotation. That is the DAML-S Virtual Machine (DSVM).
Designed at Intelligent Software Agents Laboratory (reference at [Paolucci et al., 2003]), the DSVM is basically a software module able to control the interaction with a Web service in accordance with its DAML-S description. It is invoked by a requester willing to start interacting with a DAML-S Web service. Granting semantics to DAML-S descriptions through some proper rules, the DSVM uses the Service Grounding to transform the abstract information exchanges of the Service Model into concrete message content. The execution of the appropriate DAML-S Service Model is thereby launched, thus establishing the required interaction.

The DAML-S broker implements the model depicted in figure 15 while respecting the mentioned activity protocol. Concerning with that chronological list of activities performed by a broker middle-agent, we can now better specify them as follows:

1. The requester stores the broker's Service Profile, Service Model, and Service Grounding.
2. The broker stores the Service Profile, Service Model, and Service Grounding of a provider willing to advertise its capabilities to the broker.

3. The requester sends a query about needed information to the broker in a well-known query language.
4. The broker converts the received query into a Service Profile, that is the request for a service asked by the same query.
5. The matching algorithm proposed in [Paolucci et al., 2002] is used by the broker to match the requester's Service Profile with any of the provider's Service Profiles stored by itself. Some further criteria are necessary to select only a provider among all those fitting a successful matching.
6. A further matching is tried among the non-output-parameters of the requester's Service Profile and the non-output-parameters of the selected (see previous step) provider's Service Profile. The successfully matching parameters are pruned from the provider's Service Model, and that new Service Model is notified by the broker to the requester as a request for specific extra-parameters. The provider could not truly execute the required service without those extra-parameters.
7. For the requester to execute the new Service Model, the broker should modify its own Service Model during the transaction. The requester is in fact dealing with the broker Web service w.r.t. the Service Model exposed by the broker at the very beginning of the interaction. Since DAML-S does not support the dynamical modification of the published Service Model, DAML-S language was expanded with an ad hoc statement. The requester can thus execute the new Service Model of the broker, consequently recognizing the lack of parameters needed to fulfil the required service. A new query to the broker is then formulated by the requester, this time hopefully containing the needed parameters.
8. A transformation algorithm is possibly used by the broker to supply the provider with all the non-output-parameters required to correctly execute the service. That transformation is motivated by the fact that an immediate matching between the non-output-parameters supplied by the requester and the non-output-parameters required by the provider could not always take place (the matching process mentioned at step 6 is analogously motivated).
9. Upon the receipt of the required inputs among the non-output-parameters, the provider performs the asked service and replies back to the broker the resulting information.
10. A further transformation could possibly happen for the broker to forward the provider's reply to the requester. However this would be very rare to take place, since the matching algorithm (see step 5, and section 2 of this chapter for more details) for the provider's choice assures the correspondence of the service outputs. Therefore we assume this transformation negligible.

As a general remark, we point out that the broker needs to ground every process of a newly exposed Service Model to a concrete operation (see DAML-S rough description above). That should happen every time the broker appropriately routes queries and replies between requester and provider. The implementation of a common concrete operation for each new grounding could work as a universal port for the low level (i.e. at the communication layer) exposition of new Service Models. DAML-S grounding language should eventually be able to ground each distinct Service Model to that same concrete operation. Fortunately, DAML-S grounding language is flexible enough to perform that treatment.

3.2. Open Issues

From the previous section the following issues have arisen to be necessary for the DAML-S broker implementation:

- adoption of a standard query language for the initial query from the requester to the broker (see step 3);
- requester's query conversion into a request formulated as a Service Profile (at step 4);
- algorithm to select the best fitting provider among all those reported by the matching engine applied at step 5;
- algorithm to determine non-output-parameters matching (mentioned at step 6);
- algorithm to transform inputs among the non-output-parameters supplied by the requester into exactly those inputs needed by the provider for executing the service;
- modification of the DAML-S service modelling language to include a statement devoted to dynamically execute a new Service Model;
- algorithm to generate a new grounding for the atomic process of a new Service Model over a common concrete operation executed by the broker;
- updating of the
- synchronization of the broker to cope with its concurrent execution of the provider's Service Model and the requester's execution of each brand new Service Model generated by the broker itself.

Dealing with all the advised issues goes beyond the scope of our discussion. The last four needs were in fact solved at the Intelligent Software Agent Laboratory (Carnegie Mellon University). So, we address the reader to the upcoming papers for an exhaustive explanation of the developed solutions.
In the next chapters we will instead envisage solutions to the first five issues.

V Common Threads

1. Open Issues

In chapters 3 and 5, we discussed the SEWASIE Brokering Agent (BA) and the DAML-S Broker (DSB), respectively. In both cases, our analysis resulted in some open issues still to be tackled. In order to ease a quick comparison, here we report them again.
For the BA:

1. an ODL_{I3} ontology serving as:
    a. Knowledge Base (KB) to interprete queries posed by a SEWASIE Query Agent (QA);
    b. Kowledge Base (KB) gathering some relevant information from the supervised SINodes;
2. a matching operator for the incoming queries against the cited ontology;
3. a mapping function connecting the cited ontology with the ontologies of the underlying SINodes;
4. a transport and communication layer to let agents communicate each other.

For the DSB:

1. adoption of a standard query language for the initial query from the requester;
2. requester's query conversion into a request formulated as a Service Profile;
3. algorithm to select the best fitting provider among all those reported by the matching engine;
4. algorithm to determine non-output-parameters matching;
5. algorithm to transform inputs among the non-output-parameters supplied by the requester into exactly those inputs needed by the provider for executing the service;
6. modification of the DAML-S service modelling language to include a statement devoted to dynamically execute a new Service Model;
7. algorithm to generate a new grounding for the atomic process of a new Service Model over a common executable operation;
8. updating of the synchronization to cope with the concurrent execution of the provider’s Service Model and the requester’s execution of each brand new Service Model.

As we already said at the bottom of chapters 3 and 5, our work is focused only on open issues 2, 4 for the BA, and on the first five open issues for the DSB.
We notice that both the BA and the DSB implement a middle-agent which is committed to manage an incoming query for some information. The middle-agent will process such a query posed by an information requester in order to discover an information provider capable to successfully answer it. We refer to the whole process as *Query Processing*.
We indentified the common threads of the pointed, still unsolved issues as two subtasks of Query Processing. Namely, *Query Abstraction* and *Query Rewriting*. We address them under the an inclusive process that we call *Query Reformulation*.
Query Reformulation is aimed at producing a query that can be matched and satisfied with the capabilities advertised by a discovered information provider. Thus, Query Processing must incorporate an ulterior process delegated to find out an opportune information provider. Such a process is the so called *Semantic Matching*.
Summing up, our vision of Query Processing comprises two mutual activities: Query Reformulation and Semantic Matching. Extensively, Query Processing is thus articulated in:

1. Query Abstraction,
2. Semantic Matching, and
3. Query Rewriting.

In the sequel we will see how Query Reformulation tackles both issue 2 for the BA and issues 2, 3, 4, 5 for the DSB.
However, we must priorly depict the context where Query Reformulation takes places. This leads to our contributions for issue 4 of the BA, and for issue 1 of the DSB. In other words, we must first devise an abstract model for the BA as a middle-agent . Consequently, we must posit the conditions where Query Reformulation is endevoured, and eventually propose an acceptable query language for the DSB.

2. Interaction Protocol for SEWASIE Brokering Agent

W.r.t. SEWASIE architecture (see chapter 3), here we simplify the working environment for the BA with the following limitations:

- single BA (that means no peer to peer network of BAs) interleaving between QAs and the network of SEWASIE Information Nodes (SINodes);
- common shared ontology between QAs, BA, and SINodes;
- single-shot query (that is one of the premises for the interaction protocols presented in chapter 2);
- just one best fitting SINode to answer the query (hence, no data reconciliation to assembly the answer).

Leaving from these standpoints, we now suggest how a middle-agent interaction protocol may be devised for the BA by adapting the interaction protocols introduced in chapter 2 for a matchmaker and a broker. In figure 16, the properly shaped sequence diagram has been drawn.

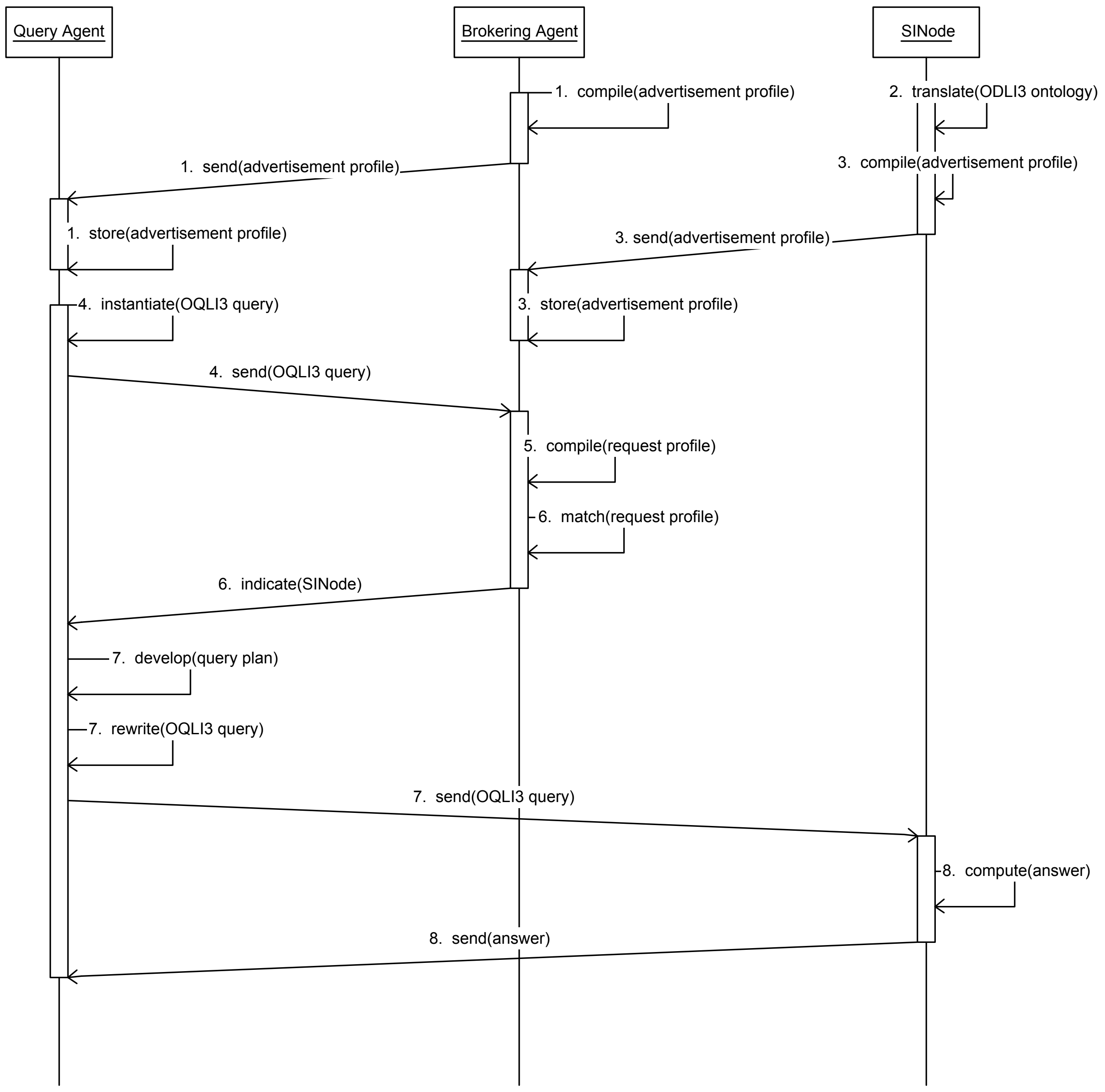

Query Agent
Brokering Agent
SINode
1. compile(advertisement profile)
2. translate(ODLI3 ontology)
1. send(advertisement profile)
3. compile(advertisement profile)
1. store(advertisement profile)
3. send(advertisement profile)
4. instantiate(OQLI3 query)
3. store(advertisement profile)
4. send(OQLI3 query)
5. compile(request profile)
6. match(request profile)
6. indicate(SINode)
7. develop(query plan)
7. rewrite(OQLI3 query)
7. send(OQLI3 query)
8. compute(answer)
8. send(answer)

Figure 16. Sequence diagram for the SEWASIE Brokering Agent interaction protocol.

As it is pointed out by the diagram above, the BA receives an advertisement profile from an SINode and compiles a requester's query into a request profile. Thus, the BA acts as a semantic Web service (see chapter 4). In fact, in order to fit out the BA with the required matching functionalities (open issue 2), we have envisaged the BA interactions with QAs and SINodes within the Service Profiles matching dialectic. Thereby, we assumed that SINodes advertise their query answering capabilities in the form of Service Profiles against the shared ontology by the BA. This way, we can apply the matching algorithm used by the DSB (see chapter 4) as the required matching operator for a request equally formulated as Service Profile against the shared ontology by the BA. Keeping this approach, we might alternatively enforce the matching functionalities of the BA by means of any other

procedure pursuing information providers discovery on the basis of Service Profiles matching.
Essentially, this constitutes the Semantic Matching process for both the BA and the DSB.
Here, we just want to propose a general approach to cope with information providers (specifically, SINodes) discovery by the BA. The implementation details are beyond the scope of our work. By the way, the DSB was regarded as a paradigmatic, fully implemented broker middle-agent executing Semantic Matching through Service Profiles matching.
So, we can explicitly describe the sequence of activities represented by the diagram above as the following step-thorugh protocol:

1. A QA stores the advertisement compiled and sent by the BA in the form of a Service Profile. So, that QA knows the location and the matching functionalities of the BA.
2. A SINode wishing to advertise its query answering capabilities translates its own local ontology from ODL_{I3} (see [Bergamaschi et al., 2001]) to a proper language (e.g. OWL, see [OWL Reference, 2003]) for the formulation of a Service Profile.
3. This SINode compiles an advertisement Service Profile that is sent to the BA and stored by the BA itself.
4. The earlier considered QA instantiates and sends a query about needed information to the BA in a well-known query language (i.e. OQL_{I3}).
5. The BA converts the received query into a Service Profile, that is the request for an information service asked by the same query.
6. A matching algorithm is used by the BA to match the QA's Service Profile with any of the SINode's Service Profiles stored by itself. Some further criteria are necessary to select only a SINode among all those fitting a successful matching. The selected SINode is indicated by the BA to the QA. Besides, some relevant information about the selected SINode are attached to the message sent by the BA to the QA.
7. Depending on the received indication and accompanying attachments, the QA elaborates a query plan and consequently rewrites the query in an adequate fashion for the selected SINode. Such a reinstantiated query is then sent to the identified SINode.
8. Upon the receipt of the query, the SINode retrieves the asked information and replies back to the QA the computed answer.

3. Query Language for DAML-S Broker

Since the recently designed DAML Query Language (DQL, reference paper at [Fikes et al., 2002]) is a language and protocol supporting agent to agent query-answering dialogues using knowledge represented in DAML+OIL or OWL (see [DAML+OIL Reference, 2001]and [OWL Reference, 2003], respectively), it seems perfectly natured to serve as query language for the DSB. In fact, we need a formalism for an unambiguous template that the initial query sent to the DSB must fulfil. The DSB will then parse such a query in order to extract the parameters for the compilation of the corresponding Service Profile.
Although DQL developers do not provide an external syntax for the language, they specify the types of objects that are passed between two agents during a query-answering dialogue (thus, the agents cast the roles of a querying *client* and an answering *server*). These objects cannot be else than queries and answers. In particular, a *DQL query* contains a *query pattern*, an *answer KB pattern*, a *must-bind variables list*, a *may-bind*

variables list, and, optionally, an *answer pattern*, a *query premise*, a *justification request*, and an *answer bundle size bound*.
The query pattern is a collection of DAML+OIL sentences (i.e. a Knowledge Base) in which some literals and/or URIrefs have been replaced by variables.
The answer KB pattern consists of set of DAML sentences that are used by the server in answering a query.
Each variable that occurs in a DQL query (i.e., in either the query pattern or the answer KB pattern) is considered to be a must-bind variable, a may-bind variable, or a don't-bind variable. The must-bind variables list and the may-bind variables list allow DQL to support existentially quantified answers by enabling the client to designate some of the query variables for which answers will be accepted with or without bindings. Answers are required to provide bindings for all the must-bind variables, and may provide bindings for any of the may-bind variables. Variables in a query that are not in either the must-bind or may-bind variables lists are don't-bind variables. Answers are not required to provide bindings for any of the don't-bind variables.
Somehow recalling rule-based languages (see [Vianu, 1997]), a query premise is a DAML+OIL KB or a KB reference and it facilitates the representation of "if-then" queries. In fact, a query premise can preceed the query pattern, thus forming an "if-then" sentence where the query premise represents the protasis and the query pattern represents the apodosis (i.e., the query premise entails the query pattern).
For the sake of brevity, we skip the definition of the remaining optional components of a DQL query, in that they are not significant for our purposes.
In fact, we have reported some of the DQL abstract specifications in order to illustrate how the main parameters of a Service Profile can be derived from a DQL query. Those parameters are the ones describing the functional behaviour of a service, i.e. *inputs*, *outputs*, *preconditions*, and *effects* (as we said at paragraph 1.3, chapter 4).
Essentially, inputs will be all the non-variable terms in the query pattern that are either logical subjects or logical objects of the DAML+OIL sentences forming the same query pattern. Outputs will be all the variable terms in the query pattern that are either logical subjects or logical objects of the DAML+OIL sentences forming the same query pattern.
Preconditions can be analogously extracted from the terms of an optional query premise.
Some non-variable terms of the query pattern might be alternatively considered as preconditions or effects rather than inputs. Similarly, some variable terms might be viewed as effects rather than outputs. Unfortunately, no syntactic criterion is actually available to certainly state such a discernment. Possibly, a semantic interpretation of the corresponding DAML+OIL sentences could supply the required methodology. Furthermore, it should be priorly resolved a sort of natured overlap between both inputs/preconditions and outputs/effects at the specification level in the Service Profile.
For a deeper treatment of problems concerning with DAML-S parameters mapping, refer to [Burnstein, 2003].

4. Query Processing

Within this section, we will basically illustrate how interesting subtasks of Query Processing overcome open issues 2, 3, 4, and 5 of the DSB.
As it has been suggested in section 2, we can tackle open issue 2 of the BA by adopting the Service Profile based approach implemented by the DSB. Thus, solutions for the open issues of the DSB will equally apply to the BA.
In particular, we will explain how Query Abstaction solves open issue 2 of the DSB, while

Query Rewriting works out open issues 3, 4, and 5. In order to supply a self-contained description of Query Processing, we will also report some brief considerations about Semantic Matching.

Being consistent with the assumptions stated at the beginning of section 2 for the BA, the proposed approach to Query Processing is figured out within a context comprising a network of information (or, more generally, service) requesters, a single broker (specifically, a DSB), a network of information (or, more generally, service) providers, and a common shared ontology.
We recall that this section contains just a shallow introduction to Query Reformulation (i.e. Query Abstraction and Query Rewriting). For a formal specification, the reader is addressed to chapter 8.

4.1. Query Abstraction

Query Processing demands this subtask to abstract terms in the requester's query so that they can take part in the compilation of the corresponding request formulated as a Service Profile. Since the aim of this compiling process is to make the subsequent matching process effective, abstracted terms must represent capabilities enforcing the matching process. Our assumption is that concepts rather than instances or variables can represent capabilities. Clearly, Query Abstraction from instances or variables to concepts is performed w.r.t. the common reference ontology.
For example, if the requester's query is “What will the temperature be in Pittsburgh tomorrow?”, then terms like temperature” and “tomorrow” do not need any abstraction, in that they are already concepts w.r.t. the reference ontology. Instead, abstraction is required for the instance “Pittsburgh”. Scrolling all the possible meanings (i.e. corresponding concepts) for “Pittsburgh” in the reference ontology, we might abstract “Pittsburgh” either to “city”, or to “city in the US”, or to “city in Pennsylvania”, or even to “location in the northern hemisphere”, and so on. As the chosen abstraction will considerably affect the subsequent matching with any of the advertisements stored by the broker, in chapter 8 it will be also proposed some useful heuristics to enforce the choice of a "good" abstraction.

4.2. Semantic Matching

The scope of this process is to discover a service provider which is capable to supply the information (or, more generally, the service) asked by the requester. Semantic Matching can be performed through any algorithm achieving requester's and provider's Service Profiles matching. Some examples in the current state of the are [Di Noia et al., 2003] and [Benatallah et al., 2003]. As it is described in greater details at section 2 of chapter 4, the DSB exploites the matching algorithm proposed by [Paolucci et al., 2002]. Here, we just recall that such an algorithm ends up with the selection of an adequate provider for the request submitted to the DSB. The first matching criterion is the subsumption relation between the request Service Profile's outputs and the provider’s advertisement profile's outputs. In fact, the outputs required by the request must be at least subsumed by the outputs promised by the provider’s advertisement. Instead, the matching rules regarding the Service Profile's inputs do not strictly constraint the final selection. The inputs required by the selected provider can actually differ from the inputs supplied by the submitted request.

4.3. Query Rewriting

Generally speaking, this is the subtask of Query Processing which aims at submitting an understandable request to a given service provider. Recalling that from our standpoint a request is a Service Profile compiled from an abstracted query (which, in turn, is the query resulting from the Query Abstraction process on the initial requester's query), such an abstracted query must be rewritten so that all its terms are meaningful for the addressed provider.

In practice, this is achieved by attempting to transform each term of the abstracted query into semantically equivalent terms, which are the parameters required by the service provider to successfully satisfy the submitted request.

Therefore, if Query Rewriting is performed after Semantic Matching according to [Paolucci et al., 2002], it is sensible to operate it just over non-outputs parameters of the provider's Service Profile. Hence, it will be reduced to a solution for open issue 4 of the DSB. Similarly, if Query Rewriting is restricted to provider's Service Profile inputs only, it can be viewed as a solution for open issue 5 of the DSB. At section 3 of chapter 8 it will be presented an algorithm dealing exactly with the latter problem.

In the end, we notice that the degree of success of Query Rewriting applied to non-outputs parameters can be a reliable criterion to rank different providers equally fitting Semantic Matching after the execution of the algorithm described in [Paolucci et al., 2002]. Hence, a solution for open issue 3 of the DSB will be straightforwardly obtained.

5. About Source Capabilities Descriptions

In chapter 1, our discussion was opened and motivated by three interrelated questions:

4. *How can a broker determine whether a query can be successfully answered by an Information Source that advertises with the broker itself?*
5. *How can a mediator determine whether a query can be successfully answered by the wrapped Information Sources below it?*
6. *How can a wrapper determine whether a query can be successfully answered by the underlying Information Source?*

The Capabilities-Based Rewriting (CBR) and the Query Expressibility Decision (QED) problems were drawn out from those questions. Indeed, descriptions of an Information Source (IS) in terms of *query capabilities* and *content capabilities* were pointed out as necessary to deal with query answering in mediator-based systems.

Now, after the advised approach to Query Processing, we observe that there is no need for a language to express querying capabilities of an IS. Moreover, there is no need for algorithms to decide whether a query is supported by those capabilities (i.e., the QED problem) or to rewrite an external query into queries supported by some integrated ISs (i.e., the CBR problem).

Service Profiles provide a sort of querying capabilities descriptions. Since Query Abstraction substantially maps queries to Service Profiles, a query can be answered only if

there is an advertised profile with successfully matching outputs and successfully matching inputs (perhaps after Query Rewriting). Therefore, the CBR basically consists of mapping a query to a Service Profile and the QED is viewed as profiles matching.
Roughly speaking, Query Processing within the Semantic Web environment allows us to state a correspondence between a mediator-based architecture and a Web service architecture, where querying capabilities are expressed through Service Profiles and a query becomes an information request formulated as a Service Profile.

VI Description Logics, OWL, and ODL_{I3}

1. Description Logics

1.1. Basics

Quoting from [Baader and Nutt, 2002], *Description Logics* (DLs) "is the most recent name for a family of knowledge representation formalisms that represent the knowledge of an application domain (the "world") by first defining the relevant concepts of the domain (its terminology), and then using these concepts to specify properties of objects and individuals occurring in the domain (the world description)".
Unlike some of their predecessors (e.g., semantic networks and frame-based formalisms), DLs emphasize two distinguishing features:

1. they are equipped with a formal, logic-based semantics;
2. reasoning is a central service which allows one to infer implicitly represented knowledge from the knowledge that is explicitly contained in the described knowledge base.

By means of DLs, the formalization of a Knowledge Base (KB) comprises two components, the *TBox* and the *ABox*. The former component introduces the *terminology*, i.e. the vocabulary of an application domain. The latter component contains assertions about named individuals in terms of this vocabulary.
More specifically, the vocabulary consists of *concepts* denoting sets of individuals, and *roles* denoting binary relationships between individuals.
At the primitive level, such a DL formalizes a KB in terms of *elementary concept descriptions* constituted by *atomic* (or *primitive*) *concepts* and *atomic* (or *primitive*) *roles*.
In order to qualify an individual as a member of either a concrete set (e.g., an individual can be either an integer, or a string, or a Boolean, or a real, etc) or an abstract set (i.e. the concept mentioned up to now), individuals can be categorized into *values* and *objects*, respectively. Consequently, the notion of concept is replaced by the notion of *type*, which is further refined as either *value type* (for values) or *class type* (for objects). The term *attribute* is used instead of the term role.
At the primitive level, such a DL thus formalizes a KB in terms of *elementary type descriptions* constituted by *atomic* (or *primitive) types* and *atomic* (or *primitive*) *attributes*.
Inductively, complex descriptions can be built from elementary descriptions with adequate *constructors*. Thereby, elementary descriptions are generalized into either *concept descriptions* when involving concepts and roles, or *type descriptions* when involving types and attributes. Various families of DLs are then distinguished by the constructors they provide.
Formal semantics of a DL are usually given via *interpretations* I that consist of a non-empty set $I(\Delta)$ (i.e. the *domain of an interpretation*) and an *interpretation function* $I(\cdot)$. Therefrom, $I = (I(\Delta), I(\cdot))$.

In order to supply semantics for elementary concept descriptions, the interpretation function assigns to every atomic concept name *A* a set $I(A) \subseteq I(\Delta)$, and to every atomic role name *R* a binary relation $I(R) \subseteq I(\Delta) \times I(\Delta)$, where Δ is the set of concept names defined w.r.t. a given TBox.
In order to supply semantics for type descriptions, the interpretation function is extended so that it assigns to every type description *T* a set $I(T) \subseteq I(2^V)$, where *V* is the smallest set comprising all the values and all the objects identifiers defined within a given ABox w.r.t. a given TBox.
The interpretation function can be specifically extended in order to satisfy a list of mandatory equations, where the argument on the left-hand side is a concept/type description, and the argument on the right-hand side is the corresponding meaning. Such expressions constitute the *model-theoretic semantics* of the DL.

DLs support some typical inference patterns such as classification of concepts and individuals, concept descriptions satisfiability, consistency of an ABox, concept descriptions equivalence, and concept descriptions disjointness.
Classification of concepts determines subconcept/superconcept relationships (i.e. *subsumption* relationships) between concepts of a given terminology, and thus allows one to structure the terminology in the form of a subsumption hierarchy.
Classification of individuals determines whether a given individual is always an instance of a certain concept.
Satisfiability is the problem of checking whether a concept description does not necessarily denote the empty concept. Satisfiability is a special case of subsumption: if the subsumer concept is the empty concept then the subsumee concept has a not satisfiable (i.e. contradictory) description. In a formal account, a concept description *C* is satisfiable w.r.t. to a TBox *T* if there exists an interpretation *I* s.t. *I(C)* is non-empty. In this case, *I* is also said to be a *model* of *C*.
As a consequence, a set of assertions within an ABox *A* is *consistent* w.r.t. a given TBox *T* if there is an interpretation that is a model of both *A* and *T*.
Two concept descriptions *C* and *D* are *equivalent* w.r.t. a TBox *T* if they have the same interpretation according to all of their models w.r.t. *T*.
By the contrary, two concept descriptions *C* and *D* are *disjoint* w.r.t. a TBox *T* if the intersection of their interpretations according to all of their models w.r.t. *T* is always an empty concept.
Because of their motivating orientation towards knowledge representation systems, DLs must secure decidable reasoning tasks. In other words, reasoning procedures should always terminate in a finite -hopefully acceptably short- time both for positive and for negative answers.
As it was firstly showed by [Levesque and Brachman, 1987], a fundamental inference problem like subsumption is intractable (i.e. not polynomially solvable) even for very inexpressive languages. Generally speaking, decidability and complexity of the inference problems depend on the expressive power of the DL at hand. Therefore, a trade-off between the expressivity and the complexity of reasoning services necessarily affects the design of any DL.

As a prerequisite for the next sections, we will now report the formal semantics for two distinct DLs, namely $\mathcal{ALC}$ and $\mathcal{OLC}$.

1.2. $\mathcal{ALC}$

"$\mathcal{ALC}$" is an attributive ($\mathcal{A}$) language ($\mathcal{L}$) with complements ($\mathcal{C}$).
In $\mathcal{ALC}$, *concept descriptions C*, *D* are usually given according to the following syntax (refer to [Baader and Nutt, 2002] for a detailed explanation):

$C, D \rightarrow A$ |	*(atomic concept)*
T |	*(top concept)*
$\perp$ |	*(bottom concept)*
$\neg A$ |	*(atomic negation)*
$C \sqcap D$ |	*(intersection)*
$C \cup D$ |	*(union)*
$\forall R.C$ |	*(value restriction, or universal quantification)*
$\exists R.C$	*(full existential quantification)*

where the following symbology has been used:

A	*(atomic concept name)*
R	*(atomic role name)*
T	*(top concept)*
$\perp$	*(bottom concept)*
$\neg$	*(complement operator)*
$\sqcap$	*(intersection operator)*
$\cup$	*(union operator)*
$R.C$	*(role restriction)*
$\forall$	*(universal quantificator)*
$\exists$	*(existential quantificator).*

Accordingly, $\mathcal{ALC}$ is defined by the following model-theoretic semantics:

$I(T) = I(\Delta)$
$I(\perp) = \varnothing$
$I(\neg C) = I(\Delta) \setminus I(C)$
$I(C \sqcap D) = I(C) \cap I(D)$
$I(C\ U\ D) = I(C) \cup I(D)$
$I(\forall R.C) = \{a \in I(\Delta) \text{ s.t. } \forall b.\ (a,b) \in I(R) \Rightarrow b \in I(C)\}$
$I(\exists R.C) = \{a \in I(\Delta) \text{ s.t. } \exists b.\ (a,b) \in I(R) \wedge b \in I(C)\}$

where the following auxiliary symbology has been used:

a, b	*(individual identifiers)*
Δ	*(set of concept names)*
$I(\cdot)$	*(interpretation function from C,D to Δ)*
$\Rightarrow$	*(logical implication)*

1.3. $\mathcal{OLC}$

"$\mathcal{OLC}$" is an object ($\mathcal{O}$) language ($\mathcal{L}$) with complements ($\mathcal{C}$).
In $\mathcal{OLC}$, the set of *finite type descriptions S* (where $S = \{ S_1, S_2, \dots, S_k \}$) is usually given according to the following syntax (refer to [SEWASIE D2.1] for a detailed explanation):

$S \rightarrow N \mid$	*(type)*
$T \mid$	*(top type)*
$\perp \mid$	*(bottom type)*
$\neg S \mid$	*(negation)*
$S_1 \sqcap S_2 \mid$	*(intersection)*
$S_1 \cup S_2 \mid$	*(union)*
$\{S\}_\forall \mid$	*(universal set quantification)*
$\{S\}_\exists \mid$	*(existential set quantification)*
$[a_1{:}S_1, \dots, a_k : S_k] \mid$	*(attributes description)*
$\Delta S \mid$	*(class description)*
$p \theta d \mid$	*(atomic predicate as range restriction)*
$p\uparrow$	*(atomic predicate as path undefinedness)*

where the following symbology has been used:

N	*(set of type names)*
T	*(top type)*
$\perp$	*(bottom type)*
$\neg$	*(complement operator)*
$\sqcap$	*(intersection operator)*
$\cup$	*(union operator)*
$\{\dots\}_\forall$	*(universal set quantifier)*
$\{\dots\}_\exists$	*(existential set quantifier)*
$\{\dots\}$	*(set constructor)*
$[\dots]$	*(tuple constructor).*
a_i	*(attribute name)*
$a_i : S_i$	*(attribute)*
Δ	*(class constructor)*
θ	*(equality, inequality, total order relations, i.e.* $=, \neq, >, <, \geq, \leq$*)*
d	*(value)*
p	*(attribute path)*
$\uparrow$	*(attribute undefinedness operator)*,

Accordingly, $\mathcal{OLC}$ is defined by the following model-theoretic semantics:

$I(T) = V$
$I(\perp) = \emptyset$
$I(B) = I_B(B)$
$I(\{S\}_\forall) = \{M \text{ s.t. } M \subseteq I(S)\}$
$I(\{S\}_\exists) = \{M \text{ s.t. } M \cap I(S) \neq \emptyset\}$
$I([a_1{:}S_1, \dots, a_k : S_k]) = \{t : A \rightarrow V \text{ s.t. } t(a_i) \in I(S_i), 1 \leq i \leq k\}$
$I(S_1 \sqcap S_2) = I(S_1) \cap I(S_2)$
$I(S_1 \cup S_2) = I(S_1) \cup I(S_2)$
$I(\neg S) = V \setminus I(S)$
$I(\Delta S) = \{o \in O \text{ s.t. } \delta(o) \in I(S)\}$

$I(p\theta d) = \{v \in V \text{ s.t. } I(p)(v)\theta d\}$
$I(p\uparrow) = \{v \in V \text{ s.t. } v \notin dom(I(p))\}$

where the following auxiliary symbology has been used:

B	*(set of base values designators, e.g., Integer, String, Boolean, and Real)*
D	*(set of base values, i.e. the union of the integers, the strings, the Booleans, and the reals)*
O	*(set of object identifiers)*
V	*(set of all the values over O, i.e. the smallest set containing D and O)*
$I(\cdot)$	*(interpretation function from S to* 2^V*)*
$I_B(\cdot)$	*(fixed interpretation function from B to* 2^D*)*
A	*(set of attribute names)*
t	*(tuple value)*
d	*(base value)*
o	*(object identifier)*
δ	(total value function from O to V).
$dom(I(\cdot))$	(domain of the interpretation function $I(\cdot)$)

2. OWL

2.1. Overview

OWL (see [OWL Reference, 2003]) is a semantic markup language for publishing and sharing ontologies on the World Wide Web. Since an ontology may include descriptions of classes, properties and their instances, OWL can be used to:

- formalize a domain by defining classes and properties of those classes;
- define individuals and assert properties about them;
- reason about these classes and individuals in the measure permitted by the language semantics.

Because of these fundamental characteristics and its increasing diffusion all over the Semantic Web and Web services community, we chose OWL (precisely, OWL DL) ontologies to serve as the knowledge bases supporting Query Reformulation.
Beyond the expressive power, we were obviously interested in the reasoning capabilities backed by OWL. Since OWL was developed as a vocabulary extension of RDF (refer to [RDF Reference, 2003]) and is derived from DAML+OIL (refer to [DAML+OIL Reference, 2001]), OWL design was strongly influenced by Description Logics (DLs), frames paradigm, and RDF syntax. In particular, the formal specification of OWL language was affected by DLs, the abstract syntax (refer to [OWL Abstract Syntax and Semantics, 2002]) was influenced by the frames paradigm, while the exchange syntax (see [OWL Reference, 2003]) was designed to be upward compatible with RDF.
The formal specification of an ontology language includes the supported constructors -clarified by well-defined semantics-, and the kinds of axioms that can be asserted through those constructors. Both the expressive power and the allowed inferences are consequently determined by the formal specification. OWL's genesis on DLs inheritance was particularly evident in the choice of the language constructors -and related interpretations- in such a way that different degrees of expressive power could be

achieved without forgetting needs for a complete and decidable reasoning.
Thus, OWL splits off into three species:

- OWL Lite, featuring classification hierarchies and simple constraints;
- OWL DL, supporting maximum complexity without losing computational completeness and decidability;
- OWL Full, as expressive as RDF but with no computational guarantee.

For an accurate analysis of the trade-off between expressive power and complexity characterizing each of those species, it is worthy to trace a comprehensive list of OWL constructs looking back at some of OWL's predecessors. Specifically, tables 3, 4, and 5 offer a synoptic glance at OWL constructs and axioms compared with equivalent OIL primitives, and DAML+OIL elements. As a matter of fact, either of those languages was conceived as a RDF vocabulary extension.

Basic constructs			
OIL primitive	**DAML+OIL element**	**OWL element**	**Logical meaning**
-	*daml:Thing*	*owl:Thing*	top concept
-	*daml:Nothing*	*owl:Nothing*	bottom concept
class-def	*daml:Class*	*owl:Class*	primitive concept (class)
-	-	*owl:DataRange*	concrete domain (datatype)
slot-def	*daml:Property* *daml:ObjectProperty*	*owl:ObjectProperty*	role (property)
-	*daml:Property* *daml:DatatypeProperty*	*owl:DatatypeProperty*	concrete domain attribute (datatype property)
inverse	*daml:inverseOf*	*owl:inverseOf*	inverse role
symmetric	-	*owl:SymmetricProperty*	simmetric role
trasnsitive	*daml:TransitiveProperty*	*owl:TransitiveProperty*	transitive role
AND	*daml:intersectionOf*	*owl:intersectionOf*	conjunction
OR	*daml:unionOf*	*owl:unionOf*	disjunction
NOT	*daml:complementOf*	*owl:complementOf*	negation
-	*daml:oneOf*	*owl:oneOf*	enumeration
slot-constraint	*daml:Restriction… daml:onProperty*	*owl:Restriction… owl:onProperty*	property restriction
has-value	*daml:hasClass*	*owl:someValuesFrom*	existential quantification
value-type	*daml:toClass*	*owl:allValuesFrom*	universal quantification
-	*daml:hasValue*	*owl:hasValue*	universal quantification on individuals
cardinality	*daml:cardinality*	*owl:cardinality*	number restriction
max-cardinality	*daml:maxCardinality*	*owl:maxCardinality*	qualified number restriction
min-cardinality	*daml:minCardinality*	*owl:minCardinality*	qualified number restriction

Table 3. Synoptic view of OWL extensions to RDF vocabulary, compared with analogous OIL primitives and DAML+OIL elements. Basic constructs.

Derived constructs and axioms			
OIL primitive	**DAML+OIL element**	**OWL element**	**Logical Meaning**
subClassOf	*daml:subClassOf*	*rdfs:subClassOf*	concept (class) hierarchy
-	*daml:sameClassAs*	*owl:equivalentClass*	extensional equivalence between concepts
-	*daml:disjointWith*	*owl:disjointWith*	concept (class) disjunction
domain	*daml:domain*	*rdfs:domain*	role subject (property domain)
range	*daml:range*	*rdfs:range*	role object (property range)
subslot-of	*daml:subPropertyOf*	*rdfs:subPropertyOf*	role (property) hierarchy
-	*daml:UniqueProperty*	*owl:FuncitonalProperty*	feature (functional property)
-	*daml:UnambiguousProperty*	*owl:InverseFunctionalProperty*	inverse feature (inverse, funtional property)
-	*daml:samePropertyAs*	*owl:equivalentProperty*	extensional equivalence between roles
-	*daml:type*	*rdf:type*	individual instantiation
-	*daml:equiivalentTo* *daml:sameIndividualAs*	*owl:sameAs*	intensional equality between individuals
-	*daml:differentIndividualFrom*	*owl:differentFrom*	intensional un-equality between individuals
-	-	*owl:AllDifferent…* *owl:distinctMembers*	UNA

Table 4. Synoptic view of OWL extensions to RDF vocabulary, compared with analogous OIL primitives and DAML+OIL elements. Derived constructs and axioms.

Annotations		
OIL primitive	**DAML+OIL element**	**OWL element**
ontology-container	*daml:Ontology*	*owl:Ontology*
import	*daml:imports*	*owl:imports*
date, title, …	*daml:versionInfo*	*owl:versionInfo*
-	-	*owl:priorVersion*
-	-	*owl:backwardCompatibleWith*
-	-	*owl:incompatibleWith*
-	-	*owl:DeprecatedClass*
-	-	*owl:DeprecatedProperty*
-	-	*owl:AnnotationProperty*
-	-	*owlOntologyProperty*

Table 5. Synoptic view of OWL extensions to RDF vocabulary, compared with analogous OIL primitives and DAML+OIL elements. Annotations.

2.2. OWL Species

Referring to the tables above we can now point out with more details the distinguishing features of the three OWL species.

2.2.1. OWL Full

It contains all the OWL constructs and, moreover, provides free, unconstrained use of RDF/RDFS constructs. For instance, OWL Full allows classes to be treated as individuals and all data values are also considered to be part of the individual domain.

2.2.2. OWL DL

It has been said earlier that OWL DL guarantees complete and decidable inferences, respectively intending that all entailments (see section 1) are certainly computed and that all computations will terminate in finite time.
These properties are secured by a number of constraints placed on the use of language constructs:

- Pairwaise separation between classes, datatypes, datatype properties, object properties, annotation properties, ontology properties, individuals, data values, and built-in vocabulary. In other words, all the domains of discourse are kept disjoint.
- Datatype properties cannot be characterized as inverse, functional, inverse-functional, symmetric, or transitive properties. These specifications are reserved to object properties.
- No cardinality constraints can be placed on transitive properties, their inverses, or any of their subproperties.
- Annotations are allowed on classes, properties, individuals, and ontology headers (i.e. the introductory sections of an ontology) only. Furthermore they have to have a well-formatted typing triple (see [OWL Reference] for details). At last, property axioms cannot be used with annotation properties.
- All axioms (i.e. assertions) must be well-formed, thus implying that all classes and properties that one refers have to be explicitly typed as OWL classes or properties, respectively. Equally, facts (i.e. axioms about individuals) concerning with individual equality and difference have to be about named individuals.

2.2.3. OWL Lite

In addition to the restrictions on OWL constructs prescribed by OWL DL, OWL Lite imposes some further constraints:

- Enumeration, disjunction, negation, existential quantification on individuals, class disjunction, and datatype range are forbidden.
- Class equivalence, Class hierarchies, class conjunction have to be asserted w.r.t. named classes or restrictions, only.
- Existential and universal quantifications have to regard named classes or named datatypes, only.

- Individuals instantiations have to be asserted w.r.t. named classes or restriction, only.
- Property domains have to refer to named classes, only.
- Property ranges have to refer to named classes or datatypes, only.

As it is stressed in [Horroks et al., 2003], the just quoted requirements let OWL DL and OWL Lite retain decidability. More precisely, as well as OIL was crafted to maintain a direct mapping with the $\mathcal{SHIQ}$ Description Logic (check [OIL Technical Report] for more details), also OWL DL and OWL Lite are very close to some Description Logics tightly influenced by $\mathcal{SHIQ}$. Namely, OWL DL is very close to $\mathcal{SHOIN}$(D) Description Logic, while OWL Lite is similar to $\mathcal{SHIF}$(D) Description Logic. Hence, the complexity of reasoning in OWL DL and OWL Lite can be referred to the complexity of inference problems faced by the respectively equivalent Description Logics. As those DLs are aimed to deal with complete and decidable inferences, OWL DL and OWL Lite keep completeness and decidability for their reasoning problems, too

2.3. Reasoning in OWL

$\mathcal{SHOIN}$(D) is a Description Logic derived from $\mathcal{SHOQ}$(D), in that $\mathcal{SHOQ}$(D) is extended with inverse roles ($\mathcal{I}$) and restricted to unqualified number restrictions ($\mathcal{N}$ rather than $\mathcal{Q}$).
$\mathcal{SHOQ}$(D) is in turn a member of the $\mathcal{SH}$ family of DLs, while it adds the ability to define a class by enumerating its instances and support for datatype and values (hence, the suffix $\mathcal{O}$ (D)).
Similarly, $\mathcal{SHIF}$(D) augments the $\mathcal{SH}$ core with features and datatypes (hence, the suffix $\mathcal{F}$(D)).
$\mathcal{SH}$ stands for a DL which adds role hierarchies to the logic $\mathcal{S}$, which is an alternative name for $\mathcal{ALC}_{\mathcal{R}+}$.
In the end, $\mathcal{ALC}_{\mathcal{R}+}$ is an extension of the ultimate kernel $\mathcal{ALC}$ (earlier introduced), in that it includes transitively closed primitive roles ($\mathcal{R}^{+}$).

Concerning with complexity, inference in $\mathcal{SHOQ}$(D) is of worst-case non-deterministic exponential time, whereas inference in $\mathcal{SHIF}$(D) is of worst-case deterministic exponential time.
In [OWL Abstract Syntax and Semantics, 2002], they provide a model-theory for OWL. At this point, we can extract from that theory the fragments regarding OWL DL ontology language as an ultimate proof for the close binding with DLs. Tables 6 and 7 show the mappings between each OWL DL language element (except for annotations) and the correspondent DL description, further clarified by model-thoretic semantics.
The symbology for the quoted tables is committed to the following notations:

Δ	*(set of class names)*
Δ_D	*(set of datatype names)*
$C, C_1, C_2,$	*(class -abstract domain- descriptions)*
$D, D_1, D_2,$	*(datatype -concrete domain- description)*
$I(\cdot)$	*(interpretation function from a generic C to Δ)*
$I_D(\cdot)$	*(interpretation function from a generic D to Δ_D)*
$o, o_1, o_2, \ldots, o_l$	*(class individuals -objects- identifiers)*
$v, v_1, v_2, \ldots, v_m$	*(datatype values)*

#M	*(cardinality of a set M)*
R, R_1, R_2	*(role -object property- descriptions)*
U, U_1, U_2	*(attribute -datatype property- descriptions)*
$\Rightarrow$	*(logical implication)*

Basic constructs			
OWL DL element	**DL description**	**Interpretation**	**Logical meaning**
owl:Thing	T	$I(T) = I(\Delta) \cup I_D(\Delta_D)$	top concept
owl:Nothing	$\perp$	$I(\perp) = \varnothing$	bottom concept
owl:Class	C	$I(C) \subseteq I(\Delta)$	primitive concept (class)
owl:DataRange	D	$I(D) = I_D(D) \subseteq I_D(\Delta_D)$	concrete domain (datatype)
owl:ObjectProperty	R	$I(R) \subseteq I(\Delta) \times I(\Delta)$	role (property)
owl:DatatypeProperty	U	$I(U) \subseteq I(\Delta) \times I_D(\Delta_D)$	concrete domain attribute (datatype property)
owl:inverseOf	R^-	$I(R^-) = (I(R))^-$	inverse role
owl:SymmetricProperty	^-R	$I(^-R) = {}^-(I(R))$	simmetric role
owl:TransitiveProperty	R^+	$I(R^+) = (I(R))^+$	transitive role
owl:intersectionOf	$C_1 \prod C_2$	$I(C_1 \prod C_2) = I(C_1) \cap I(C_2)$	conjunction
owl:unionOf	$C_1 \cup C_2$	$I(C_1 \cup C_2) = I(C_1) \cup I(C_2)$	disjunction
owl:complementOf	$\neg C$	$I(\neg C) = I(\Delta) \setminus I(C)$	negation
owl:one of	$\{o_1, o_2, \ldots, o_l\}$ $\{v_1, v_2, \ldots, v_m\}$	$I(\{o_1, o_2, \ldots, o_l\}) = \{I(o_1), I(o_2), \ldots, I(o_n)\} \subseteq I(\Delta)$ $I(\{v_1, v_2, \ldots, v_m\}) = \{I_D(v_1), I_D(v_2), \ldots, I_D(v_m)\} \subseteq I_D(\Delta_D)$	enumeration
owl:Restriction… *owl:onProperty*	$R.C$ $U.D$	$I(R.C) = \{o_1 \text{ s.t. } \forall o_2 \in I(C) \Rightarrow o_2.\ (o_1, o_2) \in I(R)\}$ $I(U.D) = \{o \text{ s.t. } \forall v \in I_D(D) \Rightarrow v.\ (o, v) \in I(R)\}$	property restriction
owl:someValuesFrom	$\exists R.C$ $\exists U.D$	$I(\exists R.C) = \{o_1 \text{ s.t. } \exists o_2.\ (o_1, o_2) \in I(R) \wedge o_2 \in I(C)\}$ $I(\exists U.D) = \{o \text{ s.t. } \exists v.\ (o, v) \in I(U) \wedge v \in I_D(D)\}$	existential quantification
owl:allValuesFrom	$\forall R.C$ $\forall U.D$	$I(\forall R.C) = \{o_1 \text{ s.t. } \forall o_2.\ (o_1, o_2) \in I(R) \Rightarrow o_2 \in I(C)\}$ $I(\forall U.D) = \{o \text{ s.t. } \forall v.\ (o, v) \in I(U) \Rightarrow v \in I_D(D)\}$	universal quantification
owl:hasValue	$R : o_2$ $U : v$	$I(R : o_2) = \{o_1 \text{ s.t. } (o_1, I(o_2)) \in I(R)\}$ $I(U : v) = \{o \text{ s.t. } (o, I(v)) \in I(U)\}$	universal quantification on individuals
owl:Cardinality	$= n\ R$ $= n\ U$	$I(\geq n\ R) = \{o_1 \text{ s.t. } \#(\{o_2.\ (o_1, o_2) \in I(R)\}) = n\}$ $I(\geq n\ U) = \{o \text{ s.t. } \#(\{v.\ (o, v) \in I(U)\}) = n\}$	number restriction
owl:maxCardinality	$\geq n\ R$ $\geq n\ U$	$I(\geq n\ R) = \{o_1 \text{ s.t. } \#(\{o_2.\ (o_1, o_2) \in I(R)\}) \geq n\}$ $I(\geq n\ U) = \{o \text{ s.t. } \#(\{v.\ (o, v) \in I(U)\}) \geq n\}$	qualified number restriction
owl:minCardinality	$\leq n\ R$ $\leq n\ U$	$I(\leq n\ R) = \{o_1 \text{ s.t. } \#(\{o_2.\ (o_1, o_2) \in I(R)\}) \leq n\}$ $I(\leq n\ U) = \{o_1 \text{ s.t. } \#(\{v.\ (o, v) \in I(U)\}) \leq n\}$	qualified number restriction

Table 6. DL syntax and semantics for OWL DL language elements. Basic constructs.

Derived constructs and axioms			
OWL DL element	**DL description**	**Interpretation**	**Logical meaning**
rdfs:subClassOf	$C_1 \subseteq C_2$	$I(C_1) \subseteq I(C_2)$	concept (class) hierarchy
owl:equivalentClass	$C_1 = C_2$	$I(C_1) = I(C_2)$	extensional equivalence between concepts
owl:disjointWith	$C_1 \sqcap C_2 = \perp$	$I(C_1) \cap I(C_2) = \varnothing$	concept (class) disjunction
rdfs:domain	$\geq 1\ R \subseteq C$ $\geq 1\ U \subseteq C$	$I(R) \subseteq I(C) \times I(\Delta)$ $I(U) \subseteq I(C) \times I_D(\Delta_D)$	role subject (property domain)
rdfs:range	$C_1 \subseteq \forall R.C$ $D_1 \subseteq \forall U.D$	$I(R) \subseteq I(\Delta) \times I(C)$ $I(U) \subseteq I(\Delta) \times I_D(D)$	role object (property range)
rdfs:subPropertyOf	$R_1 \subseteq R_2$ $U_1 \subseteq U_2$	$I(R_1) \subseteq I(R_2)$ $I(U_1) \subseteq I(U_2)$	role (property) hierarchy
owl:FunctionalProperty	$C \subseteq \leq 1\ R$ $D \subseteq \leq 1\ U$	$I(R) \subseteq I(\Delta) \times I(C)$ $I(U) \subseteq I(\Delta) \times I_D(D)$	feature (functional property)
owl:InverseFunctionalProperty	$C \subseteq \leq 1\ R^-$	$I(R^-) \subseteq I(\Delta) \times I(C)$	inverse feature (inverse, funtional property)
owl:equivalentProperty	$R_1 = R_2$ $U_1 = U_2$	$I(R_1) = I(R_2)$ $I(U_1) = I(U_2)$	extensional equivalence between roles
rdf:type	$C(o)$ $D(o)$	$o \in I(C)$ $v \in I_D(D)$	individual instantiation
owl:sameAs	$o_1 = o_2$	$I(o_1) = I(o_2)$	intensional equality between individuals
owl:differentFrom	$o_1 \neq o_2$	$I(o_1) \neq I(o_2)$	intensional disequality between individuals
owl:AllDifferent…owl:distinctMembers	$o_1 \neq o_2 \neq \ldots \neq o_l$	$I(o_1) \neq I(o_2) \neq \ldots \neq I(o_l)$	UNA

Table 7. DL syntax and semantics for OWL DL language elements. Derived constructs and axioms.

3. ODL$_{I3}$

3.1. Overview

As our investigation about Query Reformulation is aimed to provide a solution to some open issues concerning with both the DAML-S broker developed at Intelligent Software Agent Laboratory (Carnegie Mellon University, USA) and the Brokering Agent designed by the SEWASIE consortium (EU), it urges to be compliant with the modelling language adopted within the SEWASIE sytstem, too. That language is ODL$_{I3}$ (see chapter 3).
Despite its close relativeness to ODL (refer to [ODL]), the object data modeling language ODL$_{I3}$ introduces some extension w.r.t. ODL.
The capability of supporting value-types motivated the creation of ODL as an extended Description Logic. Generally speaking, the user can dispose of the following ODL's features:

- a system of *base types*, such as *string*, *boolean*, *integer*, *real*;
- the type constructors *tuple*, *set*, and *class* which make it possible to render *class types* -or *classes*-(i.e. sets of *objects* with an identity and a value) and *value types* (i.e. sets of complex, finitely nested *values* without object identity);
- the *intersection* operator, allowig simple and multiple inheritance between types;
- named types, specifically *primitive* (i.e. the implementor has to resolve the interpretation of the element declaring its membership to a given class) or *virtual* (i.e. the interpretation of the element can be automatically computed).

ODL$_{I3}$ was subsequently crafted for the description of both the local source ontologies and the global ontology of an SINode (see chapter 3). Therefore, it enhanced ODL with the following characteristics:

- the *union* operator, to express disjoint data structures in the definition of a class;
- the *optional* operator, to specify that an attribute (i.e. a qualifying property for a type) may be optional for an individual;
- integrity constraints rules in the form of *if then* rules;
- *intensional* relationships through terminalogical relationships between classes and attributes names (i.e. terms), which are:
  - o *SYN* (SYNonym-of), expressing synonymy between two terms;
  - o *BT* (Broader Terms), to say that a term has a broader, more general meaning than another;
  - o *NT* (Narrow Terms), for hyponymy between two terms, that is the opposite of the forementioned hypernymy;
  - o *RT* (Related Terms), or positive association to state that two terms are generally used together in the same context;
- *extensional* relationships, one for each intensional relationship in order to strengthen them at the instance level, by imposing that the corrispondent terminological relationship applies on all the individuals of the named classes/attributes;
- *mapping* rules, in order express relationships holding between the global ontology and the substrating local ontologies.

3.2. Reasoning in ODL$_{I3}$

To perform inferences that will be useful for semantic integration, ODL$_{I3}$ descriptions are translated into a Description Logic language merely internal to SEWASIE. That DL is $\mathcal{OLCD}$. As a matter of fact, the semantics formalized by $\mathcal{OLCD}$ may be interpreted as an extension to ODL features, too. In particular, $\mathcal{OLCD}$ augments ODL with:

- the *comparison* operator;
- the *union* operator;
- the *complement* operator;
- the *existential* quantificator;
- the *universal* quantificator;
- *attribute path* specifications (i.e. dot-separated sequences of attributes expressing navigation through class types and value types), but not *path relations* (i.e. comparisons between two attribute paths);
- the capability of expressing cyclic descriptions (i.e. circular references in type name descriptions)

After all, we can identify the Description Logic $\mathcal{OLC}$ (earlier formalized) as the common

substratum both for $\mathcal{OLCD}$ -which increments the expressiveness of $\mathcal{OLC}$ with cyclic descriptions ($\mathcal{D}$)- and for ODL -thus, for ODL$_{I3}$, too.
The translation from ODL$_{I3}$ to $\mathcal{OLCD}$ is then straightforward. As we said, that is the way to figure out reasoning in ODL$_{I3}$.

4. Comparative Analysis

4.1. Synopsis for OWL DL, ODL$_{I3}$, and relative DLs

The ultimate scope of our digression about ODL$_{I3}$ is the comparison of the semantics associated to OWL DL with those formalized by ODL$_{I3}$ itself. A positive response from this analysis implies that our approach to Query Reformulation can be supported by ODL$_{I3}$ ontologies and, by turn, that required inference services for Query Reformulation can be performed by an $\mathcal{OLCD}$-based reasoning tool.
Table 8 ummarizes the comparison among OWL DL and ODL$_{I3}$ elements. On the right side, it is also reported whether each description is interpretable according to the respecitvely underlying DLs.

ODL$_{I3}$ construct/axiom	**OWL DL construct/axiom**	**Logical meaning**	**Interpretable with $\mathcal{SHOIN}$(D)**	**Interpretable with $\mathcal{OLCD}$**
-	*owl:Thing*	top concept	♦	♦
-	*owl:Nothing*	bottom concept	♦	♦
interface	*owl:Class*	concept (class)	♦	♦
view				
B	*owl:DataRange*	concrete domains (datatypes or base types)	♦	♦
attribute	*owl:ObjectProperty*	role (property)	♦	♦
	owl:DatatypeProperty	concrete domain attribute (datatype property)	♦	♦
-	*owl:inverseOf*	inverse role	♦	-
-	*owl:SymmetricProperty*	simmetric role	♦	-
-	*owl:TransitiveProperty*	transitive role	♦	♦
key	*owl:FunctionalProperty* *(in OWL DL, not applicable to Datatype Properties)*	feature (functional property for a concep, or key for a data structure)	♦	-
foreign key … references				
and	*owl:intersectionOf*	conjunction	♦	♦
union	*owl:unionOf*	disjunction	♦	♦
-	*owl:complementOf*	negation	♦	♦

*	-	optional attribute	-	♦
range	*owl:oneOf*	enumeration	♦	♦
enum				
exists	*owl:someValuesFrom*	existential quantification	♦	♦
attribute set	*owl:allValuesFrom*	universal quantification	♦	♦
forall				
-	*owl:hasValue*	universal quantification of individuals	♦	♦
-	*owl:maxCardinality*	qualified number restriction	♦	-
-	*owl:minCardinality*	qualified number restriction	♦	-
:	*rdfs:subClassOf*	concept (class) hierarchy	♦	♦
bt_{ext}	*rdfs:subClassOf*	extensional, terminological hierarchy	♦	♦
nt_{ext}				
syn_{ext}	*owl:equivalentClass*	extensional equivalence between concepts	♦	♦
rel_{ext}	*owl:ObjectProperty*	extensional relation between concepts	♦	♦
-	*owl:disjointWith*	concept (class) disjunction	♦	♦
attribute	*rdfs:domain*	role subject (property domain)	♦	♦
	rdfs:range	role object (property range)		
-	*rdfs:subPropertyOf*	role (property) hierarchy	♦	-
-	*owl:InverseFunctionalProperty*	inverse feature (inverse, funtional property)	♦	-
-	*owl:equivalentProperty*	extensional equivalence between roles	♦	-
rule	*owl:Restriction… owl:onProperty*	property restriction	♦	♦
-	*owl:cardinality*	number restriction	♦	-
-	*rdf:type*	individual instantiation	♦	♦
syn	*owl:sameAs*	intensional, terminological equality between individuals	♦	♦
bt	-	intensional,	-	-

nt	-	terminological hierarchy	-	-
rel	-	intensional, positive association	-	-
mapping rule	*URI reference*	relationship between integrated schema and local sources	-	-
-	*owl:differentFrom*	Intensional un-equality between individuals	♦	♦
-	*owl:AllDifferent… owl:distinctMembers*	UNA	-	-

Table 8. Synoptical comparison between OWL DL, ODL$_{I3}$, and respective DLs.

4.2. Sample Translations

As a concrete example for the asserted correspondences between OWL DL language and ODL$_{I3}$, here we report some excerpts of ontology translations from OWL DL to ODL$_{I3}$ and vice versa. We refer to the Appendix for a larger sample of a translation from an OWL DL ontology to ODL$_{I3}$ descriptions, and for a complete translation from an ODL$_{I3}$ schema to OWL DL triples.

4.2.1. From OWL DL to ODL$_{I3}$

The source fragment for the translation is taken from the "mad cow" OWL DL ontology (available at [“mad cows” OWL ontology]).

```
<owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#white+van+man">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#man"/>
                    <owl:Restriction>
                        <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#drives"/>
                        <owl:someValuesFrom>
                            <owl:Class>
                                <owl:intersectionOf rdf:parseType="Collection">
                                    <owl:Restriction>
                                    <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#has+colour"/>
                                    <owl:someValuesFrom>
                                    <owl:Class>
                                    <owl:oneOf rdf:parseType="Collection">
                                    <owl:Thing
rdf:about="http://cohse.semanticweb.org/ontologies/people#white"/>
                                    </owl:oneOf>
                                    </owl:Class>
                                    </owl:someValuesFrom>
                                    </owl:Restriction>
                                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#van"/>
                                </owl:intersectionOf>
                            </owl:Class>
                        </owl:someValuesFrom>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
```

The correspondent ODL$_{I3}$ descriptions are as follows:

```
interface man ()
  { };
interface white ()
  { };
interface van ()
  { };

rule rule1 forall x in man:
  (  exists x1 in x.drives:
       (  x1 in van  ) and
          (  exists x2 in x1.has+colour:
               (  x2 in white  )  )  )
  then
    x in man+van+white;

// Giving for granted that the range of drives is always of type van, we might
equally write:
// forall x in man:
//   ( exists x1 in x.drives:
//       (  x1.has+colour in white  )  )
//   then
//     x in man+van+white;
```

4.2.2. From ODL$_{I3}$ to OWL DL

Now, the source fragment for the translation are ODL$_{I3}$ descriptions extracted from the "storage" schema (available at [“storage” ODL$_{I3}$ schema]):

```
 interface Manager
( extent managers
  keys name)
{  attribute string                name;
   attribute range {40000, 100000} salary;
   attribute range{1, 15}          level;
};

rule rule1 forall X in Manager:   X.level >= 5 and X.level <= 10
```

```
  then                           X.salary >= 40000 and X.salary <= 60000 ;
```

The corresponding OWL DL triples are the following ones:

```
<owl:DatatypeProperty rdf:ID="name">
  <rdf:type rdf:resource="&owl;#FunctionalProperty"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="salary"/>
<owl:DatatypeProperty rdf:ID="level"/>

<owl:Class rdf:ID="Manager">
  <name rdf:resource="&xsd;#string"/>
  <salary>
    <owl:DataRange>
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#positiveInteger">40000</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">100000</rdf:first>
              <rdf:rest rdf:resource="&rdf;#nil"/>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
    </owl:DataRange>
  </salary>
  <level>
    <owl:DataRange>
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#positiveInteger">1</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">15</rdf:first>
              <rdf:rest rdf:resource="&rdf;#nil"/>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
    </owl:DataRange>
  </level>
</owl:Class>

<owl:Class rdf:ID="MiddleLevelManager">
  <rdfs:subClassOf rdf:resource="#Manager"/>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#level"/>
      <owl:allValuesFrom rdf:resource="#MiddleLevel"/>
    </owl:Restriction>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#level"/>
      <owl:allValuesFrom rdf:resource="#MiddleSalary"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:DataRange rdf:ID="MiddleLevel">
      <owl:oneOf>
        <rdf:List>
```

```
          <rdf:first rdf:datatype="&xsd;#positiveInteger">5</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">6</rdf:first>
              <rdf:rest>
                <rdf:List>
                  <rdf:first rdf:datatype="&xsd;#positiveInteger">7</rdf:first>
                  <rdf:rest>
                    <rdf:List>
                      <rdf:first
rdf:datatype="&xsd;#positiveInteger">8</rdf:first>
                      <rdf:rest>
                        <rdf:List>
                          <rdf:first
rdf:datatype="&xsd;#positiveInteger">9</rdf:first>
                          <rdf:rest>
                            <rdf:List>
                              <rdf:first
rdf:datatype="&xsd;#positiveInteger">10</rdf:first>
                              <rdf:rest rdf:resource="&rdf;#nil"/>
                            </rdf:List>
                          </rdf:rest>
                        </rdf:List>
                      </rdf:rest>
                    </rdf:List>
                  </rdf:rest>
                </rdf:List>
              </rdf:rest>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
</owl:DataRange>

<owl:DataRange rdf:ID="MiddleSalary">
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#positiveInteger">40000</rdf:first>
          <rdf:rest>
            ...
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">45000</rdf:first>
              <rdf:rest>
                ...
                <rdf:List>
                  <rdf:first
rdf:datatype="&xsd;#positiveInteger">50000</rdf:first>
                  <rdf:rest>
                    ...
                    <rdf:List>
                      <rdf:first
rdf:datatype="&xsd;#positiveInteger">52000</rdf:first>
                      <rdf:rest>
                        ...
                        <rdf:List>
                          <rdf:first
rdf:datatype="&xsd;#positiveInteger">55000</rdf:first>
                          <rdf:rest>
                            ...
                            <rdf:List>
                              <rdf:first
rdf:datatype="&xsd;#positiveInteger">60000</rdf:first>
                              <rdf:rest rdf:resource="&rdf;#nil"/>
                            </rdf:List>
                          </rdf:rest>
                        </rdf:List>
                      </rdf:rest>
                    </rdf:List>
                  </rdf:rest>
                </rdf:List>
              </rdf:rest>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
</owl:DataRange>
```

Since the ODL$_{I3}$ construct *range* has not got a direct counterpart among OWL DL constructs, we had to translate it by means of an enumeration (hereby using the constructs *owl:oneOf* and and *rdf:List*). Hence, a comprehensive hard-coded translation is practically too long to be done and showed. Nevertheless, we presume that a software tool implementing a conversion algorithm from ODL$_{I3}$ to OWL DL language will easily achieve such a translation, even producing a huge OWL DL ontology as result.

VII Concept Rewriting

1. Terminology Transparency for Semantic Interoperability

Open systems framework for social interaction posits semantics as a matter of continuous negotiation and evolution in an environment of uncertain and incomplete information, which preserves the autonomy of the information sources, and yet allows collaboration and cooperation in the presence of conflicts.
Another approach to support a more general notion of semantics is to relate the content and representation of information resources to entities and concepts in the real world. To this effort, the limited forms of operations and axiomatic semantics of a particular representational or language framework are not sufficient. So, *semantic interoperability* (see [Ouksel and Sheth, 1999]) should support high-level, context-sensitive information requests over heterogeneous information resources, hiding system, syntax, and structural heterogeneity.
For semantic interoperability achievement, four enablers and capabilities were identified:

- *Terminology (and language) transparency*
  This will allow a user to choose a terminology of his or her choice while allowing the information source to subscribe to a related but different terminology.
- *Context-sensitive information processing*
  So, the information system will recognize or understand the context of an information need and use it to limit information overload, both by formulating more precise queries used for searching information sources and by filtering and transforming the information before presenting it to the user.
- *Rules of interaction mechanisms*
  This is not a standardization of semantics as reference terminologies. Rather, these mechanisms formally specify the format of messages and the data types on communicated semantic and pragmatic information without any infraction on the substance being communicated, and the exchange protocols. These rules provide means for the interacting parties to reach agreements on norms, responsibilities and commitments.
- *Semantic correlation*
  This will allow the representation of semantically related information regardless of distribution and heterogeneity (including various forms of media) by the user or the third party, and their use for obtaining all forms of relevant information anywhere.

From this brief exposition, we conclude that terminologies are a key component for the solution of semantic interoperability.
A *terminology* can be regarded as the vocabulary of a certain domain of discourse. By defining an *ontology* as a specific vocabulary accompanied by relationships used to describe certain aspects of reality, and a set of explicit assumptions regarding the intended meaning of the specified vocabulary (see [Guarino, 1998]), it can been seen that an ontology embodies a terminology while extending it with relationships and some explicit

assertions. Therefore, ontologies are naturally committed to represent knowledge for an information source. Support and use of multiple, independently-developed ontologies is then important for developing scalable information systems with multiple information producers and consumers. One challenging issue in supporting semantic interoperability is how to allow both users both users and providers to subscribe to existing ontologies of their choice or create a new one. Viewing an ontology as a domain specific terminology, we inscribed this issue in the cited problem of *terminology transparency*.
Within this perspective, processing an information request represented in terms of one ontology in an environment with information resources that subscribe to different (but related and relevant) ontologies requires to cope with the general problem of *term rewriting* (that is addressed by [Baader and Nipkow, 1998]). For our present aims, such a huge theoretical and actual issue may be reduced to the theory of *rewriting concepts using terminologies* (that is introduced by [Baader et al., 2000]). This may involve using inter-ontological relationships, such as synonym, hypernym, or homonym, and other possible domain specific relationships. In addition, ad hoc operators may be specified and exploited in order to reformulate an information request on the basis of the terminology understood by some information sources.
In the sequel, we first recall the formal handling of rewriting concepts using terminologies. We then provide some concrete example of major integration systems (like OBSERVER and SIMS) facing concept rewriting for information request processing.

2. Rewriting Concepts Using Terminologies

2.1. Formalization

Given a concept expressed in a source language, *concept rewriting* aims at finding a concept, possibly expressed in a target language, which is related to the given concept according to equivalence, subsumption, or some other relation. In order to specify the rewriting, one can provide a suitable set of constraints between concepts in the source language and concepts in the target language. In other informal words, given a TBox T (i.e., a set of concept definitions) and a concept description C that does not contain concept names defined in T, can this description be rewritten into a “related better” description E by using (some of) the names defined in T?
A purely formal accounting for the concept rewriting problem can be given within the Description Logics (DLs) framework (see chapter 6, section 1).
W.r.t. DLs, a *TBox* T is a finite set of *concept definitions* of the form $A = C$, where $A \in N_C$ is a *concept name* and C is a *concept description* (N_C is the set of concept names). The concept name A is a *defined name* in the TBox T iff it occurs on the left-hand side of a concept definition in T; otherwise, A is called *primitive name*.
For a given DL $\mathcal{L}$, we talk about *$\mathcal{L}$-concept descriptions* and *$\mathcal{L}_s$-TBoxes*, if all constructors occurring in the concept descriptions and concept definitions belong to $\mathcal{L}$.
Let N_R be a set of *role names* and N_P a set of primitive names, and let $\mathcal{L}_s$, $\mathcal{L}_d$, and $\mathcal{L}_t$ be three DLs (the source, destination, and TBox-DL, respectively). A *rewriting problem* is given by

- an $\mathcal{L}_t$-TBox T containing only role names from N_R and primitive names from N_P; the set of defined names occurring in T is denoted by N_D;

- an $\mathcal{L}_s$-concept description *C* using only the names from N_R and N_P;
- a binary relation $\rho \subseteq \mathcal{L}_s \times \mathcal{L}_d$ between $\mathcal{L}_s$- and $\mathcal{L}_d$-concept descriptions.

An $\mathcal{L}_d$-rewriting of *C* using *T* is an $\mathcal{L}_s$-concept description *E* built using names from N_R and $N_P \cup N_D$ such that $C \rho E$.

2.2. Application to Query Processing

When dealing with information integration systems, a *query* is the instantiation of an *information request* . Hence, *information request processing* is delegated to *query processing*. A classification of various approaches for query processing in global information systems is as follows:

- Syntactic *keyword-based* and navigational approaches in which the query is a set of keywords. There are little or no semantics associated with this approach and the user has to do most of the information filtering and correlation. However, this approach is simple to use and support.
- A *global* (common) *ontology-based* approach which supports expression of complex constraints as a part of the query. This involves development and integration of domain specific ontologies into a common global ontology and partitioning it into microtheories. This approach transfers the burden of information correlation and filtering on the query processing system. It can be very difficult to support because of the complexity involved in integrating the ontologies and maintaining consistency across concepts (originally) from different ontologies.
- A group of *loosely coupled* approaches where instead of integrating the pre-existing ontologies, interoperation across them is achieved via terminological relationships represented between terms across the ontologies. Answers are expected to be inferior (w.r.t. the previous approach) as semantic relationships are approximated using terminological ones. However, loosely coupled approaches are scalable, extensible and easier to support.

2.2.1. OBSERVER

OBSEVER (see [Mena et al., 1996]) extends or builds upon some of the loosely coupled approaches by using metadata to capture the information content of the repositories. Intensional descriptions abstract from the structure and organization of the individual repositories as intensional metadata. The user queries the system by expressing his information needs using intensional metadata descriptions represented using the DL-based data model CLASSIC ([Brachman et al., 1991]).
Reminding the semantic interoperability issues mentioned at the beginning of this chapter, the most critical problem in characterizing the information content is that of different vocabularies used to describe similar information across domains. This leads to different terms and constraints being used to characterize similar information. Interoperation across ontologies is achieved by traversing semantic relationships defined between terms across ontologies. User queries are rewritten in a semantics-preserving manner by replacing them with synonym terms from different ontologies; hyponym and hypernym terms can also be used and the loss of information is measured. The key objective of this

approach is to reduce the problem of knowing the structure and semantics of data in the huge number of repositories in a global information system to the significantly smaller problem of knowing the synonym relationships between terms across ontologies.
This decreased complexity is granted by the main assumption behind the system's development. Namely, the number of relationships between terms across ontologies is an order magnitude smaller than the number of terms relevant to the system.
The figure below illustrates the basic elements of OBSERVER's architecture.

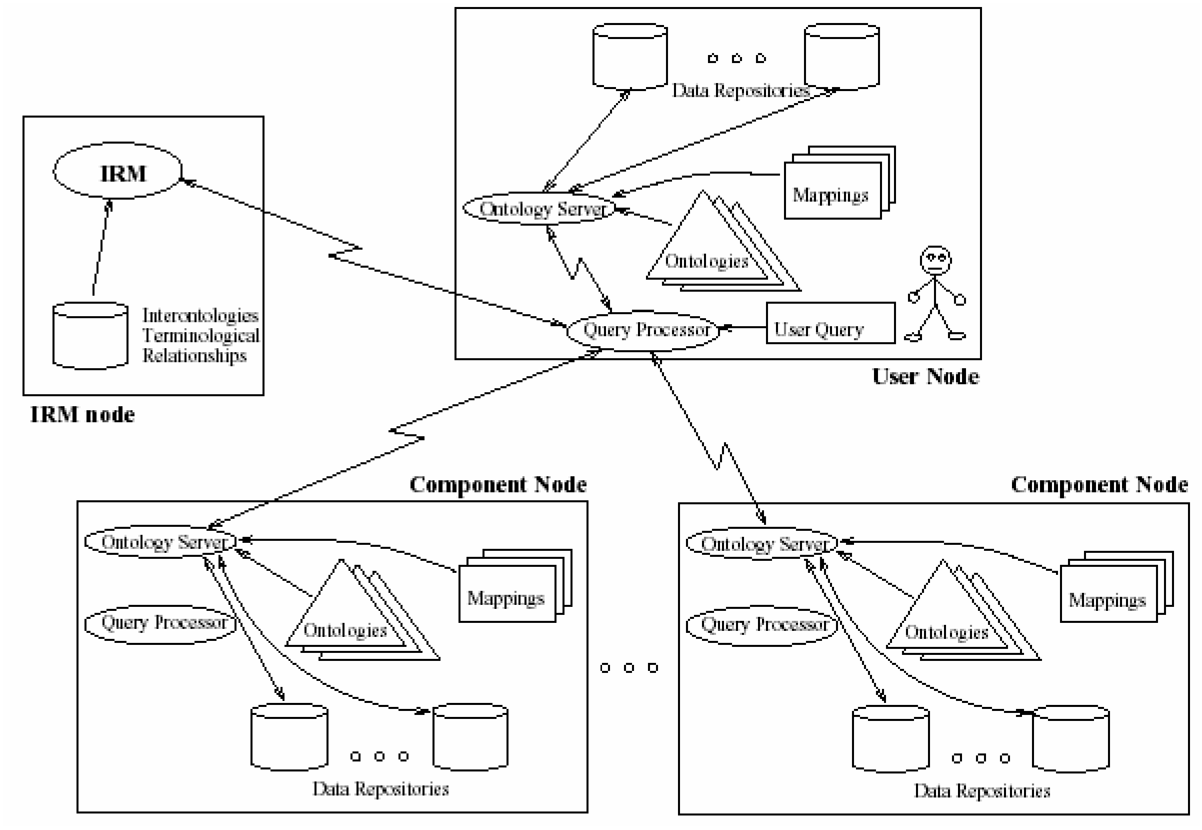


Figure 17. OBSERVER's architecture.

We know proceed ahead with a quick description of the depicted modules effectively performing concept rewriting .

- Query Processor
  It takes as input a user query expressed in CLASSIC using terms from a chosen user ontology. The query processor navigates other component ontologies of the global information system and translates terms in the user query into component ontologies preserving the semantics of the user query. The resulting (sometimes partial) translation of the query at the component ontologies enables identification of the relevant information at the underlying data repositories. Partial translations, if any, are combined at the present ontology with those determined at previous ontologies such that all constraints in the user query are translated.
- Interontology Relationships Manager (IRM)
  Synonyms, hyponym and hypernym relationships between the terms in various ontologies are represented in a declarative manner in an independent repository. The Query Processor is allowed to interrogate the IRM for the consultation of this repository in order to successfully perform term translation.

The algorithm executed to transform a given user query into an equivalent one in terms of

a target ontology is sketched in figure 18. It is worthy to remark that this procedure takes into account only synonym relationships, definition of terms, and equivalent terms.
Summing up in a formal fashion, we can say that:

- concept rewriting traslated from an ontology (i.e. user ontology) to another (i.e. component ontology) both designed with CLASSIC (frame-based) knowledge representation language (we refer to the equivalent DL as CLASSIC DL), hence $\mathcal{L}_s = \mathcal{L}_d = \mathcal{L}_t = \mathit{CLASSIC\ DL}$;
- concept rewriting is based on the terminology (TBox) *T* of the target ontology (i.e. component ontology);
- concept rewriting is pursued according to equivalence ($\equiv$), synonymy (*syn*), hypernymy (*hyper*), hyponymy (*hypo*) relationships between terms of the user ontology and each target ontology in turn, hence $\rho = \equiv$ aut *syn* aut *hypo* aut *hyper*.

2.2.2. SIMS

SIMS (see [Arens et al., 1996]) provides an implementation of the global ontology-based approach.
A shared ontology used to describe the contents of sources available within some application domain is embodied by an appositely created domain model. Loom (ref. to [MacGregor, 1990]) is used as the knowledge representation language to establish a fixed vocabulary describing objects in the domain, their attributes, and the relationships among them.
An information source is incorporated into SIMS by first modelling its own contents and then by relating concepts and roles of the built model to corresponding concepts and roles of the domain model. Further concepts not directly represented by any source may augment the minimal model (i.e. the smallest model that provides sufficient domain-level entities to support reference to every existing information source concept and role) to a more readable though more complex model for the information sources.
SIMS accepts queries in its high-level uniform language. Thus, Loom statements interpreted as queries to SIMS do not need to contain information describing which sources are relevant to finding their answers or where they are located, in that there is no fixed mapping form a query to the sources used to answer it. Queries to SIMS do not even need to state how information obtained from different sources should be joined or otherwise combined or manipulated. It is the task of SIMS to determine how to efficiently and transparently retrieve and integrate the data necessary to answer a query.
Within this framework, query processing is primarily composed by query reformulation. This component identifies the sources of information that are required in order to answer a query and determines how data from them must be combined to produce precisely what the user requested. The user's query expressed in domain terms is thus reformulated into queries to specific information sources. Concept rewriting is the essence of this process.
Steps in the reformulation task consist of application of any of several available reformulation operators. The application of each operator rewrites a number of clauses of the given query into a different, but semantically equivalent, set of clauses. Operators are repeatedly applied until the resulting query explicitly refers to information sources that contain the needed information. Furthermore, the resulting query will make explicit how information from the various sources must be combined to result in an answer to the original query posed to SIMS.
Notice that the applicability of these operators depends on the models, the given query,

and the available information sources.
A brief overview of the reformulation operators can be outlined as follows:

- the choose-source operator selects the appropriate information source corresponding to a domain-level term used in a query;
- queries containing domain clusters (i.e. a collection of clauses that refer to some variable and to any constraint upon it, while at least some clauses being at the domain level) that have roles intended for mapping to different sources are split up into several subqueries, each referring only to attributes present in a single information source, by the decompose operator;
- when dealing with terms of an augmented model rather than a minimal model, the substitute operators use the underlying definition of those terms w.r.t. the minimal model to replace the terms themselves;
- infer-equivalences operators rewrite clauses using concepts and role constraints to figure out equivalence to an alternative query.

Concluding with a formal reference to the logic theory of concept rewriting expressed above, we can say that:

- concept rewriting is from terms of the domain model into terms of any of the information source models, both kinds of models being designed with the same DL (which we address as Loom DL), hence
  $\mathcal{L}_s = \mathcal{L}_d = \mathcal{L}_t =$ *Loom DL;*
- concept rewriting is based on the terminology (TBox) *T* of some of the information sources models;
- concept rewriting is performed through the iterative and combined application of the reformulation operators (i.e., *choose-source*, *decompose*, *substitute*, *infer-equivalences*), hence

$\rho =$ *choose-source* $\circ$ *decompose* $\circ$ *substitute* $\circ$ *infer-equivalences*.

```
Procedure translatePreservingSemantics(user-query,target-ontology)
  begin
  fills=0;
  for each constraint in the user query do
    begin
    for each component of the constraint DO
      begin
      case component is:
        term: /* concept or role */
         if exists synonym from user to target ontology then
           begin
          substitute(term, synonym-of-term);
          if fills then
            begin
            role=term;
            new-role=synonym-of-term;
           end
         end
     else
       begin
       if it is a defined term then
         begin
          substitute(term, definition-of-term);
          translate-preserving-semantics(definition-of-term, target-ontology);
        end
      end
     value:
       if fills and exists transformer function between role and new-role then
         begin
         substitute(value, equivalent-value);
         fills = 0;
         end
    operator: (* all, at-least, fills, … *)
      if it is "fills" then
       begin
       fills := 1;
       end
     else
       begin
       fills := 0;
       end
    end
  end
end
```

Figure 18. Algorithm for query translation in OBSERVER.

VIII Rationale for Query Reformulation

1. Working Hypothesis

- All the actors (i.e., requester, broker, and provider) share a common Knowledge Base (KB)
- The KB must be represented in a FOL compliant language, such that sound, complete, and decidable inferences can be performed.
- Against the shared KB, the requester is allowed to pose queries formalized as a positive literals. The literals must be FOL triples. The query may contain variables.
- Open World Assumption (OWA). The absence of information is indicated by the state of truth *nil* rather than *false*, in that some more information might be added to recover the encountered lack.

While satisfying these conditions, ontologies have been chosen to formalize the common KB, and OWL DL has been assumed as the ontology description language (check section 2, chapter 6 for more details about OWL DL).

2. Formal Conventions

In order to successfully undertake the semantic matching, the broker must abstract the query into a request for a service. This service is described by means of parameters that are specified in the supported KB.
The KB is organized as an ontology.
Each statement of the KB is a positive literal expressed using a FOL-based language. As it has been assumed that the ontology is written in OWL DL, each sentence of the KB is a FOL triple.
The minimal request expressible in terms of the ontology must be a sentence of the ontology itself. That triple-like sentence is the description of a binary predicate.
Let us properly formalize this logical theory according to a fragment of FOL.

2.1. Syntax

The signature consists of the countable infinite supply of symbols:

- *Variable* symbols: *n*, *o*, *p*, …, *x*, *y*, *z*
- *Individual constant* symbols: *a*, *b*, *c*, …, *k*, *l*, *m*
- *Unary predicate* symbols: *A*, *B*, *C*, …, K, L, M

- *Binary predicate* symbols: *N*, *O*, *P*, …, *X*, *Y*, *Z*
- A *term t* is defined as follows:
  $t \rightarrow x$ | (variable)
  a | (constant)
  A | (unary predicate)
- An *atomic formula* ϕ is defined as follows:
  $\phi \rightarrow P(t_1, t_2)$ |
  $A(a)$
- The *entailment* , or *logical implication*, is indicated by $\models$,
  while an *inference*, or *inferential calculation*, or *reasoning service*, is indicated by $\vdash$ (the arrow used so far, i.e. →, standed for *syntactic derivation*).
- Besides the notation we have just reported, we are committed to the conventional symbology for logic and set theory writings.

2.2. Alternative Terminology

A *triple* is an atomic formula where the predicate symbol is a binary predicate.
An atomic formula where the predicate symbol is a unary predicate is regarded as the instantiation of an individual constant.
Hereafter, being compliant with Object Oriented languages and Description Logics terminology, we will freely use words such as *individual*, or *instance* to call individual constant. Similarly, we will name unary predicate as *entity*, *concept*, or *class*. A binary predicate might then be deliberately called *role*, *property*, or *relation*. At last, atomic formula is taken as a synonym for *atom*, *literal*, *sentence*, or *statement*.
The query to be processed is formalized as triple. Therefore we can always identify the logic subject and the logic object of the binary predicate constituting the query. Hereafter, when arguing about a query we will reference the logical subject and the logical object simply as the *subject* and the *object* of the query, while the binary predicate will be simply addressed as the *predicate* of the query.
E.g., the query

$$Brother(kingJohn, x)$$

is a statement where the variable *x* is the object of a the predicate *Brother* with subject *kingJohn*.

3. Query Reformulation

Once the query has been posed to the broker by the requester it must go through a three-phase processing (i.e. *Query Processing*) in order to be successfully submitted to a provider.

1. Query Abstraction
2. Semantic Matching
3. Query Rewriting

Query Abstraction and *Query Rewriting* constitute the original contribution of our work. As it has been addressed in chapter 5, *Query Reformulation* consists of them both as a whole.

3.1. Query Abstraction

This process ends up with an abstracted query, i.e. each term of the query is a concept rather than an instance or a variable. The resulting abstracted query is the request employed by the broker for the semantic matching with the stored advertisements (see chapters 4 and 5).
Query Abstraction clearly depends on the shape of the initial query. We enumerate all the possible cases for the initial query formulation:

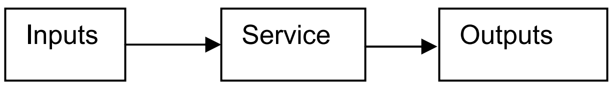
1. P(A, B)
2. P(A, b)
3. P(a, B)
4. P(a, b)
5. P(A, y)
6. P(x, B)
7. P(x, y)
8. P(a, y)
9. P(x, b)

Case analysis:

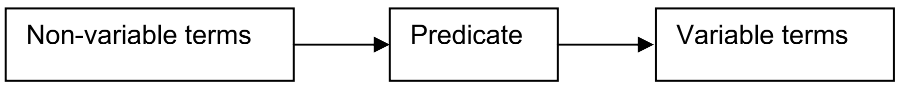
- doesn't require any abstraction, since all the terms in the query already represent concepts;
- (2), (3), (4) are analogous, since they require abstraction from instances to concepts;
- (5), (6), (7) need abstraction from variable to concepts;
- (8), (9) need abstraction from both instances or variable to concepts.

3.1.1. Abstraction and Service Representation

Keeping in mind that the abstracted query is equivalent to a request for a service, we must identify the parameters describing the service within the query itself.
The easiest way to characterize a service is describing it as a transformation from some input parameters to some output results.



To state a correspondence between a query/request and a service we must extract the service parameters from the query/request. Considering the known information as *inputs* and the information to be retrieved as *outputs*, it is straightforward to regard the non-variable terms in the query as inputs and the variable terms in the query as outputs for the service. In the same fashion, the predicate of the query can be viewed as a sort of shallow description of the transformation accomplished by the service.



Queries where no variable is specified are *Boolean queries*, in that there is no information to be retrieved in order to bind some unknown parameter (i.e., variable).

3.1.2. Abstraction Process

Reminding the query analysis we have done above, we must deal with two alternative kinds of abstraction:

- from instance to concept (*Instance Abstraction*);
- from variable to concept (*Variable Abstraction*).

3.1.2.1. Instance Abstraction

Given an individual name and the reference ontology, we can usually infer with sound and decidable reasoning the classes wherefrom the named individual was instantiated. So, we abstract the given individual to the closest class in the ontology hierarchy that comprises the individual itself.
If such a class existis but is not computable (i.e. it cannot be inferred even though it is entailed by the KB, thus meaning that reasoning is not complete), or it cannot be correctly inferred (i.e. the computed assertion is not entailed by the KB, thus meaning that reasoning is not sound), or it cannot be calculated in a finite time (i.e. the current inference is not decidable), then Instance Abstraction fails.

Query: P(a, B)
if $\exists A$ s.t. $a \vdash A(a) \wedge KB \models A(a)$
then Abstract(a) = A.
Hence,
abstracted query: P(A, B)

3.1.2.2. Variable Abstraction

This time we cannot substitute the variable term with a class term in the query by reasoning on the considered term only. We don’t have enough clues to deduct an acceptable concept for the given variable looking at the given variable only. Hence we widen our glimpse at the other elements of the query. A useful element is the predicate. In fact we always know the logical meaning of the considered term w.r.t. the predicate of the query. Any term is committed to be either the subject or the object of the query (see case analysis above).
We remind that, parsing the ontology, on a side we can usually deduce which classes are entailed to belong to the domain of a given property, and, on the other side, we can usually deduce which classes are entailed to belong to the range of a given property.
Therefore, in case the variable of the query to be abstracted is the subject of the predicate, we substitute it with *a* class derived with sound and decidable reasoning as a domain class for the given predicate in the reference ontology. By the other hand, in case the variable term to be abstracted is the object of the query predicate, we substitute it with *a* class derived with sound and decidable reasoning as a range class for the given predicate in the ontology. Once more, we stress that if such inferences are neither sound, or decidable, then Variable Abstraction fails.

- *Query: $P(x, B)$*
  if $\exists A_i \in \{A_1, A_2, \ldots, A_n\}$ s.t. $\exists C: P \vdash P(A_i, C) \wedge KB \models P(A_i, C)$
  then Abstract(x) = A_i.
  Hence,
  abstracted query: $P(A_i, B)$
- *Query: $P(A, y)$*
  if $\exists B_i \in \{B_1, B_2, \ldots, B_m\}$ s.t. $\exists C: P \vdash P(B_i, C) \wedge KB \models P(B_i, C)$
  then Abstract(y) = B_i.
  Hence,
  abstracted query: $P(A, B_i)$

3.2. Query Rewriting

This process is aimed to supply the provider with a fully understandable query.
Semantic Matching guarantees a strict correspondence between the abstracted query's (i.e., request's) outputs and the provider's advertisement's outputs, whereas such a correspondence is not guaranteed with regard of the respective inputs. So, Query Rewriting must be committed to transform the request's inputs into comprehensible inputs for the chosen provider.
Once the provider has been selected, we know the required inputs for the fulfilment of the requested service in terms of classes of the reference ontology. On the opposite side, the inputs supplied by the requester's query can be either classes or individuals (see *Abstraction and Service Representation* above).
Besides, since we are dealing with FOL triples only, we can safely identify the correspondence between the terms in the request and terms in the advertisement. Namely, the subject term in the request will correspond with the subject term in the advertisement, while the object term in the request will correspond with the object term in the advertisement. Hence, there's no chance of misleading correspondence between request's inputs/outputs and advertisement inputs/outputs.
We can now list down the possible cases for the relationship between the requester's input and the correspondent provider's input.

Case	Requester's Input	Provider's Input
1	A	A
2	B	A
3	$a \rightarrow A(a)$	A
4	$b \rightarrow B(b)$	A

Case analysis:
- evidently, (1) doesn't need any rewriting;
- (2) requires the rewriting of a concept into a different concept;
- (3) doesn't need any rewriting;
- (4) needs the rewriting of an instance into an instance of a different class.

It is worthy to further explain that case (3) doesn't require any rewriting because the instance *a*, that is the input in the requester's query, belongs to the same class advertised by the provider as the required input. Even though the individual term in the requester query was previously abstracted into a class in order to perform the semantic matching,

that individual term can be submitted as it is to the provider once the semantic matching has been successfully performed. Advertising class *A* as the required input, the provider actually expects for any instance of the class *A* as the concrete input for the service performance.
Case (2) and (4) put in evidence how the rewriting process can be broken down into two simpler sub-processes:

- transforming of a concept into another concept (Concept Transformation);
- rewriting of an instance into another instance (Instance Rewriting).

It actually turns out that case (2) simply requires concept transformation, whilst case (4) requires both concept transformation and the subsequent instance rewriting.

3.2.1. Concept Transformation

Given a root concept to be transformed and the target concept for the transformation, the scope of this process is to find the way to pass from the root concept to the target concept. Since an ontology represents the relationships between two classes as properties (i.e. binary predicates), transforming a root class into the target class essentially means to find a property linking those two classes. We are not interested in the verse of the considered property, i.e. the root class may belong to the property domain as well as to the property range while the target class may respectively belong to the property range as well as to the property domain. We just want to find a connection between two classes, no matter which verse this connection goes. Once we have found such a binding, we can replace the root class with the target class. Having the root class, the target class, and a property relating them we can always move from a class to the other one and vice-versa by exploiting the property.
This process can be iterated along a path of classes. In fact, a one-step path is the easiest case: we are able to find a property immediately relating the root class to the target class. Unfortunately, that doesn't happen so frequently. So we must step through some ulterior classes in order to reach the target class from the root class. Keeping track of each *(root class, property, target class)* triple composing the path, we can easily backtrack and forth the path itself , i.e. the concept transformation process is always reversible. This is the fundamental condition for the process to be applicable. In case the process were not reversible, it would be ambiguous too. In other words, an irreversible concept transformation would be irresolvable since we would not know where to or where from transform a given concept.
Therefore, it is necessary to record all the steps of the ation path. The symbolic representation to achieve the path tracking is a tree graph where the nodes are the related classes and the links are the relating properties.
The final question naturally arising is about the kind of properties we are dealing with. As we assumed to address either classes, properties, and individual to an ontology, we can basically handle two kinds of properties:

- *is-a* links;
- *non-is-a* links.

Sample properties of the first type are subsumption and equivalence relationships but also conjunction operated in a certain manner.
Sample properties of the second type are any designer-defined property but also disjunction and conjunction, if properly instantiated.
Here we have given just some hints about this distinction. A deeper investigation will turn

out to be meaningful in the next section (i.e., *Instance Rewriting*).
As we have done with Query Abstraction, in the following we introduce a formal description of *Concept Transformation*.

Root concept: $A = A_0$, target concept: $B = A_n$.
A can be transformed into B iff
$\exists P_1$ *s.t.* $(P_1 \vdash P_1(A_0, A_1) \wedge KB \models P_1(A_0, A_1)) \vee (P_1 \vdash P_1(A_1, A_0) \wedge KB \models P_1(A_1, A_0))$,
$\exists P_2$ *s.t.* $(P_2 \vdash P_2(A_1, A_2) \wedge KB \models P_2(A_1, A_2)) \vee (P_2 \vdash P_2(A_2, A_1) \wedge KB \models P_2(A_2, A_1))$,
…
$\exists P_n$ *s.t.* $(P_n \vdash P_n(A_{n-1}, A_n) \wedge KB \models P_n(A_{n-1}, A_n)) \vee (P_n \vdash P_n(A_n, A_{n-1}) \wedge KB \models P_n(A_n, A_{n-1}))$,
Hence,
$Path(A_0, A_n) = Path(A_0.A_1.A_2. \ldots .A_{n-1}.A_n)$
where $(A_{i-1}.A_i. \ldots)$ is the standard dot-separated notation for path navigation.
$n \geq 1$ is the length of the path (obviously, $n = 0$ iff $A \equiv B$).

A path can be indeed expanded as a set of interrelated triples:
$Path_{Triples}(A_0, A_n) = \{(A_0, P_1, A_1), (A_1, P_2, A_2), \ldots, (A_{n-1}, P_n, A_n)\}$.
Since $(P_i \vdash P_i(A_{i-1}, A_i) \wedge KB \models P_i(A_{i-1}, A_i)) \vee (P_i \vdash P_i(A_i, A_{i-1}) \wedge KB \models P_i(A_i, A_{i-1}))$,
for any path triple (A_{i-1}, P_i, A_i) we can define the function
$\mathcal{P}_S : A_{i-1} \rightarrow A_i$ *with* $i = 1, \ldots, n$
as the path step transformation.
Thereby, we define the function
$\mathcal{P} : A_0 \rightarrow A_n$, $\mathcal{P} = \mathcal{P}_S(A_1) \circ \mathcal{P}_S(A_2) \circ \ldots \circ \mathcal{P}_S(A_{n-1}) \circ \mathcal{P}_S(A_n)$,
where $\circ$ is the standard symbol of function composition,
as the path transformation.

Although the path tranformation $\mathcal{P}$ is effectively obtained by the joint application of binary predicates, we could not define $\mathcal{P}$ itself as a binary predicate for we don't know which is the logical subject and which is the logical object of $\mathcal{P}$. In fact, despite of the directionality of $\mathcal{P}$ (that is, from A_0 towards A_n) it might happen that both A_0 and A_n are domains or ranges of different predicates realizing some path step tranformation $\mathcal{P}_S$ (e.g., $P_1 \vdash P_1(A_0, A_1), \ldots, P_n \vdash P_n(A_n, A_{n-1})$). The definition of $\mathcal{P}$, and more specifically the definition of $\mathcal{P}_S$, beyond notifying that a path transformation essentially results from the combined application of some predicates P_i, confer a functional characterization to the Concept Transformation process.
For instance, if we exploit the predicate P_i to go from the concept A_{i-1} to the concept A_i and we want to move further from the concept A_i to the concept A_{i+1} then we will apply a further predicate P_{i+1} to the concept A_i in order to finally reach the concept A_{i+1}. We have earlier summarized this iter with the combined application of the function $\mathcal{P}_S$ that enables the concept A_{i-1} to be substituted with the concept A_{i+1}. Doing that, we don't specify any logical aspect of the global transformation arguments -that are A_{i-1} and A_{i+1}.
Ultimately, the path transformation $\mathcal{P}$ is a sort of black box that receives a concept A_0 as input and returns a related concept A_n as output. The logical account is relevant to the predicates used to get the path step transformation $\mathcal{P}_S$. Just operating with such predicates we are forced to specify and keep notice of the respective subjects and objects.

3.2.2. Instance Rewriting

Given a root instance to be transformed and the target concept for the transformation, the scope of this process is to find the way to pass from the root instance to some instance of the target concept. As Instance Rewriting is performed after Concept Transformation, the path to go from the root concept (i.e., the concept embracing the root instance) to the target concept is well known. Instance rewriting can then be reduced to the task of populating the known path with individuals. By parsing the reference ontology we can actually infer which individuals constitute the current instantiation of a given property. After Concept Transformation we should know which property is relating two given classes. After Instance Rewriting we should know which individuals of the given classes are related through the given property.
Instance Rewriting is successfully accomplished if we are able to find a chain of related individuals along the path of classes produced by Concept Transformation.

Root instance: $a = a_0$,
root concept: $A = A_0$ *s.t.* $a_0 \vdash A_0(a_0) \wedge KB \models A_0(a_0)$,
target concept: $B = A_n$,
$Path_{Triples}(A_0, A_n) = \{(A_0, P_1, A_1), (A_1, P_1, A_2), ..., (A_{n-1}, P_n, A_n)\}$.
a_0 *can be rewritten into an instance* a_n *s.t.* $a_n \vdash A_n(a_n) \wedge KB \models A_n(a_n)$ *iff*
$\exists a_1$ *s.t.* $(P_1 \vdash P_1(a_0, a_1) \wedge KB \models P_1(a_0, a_1)) \vee (P_1 \vdash P_1(a_1, a_0) \wedge KB \models P_1(a_1, a_0))$,
$\exists a_2$ *s.t.* $(P_2 \vdash P_2(a_1, a_2) \wedge KB \models P_2(a_1, a_2)) \vee (P_2 \vdash P_2(a_2, a_1) \wedge KB \models P_2(a_2, a_1))$,
…
$\exists a_n$ *s.t.* $(P_n \vdash P_n(a_{n-1}, a_n) \wedge KB \models P_n(a_{n-1}, a_n)) \vee (P_n \vdash P_n(a_n, a_{n-1}) \wedge KB \models P_n(a_n, a_{n-1}))$.

The success of this operation depends on the type and directionality of properties composing the path and, obviously, on the current set of individuals asserted for the reference ontology.
Let us point out how the property type, accompanied by concept non-empty extension, can be a sufficient condition of the effective realization (i.e. instantiation) of the relationship between two concepts.

Path step predicate: $P_i(A_{i-1}, A_i)$ *with* $i = 1, ..., n$
Path step root concept: A_{i-1}, *path step target concept:* A_i
Non-empty concept extension: $\exists a_{i-1}$ *s.t.* $a_{i-1} \vdash A_{i-1}(a_{i-1}) \wedge KB \models A_{i-1}(a_{i-1})$, $\exists a_i$ *s.t.* $a_i \vdash A_i(a_i) \wedge KB \models A_i(a_i)$
P_i *is an is-a link such that* A_{i-1} *is subsumed by* A_i,
i.e., $\forall a_{i-1}$ *s.t.* $a_{i-1} \vdash A_{i-1}(a_{i-1}) \wedge KB \models A_{i-1}(a_{i-1})$
$\exists a_i$ *s.t.* $a_i \vdash A_i(a_i) \wedge KB \models A_i(a_i)$, $a_{i-1} = a_i$
Hence,
$P_i(A_{i-1}, A_i) \vdash P_i(a_{i-1}, a_i)$

Clearly, this condition is still valid if A_{i-1} *is subsuming* A_i, though in that case the root concept must be A_i rather than A_{i-1} and the if-clause of the theorem must be reformulated:

P_i *is an is-a link such that* A_{i-1} *is subsuming* A_i,
i.e., $\forall a_i$ *s.t.* $a_i \vdash A_i(a_i) \wedge KB \models A_i(a_i)$
$\exists a_{i-1}$ *s.t.* $a_i \vdash A_{i-1}(a_{i-1}) \wedge KB \models A_{i-1}(a_{i-1})$, $a_i = a_{i-1}$
Hence,
$P_i(A_{i-1}, A_i) \vdash P_i(a_{i-1}, a_i)$

W.r.t. OWL DL language and given for granted non-empty concept extensions, we now list the possible property assertions which make sure the realization (i.e., validation at

instance level) of the link between two concepts.
Assuming *A* as the root class and *B* as the target class, we have

- *A equivalentClass B*,
- *B equivalentClass A*,
- *A intersectionOf B*,
- *A subClassOf B*.

However, Concept Transformation can result with any kind of property. As a consequence, a path at the concept level -signalled to go from a root class *A* to a target class *B*- might not be applicable as it is not completely instantiated. Such a condition must be meant in the sense that there exists at least a path triple $(A_{i-1}, P_i, A_i) \in Path(A, B)$ for which
either

a) $(not \exists\, a_{i-1}\ s.t.\ a_{i-1} \vdash A_{i-1}(a_{i-1}) \wedge KB \models A_{i-1}(a_{i-1})) \vee (not \exists\, a_i\ s.t.\ a_i \vdash A_i(a_i) \wedge KB \models A_i(a_i))$,

or

b) $\exists\, a_{i-1}\ s.t.\ a_{i-1} \vdash A_{i-1}(a_{i-1}) \wedge KB \models A_{i-1}(a_{i-1}),\ \exists\, a_i\ s.t.\ a_i \vdash A_i(a_i) \wedge KB \models A_i(a_i)$,
but, $\forall\, a_{i-1}, a_i$ *satisfying the clause above, it never happens that*
$(a_{i-1}, a_i \vdash P_i(a_{i-1}, a_i) \wedge KB \models P_i(a_{i-1}, a_i)) \vee$
$(a_{i-1}, a_i \vdash P_i(a_i, a_{i-1}) \wedge KB \models P_i(a_i, a_{i-1}))$,

or

c) $\exists\, a_{i-1}\ s.t.\ a_{i-1} \vdash A_{i-1}(a_{i-1}) \wedge KB \models A_{i-1}(a_{i-1}),\ \exists\, a_i\ s.t.\ a_i \vdash A_i(a_i) \wedge KB \models A_i(a_i)$,
$(a_{i-1}, a_i \vdash P_i(a_{i-1}, a_i) \wedge KB \models P_i(a_{i-1}, a_i)) \vee (a_{i-1}, a_i \vdash P_i(a_i, a_{i-1}) \wedge KB \models P_i(a_i, a_{i-1}))$,
but, $\forall\, a_{i-1}, a_i$ *satisfying the sentence above, it happens that* a_{i-1} *does not satisfy either of condition a) or b) w.r.t. the preceding triple in the path or* a_i *does not satisfy either of condition a) or b) w.r.t. the next triple in the path*.

We don't prevent a priori the consideration of such a path (i.e., it exists at concept level, but it is not exploitable at instance level) because we are working with the OWA. Simply, when instance rewriting cannot be successfully achieved because the accounted path is not completely instantiated we quit the process.
As a final remark, we notice that concept satisfiability alone is a necessary condition for instance rewriting success. Moreover, if the assertions about the examined concepts are consistent, results after instance rewriting will be consistent as well.

3.2.3. Concept Rewriting Formalization

In the end, we can describe Query Rewriting via the canonical formalism for concept rewriting introduced at section 2 of chapter 7.
So, we can say that:

- concept rewriting is performed from terms of the requester's abstracted query (i.e. the request) to terms of the provider's advertisement, both request and advertisement being expressed in OWL DL ontology language. Hence
$\mathcal{L}_s = \mathcal{L}_d = \mathcal{L}_t = \mathcal{SHOIN}(D)$;
- concept rewriting is based on the terminology *T* of the shared OWL DL ontology;
- concept rewriting is executed through the iterative and combined application of any of the following *is-a* or *non-is-a* links defined within the OWL DL reference ontology:
    - ❑ *equivalentClass*,
    - ❑ *subClassOf*,
    - ❑ *intersectionOf*,
    - ❑ *unionOf*,

- *ObjectProperty*,
- *DatatypeProperty*.

Hence,

ρ = *equivalentClass* o *subClassOf* o *intersectionOf* o *unionOf* o *ObjectProperty* o *DatatypeProperty*,

where, with an abuse of symbolism, o now indicates "iterative and combined application" of the addressed links instead of function composition as usual.

4. Semantic Similarity Heuristics

4.1. Semantic Similarity

Since we took an ontology as the support KB, we applied some work derived from semantic networks theory to our approach. In particular we resorted to semantic similarity to enhance the search algorithm for Concept Transformation with some simple heuristics.
In [Quillian, 1968] a *semantic network* is defined as a "representational format [that would] permit the 'meanings' of words to be stored, so that humanlike use of these meanings is possible". Hence, we can synthesise a semantic network as a "network of hierarchically or associatively linked concepts", and an ontology (see section 1, chapter 7) perfectly casts this definition.
From spread activation theory (again, refer to [Quillian, 1968]), it arose the assumption that a semantic network is organized along the lines of *semantic similarity*. Namely, the more properties two concepts share in common, the more links there are between the same concepts, and the more closely related they are.
Incidentally, as it is stressed out in [Rada et al., 1989], semantic similarity is a special case of *semantic relatedness*. Nevertheless, for our purposes, we will restrict our vision of relatedness as similarity, and we will hereafter use *relevance* as a synonym for similarity.

Principally depending on the available network, semantic similarity can be quantified with different metrics (for a quick overview see [Hale, 1998]). In a taxonomy (i.e. a semantic network exclusively based on *is-a* links, which allow for concept generalization and specification), a natural way to evaluate semantic similarity is to measure the *distance* between the nodes corresponding to the items being compared. Such a distance is taken by counting the edges on the path to go from a node to the other one across the *is-a* links of the given taxonomy.
However, the edge-counting method relies on the unrealistic hypothesis that *is-a* links in a taxonomy represent uniform distances. Therefore, in [Resnik, 1995] the notion of *information content* was proposed as a more accurate metric for semantic similarity. In a *is-a* taxonomy permitting multiple inheritance, the information content approach grounds semantic similarity between two concepts on the extent to which they share information in common. In fact, the unrealistic assumption of uniform edge-distances is avoided by coupling each concept in the taxonomy with the estimated probability of encountering an instance of that same concept in the taxonomy itself. The trustworthiness of such an estimate may be indeed enforced by accounting it on an external *corpus* of words.
Thus quantifying the information content of a concept as negative the log of the associated probability, the semantic similarity of any pair of concepts in the taxonomy is indexed by the information content of the most specific common subsumer. Since the introduced

likelihood increases when informativeness of the relative concept decreases and vice-versa, semantic similarity is directly proportional to the amount of information shared in common by the compared concepts.

4.2. WordNet-Based Lexical Similarity

We borrowed those intuitions to roughly denote *lexical similarity* between two concepts. Here we have deliberately used the term lexical rather than semantic, in that we are specializing semantic similarity on the basis of a lexical KB. Because of its accurate design and its easy accessibility, we chose WordNet (accurately presented in [Miller et al., 1990]) to serve as such a KB.
However, we remind that Query Reformulation is figured out w.r.t. an OWL DL ontology. So, to assert any lexical relevance between two ontology concepts, we must bypass a basic discrepancy between WordNet and our OWL DL ontology.
Essentially, WordNet divides English words into four syntactic categories, that are nouns, verbs, adjectives, and adverbs. Those categories are in turn organized into sets of synonyms (i.e. *synsets*), each representing a lexicalised concept. Therefore, WordNet characterizes any English word as a *lemma* with a number of associated synsets.
Semantic relations ultimately link either *word forms* (i.e. lemmas) or synsets (i.e. meanings, or *word senses*). Concerning with lemmas, those relations are *synonymy*, and *antonymy*. Concerning with word senses, those relations are *hyponymy*, *hypernymy*, *meronymy*, *holonymy*, *toponymy*, and *entailment*.
In a radically different fashion, an OWL ontology is a hierarchical, associative network of URIs. What we theoretically refer to as a KB concept, in practice is a URI defined within an ontology.
Hence, we will be able to exploit WordNet semantic relations only if we first bind every ontology URI with some WordNet synsets. Thereby, we will treat ontology URIs as WordNet lemmas, so that we will be able to associate a set of meanings with each ontology URI.
By the way, the possibility of such a semantic annotation is already self-included in ODL_{I3} ontology language used by the MOMIS system (refer to [Bergamaschi et al., 2001] for more details).
Our method to evaluate lexical relevance between two (ontology) concepts *A* and *B* articulates in the following steps:

Granting the bound set of meanings for *A* and *B*, respectively referred to as *Syn(A)* and *Syn(B)*,

1. we compute the intersection of Syn(A) and Syn(B);
2. if that intersection is not empty, we will say that *A* is relevant to *B* in a "certain degree" (jump to next paragraph for more details);
3. if that intersection is empty, we will compute all the respective hypernyms for each meaning compounded by *Syn(A)* and *Syn(B)*;
4. we compare the retrieved sets of hypernyms to get the common ones;
5. for each common hypernym, we count the number of synsets it is far from the respective root synset in *Syn(A)* and *Syn(B)*; the sum of the globally counted edges (i.e. synsets) to go from a root synset in *Syn(A)* to a root synset in *Syn(B)* passing through a common hypernym is referred to as an edge-based distance.
6. If the any of such edge-based distances is less than a predefined threshold, then we will say that *A* is relevant to *B* in a "certain degree" (again, see next paragraph for a deeper explanation)

4.3. Heuristics for Query Reformulation

Originally concerning with the MOMIS system and then involving the SEWASIE Information Node (see section 2, chapter 3) as its natural descendant, in [Bergamaschi et al., 2001] it is explained how an effective integration process to build a global ODL_{I3} class from local ODL_{I3} classes distributed in local source schemas leans on the notion of *affinity*. That way, they identify ODL_{I3} classes that describe the same or semantically related information in different source schemas and give a measure of the level of matching of their structure. ODL_{I3} are thus analysed and compared by means of a *global affinity* coefficient as the linear combination of a *structural affinity* coefficient and a *name affinity* coefficient.
Analogously, in the present context of Query Reformulation, we denote semantic relevance between two concepts *A* and *B* of the support ontology by means of a *global relevance* coefficient (*GR*) as the linear combination of a *structural relevance* coefficient (*SR*) and a *lexical relevance* coefficient (*LR*):

$GR(A, B) = w_{SR} \cdot SR(A, B) + w_{LR} \cdot LR(A, B)$
where $w_{SR}, w_{LR} \in [0, 1], w_{SR} + w_{LR} = 1.$

w_{SR} and w_{LR} are optional coefficients to assess the weight of each partial coefficient in computing the global relevance coefficient.

The so-defined semantic relevance was used as an heuristic aid for two subprocesses of Query Reformulation. Namely, Variable Abstraction and Concept Transformation (see section 3). Accordingly, we are about to introduce how the structural relevance coefficient and the lexical relevance coefficient must be computed.

4.3.1. Structural Relevance

It is based on the ontology relationships linking two concepts after Variable Abstraction or Concept Tranformation. As in the former case just a relationship is used (that one expressed by the orginal query predicate), structural relevance cannot discriminate the process.
Contrarily, in the latter case we have a path from a root concept to the target concept substained by a number of relationships. However, those relationships can be of some prefixed types. Therefore, we can deliberately assign a strength coefficient (σ_{rel}) to each type of relationship and calculate the structural relevance coefficient of the root concept w.r.t. the target concept as the sum of the strength coefficients of all the relationships exploited across the path. Explicitely, we have:

strength coefficients: $\sigma_{eq} \geq \sigma_{sc} \geq \sigma_{in} \geq \sigma_{un} \geq \sigma_{op} \geq \sigma_{dp}$,
respectively assessing structural strength for equivalentClass, subClassOf, intersectionOf, unionOf, ObjectProperty, DatatypeProperty.
$SR(A_0, A_n) = (\sigma_{rel})_1 + (\sigma_{rel})_2 + \ldots + (\sigma_{rel})_{n-1} + (\sigma_{rel})_n$
where $(\sigma_{rel})_i \in \{\sigma_{eq}, \sigma_{sc}, \sigma_{in}, \sigma_{un}, \sigma_{op}, \sigma_{dp}\}$ *is the strength coefficient associated with the type of the relationship exploited at the i-th* ($i = 1, \ldots, n$) *step of* $Path(A_0.A_1.A_2. \ldots .A_{n-1}.A_n)$.

4.3.2. Lexical Relevance

Recalling the method to assess WordNet-based lexical similarity between two concepts *A* and *B* proposed above, the lexical relevance coefficient quantifies the result of such lexical relevance assessment.
In case the quoted algorithm ends up at step 2, lexical relevance may be quantified by the respective weight of the meanings composing the non-empy intersection of *Syn(A)* and *Syn(B)*. In fact, when associating a set of word senses (that are exactly *Syn(A)* and *Syn(B)*) with an ontology URI (respectively, the concepts *A* and *B*) it is possible to feature each bound synset with a distinct weight.
In case the presented algorithm terminates at step 6, the lexical relevance coefficient may be quantified as the information content (see above) of the common hypernym which has been considered.

Applying lexical relevance to Variable Abstraction is a straightforward procedure: the concepts to be compared will be the potential abstracted concept and the remaining concept in the abstracted query. All the concepts admissible for Variable Abstraction could actually be sorted according to the respective lexical relevance to the other concept in the abstracted query. Then, the best ranked concept will be regarded as the final abstracted concept.

Applying lexical relevance to Concept Transformation implies an iterative process:
1. the lexical relevance coefficient between each concept along the path and the prefixed target concept may serve as an heuristics to decide whether the concept on the path is worthy to be taken into account of to be discarded;
2. once a path has been determined, a lexical relevance coefficient for the whole iter may be calculated by summing all the single coefficients from step 1.

At last, it urges to advise that all the used constants -such as the weight of the meanings associated with a URI, and the information content of the common hypernyms- must be arranged in a convenient way for the whole scope of the process. For example, the quoted weights should be proportional to the information contents so that the algorithm at section 4.2 produces comparable results aside from its termination at step 2 or step 6. Moreover it should be taken into account that lexical relevance coefficients might be addictive components for a global relevance coefficient (see next paragraph).

4.3.3. Global Relevance

For Variable Abstraction ending with an abstracted query $P(A_i, B)$, the global relevance coincides with the lexical relevance. Hence,

$$GR(A_i, B) = LR(A_i, B)$$

For Concept Transformation resulting with $Path(A_0.A_1.A_2. \ldots .A_{n-1}.A_n)$, the global relevance of the root concept to the target concept w.r.t. to the current path has both the structural and lexical components:

$GR(A_0, A_n) = w_{SR} \cdot SR(A_0, A_n) + w_{LR} \cdot LR(A_0, A_n)$
where the computation of $SR(A_0, A_n)$ is explained at paragraph 4.3.1, while
$LR(A_0, A_n) = LR(A_1, A_n) + LR(A_2, A_n) + \ldots + LR(A_{n-2}, A_n) + LR(A_{n-1}, A_n)$

Thereby, all the paths produced by Concept Transformation might be ranked by the respective global relevance coefficient. Incidentally, the discovered paths might be first ranked by their length.

IX Reasoning Tools

All the inference problems underlying Query Reformulation are practically solved through the inference services made available by a *reasoning tool* (alternatively called *reasoning engine* or, simply, *reasoner*). Nowadays, some of the major reasoners potentially exploitable are ODB-Tools, JTp, DAMLJessKB, FaCT, and RACER. The present chapter furnishes the outlines of these systems. A greater regard will be devoted to RACER, in that it is the one effectively used for our current implementation of Query Reformulation.

1. ODB-Tools

ODB-Tools (Object DataBase-Tools, for a detailed treatment refer to [Beneventano et al., 1997]) is a Description Logics (DLs) based system for schema validation and semantic query optimization in Object-Oriented (O-O) databases.
ODB-Tools was originally based on the DL called $\mathcal{OCDL}$ (Object Constraints Description Language, see [Beneventano et al., 1996]), and it could accept an ODL (Object Data Language, reference at [ODS, 1997]) schema to be validated.
ODB-Tools was conceived as an integrated component of the MOMIS system (described by [Bergamaschi et al., 2001]). Afterwards, ODB-Tools was embedded into the SEWASIE system (see chapter 3) as its reasoning engine.
ODB-Tools has been extended to perform inferences relying on the DL called $\mathcal{OLCD}$ (see chapter 6, section 3) and the data modelling language is ODL_{I3} (generally sketched always in chapter 6, section 3).
The former query language for the queries to be optimized by ODB-Tools was OQL (Object Query Language, see [OQL]). After the commitment to ODL_{I3}, OQL has been parallely replaced by OQL_{I3} (see [ODS, 1997]).
In order to achieve schema validation and semantic query optimization, ODB-Tools provides inference services such as subsumption computation, incoherence detection, and canonical form computation.
ODB-Tools was developed at the University of Modena and Reggio Emilia in the late 90s.

2. JTP

JTP (exstensively, Java Theorem Prover, described by [Fikes et al., 2003]) is an O-O modular reasoning system. It employs an architecture for hybrid reasoning (the *JTP architecture*), and an easily extensible library of general-purpose reasoning systems components (the *JTP library*).

In JTP architecture, a reasoning system consists of modules -properly called reasoners- that are basically classified into two types: backward-chaining reasoners (i.e., they process goals and return proofs for the answers they provide), and forward-chaining reasoners (i.e., they process assertions substantiated by proofs, and draw conclusions).
Those reasoners are gathered by the JTP library. Among the others, such a collection includes evaluable relation reasoners (e.g., the provide standard functionality for arithmetic operators), Horn-clause reasoners (that process Horn clauses), an essential model elimination theorem prover (which is the core of the whole reasoning system and is based on the algorithm described by [Stickel, 1985]; incidentally, for a rough definition of a *theorem prover*, the reader is suggested to jump to section 6 of this chapter), and a generalized modus ponens knowledge store with the relative reasoners.
Above all the reasoning components, at the top level of abstraction in the JTP architecture, the implemented system uses a First-Order Logic (FOL) representation language, which is KIF 3.0 (the reference document is [Genesereth and Fikes, 1992]). Hence, the system can be queried with FOL sentences respecting the KIF 3.0 format and it can load any FOL compliant Knowledge Base. This way, even sentences of a DAML+OIL or OWL ontology can be stored by the JTP system and compiled into FOL axioms. FOL queried are asked against those FOL translations of Description Logic (DL) expressions. Thus, accomplishing any kind of FOL inference by backwards or forwards chaining on FOL axioms is computationally expansive as compared to results for DL-based reasoners. As a matter of fact, JTP was not able to load the well-known wine ontology (available at ["wines" DAML+OIL ontology]) even after several minutes of continuos processing (the testbed machine was equipped with a CPU running at 1 GHz of clock frequency, with 256 MB of central memory).
In the late 90s, JTP was developed by Gleb Frank in Knowledge System Laboratory of Computer Science Department in Stanford University (CA).

3. DAMLJessKB

On the surface, DAMLJessKB (refer to [Koepena and Regli, 2002]) is advertised as a DL-based reasoner perfoming inference w.r.t. DAML+OIL ontology language. Truly, reasoning is carried out by the production system (see section 6 of this chapter for a quick definition) called JESS (Java Expert System Shell, refer to [Friedman-Hill, 1995]). Rather than directly incorporating DAML+OIL axiomatic semantics as proofs of a theorem prover (like JTP mainly does), DAMLJessKB indirectly uses the DAML+OIL Knowledge Base as input (i.e. the *facts*) of the JESS production system. Rules are then derived from the semantics of the language, thus representing the heuristic knowledge of the expert system (i.e. the production system itself). Therefore, DAMLJessKB supports any reasoning service motivated by DAML+OIL ontology language and it inevitably tumbles on the undecidable semantics of DAML+OIL. E.g., DAML+OIL makes no distinction between classes and individuals and it even blurrs the separation between datatypes and classes. At section 2, chapter 6, it was seen as these sample issues were appropriately kept away from OWL DL language just to secure decidability for supported inferences. Nevertheless, DAMLJessKB is inclined towards full compatibility with DAML+OIL, even being aware of the high computational complexity that is due.
As a final remark, we point out that DAMLJessKB embodies the Closed World Assumption (CWA), i.e., "if a fact does not exist it is assumed to be false".
Recently, a prototypical version of DAMLJessKB compliant with OWL ontology language

has been released. Quite obviously, it has been called OWLJessKB.
DAMLJessKB developement was supported in part by National Science Foundation (NSF), Knowledge and Distributed Intelligence in the Information Age (KDI) Initiative Grant CISE/IIS-9873005, CAREER Award CISE/IRIS-9733545 and ONR Grant N00014-01-1-0618.

4. FaCT

The FaCT (Fast Classification of Terminologies, main reference at [Horroks, 1999]) is a DL classifier that can also be used for modal logi satisfiability testing. It includes two reasoners, one for the DL called $\mathcal{SHF}$ and another for the DL called $\mathcal{SHIQ}$ (see [Horrocks et al., 2000]). FaCT supports reasoning with KBs containing general concept inclusion axioms (i.e. they state the subsumption relation between two concept terms). Development of ABox resoning has been recently released. Although FaCT was originally developed to work with the standard KRSS fuctional interface (described by [Patel-Schneider and Swartout , 1993]), now it has been added the possibility of a CORBA-based client server configuration.
In an Open World Assumption (OWA), the system implements optimization techniques, including axiom absorption, lexical normalization, semantic branching search, simplification, dependancy directed backtracking, heuristic guided search and caching (refer to [Horroks and Patel-Schneider, 1999]), which have become standard for sound and complete tableaux-based algorithms (see [Patel-Schneider, 1998]).
FaCT is part of the Camelot Project at the University of Manchester (UK).

5. RACER

The RACER (Renamed Abox and Concept Expression Reasoner) system (see [Haarlsev and Möller, 2003]) is a knowledge representation system implementing a highly optimised tableau calculus for a very expressive DL, namely $\mathcal{ALCQHI}_{\mathcal{R}+}$ -also known as $\mathcal{SHIQ}$ (see [Horrocks et al., 2000]).

- Based on $\mathcal{SHIQ}$, RACER offers reasoning services for multiple TBoxes and for multiple ABoxes as well. RACER supports the specification of general terminological axioms. A TBox may contain general concept inclusion axioms. Multiple definitions or even cyclic definitions of concepts can be handled by RACER.
- Besides, RACER augments $\mathcal{SHIQ}$ with concrete domains thus providing algebraic reasoning for min/max restrictions over the integers, linear polynomial (in-)equations over the reals or cardinals with order relations, non-linear multivariate polynomial (in-)equations over complex numbers, equalities, and inequalities of strings.
- Moreover, RACER is able to deal with functional roles, i.e. features ($\mathcal{F}$). Since $\mathcal{SHIQ}$ is able to represent inverse roles ($\mathcal{I}$), the further accounting of $\mathcal{F}$ in the logic would lead to the loss of the finite model property. So, although $\mathcal{SHIQ}$ excludes $\mathcal{F}$, RACER's ability to deal with both $\mathcal{F}$ and $\mathcal{I}$ implied the development of a sophisticated algorithm to find finite representations of infinite models in order to guarantee termination. In addition, it was necessary to restrict the use of transitive

roles ($\mathcal{R}^+$) in number restrictions and to keep disjointed $\mathcal{F}$ and $\mathcal{R}^+$ in order to maintain decidability.

- As pointed out among the motivations for the choice of RACER as our effective reasoning tool, RACER 1.7 can directly read knowledge bases specified w.r.t. the OWL, DAML+OIL, RDF/RDFS standard, although there are some restrictions on some OWL and DAML+OIL language expressions, and the implementation of the OWL interface is still prototypical. Keeping our focus on this latter interface, OWL documents are interpreted w.r.t. the OWL-DL language. For instance, as OWL-DL doesn't admit to treat classes or roles as individuals (see section 2, chapter 6), intensional equality for classes or roles is not supported. In fact, DL systems can provide only an approximation for true nominals.
- As other description logic systems, RACER employs the OWA for reasoning. This means that what cannot be proven to be true is not believed to be false.
- Diverting from $\mathcal{SHIQ}$, RACER also employs the UNA. This means that all individuals used in an ABox are assumed to be mapped to different elements of the universe, i.e. two individuals cannot refer to the same domain element.
- RACER implements the HTTP-based quasi-standard DIG for interconnecting DL systems with interfaces and applications using an XML-based protocol. RACER also implements most of the functions specified in the older KRSS [Patel-Schneider and Swartout , 1993].

Based on the logical semantics of the representation language, different kinds of queries are defined as inference problems. Hence, answering a query is called providing inference service. Given a TBox, we exploited only the following RACER inference services:

- Concept testing w.r.t. a TBox:
  Is a certain concept name referring to a concept specified in a TBox?
- Concrete domain attribute testing w.r.t. a TBox:
  Is a certain role name to be interpreted as a reference to a concrete domain attribute in a TBox?
- Concept equivalence w.r.t. a TBox:
  Are the specified concepts equivalent in the given TBox? Or, to determine all the equivalent concepts for a given concept.
  We remind that equivalence is always intended in the extensional denotation, thus differing from the intensional characterization of equality. Namely, two concepts are equivalent if and only if they collect the same set of individuals (i.e., they have the same extension), whereas two concepts are equal if and only if they are exactly the same concept (i.e., they have the same intensional meaning).
- Concept subsumption w.r.t. a TBox:
  Is there a subset relationship between the set of objects described by two concepts?
- Determine the parents and children of a concept w.r.t. a TBox.
  The parents of a concept are the most specific concept names mentioned in a TBox which subsume the concept. The children of a concept are the most general concept names mentioned in a TBox that the concept subsumes.
- Computation of the type of a concrete domain attribute w.r.t. a TBox.
- Evaluation of the domain and range for a role w.r.t. a TBox.
  The domain and range for a given role are respectively constituted by all the concepts expressively declared in a TBox as domain or range for the role itself.
- Retrieval of all the transitive roles and features w.r.t. a TBox while distinguishing all the roles specifically defined as concrete domain attributes.

If also an ABox is given, we exploited RACER with the following types of inferences:

- Instance testing w.r.t. an ABox:
  Is the object for which an individual stands a member of the set of objects asserted by an ABox?
- Instance retrieval w.r.t. an ABox and a TBox.
  Find all individuals from an ABox such that the objects they stand for can be proven to be a member of a set of objects described by a certain concept.
- Computation of the direct types of an individual w.r.t. an ABox and a TBox.
  Find the most specific concept names from a TBox of which a given individual is an instance.
- Computation of the types of an individual w.r.t. an ABox and a TBox.
  Find all the concept names from a TBox of which a given individual is an instance.
- Computation of the fillers of a role with reference to an individual.
- Computation of the predecessors of a role, that is restricted not to be a concrete domain attribute, with reference to an individual.
- Retrieval of all the pair of individuals related by a given role w.r.t. an ABox and a TBox.

Required inference services not implemented by RACER:

- Retrieval of all the predecessors of a concrete domain attribute w.r.t. a TBox and an ABox.
  In practice, we made up this reasoning skill by taking advantage of the RACER's ability to compute the fillers of a concrete domain attribute w.r.t. an individual. Basically, we first determined all the individuals asserted by an ABox and then we checked for each individual if there existed any fillers for a concrete domain attribute. A positive response meant that the current individual was to be intended as a predecessor of the concrete domain attribute.
- Computation of the predecessors of a concrete domain attribute w.r.t. an individual. Again, this needed though explicitly missing service was built up by profiting of the RACER's inference to retrieve the fillers of a concrete domain attribute w.r.t. an individual. Basically, we iterated the procedure explained above. We first retrieved all the individuals populating an ABox. Then, we determined the fillers of a concrete domain attribute w.r.t. either retrieved individual. Then we looked up the determined fillers to check whether one of them was the initially given individual. A positive response eventually meant that the individual for which the fillers were retrieved was to be intended as a predecessor of the concrete domain attribute w.r.t. the initially given individual.
- Concrete domain object testing w.r.t. an ABox and a TBox:
  Can a given value be interpreted as a concrete domain object rather than as a proper concept?
  We didn't overcame this lack in RACER's implementation, in that we decided to neglect concrete domain checking for possible concrete domain objects. In fact, it turns out that RACER signals an error, though carrying on the next tasks, when it is commanded to perform concept inferences on concrete domain objects. We assumed that error message as an effective, but subtle, concrete domain object testing.

RACER was initially developed at the University of Hamburg, Germany. However, nowadays RACER is actively supported and future releases are developed at Concordia University in Montreal, Canada, and at the University of Applied Sciences in Wedel, Germany.

6. Comparative Summary

We will conclude the present chapter with a comparative view over the main characteristics of the outlined reasoners. We will stress a first distinction between DL-based reasoners and non-DL-based reasoners. The latters can then roughly fall into two categories: theorem provers and production (or expert) systems.
Theorem provers essentially reason by backward-chaining, whereas production systems lean on forward-chaining. By an historical standpoint, theorem provers descend from Prolog (see [Sterling and Shapiro, 1986]), whereas production systems implement the Rete algorithm (see [Forgy, 1982]). However, a sharp separation is merely ideal. As we said, even though JTP is based on a model elimination theorem prover, it equally performs forward-chaining reasoning (in fact, it is known as an *hybrid* reasoning system). Similarly, even if JESS is based on the Rete algorithm, it includes a kind of backward-chaining, too
In the end, reasoners will be distinguished by the respective representation or query language and by the optional provision of public Java language Application Programming Interfaces.
Table 9 summarizes the undertaken analysis.

	Non-DL-based		**DL-based**					
Reasoners	**Theorem prover**	**Production system**	**Implemented DL**	**KB representation language**	**Query language**	**OWA**	**CWA**	**Java API**
ODB-Tools	-	-	$\mathcal{OCDL}$, $\mathcal{OLCD}$	ODL, ODL_{I3}	OQL, OQL_{I3}	♦	-	-
JTP	♦	-	-	KIF 3.0	KIF 3.0	♦	-	♦
DAMLJessKB	-	♦	-	JESS-based	JESS-based	-	♦	♦
FaCT	-	-	$\mathcal{SHIQ}$, $\mathcal{SHF}$	KRSS-based	KRSS-based	♦	-	♦
RACER	-	-	$\mathcal{SHIQ}$	KRSS-based	KRSS-based	♦	-	♦

Table 9. Comparative view of some reasoning engines.

X Implementation

1. Overview

Code development was pursued trying to achieve the highest degree of modularity and flexibility.
In fact, the current implementation is supposed to be integrated with two wider projects, namely SEWASIE (refer to chapter 3), and DAML-S broker (refer to chapter 4). Moreover, the coding makes use of external components such as a reasoning engine connected to the support KB, and a DBMS server grounded on a lexical database.
So, this implementation must be naturally adaptable to any reasoning tool different from that one used in the current release, and to any procedure exploiting the lexical database.
Because of its well known portability, and in order to be consistent with the development s we are integrating, the Java programming language (see [Java]) was chosen to encode this implementation.

The general structure of the current implementation is illustrated by the following package diagram:

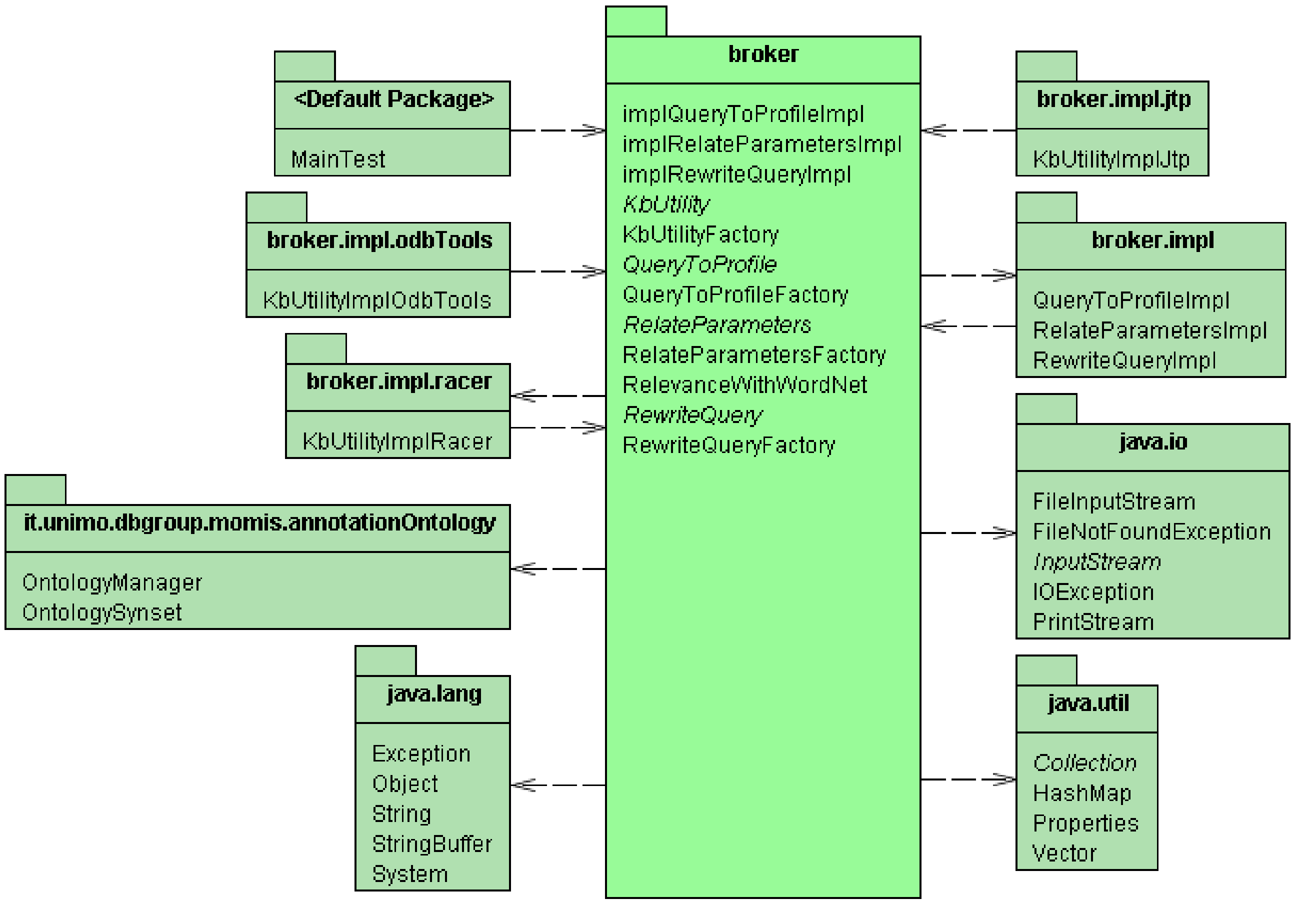


Figure 19. Package diagram for *broker*.

The package *broker* collects the Application Programming Interfaces (API) for the front end approach to Query Reformulation, while the current implementation of those interfaces is coded through the classes in the sub-package *broker.impl*. Those classes are characterized by the suffix *Impl*.
Beyond the mentioned API, the package *broker* also contains some classes specially purposed to create the required API all across the rest of the coding. Those classes are distinguished by the suffix *Factory*.

On a side, the package *broker.impl* should contain as many sub-packages as the number of reasoners potentially exploitable in the current implementation. At the moment we named three alternative reasoners -ODB Tools, JTP, and RACER- although just one -RACER- was effectively exploited. The corresponding sub-packages are be *broker.impl.*odbTools, *broker.impl.jtp*, and *broker.impl.racer*, respectively.

Falling into greater details, the APIs *QueryToProfile* and *RewriteQuery* respectively gather the methods to cope with the Query Abstraction and Query Rewriting processes for Query Reformulation.
Towards the same goal is the interface *RelateParameters*, since it includes the methods to execute the main algorithm to perform Query Rewriting. Those methods are actually invoked inside the interface *RewriteQuery*.

KbUtility is the interface module which guarantees the plug-in connection with any reasoner and KB supported by the reasoner itself.

QueryToProfileFactory, *RewriteQueryFactory*, *RelateParametersFactory*, *KbUtilityFactory* are the classes purposed only to create the above mentioned interfaces: This manner, reater extensibility and readability are retained.

We can find the current implementation of the interfaces for Query Reformulation in the classes *QueryToProfileImpl*, *RewriteQueryImpl*, and *RelateParametersImpl*.
Obviously, the interface *KbUtility* cannot have a unique implementation, in that it must refer to the currently used reasoner. Heretofore, we developed just an implementation for that interface: *KbUtilityImplRacer* (to say the truth, the implementation *KbUtilityImplJtp* was priorly coded, but it was then discarded because the exploited reasoner -JTP- did not provide acceptable performances).

In the sequent sections we will analyse each main component of the outlined implementation, paying great attention to the executed algorithms.

2. Detailed Analysis and Algorithms

2.1. *QueryToProfile*

The method *abstractQuery* of this class performs the actual Query Abstraction. In the class

QueryToProfileImpl (see the relative class diagram in figure 20) this is accomplished by executing the algorithm in figure 21.

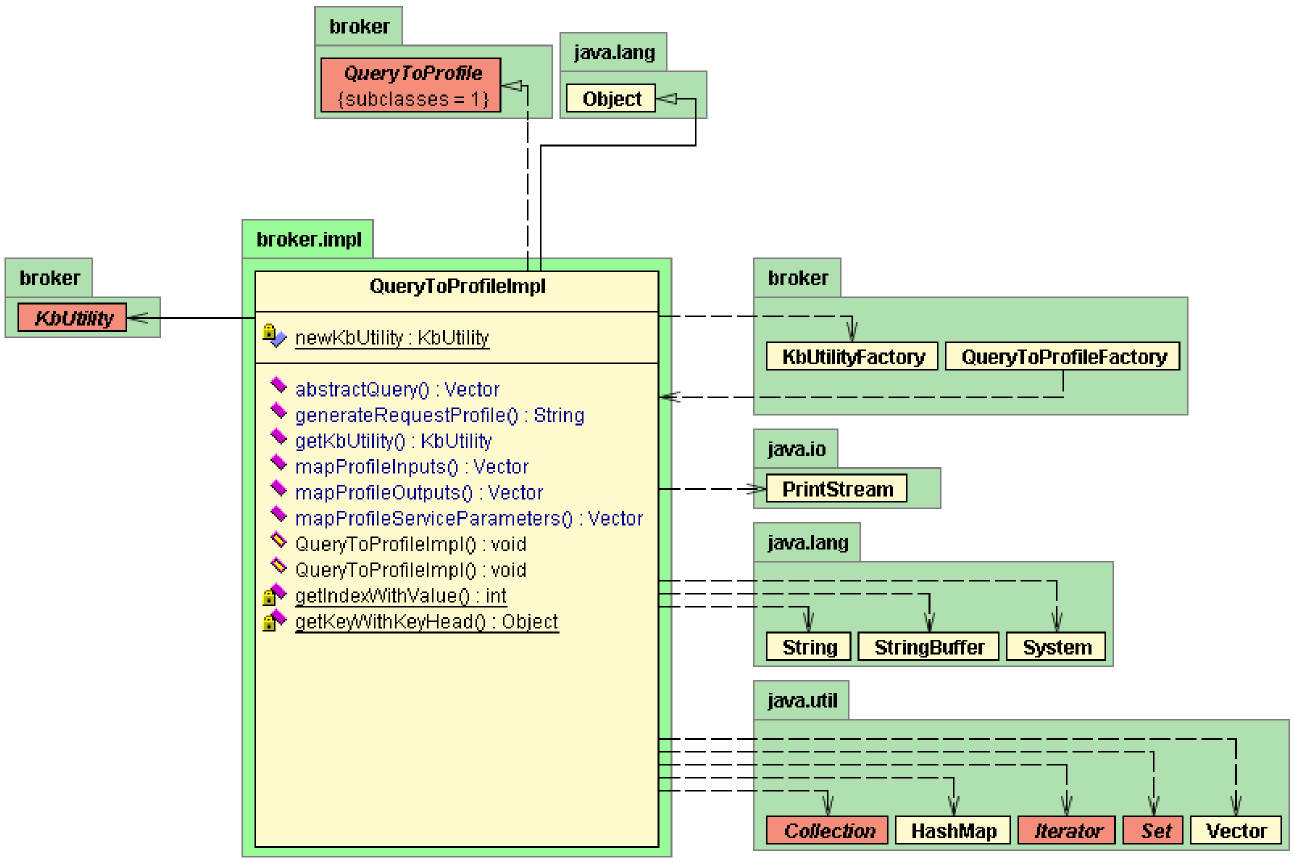


Figure 20. Class diagram for *QueryToProfileImpl*.

```
Vector Function abstractQuery (Vector query) :
 begin
  abstractedQuery := query;
  for each term t in the abstractedQuery that is either the subject or the object of the abstractedQuery do
   if (t is not a concept)
    then
      {if (t is an instance)
        then
         {t := first concept found as the concept from which t was instantiated;
         }
        else
         {if (t is the subject of the query)
           then
            {if (the predicate of the query has some domain concept)
              then
               {t := first concept found as a domain of the predicate of the query;
               }
              else
               {t:= "NIL";
               }
            }
           else
            {if (the predicate of the query has some range concept)
              then
               {t := first concept found as a range of the predicate of the query;
               }
              else
               {t:= "NIL";
               }
            }
         }
      }
  done
  return abstractedQuery;
 end
```

Figure 21. Principal algorithm for Query Abstraction.

2.2. *RelateParameters*

This interface compounds several methods invoked to find a relation such that from a given concept of the provided KB we can obtain another concept of the same KB. This is accomplished by building a path of concepts where the start point and the end point are given. In order to come up with such a path the KB is explored thoroughly by building up a search space network organized as a tree graph.
The root node of the search network is the concept given as the start point of the path. The children nodes are created by exploring the knowledge base in order to find the concepts anyway related to the root node.
In a first instance, the subsumption relationship is considered, while any other kind of relationship is taken into account in the sequel (look at chapter 8 for more details). Running across those relationships, the search terminates when the KB has been exhaustively parsed, i.e. when all the concepts reachable from the given root node have been examined once. Thus, the core of the search algorithm is the KB exploration to dig out all the concepts related to a given concept. With a breadth first strategy, that

exploration is recursively applied to each node generated in the search space until the the termination criterion described above is satisfied (for more details about search algorithms strategies see [Winston, 1992]).
During seeking, we assume that the target concept is hit as soon as an encountered concept satisfies at least one of the following conditions:

- it has the same name of the target concept (check up through string equality);
- it has the same extension (i.e., it is composed by the same set of individuals) of the target concept (check up through an appropriate call to the reasoning engine);
- it is subsumed by the target concept (again, that is checked through an appropriate call to the reasoning engine).

We remind that the search network is constituted by all the concepts explored while the search algorithm is running. Indeed, a parallel network is created at the same time. It tracks all the relations exploited to move from a concept to another along the search network. It basically stores a declarative identification of the links between two nodes of the search tree.
Those graphs are represented as *HashMap* in the Java programming language. Each node of the search network is tightened to the generating link by the key assignment in the proper *HashMap*. While two *HashMap* are used -one for the nodes, the other for the links-, there is a unique set of keys for those two *HasMap* so that through the same key we can retrieve a node and the generating link from the respective *HashMap*.
Moreover, the keys are moulded in such a way that the ancestor hierarchy is always deducible. As each element in both of the *HashMap* represents a node of a tree graph, each key is composed by a prefix which is the key of the parent node and an appended suffix which allows for the unique identification of the current node.
During the search for the target concept, as soon as it has been hit, the walked path to reach it is stored as the sequence of the identifying keys for the touched concepts up to the target.
It goes without saying that more than a path leading to the target concept might be found. So, when the whole search has been completed, all the discovered paths are ranked according to their length (with path length we mean the number of concepts forming the path itself).
Summing up, the search algorithm:

- terminates in a finite time (hence, it is decidable);
- guarantees that all the possible paths connecting the root concept to the target concept are found (hence, it is complete);
- sorts those paths from the shortest one to the longest one;
- employs a breadth first strategy.

A final annotation regards the modular structure of the coding. Since the whole search algorithm is distributed into several small steps, each of those steps was implemented in a separated method of the classes *RelateParameters* and *RelateParametersImpl* (see the relative class diagram in figure 22).
Whereas *RelateParameters* comprises the declarations of the methods directly invoked to execute the algorithm, *RelateParameters* compaunds the definitions of those methods and the definitions of some ulterior methods exploited by the former ones.
Later on, in the case study (section 3 of the present chapter), we will show a possible search network -that is a tree graph whose nodes are concepts defined in the KB- and the correspondent links -i.e. a tree graph whose nodes are relations defined in the KB.

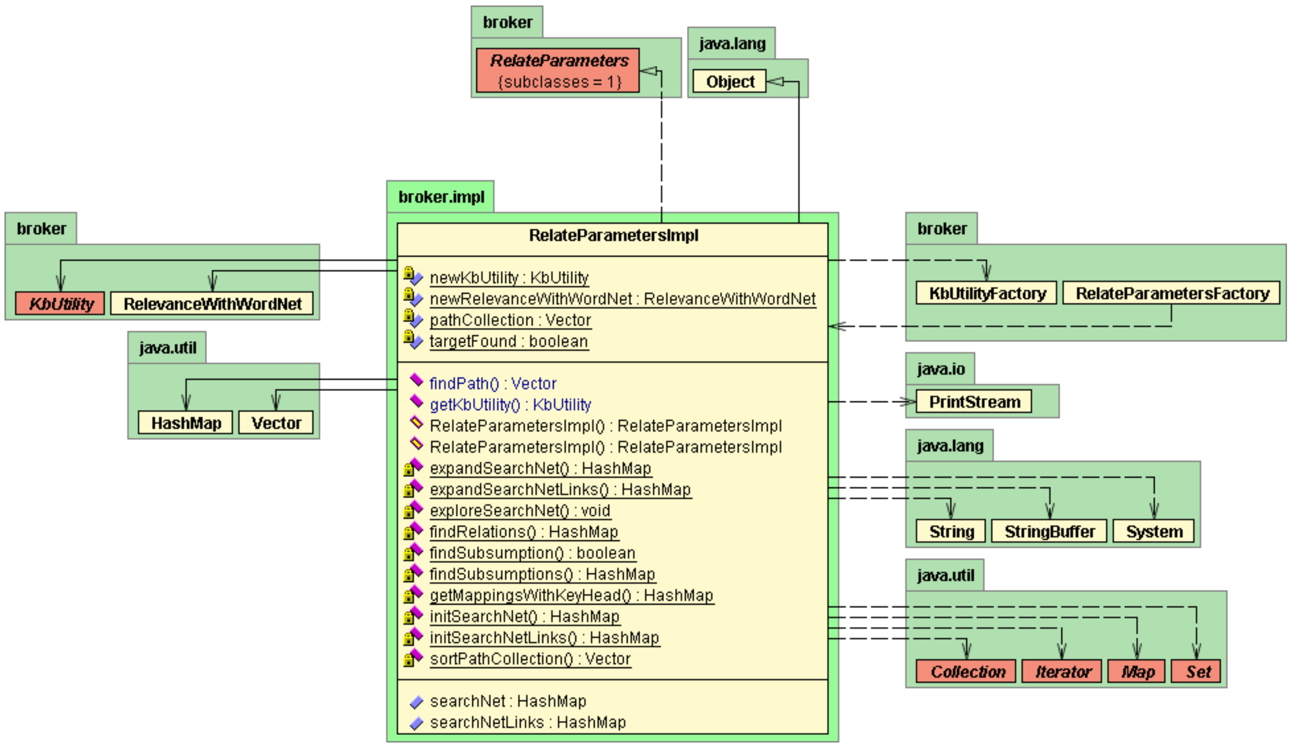

broker
RelateParameters
{subclasses = 1}
java.lang
Object
broker.impl
RelateParametersImpl
newKbUtility : KbUtility
newRelevanceWithWordNet : RelevanceWithWordNet
pathCollection : Vector
targetFound : boolean
findPath() : Vector
getKbUtility() : KbUtility
RelateParametersImpl() : RelateParametersImpl
RelateParametersImpl() : RelateParametersImpl
expandSearchNet() : HashMap
expandSearchNetLinks() : HashMap
exploreSearchNet() : void
findRelations() : HashMap
findSubsumption() : boolean
findSubsumptions() : HashMap
getMappingsWithKeyHead() : HashMap
initSearchNet() : HashMap
initSearchNetLinks() : HashMap
sortPathCollection() : Vector
searchNet : HashMap
searchNetLinks : HashMap
broker
KbUtility
RelevanceWithWordNet
java.util
HashMap
Vector
broker
KbUtilityFactory
RelateParametersFactory
java.io
PrintStream
java.lang
String
StringBuffer
System
java.util
Collection
Iterator
Map
Set

Figure 22. Class diagram for *RelateParametersImpl*.

The following frames depict the algorithms underlying the methods encoded in *RelateParametersImpl*.

```
Vector Function findPath(Object rootConcept, Object targetConcept)
 begin
   initialize the HashMap searchNet with the key-element mapping “0”-rootConcept;
   initialize the HashMap searchNetLinks with the key-element mapping “0”-“”;
   initialize the Vector initialPath with the element 0 at the first location (i.e., indexed by 0);
   Vector path := exploreSearchNet(searchNet, searchNetLinks, initialPath, targetConcept);
   return path
 end
```

Figure 23. Chief algorithm to find the path relating two concepts (the corresponding method is declared in *RelateParamenters*).

```
Vector Function exploreSearchNet(HashMap searchNet, HashMap searchNetLinks, Vector initialPath, Object targetConcept)
 begin
   Vector finalPath := initialPath;
   String currentKey := last element of initialPath;
   Object currentRootConcept := element of searchNet mapped with the key currentKey;
   if (targetConcept subsumes currentRootConcept)
     then
       {search terminates;
       }
     else
       {HashMap foundRelations := findSubsumptions(currentRootConcept, searchNet);
        Add to foundRelations the mappings returned by findRelations(currentRootConcept, searchNet);
        if (foundRelations is not empty)
          then
            {update searchNet with the mappings returned by expandSearchNet(searchNet, currentKey, foundRelations);
             update searchNetLinks with the mappings returned by expandSearchNetLinks(searchNetLinks, currentKey, foundRelations);
             HashMap newNodes := nodes actually added to searchNet through expandSearchNet;
             while (newNodes has one more mapping and the targetConcept has not been found) do
               get out a new current mapping from newNodes;
               String key = key in the current mapping just taken out from newNodes;
               If (element in the current mapping taken out from newNodes is equal to targetConcept)
                 then
                   {finalPath := initialPath;
                    append key as a new element to finalPath;
                    terminate search;
                   }
                 else
                   {append key as a new element to initialPath;
                    finalPath := exploreSearchNet(searchNet, searchNetLinks, initialPath, targetConcept);
                    if (targetConcept has not been found through the last invocation of exploreSearchNet either)
                      then
                        {remove the last element from initialPath;
                         finalPath := initialPath;
                        }
                   }
             done
            }
       }
   return finalPath;
 end
```

Figure 24. Main algorithm for the search net exploration.

```
HashMap Function findSubsumptions(Object rootConcept, HashMap searchNet)
 begin
   create the HashMap foundSubsumptionsTemp where each mapping is composed by the name of a
concept subsuming rootConcept (key) and the string "isSubsuming" (element);
   for each mapping in foundSubsumptionsTemp do
     if (the key of the current mapping is a concept not yet contained  by searchNet)
       then
        {put the current mapping in the HashMap foundSubsumptions keeping the same pattern for the
key-element association;
        }
   done
 return foundSubsumptions;
end
```

Figure 25. Algorithm to find subsumptions between two concepts.

```
HashMap Function findRelations(Object rootConcept, HashMap searchNet)
 begin
   create the HashMap foundRelationsTemp where each mapping is composed by the name of a
concept related to rootConcept (key) and the name of the exploited property (element);
   for each mapping in foundRelationsTemp do
     if (the key of the current mapping is a concept not yet contained  by searchNet)
       then
        {put the current mapping in the HashMap foundRelations keeping the same pattern for the key-
element association;
        }
   done
 return foundRelations;
end
```

Figure 26. Algorithm to find relations between two concepts.

```
HashMap Function expandSearchNet(HashNet searchNet, Object rootConceptKey, HashMap
foundRelations)
 begin
   while (foundRelations has one more mapping) do
     {extract a new current mapping from foundRelations;
      String key := rootConceptKey concatenated with a counter which is incremented while this
loop goes forth;
      add a new mapping to searchNet where the element coincides with the key taken from the
current mapping just extracted from foundRelations and the associated key coincides  the just
generated key;
    }

 done
 return searchNet;
end
```

Figure 27. Algorithm for expanding the search network.

```
HashMap Function expandSearchNetLinks(HashNet searchNetLinks, Object rootLinkKey,
HashMap foundRelations)
 begin
   while (foundRelations has one more mapping) do
     {extract a new current mapping from foundRelations;
      String key := rootLinkKey concatenated with a counter which is incremented while this loop
goes forth;
      add a new mapping to searchNetLinks where the element is that one taken from the current
mapping just extracted from foundRelations and the associated key is the just generated key;
     }
 done
 return searchNetLinks;
end
```

Figure 28. Algorithm for expanding the search network links.

2.3. *RewriteQuery*

The concrete Query Rewriting is performed by invoking the method *RewriteParameter* for each term of the current query. We recall that Query Rewriting is applied to the instances appearing as terms of the current query. Those instances need to be rewritten as instances of the concepts appearing in the abstracted query. In fact, if an instance that is a term of the current query was abstracted to a determined concept, then we must rewrite that instance into an instance of the concept it was abstracted to.
Once again, the method *RewriteParameters* call some more methods to fulfil its task. Those methods are defined in the class *RewriteQueryImpl* (see the relative class diagram in figure 29) and are hidden to the implementer who wants to use the interface *RewriteQuery*. Within that interface only the method *RewriteParameter* is declared.
Before introducing the implemented algorithms for Query Rewriting, it is worthy to describe the concrete criteria ruling the successful accomplishment of this process. Given two classes *A* and *B* which are consecutive steps of a Concept Transform path (see foregoing paragraph) and a pair of individuals, respectively *a* and *b*, we consider *a* as a valid instance of *B* iff at least one of the following conditions is true:

- *A* is subsumed by *B* (that is checked by calling the appropriate inference service of the reasoning engine);
- *A* and *B* are equivalent (again, checked through a proper command of the integrated reasoner);
- *A* is subsuming *B* but *a* is also a member of *B* (the reasoning tool will solve this question, too).

Otherwise, *a* can be substituted with *b* iff

- the couple *(a, b)* is a valid instance of the relationship linking A and B (once more, the reasoning engine is commanded to perform such a check).

In the frames below, we present a declarative description of all the methods closely underlying the Query Rewriting process. As we said above, the respective implementations are in the class *RewriteQueryImpl*.

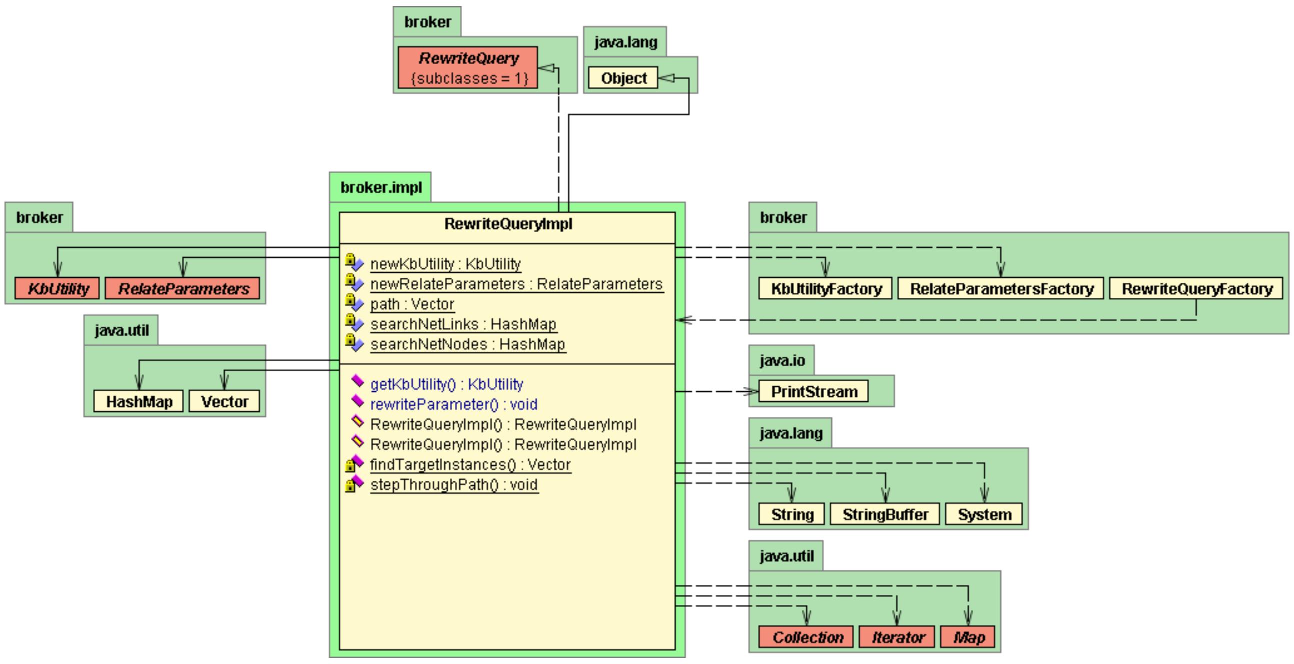


Figure 29. Class diagram for *RewriteQueryImpl*.

```
Procedure rewriteParameter(Object rootConcept, Object targetConcept, Object rootInstance)
 begin
   Vector path := findPath(rootConcept, targetConcept);
   if (path is longer than a single step)
     then
       {StepThroughPath(0, rootInstance);
       }
     else
       {there is no need to perform query rewriting;
       }
 end
```

Figure 30. Chief algorithm to rewrite a query term (the corresponding method is declared in *RewriteQuery*).

```
Procedure stepThroughPath(int pathStep, Object rootInstance)
 begin
  if (the current step is the final step of the path)
    then
      {stop term rewriting;
      }
    else
      {Vector foundTargetInstances := instances of the concept identified by the next step in the path
that are related to the given rootInstance of the concept identified by the current pathStep;
       if (foundTargetInstances is empty)
         then
           {stop term rewriting as it cannot proceed further;
           }
         else
           {for each element i in foundTargetInstances do
              stepThroughPath(pathStep + 1, i);
              done
           }
      }
 end
```

Figure 31. Algorithm to populate the path relating two concepts with proper individuals.

2.4. *KbUtility*

Essentially, this interface embeds the proper methods to connect a given reasoning tool to a given KB and let that tool perform the required inference services w.r.t. that KB. Up to now, several calls to the reasoning tool have been mentioned (for example, look back at the previous two sections, or at the algorithm sketch for Query Abstraction). However, much more are need as subprocedure of the former ones. The class diagram of *KbUtilityImplRacer* (see figure 32) collects all the implemented methods. Among them, only those within the first block (i.e. the public ones) can be invoked as attributes of *KbUtility*. Since no peculiar algorithm is here involved, we skip the usual pseudocode representations.

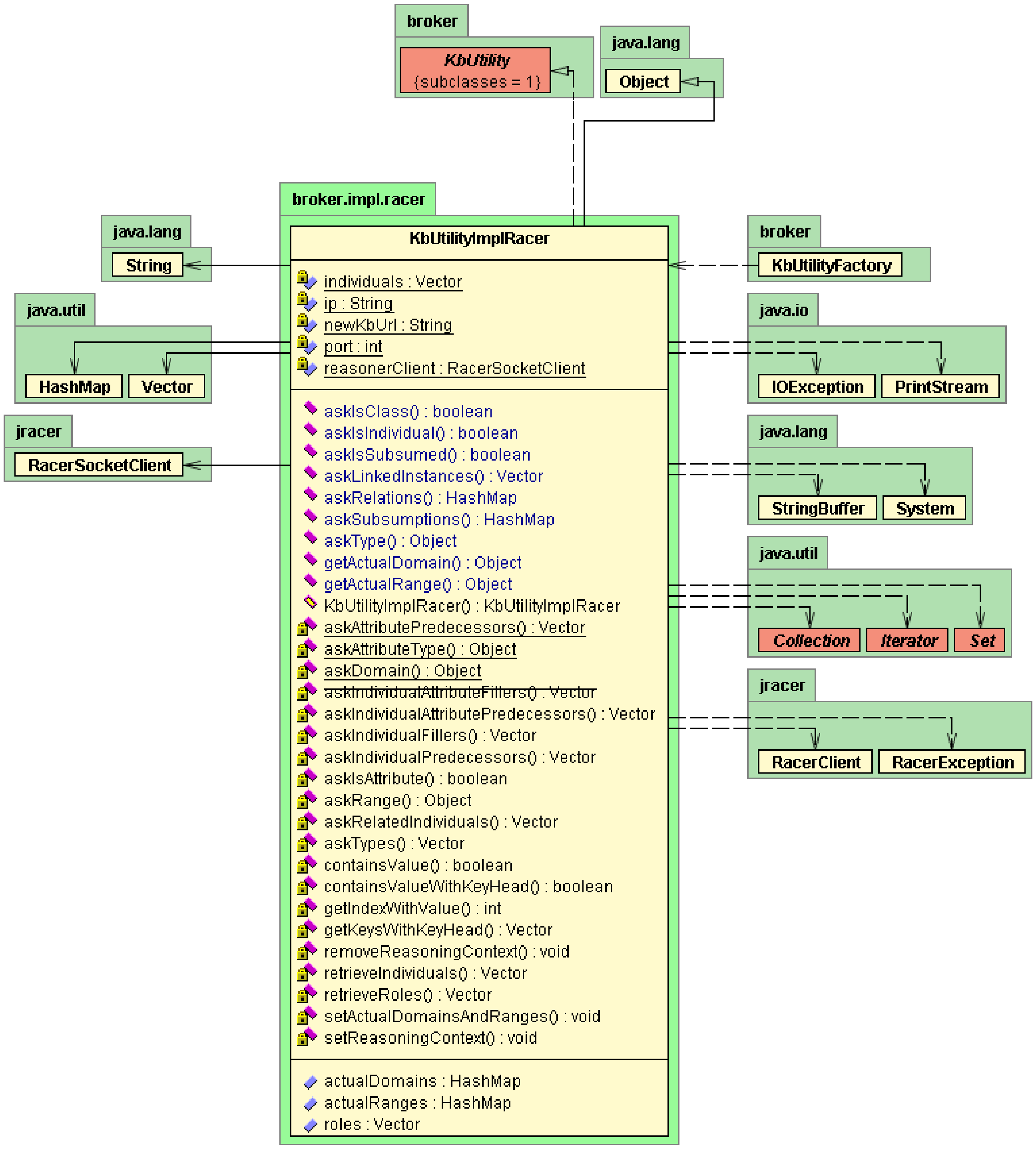

broker
KbUtility
{subclasses = 1}
java.lang
Object
broker.impl.racer
KbUtilityImplRacer
individuals : Vector
ip : String
newKbUrl : String
port : int
reasonerClient : RacerSocketClient
askIsClass() : boolean
askIsIndividual() : boolean
askIsSubsumed() : boolean
askLinkedInstances() : Vector
askRelations() : HashMap
askSubsumptions() : HashMap
askType() : Object
getActualDomain() : Object
getActualRange() : Object
KbUtilityImplRacer() : KbUtilityImplRacer
askAttributePredecessors() : Vector
askAttributeType() : Object
askDomain() : Object
~~askIndividualAttributeFillers() : Vector~~
askIndividualAttributePredecessors() : Vector
askIndividualFillers() : Vector
askIndividualPredecessors() : Vector
askIsAttribute() : boolean
askRange() : Object
askRelatedIndividuals() : Vector
askTypes() : Vector
containsValue() : boolean
containsValueWithKeyHead() : boolean
getIndexWithValue() : int
getKeysWithKeyHead() : Vector
removeReasoningContext() : void
retrieveIndividuals() : Vector
retrieveRoles() : Vector
setActualDomainsAndRanges() : void
setReasoningContext() : void
actualDomains : HashMap
actualRanges : HashMap
roles : Vector
java.lang
String
java.util
HashMap
Vector
jracer
RacerSocketClient
broker
KbUtilityFactory
java.io
IOException
PrintStream
java.lang
StringBuffer
System
java.util
Collection
Iterator
Set
jracer
RacerClient
RacerException

Figure 32. Class diagram for *KbUtilityImplRacer*.

2.5. *RelevanceWithWordNet*

As we roughly said in the introductory paragraph of this chapter, the current implementation exploits a lexical DataBase (DB) via a DBMS server. The lexical DB is

WordNet (we address to [Miller et al., 1990] for a full treatment) while the DBMS is MySql (see [MySql]). The class *RelevanceWithWordNet* actually executes a simplified version of the algorithm for semantic similarity heuristics detailed at section 4, chapter 8. That is the reason for the usage of WordNet. In particular, the *RelevanceWithWordNet* accomplishes its task (i.e. the execution of the mentioned algorithm) by means of some external methods belonging to the class *OntologyManager*. Such a module was imported from a package developed for the SINode component of the SEWASIE system (see chapter3). The useful achievement of *OntologyManager* is that it connects to a static dump of the WordNet DB via MySql Server. Thereby, it allows *RelevanceWithWordNet* to perform the needed checks for semantic similarity w.r.t. that available dump of WordNet.
Figure 33 reports the class diagram for *RelevanceWithWordNet*.

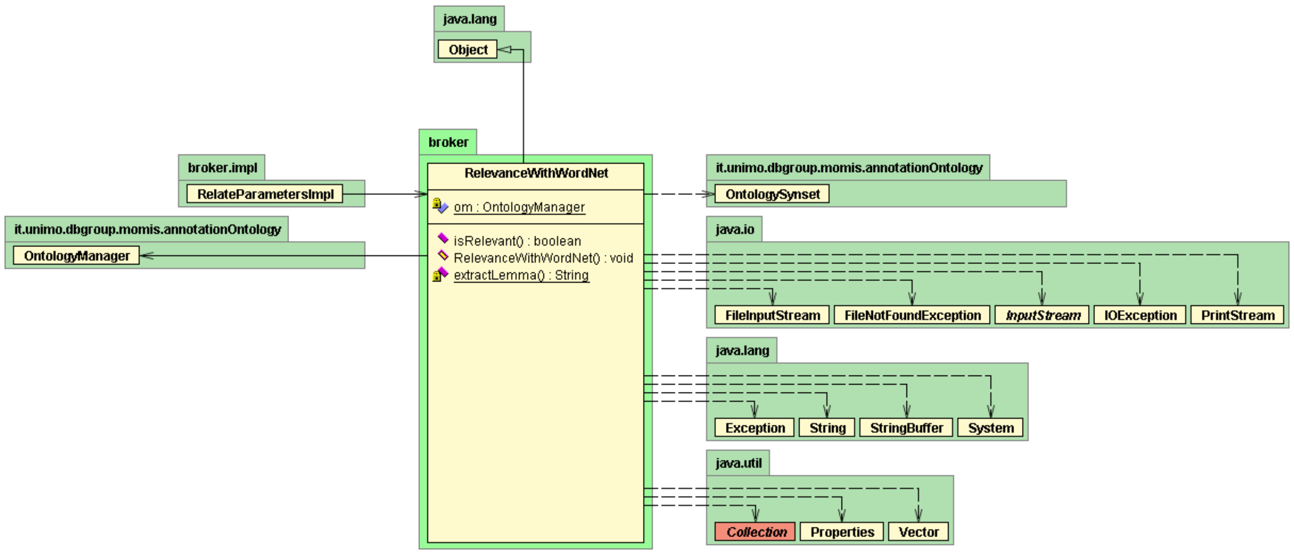


Figure 33. Class diagram for *RelevanceWithWordNet* .

3. Case Study

Testing activity on the implemented software was run w.r.t. to the "mad cows" OWL ontology (available at ["mad_cows" OWL ontology]).
From the algorithmic point of view, the main achievements of this implementation reside in the results coming from the execution of the method *findPath* encoded by class *RelateParameterImpl*.
In the following various outcomes are represented as tree graphs. As it has been said earlier, *findPath* produces two coupled search networks in order to eventually discover a path heading from a root concept to a target concept. A search networks involves the concepts examined, the other one includes the exploited intra-ontology relationships (i.e. search network links). Keeping the root concept as "person" and the target concept as "man", the drawn graphs have the following legends:

- Figure 34: search network for a depth first execution of the *RelateParameters* algorithm, arrested as soon as the target concept has been hit; no semantic similarity heuristics has been applied.

- Figure 35: Network for the exploited links, to be seen as a supplementary explanation of figure 34.
- Figure 36: search network derived from a breadth first execution of the *RelateParameters* algorithm. Here, the algorithm terminates when all the concepts reachable from the given root node have been examined once. No semantic similarity heuristics has aided the search.
- Figure 37: network for the exploited links, specifically related to figure 36.
- Figure 38: search network derived from a breadth first execution of the *RelateParameters* algorithm. Again, the algorithm terminates when all the concepts reachable from the given root node have been examined once. An elementary version of the semantic similarity heuristics defined at section 4, chapter 8 has driven the search.
- Figure 39: Network for the exploited links so as to accompany the figure 38.
- Figure 40: collection of all the paths resulting from the searchs described by the former representations.

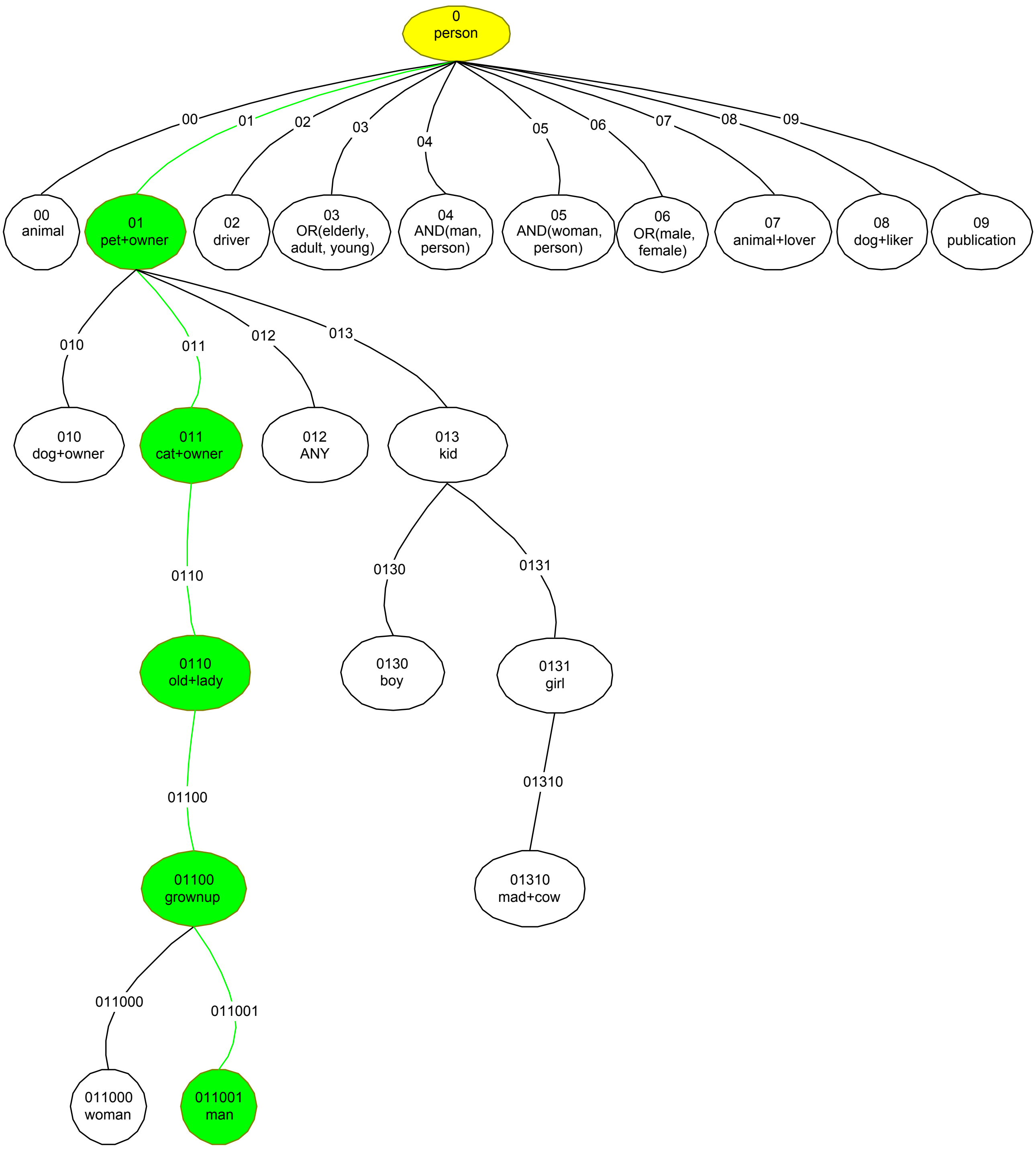


Figure 34. Search network, depth first, termination after the first hit.

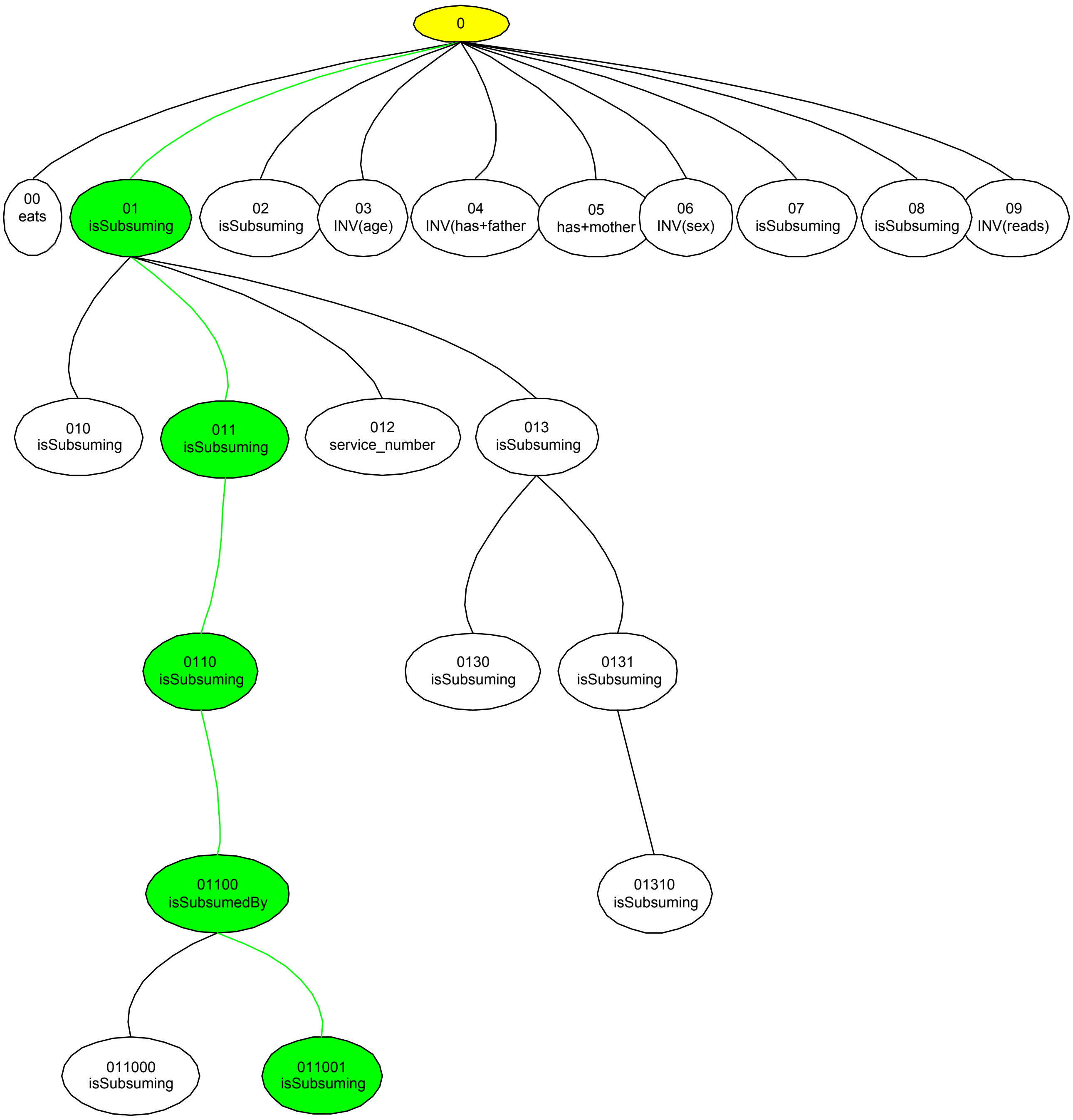


Figure 35. Search network links, depth first, termination after the first hit.

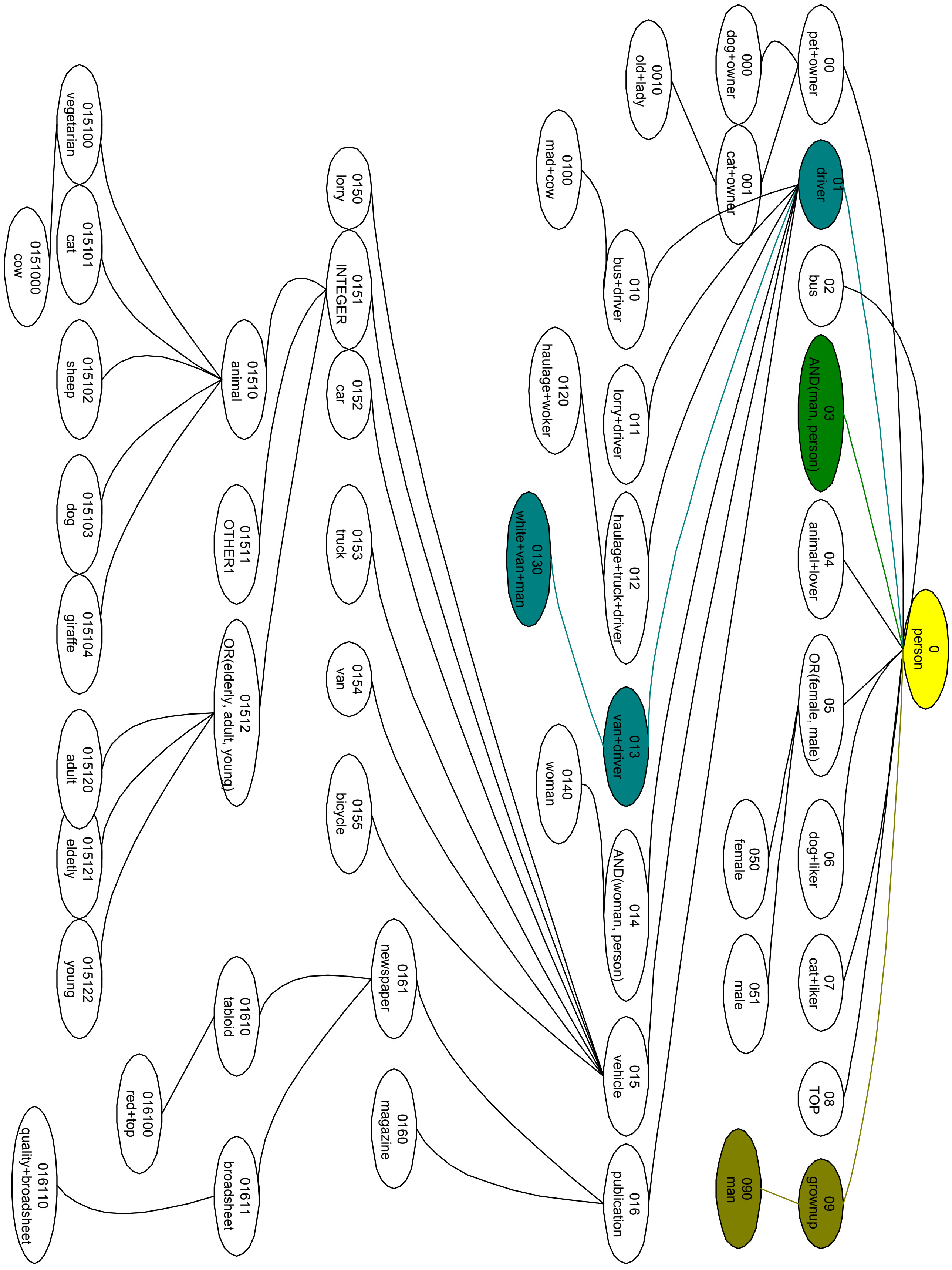


Figure 36. Search network, breadth first, exhaustive search.

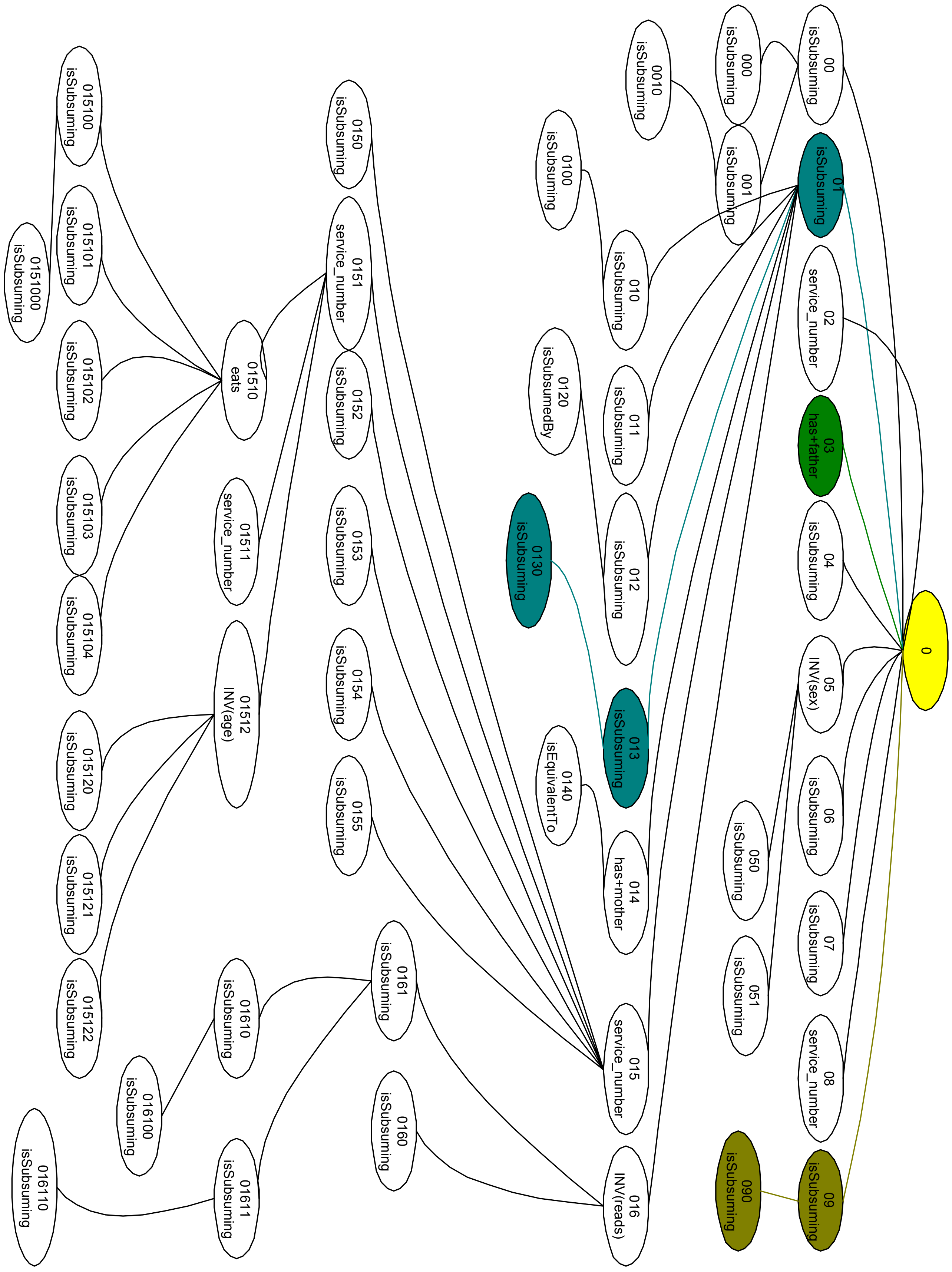


Figure 37. Search network links, breadth first, exhaustive search.

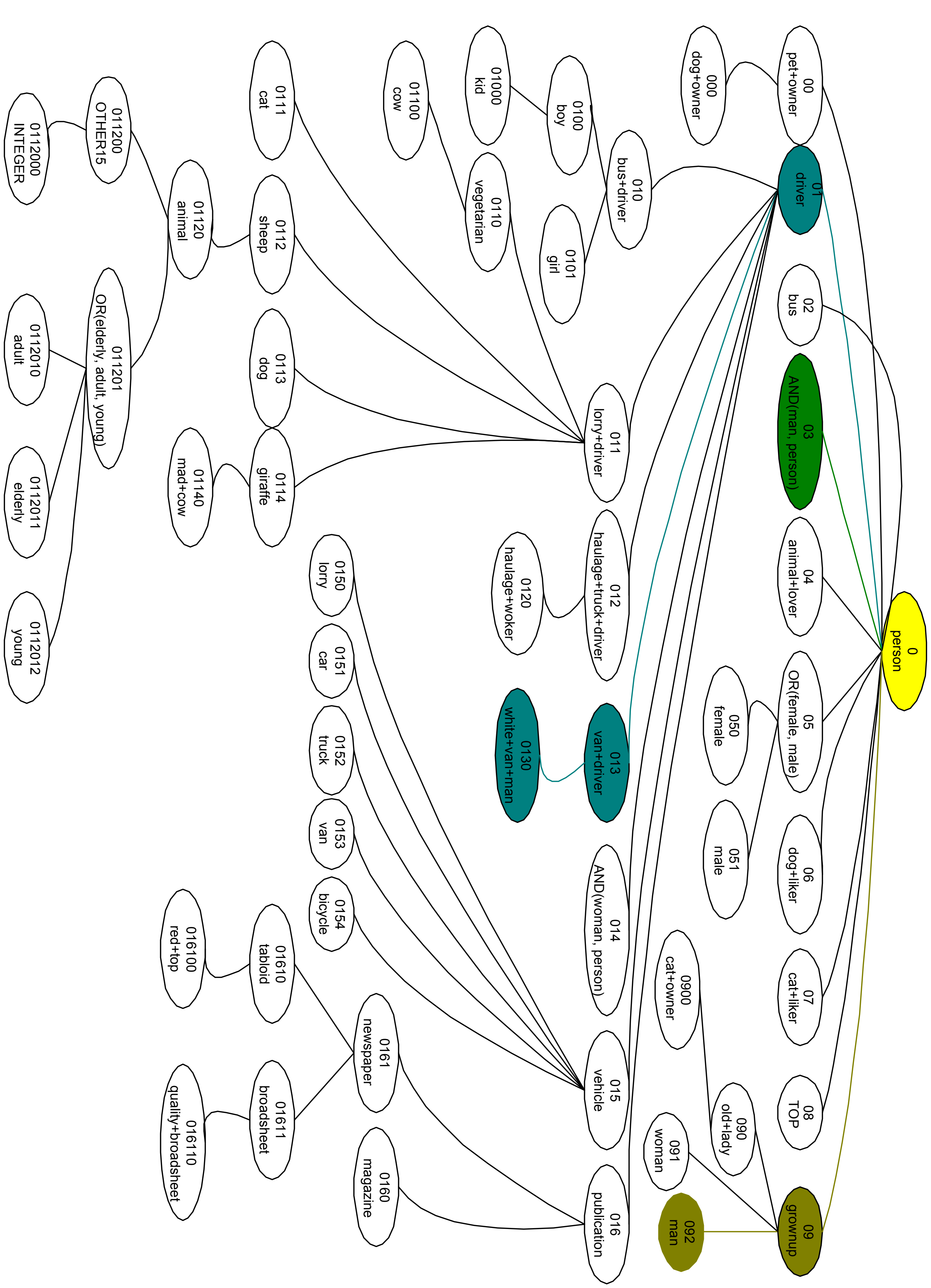


Figure 38. Search network, breadth first, exhaustive search, simplified semantic similarity heuristics.

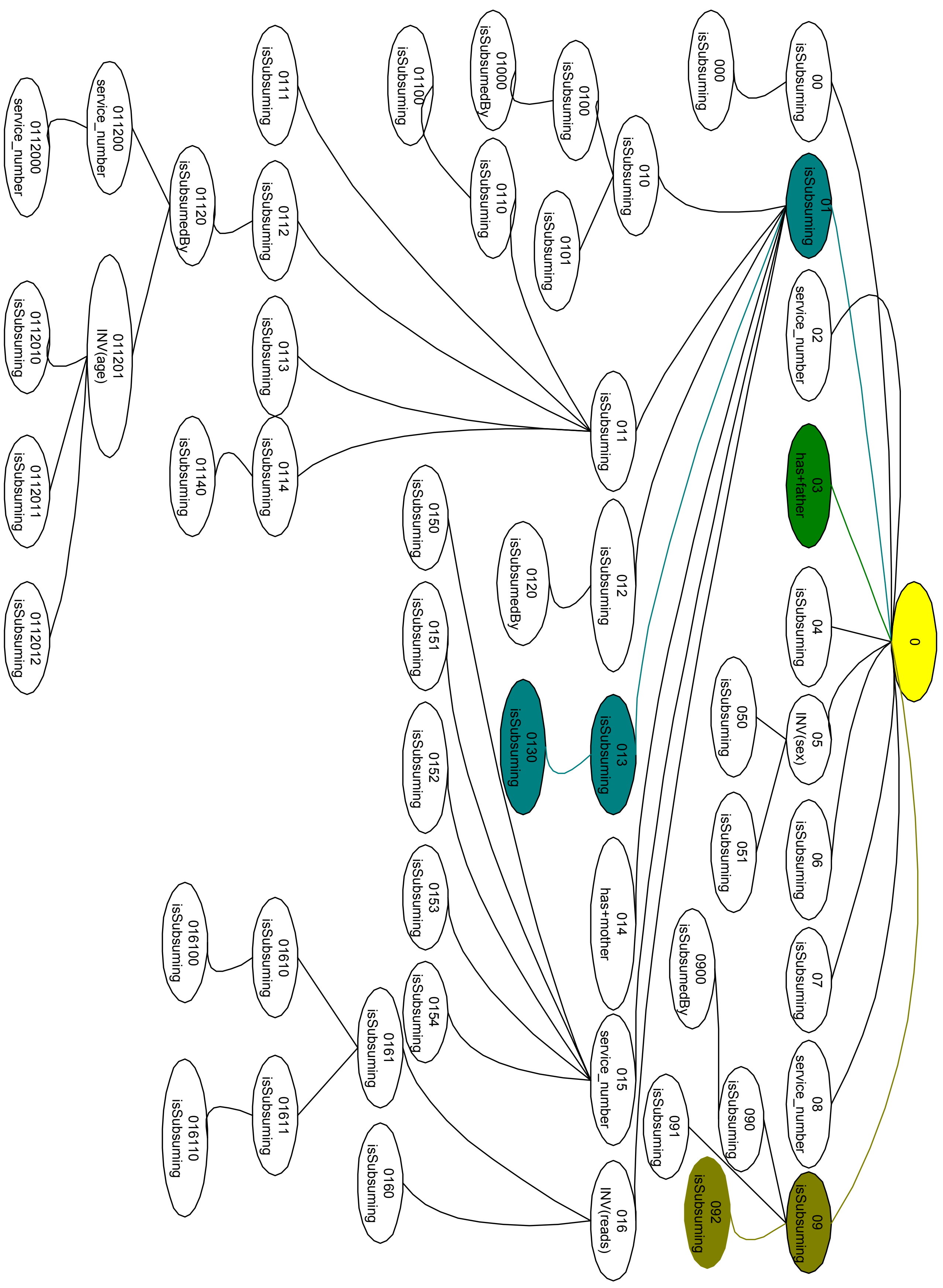


Figure 39. Search network links, breadth first, exhaustive search, simplified semantic similarity heuristics.

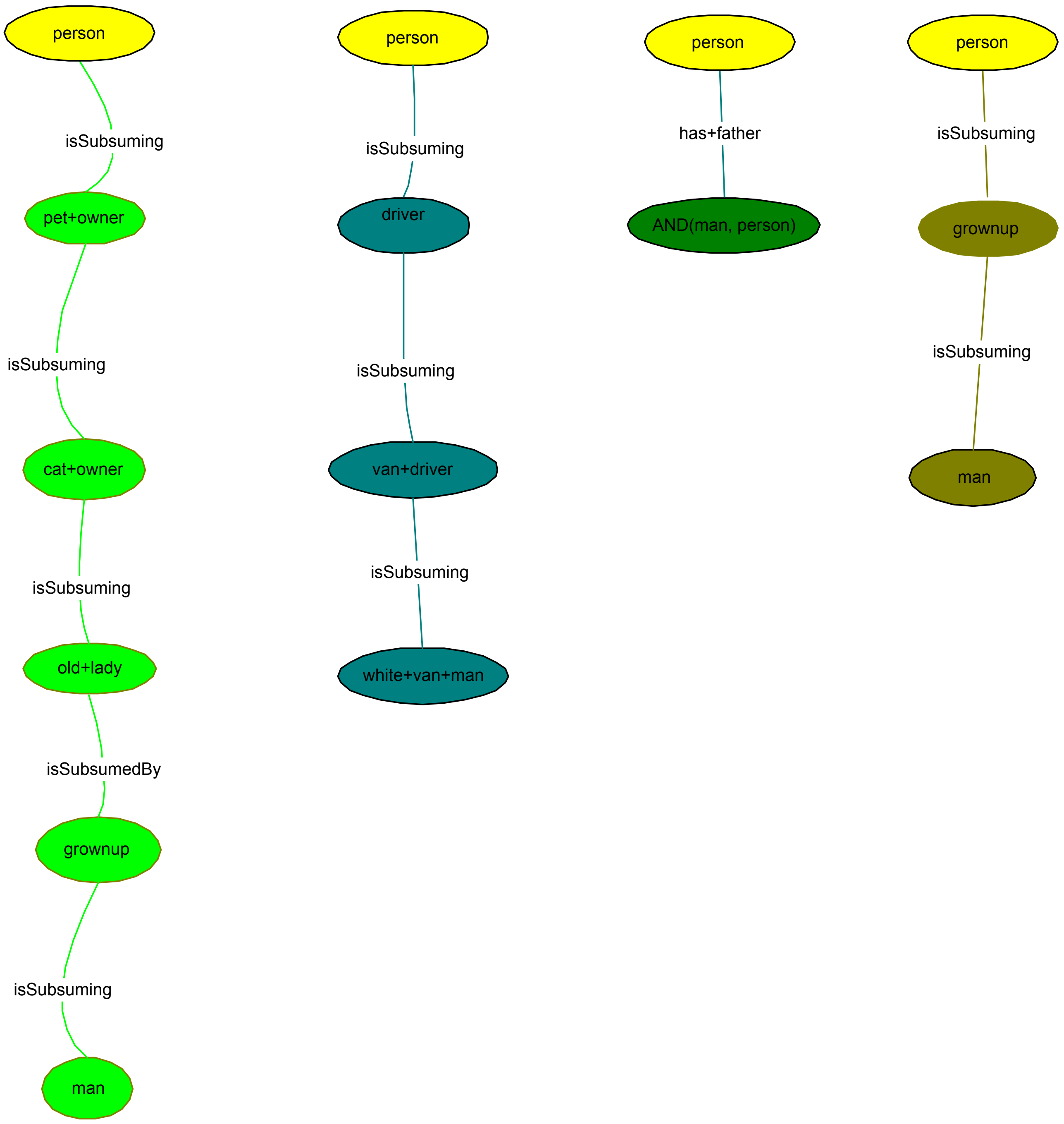


Figure 40. Paths found.

XI Conclusions and Future Work

1. Conclusions

Reviewing all the sections of this work, we can sum up the following contributions, each one drawn from a specific chapter:

- Query capabilities descriptions of an information source allow a mediator-based system to face the Capabilities-Based Rewriting and the Query Expressibility Decision problem. Providing also content capabilities description allows the sytem to fully decide about the answerability of a query (chapter 1).
- Enroling DAML-S as the (query) capability description language let Web services discover service providers through a Semantic Matching process of requesters' Service Profiles against Service Profiles advertised by service providers. Thereby, the DAML-S Broker (DSB) joins Semantic Web annotated semantics with Web services architecture as a semantic autonomous Web service (chapter 4).
- Having devised a middle-agent interaction protocol for the SEWASIE Brokering Agent (BA), the DAML-S paradigm may be applied to provide the required matching functionalities for a query against the BA ontology (chapter 5).
- Within such a framework, Query Processing by a broker middle-agent can be conceived as a multi-phase process basically involving two principal subtasks, i.e. Query Reformulation and Semantic Matching (chapter 5).
- Query Reformulation is divided into two subprocess: Query Abstraction, which produces an abstracted query properly built up for Semantic Matching; Query Rewriting, which rewrites the abstracted query so that a selected service provider is furnished with exactly the required parameters (chapter 5).
- From the operative point of view, Query Abstraction turns instance and variable terms of the initial query into concept terms of the abstracted query (i.e. Instance Abstraction and Variable Abstraction, respectively). Query Rewriting converts concept terms of the abstracted query into concept terms expressing the parameters required by the selected provider (i.e., Concept Transformation). Subsequently, instance terms of the initial (not yet abstracted) query are translated into instances of the previously transformed concepts (i.e., Instance Rewriting). Any subprocess of Query Reformulation is performed w.r.t. a reference OWL DL ontology shared by the requesters, the broker middle-agent, and the providers (chapter 8).
- Semantic similarty heuristics, also exploiting the WordNet lexical database, aids Query Reformulation (chapter 8).
- A prooved exportability of OWL DL ontology language from/to ODL_{I3} allows the reference ontology for Query Reformulation to be specified in ODL_{I3} as well as in OWL DL (chapter 6).
- In a first instance, Query Reformulation was thought to solve some still untackled issues for the DSB. Namely, requester's query conversion into a request formulated

as a Service Profile; selection of the best fitting provider among all those reported by the Semantic Matching engine; determination of non-output-parameters matching; transformation of inputs among the non-output-parameters supplied by the requester into exactly those inputs needed by the provider for executing the service (chapter 5).

- A further open issue for the DSB was the choice of a suitable query language. DQL was claimed to properly respond this problem (chapter 5).
- Since the developed algorithms for Query Reformulation are self-standing, Query Abstraction or Query Rewriting can be executed any time it is needed to respectively abstract or rewrite a query w.r.t. some target concepts. Hence, the BA might also benefit from Query Reformulation (general remark).

2. Future Work

Since Query Reformulation has been investigated for queries constructed as binary predicate triples only, upcoming research is firstly committed to deal with this shortcoming. Hence, research to deal with queries constructed as conjunctions of binary predicate triples will be understaken.

Such a work underlies the matter of *predicate interpretation*. In fact, each triple composing a conjunctive query can be abstracted and rewritten w.r.t. a conjunction of target triples only if a corrispondence among the respective predicates is formerly identified. In other words, each predicate embodied in the conjunctive query to be reformulated must match with a predicate in the conjunctive expression which describes the target for Query Reformulation. According to the motivating framework kept heretofore, an incoming conjunctive query should be reformulated w.r.t. a Service Profile advertisement which is itself expressable as a conjunction of binary predicate triples. In particular, such a reshaping mainly involves Query Rewriting.
By the way, it is also necessary to revise the whole Semantic Matching method, so that predicate matching is taken into account.

Going back to the current standpoint, that is Query Reformulation w.r.t. Service Profiles, it should be possible to perform Abstraction and Rewriting over any parameter constituting a Service Profile (i.e. input, output, precondition, effect, service parameter, etc). At the moment, operative procedures have been formulated w.r.t. inputs, only. Future work should address Query Reformulation over the remaining parameters, too.

However, Query Abstraction depends on the interpretation of the terms to be abstracted as peculiar parameters, while Query Rewriting must be consistent with the outcomes of Semantic Matching. Above all, it is then necessary to extract the most significant Service Profile parameters (at least inputs, outputs, preconditions, effects) from the query to be processed (thus, reformulated, too). This problem has been marginally treated at section 3, chapter 5, specifically concerning with DQL (DAML Query Language) queries. A deeper investigation is then mandatory, accurately checking OQL_{I3} queries (see chapter 3) interpretation as well. Full applicability to Query Processing within the SEWASIE framework would be thus achieved.

Keeping the focus on running the implementad software for Query Reformulation within the SEWASIE architecture, further developments might be devoted to the usage of ODB-Tools as the actual reasoning engine (as a valid alternative to RACER).

Instead, a satisfactory implementation of the DSB would eventually require an interface to accept DQL queries. About this, a preliminary survey has already been done. So, the main advise would be to adapt the client package of the DQL toolkit, publicly available at [DQL], for the needs of the actual DSB query interface.

In the end, the applied algorithms for Query Reformulation might be optimized with some supplementary and/or complementary heuristics (in addition to the current one, which is based on semantic similarity).

Appendix

1. Translations

1.1. From OWL DL to ODL_{I3}

1.1.1. Fragments of "mad cows" OWL DL Ontology

```
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#white+van+man">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#man"/>
                    <owl:Restriction>
                        <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#drives"/>
                        <owl:someValuesFrom>
                            <owl:Class>
                                <owl:intersectionOf rdf:parseType="Collection">
                                    <owl:Restriction>
                                    <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#has+colour"/>
                                    <owl:someValuesFrom>
                                    <owl:Class>
                                    <owl:oneOf rdf:parseType="Collection">
                                    <owl:Thing
rdf:about="http://cohse.semanticweb.org/ontologies/people#white"/>
                                    </owl:oneOf>
                                    </owl:Class>
                                    </owl:someValuesFrom>
                                    </owl:Restriction>
                                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#van"/>
                                </owl:intersectionOf>
                            </owl:Class>
                        </owl:someValuesFrom>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#publication"/>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#giraffe">
        <rdfs:subClassOf>
            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#animal"/>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
```

```
                <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#eats"/>
                <owl:allValuesFrom>
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#leaf"/>
                </owl:allValuesFrom>
            </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#cat+liker">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#person"/>
                    <owl:Restriction>
                        <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#likes"/>
                        <owl:someValuesFrom>
                            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#cat"/>
                        </owl:someValuesFrom>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#cat+owner">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#person"/>
                    <owl:Restriction>
                        <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#has+pet"/>
                        <owl:someValuesFrom>
                            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#cat"/>
                        </owl:someValuesFrom>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#grownup">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#person"/>
                    <owl:Restriction>
                        <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#age"/>
                        <owl:someValuesFrom>
                            <owl:Class>
                                <owl:oneOf rdf:parseType="Collection">
                                    <owl:Thing
rdf:about="http://cohse.semanticweb.org/ontologies/people#adult"/>
                                    <owl:Thing
```

```
rdf:about="http://cohse.semanticweb.org/ontologies/people#elderly"/>
                                </owl:oneOf>
                            </owl:Class>
                        </owl:someValuesFrom>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#quality+broadsheet">
        <rdfs:subClassOf>
            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#broadsheet"/>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#vehicle"/>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#newspaper">
        <rdfs:subClassOf>
            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#publication"/>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#broadsheet"/>
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#tabloid"/>
                </owl:unionOf>
            </owl:Class>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#bus+company">
        <rdfs:subClassOf>
            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#company"/>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#pet+owner">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#person"/>
                    <owl:Restriction>
                        <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#has+pet"/>
                        <owl:someValuesFrom>
                            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#animal"/>
                        </owl:someValuesFrom>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#mad+cow">
```

```
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#cow"/>
                    <owl:Restriction>
                        <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#eats"/>
                        <owl:someValuesFrom>
                            <owl:Class>
                                <owl:intersectionOf rdf:parseType="Collection">
                                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#brain"/>
                                    <owl:Restriction>
                                    <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#part+of"/>
                                    <owl:someValuesFrom>
                                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#sheep"/>
                                    </owl:someValuesFrom>
                                    </owl:Restriction>
                                </owl:intersectionOf>
                            </owl:Class>
                        </owl:someValuesFrom>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
    <owl:Class rdf:about="http://cohse.semanticweb.org/ontologies/people#boy">
        <owl:equivalentClass>
            <owl:Class>
                <owl:intersectionOf rdf:parseType="Collection">
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#person"/>
                    <owl:Restriction>
                        <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#sex"/>
                        <owl:someValuesFrom>
                            <owl:Class>
                                <owl:oneOf rdf:parseType="Collection">
                                    <owl:Thing
rdf:about="http://cohse.semanticweb.org/ontologies/people#male"/>
                                </owl:oneOf>
                            </owl:Class>
                        </owl:someValuesFrom>
                    </owl:Restriction>
                    <owl:Restriction>
                        <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#age"/>
                        <owl:someValuesFrom>
                            <owl:Class>
                                <owl:oneOf rdf:parseType="Collection">
                                    <owl:Thing
rdf:about="http://cohse.semanticweb.org/ontologies/people#young"/>
                                </owl:oneOf>
                            </owl:Class>
                        </owl:someValuesFrom>
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>
    <owl:Class rdf:about="http://cohse.semanticweb.org/ontologies/people#bus">
        <rdfs:subClassOf>
            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#vehicle"/>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://cohse.semanticweb.org/ontologies/people#car">
        <rdfs:subClassOf>
            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#vehicle"/>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://cohse.semanticweb.org/ontologies/people#cat">
        <rdfs:subClassOf>
            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#animal"/>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://cohse.semanticweb.org/ontologies/people#cow">
        <rdfs:subClassOf>
            <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#vegetarian"/>
        </rdfs:subClassOf>
    </owl:Class>
    <owl:Class rdf:about="http://cohse.semanticweb.org/ontologies/people#dog">
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty
rdf:resource="http://cohse.semanticweb.org/ontologies/people#eats"/>
                <owl:someValuesFrom>
                    <owl:Class
rdf:about="http://cohse.semanticweb.org/ontologies/people#bone"/>
                </owl:someValuesFrom>
            </owl:Restriction>
        </rdfs:subClassOf>
```

1.1.2. Corresponding ODL$_{I3}$ Descriptions

```
interface white+van+man ()
  { };
interface man ()
  { };
interface white ()
  { };
interface van ()
  { };
interface publication ()
  { };
interface animal ()
  { };
interface leaf+eater ()
  { };
interface giraffe : animal, leaf+eater ()
  { };
interface cat+liker ()
  { };
interface cat+owner ()
  { };
interface grownup ()
  { };
```

```
interface quality+broadsheet : broadsheet ()
  { };
interface vehicle ()
  { };
interface newspaper : publication, broadsheet+tabloid ()
  { };
interface broadsheet+tabloid : union broadsheet, tabloid ()
  { };
interface bus+company : company, tabloid ()
  { };
interface pet+owner ()
  { };
interface mad+cow ()
  { };
interface boy ()
  { };
interface bus : vehicle ()
  { };
interface car : vehicle ()
  { };
interface cat : animal ()
  { };
interface cow : vegetarian ()
  { };
interface bone+eater ()
  { };
interface dog : bone+eater ()
  { };

rule rule1 forall x in man:
  (  exists x1 in x.drives:
       (  x1 in van  ) and
          (  exists x2 in x1.has+colour:
               (  x2 in white  )  )  )
  then
    x in man+van+white;

// Giving for granted that the range of drives is always of type van, we might
equally write:
// forall x in man:
//   ( exists x1 in x.drives:
//       (  x1.has+colour in white  )  )
//   then
//     x in man+van+white;

rule rule2 forall x in animal:
             (  x.eats in leaf  )
             then
               x in leaf+eater;

rule rule3 forall x in person:
             (  exists x1 in x.likes:
                  (  x1 in cat  )  )
             then
               x in cat+liker;

rule rule4 forall x in person:
             (  exists x1 in x.has+pet:
                  (  x1 in cat  )  )
             then
               x in cat+owner;

rule rule5 forall x in person:
```

```
             (  exists x1 in x.age:
                  (  x1 in adult  )  )
             then
               x in grownup;

rule rule6 forall x in person:
             (  exists x1 in x.age:
                  (  x1 in elderly  )  )
             then
               x in grownup;

rule rule7 forall x in person:
             (  exists x1 in x.has+pet:
                  (  x1 in animal  )  )
             then
               x in pet+owner;

rule rule8 forall x in cow:
             (  exists x1 in x.eats:
                  (  x1 in brain  )  and
                  (  exists  x2 in x1.partOf:
                       (  x2 in sheep  )  )  )
             then
               x in mad+cow;

rule rule9 forall x in person:
             (  exists x1 in x.sex:
                  (  x1 in male  )  )  and
             (  exists  x2 in x.age:
                  (  x2 in young  )  )
             then
               x in boy;

rule rule10 forall x in animal:
             (  exists x1 in x.eats:
                  (  x1 in bone  )  )
             then
               x in bone+eater
```

1.2. From ODL$_{I3}$ to OWL DL

1.2.1. ODL$_{I3}$ "storage" Schema

```
interface Material
( extent materials
  keys name)
{  attribute string name;
   attribute int risk;
   attribute set %60 string> feature;
};

interface SMaterial : Material
( extent smaterials
  keys name)
{  }
;

interface Manager
( extent managers
  keys name)
```

```
{  attribute string                name;
   attribute range {40000, 100000} salary;
   attribute range{1, 15}          level;
};

interface TManager : Manager
( extent tmanagers
  keys name)
{  attribute range{8, 12}       level;
};

interface  Storage
( extent storages
  keys name)
{  attribute string         name;
   attribute string         category;
   attribute Manager        managed_by;
   typedef struct _stock
      {
          Material         item;
          range {10, 300}  qty;
      } t_stock;
   attribute set    stock;
};

interface  SStorage : Storage
( extent storages
  keys name)
{   };

rule rule1 forall X in Manager:   X.level >= 5 and X.level <= 10
  then                            X.salary >= 40000 and X.salary <= 60000 ;

rule rule2 forall X in Material:  X.risk >=  10
  then                            X in SMaterial ;

rule rule3 forall X in Storage:   X.category = "B4"
  then                            X.managed_by in TManager ;

rule rule4 forall X in Storage:   for all X1 in X.stock: X1.item in SMaterial
  then                            X in SStorage ;

rule rule5 forall X in Storage:   forall X1 in X.stock: X1.qty >= 10 and
                                                      X1.qty <=50
  then                            X.category = "A2" ;
```

1.2.2. Corresponding OWL DL Ontology

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY this "file:c:\temp\material.owl">
]>

<rdf:RDF
  xmlns:rdf = "&rdf;#"
  xmlns:rdfs ="&rdfs;#"
  xmlns:owl ="&owl;#"
  xmlns:xsd = "&xsd;#"
  xmlns     ="&this;#"
  xmlns:this="&this;#"
 >

<owl:Ontology rdf:about="">
 <owl:versionInfo/>
 <!-- owl:imports rdf:resource="&owl;"
 owl:imports rdf:resource="&rdf;"
 owl:imports rdf:resource="&rdfs;"
 owl:imports rdf:resource="&owl;"
 -->
</owl:Ontology>

<owl:DatatypeProperty rdf:ID="name">
  <rdf:type rdf:resource="&owl;#FunctionalProperty"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="risk"/>
<owl:DatatypeProperty rdf:ID="feature"/>
<owl:DatatypeProperty rdf:ID="salary"/>
<owl:DatatypeProperty rdf:ID="level"/>
<owl:DatatypeProperty rdf:ID="category"/>
<owl:ObjectProperty rdf:ID="managed_by"/>
<owl:ObjectProperty rdf:ID="item"/>
<owl:DatatypeProperty rdf:ID="qty"/>
<owl:ObjectProperty rdf:ID="stock"/>

<owl:Class rdf:ID="Material">
  <name rdf:resource="&xsd;#string"/>
  <risk rdf:resource="&xsd;#integer"/>
  <feature rdf:resource="&xsd;#string"/>
</owl:Class>

<Material rdf:ID="m1">
  <feature rdf:datatype="&xsd;#string">raw</feature>
  <feature rdf:datatype="&xsd;#string">ready</feature>
</Material>

<owl:Class rdf:ID="SMaterial">
  <rdfs:subClassOf rdf:resource="#Material"/>
</owl:Class>

<owl:Class rdf:ID="Manager">
  <name rdf:resource="&xsd;#string"/>
  <salary>
    <owl:DataRange>
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#positiveInteger">40000</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">100000</rdf:first>
              <rdf:rest rdf:resource="&rdf;#nil"/>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
    </owl:DataRange>
  </salary>
  <level>
    <owl:DataRange>
      <owl:oneOf>
```

```
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#positiveInteger">1</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">15</rdf:first>
              <rdf:rest rdf:resource="&rdf;#nil"/>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
    </owl:DataRange>
  </level>
</owl:Class>

<owl:Class rdf:ID="TManager">
  <rdfs:subClassOf rdf:resource="#Manager"/>
  <level>
    <owl:DataRange>
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#positiveInteger">8</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">12</rdf:first>
              <rdf:rest rdf:resource="&rdf;#nil"/>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
    </owl:DataRange>
  </level>
</owl:Class>

<owl:Class rdf:ID="T_stock">
  <item rdf:resource="#Manager"/>
  <qty>
    <owl:DataRange>
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#positiveInteger">10</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">300</rdf:first>
              <rdf:rest rdf:resource="&rdf;#nil"/>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
    </owl:DataRange>
  </qty>
</owl:Class>

<owl:Class rdf:ID="Storage">
  <name rdf:resource="&xsd;#string"/>
  <category rdf:resource="&xsd;#string"/>
  <managed_by rdf:resource="#Manager"/>
  <stock rdf:resource="#T_stock"/>
</owl:Class>

<owl:Class rdf:ID="SStorage">
  <rdfs:subClassOf rdf:resource="#Storage"/>
</owl:Class>
```

```
<owl:Class rdf:ID="MiddleLevelManager">
  <rdfs:subClassOf rdf:resource="#Manager"/>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#level"/>
      <owl:allValuesFrom rdf:resource="#MiddleLevel"/>
    </owl:Restriction>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#level"/>
      <owl:allValuesFrom rdf:resource="#MiddleSalary"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:DataRange rdf:ID="MiddleLevel">
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#positiveInteger">5</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">6</rdf:first>
              <rdf:rest>
                <rdf:List>
                  <rdf:first rdf:datatype="&xsd;#positiveInteger">7</rdf:first>
                  <rdf:rest>
                    <rdf:List>
                      <rdf:first
rdf:datatype="&xsd;#positiveInteger">8</rdf:first>
                      <rdf:rest>
                        <rdf:List>
                          <rdf:first
rdf:datatype="&xsd;#positiveInteger">9</rdf:first>
                          <rdf:rest>
                            <rdf:List>
                              <rdf:first
rdf:datatype="&xsd;#positiveInteger">10</rdf:first>
                              <rdf:rest rdf:resource="&rdf;#nil"/>
                            </rdf:List>
                          </rdf:rest>
                        </rdf:List>
                      </rdf:rest>
                    </rdf:List>
                  </rdf:rest>
                </rdf:List>
              </rdf:rest>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
</owl:DataRange>

<owl:DataRange rdf:ID="MiddleSalary">
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#positiveInteger">40000</rdf:first>
          <rdf:rest>
            ...
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#positiveInteger">45000</rdf:first>
              <rdf:rest>
                ...
                <rdf:List>
                  <rdf:first
```

```
rdf:datatype="&xsd;#positiveInteger">50000</rdf:first>
                  <rdf:rest>
                    ...
                    <rdf:List>
                      <rdf:first
rdf:datatype="&xsd;#positiveInteger">52000</rdf:first>
                      <rdf:rest>
                        ...
                        <rdf:List>
                          <rdf:first
rdf:datatype="&xsd;#positiveInteger">55000</rdf:first>
                          <rdf:rest>
                            ...
                            <rdf:List>
                              <rdf:first
rdf:datatype="&xsd;#positiveInteger">60000</rdf:first>
                              <rdf:rest rdf:resource="&rdf;#nil"/>
                            </rdf:List>
                          </rdf:rest>
                        </rdf:List>
                      </rdf:rest>
                    </rdf:List>
                  </rdf:rest>
                </rdf:List>
              </rdf:rest>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
</owl:DataRange>

<owl:Class rdf:ID="HighRiskMaterial">
  <rdfs:subClassOf rdf:resource="#Material"/>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#risk"/>
      <owl:allValuesFrom rdf:resource="#HighRisk"/>
    </owl:Restriction>
    <owl:Thing rdf:about="#SMaterial"/>
  </owl:intersectionOf>
</owl:Class>

<owl:DataRange rdf:ID="HighRisk">
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#integer">10</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#integer">20</rdf:first>
              <rdf:rest>
                <rdf:List>
                  <rdf:first rdf:datatype="&xsd;#integer">30</rdf:first>
                  <rdf:rest>
                    <rdf:List>
                      <rdf:first rdf:datatype="&xsd;#integer">40</rdf:first>
                      <rdf:rest>
                        <rdf:List>
                          <rdf:first rdf:datatype="&xsd;#integer">50</rdf:first>
                          <rdf:rest>
                            <rdf:List>
                              <rdf:first
rdf:datatype="&xsd;#integer">60</rdf:first>
                              <rdf:rest rdf:resource="&rdf;#nil"/>
                            </rdf:List>
                          </rdf:rest>
                        </rdf:List>
                      </rdf:rest>
                    </rdf:List>
                  </rdf:rest>
                </rdf:List>
              </rdf:rest>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
</owl:DataRange>

<owl:Class rdf:ID="TManagedStorage">
  <rdfs:subClassOf rdf:resource="#Storage"/>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#category"/>
      <owl:allValuesFrom rdf:resource="#B4Category"/>
    </owl:Restriction>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#managed_by"/>
      <owl:allValuesFrom rdf:resource="#TManager"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:DataRange rdf:ID="B4Category">
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#string">B4</rdf:first>
          <rdf:rest rdf:resource="&rdf;#nil"/>
        </rdf:List>
      </owl:oneOf>
</owl:DataRange>

<owl:Class rdf:ID="SMaterialStorage">
  <rdfs:subClassOf rdf:resource="#Storage"/>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#stock"/>
      <owl:allValuesFrom rdf:resource="#SMaterialItemT_Stock"/>
    </owl:Restriction>
    <owl:Thing rdf:about="#SStorage"/>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="SMaterialItemT_Stock">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#item"/>
      <owl:allValuesFrom rdf:resource="#SMaterial"/>
    </owl:Restriction>
    <owl:Thing rdf:about="#T_Stock"/>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="A2Storage">
  <rdfs:subClassOf rdf:resource="#Storage"/>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#stock"/>
      <owl:allValuesFrom rdf:resource="#ShortQtyT_Stock"/>
    </owl:Restriction>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#category"/>
      <owl:allValuesFrom rdf:resource="#A2Category"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="ShortQtyT_Stock">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#qty"/>
      <owl:allValuesFrom rdf:resource="#ShortQty"/>
    </owl:Restriction>
    <owl:Thing rdf:about="#T_Stock"/>
  </owl:intersectionOf>
</owl:Class>

<owl:DataRange rdf:ID="ShortQty">
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#integer">10</rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;#integer">20</rdf:first>
              <rdf:rest>
                <rdf:List>
                  <rdf:first rdf:datatype="&xsd;#integer">30</rdf:first>
                  <rdf:rest>
                    <rdf:List>
                      <rdf:first rdf:datatype="&xsd;#integer">40</rdf:first>
                      <rdf:rest>
                        <rdf:List>
                          <rdf:first rdf:datatype="&xsd;#integer">45</rdf:first>
                          <rdf:rest>
                            <rdf:List>
                              <rdf:first
rdf:datatype="&xsd;#integer">50</rdf:first>
                              <rdf:rest rdf:resource="&rdf;#nil"/>
                            </rdf:List>
                          </rdf:rest>
                        </rdf:List>
                      </rdf:rest>
                    </rdf:List>
                  </rdf:rest>
                </rdf:List>
              </rdf:rest>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
</owl:DataRange>

<owl:DataRange rdf:ID="A2Category">
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;#string">A2</rdf:first>
          <rdf:rest rdf:resource="&rdf;#nil"/>
        </rdf:List>
      </owl:oneOf>
</owl:DataRange>
```

```
</rdf:RDF>
```

2. Acronyms

BA = (SEWASIE) Brokering Agent
BPEL4WS = Business Process Execution Language for Web Services
CBR = Capabilities-Based Rewriting
CLASSIC = CLASSification of Individuals and Concepts
CWA = Closed Word Assumption
D2I = Data to Information
DAML = Darpa Agent Markup Language
DAML-S = DAML for web Services
DBMS = DataBase Management System
DIG = Description Logic Implementation Group
DL = Description Logic
DQL = DAML Query Language
DSB = DAML-S Broker
DSVM = Daml-S Virtual Machine
FaCT = Fast Classification of Terminologies
FGFFV = Fraunhofer-Gesellshaft zur Förderung der angewandten Forschung eingetragener Verein
FOL = First Order Logic
GVV = Global Virtual View
I3 = Intelligent Information Integration
IOPE = Inputs, Outputs, Preconditions, and Effects
KB = Knowledge Base
KRSS = Knowledge Representation System Specification
MAS = Multi-Agents System
MOMIS = Mediator environment for Multiple Information Sources
OB = (SEWASIE) Ontology Builder
ODL = Object Data Language
ODM = Object Data Model
ODMG = Object Data Management Group
O-O = Object-Oriented
OIL = Ontology Inference Layer
OQL = Object Query Language
OWL = Ontology Web Language
OWA = Open World Assumption
QA = (SEWASIE) Query Agent
QED = Query Expressibility Decision
QM = (SEWASIE) Query Manager
P2P = Peer to Peer
RACER = Renamed ABox and Concept Expression Reasoner
RDF = Resource Description Framework
RDFS = RDF Schema
RWTH = Rheinisch Westfaelische Technische Hochschule Aachen
SEWASIE = SEmantic Webs and AgentS in Integrated Economies
SIMS = Services and Information Management for Decision Systems
SINode = SEWASIE Information Node
s.t. = such that
SVN = SEWASIE Virtual Network

TSIMMIS = The Stanford IBM Manager of Multiple Information Sources
UDDI = Universal Desciption Discovery and Integration
UNA = Unique Name Assumption
VDS = Virtual Data Store
w.r.t. = with respect to
WSCI = Web Service Choreography Interface
WSDL = Web Service Description Language
XML = eXtensive Markup Language
XMLS = XML Schema

3. Glossary

(autonomous) *Agent*
A system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.

Assertion
Any expression which is claimed to be true. Similarly, the act of claiming something to be true.

Attribute
A role relating two individuals, which can belong to abstract concepts as well as concrete concepts (see *Role*).
E.g., related individuals might me "John" and "Lucy" (where "John" and "Lucy" are instances of two abstract concepts like "mother" and "father"), or "John" and "3" (where "John" is an instance of an abstract concept like "father" and "3" is an instance of a concrete concept like "integer"), or even "John" and "3" (where "John" is an instance of a concrete concept like "string" and "3" is an instance of a concrete concept like "integer").

Axiom
A well formed fromula that is stipulated as unproved premise for the proof of other well formed formulas inside a formal system (see *Formal* and *Well Formed Formula*).

Broker
A middle agent essentially characterized by its interleaving position between requester and provider. It understands both the needs of a requester and the capabilities of a provider, and routes both requests and replies appropriately.

Class
A general concept, category, or classification (see *Concept*). Something used primarily to classify or categorize other things.
W.r.t. OWL ontology language, a resource of type *rdfs:Class* with an associated set of resources all of which have the class as a value of the *rdf:type* property (see *Resource* and *Property*)

Complete
Able to draw all valid inferences (see *Inference*)

Concept
A category comprising a certain set of individuals. In predicate logic, The memberiship of an individual to a concept can be expressed via a unary predicate.

Consistent
True under all intepretations. Possible to satisfy (see *Satisfiable*).

Construct
A description in a knowledge representation language.

Constructor
An element of a knowledge representation language used to create a construct (see *Construct*).

Correct
See *Sound*.

Decidable
Said of a computational process that terminates in a finite time.

Description Logics
A family of knowledge representation formalisms that represent the knowledge of an application domain (the "world") by first defining the relevant concepts of the domain (its terminology), and then using these concepts to specify properties of objects and individuals occurring in the domain (the world description).

End-Agent
Agent which needs or can offer services (see *Agent*).

Entailment
A semantic relationship between expressions which holds whenever the truth of the first guarantees the truth of the second. Equivalently, whenever it is logically impossible for the first expression to be true and the second one to be false. Equivalently, when any interpretation which satisfies the first also satisfies the second.

Extensional
A set-based theory of logic of classes, in which classes are considered to be sets, properties considered to be sets of *(object, value)* pairs, and so on. A theory which admits no distinction between entities with the same extension.

First-order predicate logic
Predicate logic in which predicates take only individuals as arguments and quantifiers only bind individual variables (see *Predicate Logic*).

Formal
Couched in language sufficiently precise as to enable results to be established using conventional mathematical techniques.

(logical) *Implication*
See *Entailment*.

Inference
An act or process of constructing new expressions from existing expressions, or result of such an act or process.
Inferences corresponding to entailments are described as *correct* or *valid* or *sound* (see *Entailment*).
An inference rule is a formal description of a type of inference.
An inference system is a set of inference rules.
An inference system may also be a sofware which generates inferences or check inferences for validity.

Information Agent
A computational software entity (i.e. an intelligent agent) that may access one or multiple, distributed, and heterogeneous information sources available, and pro-actively acquires, mediates, and maintains relevant information on behalf of its user(s) or other agents preferably just-in-time.

Information Request
A request for some generic information. Therefore, it is a special example of a service request. Optionally, it can be compiled from a given query.

Information Source
A source of data potentially wrapped and/or integrated in a complex system to provide information through data retrieval.

Intensional
Not extensional (see *Extensional*). A logic which allows distinct entities with the same extension.

Interpretation
A minimal formal description of those aspects of a world which is just sufficient to establish the truth or falsity of any expression of a logic (see *Logic*).

Logic
A formal language which expresses propositions (see *Proposition*).

Logical Axiom
An axiom that is a valid well formed formula of the language of a formal system (see *Axiom, Formal, and Well Formed Formula*).

Knowledge Base
A set of sentences in a formal language.

Matchmaker
A middle-agent that allows providers to advertise their capabilities (and useful service parameters) w.r.t. performable services, and requesters to send requests. It stores the capability advertisements that can then be queried by requesters, i.e. the capabilities exposed by the providers.

Middle-Agent
Middle-agents exist to enable interactions among end-agents (see *End-Agent*).

Model
An interpretation in which expressions of interest come out true for that interpretation (see *Interpretation*).

Model Theory
A formal semantic theory which relates expressions to interpretations (see *Semantic* and *Interpretation*).

Multi-Agents System
A loosely coupled network of problem solvers (e.g., information agents -see *Information Agent*) that work together to solve problems that are beyond the individual capabilities or knowledge of each problem solver.

Ontology
A specific vocabulary accompanied by relationships used to describe certain aspects of reality, and a set of explicit assumptions regarding the intended meaning of the specified vocabulary.

Predicate
Whatever is said of the subject of a sentence.

Predicate logic
the branch of logic dealing with propositions in which subject and predicate are separately signified, reasoning whose validity depends on this level of articulation, and systems containing such propositions and reasoning. Also called quantification theory (see *Logic*, *Predicate*, and *Proposition*).

Property
See *Role*.

Proposition
Something that has a truth-value; a statement or expression that is true or false.

Provider
An entity (e.g., an agent -see *Agent*) capable to provide the service asked by a service request, i.e. a service reply (see *Service Reply*). More specifically, it may provide the information asked by an information request (see *Information Request* and *Service Request*).

Query
A message exchanged between two entities, expressing the need for some information. Optionally, it can be instantiated from an information request. Usually, it is originated from a problem solving process.

Range

Reasoner, Reasoning, Reasoning System (Tool, Engine)

See *Inference*.

Requester
An entity (e.g., an agent -see *Agent*) issueing a service request. More specifically, it may issue an information request (see *Information Request* and *Service Request*).

Resource
An entity; anything in the universe (see *Universe*).
As a class name: the class of everything; the most inclusive category ever possible.

Role
Binary relationships between two individuals. It can be formalized as a binary predicate (see *Predicate*).

Satisfy, *Satisfiability*, *Satisfiable*
To make true.
The basic semantic relationship between an interpretation and an expression. X satisfies Y means that if the world conforms to the conditions described by X, then Y must be true (see *World*).
Able to be satisfied.

Semantic, *Semantics*
Concerned with the specification of meanings (i.e. denotations of expressions).

Service Request
A generic request for any kind service, from the execution of a composite task to simple retrieval of some specific data. Usually, a request is submitted by an agent, thus temporary assuming the connotation of a (service) requester.

Service Reply
The response to a service request. It is usually elaborated by an agent, thus temporary casting the role of a (service) provider.

Sound
Unable to draw any invalid inferences (see *Inference* and *Valid*).

Syntactic, Syntax
Concerned with the specification of expressions.

Terminology
The vocabulary of an application domain, or, more generally, of a domain of discourse.

Universe
The universal classification, or the set of things that an interpretation considers to exist.

Valid
Corresponding to an entailement (see *Entailment*).

Well Formed Formula
A string of symbols from the alphabet of the formal language that conforms to the grammar of the formal language (see *Formal*).

World
An interpretation (see *Interpretation*).

Note

Most of the definitions, expecially those concerning with the terminology used in ontology languages, have been taken from [RDF Semantics, 2003].
Any other reported meaning has been extracted from relevant papers as referenced in the body of this thesis.

Bibliography

1. [Ambite et al., 2001] Jose Luis Ambite, Craig A. Knoblock, Ion Muslea and Andrew Philpot. Compiling Source Descriptions for Efficient and Flexible Information Integration. Journal of Intelligent Information Systems, vol. 16, no. 2, pp. 149-187. 2001.

2. [Arens et al., 1993] Y. Arens, C. Y. Chee, C.-N. Hsu, and C. A. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. International Journal on Intelligent and Cooperative Information Systems, vol. 2, no. 2, pp. 127-158. 1993.

3. [Arens et al., 1996] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query Reformulation for Dynamic Information Integration. Journal of Intelligent Information Systems - Special Issue on Intelligent Information Integration. 1996

4. [Arens et al., 1996, bis] Yigal Arens and Chun-Nan Hsu and Craig A. Knoblock. Query Processing in the SIMS Information Mediator. Advanced Planning Technology, editor, Austin Tate, AAAI Press, Menlo Park, CA. 1996.

5. [Axis] Apache Axis, home page. http://ws.apache.org/axis/.

6. [Baader and Nipkow, 1998] Franz Baader and Tobias Nipkow. Term Rewriting and All That. Cambridge University Press. 1998.

7. [Baader et al., 2000] Franz Baader, Ralf Küsters, and Ralf Molitor. Rewriting Concepts Using Terminologies. Proceedings of the Seventh International Conference on Knowledge Representation and Reasoning (KR2000). 2000.

8. [Baader and Nutt, 2002] Franz Baader, Werner Nutt. Basic Description Logics. Description Logic Handbook, pp. 47-100. Cambridge University Press. 2002.

9. [Bayardo et al., 1997] R. J. Bayardo, Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, and R. Shea. InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 195-206. New York, NY. 1997.

10. [Benatallah et al., 2003] Boualem Benatallah, Mohand-Said Hacid, Christophe Rey and Farouk Toumani. Request Rewriting-Based Web Service Discovery. 2nd International Semantic Web Conference (ISWC2003). Sanibel Island, Florida, USA. October 20 - 24, 2003.

11. [Beneventano et al., 1996] Domenico Beneventano, Sonia Bergamaschi, and Claudio Sartori. Semantic Query optimization by subsumption in oodb. International Workshop on Flexible Query Answering Systems, Roskilde, Denmark. May, 1996.

12. [Beneventano et al., 1997] Domenico Beneventano, S. Bergamaschi, Claudio Sartori and Maurizio Vincini. ODB-Tools: a description logics based tool for schema

validation and semantic query optimization in Object Oriented Databases. Proceeding of the Fifth Conference of the Italian Association for Artificial Intelligence (AI*IA97): Advance in Artificial Intelligence, LNAI vol. 1321, Springer. Berlin. September 17-19, 1997.

13. [Beneventano et al., 1998] Domenico Beneventano, Sonia Bergamaschi, S. Lodi, and Claudio Sartori. Consistency Checking in Complex Object Database Schemata with Integrity Constraints. IEEE Transactions on Knowledge and Data Engineering 10(4), pp. 576-598. 1998.

14. [Bergamaschi et al., 2001] Sonia Bergamaschi, Silvana Castano, Domenico Beneventano, and Maurizio Vincini. Semantic Integration of Heterogeneous Information Sources. Special Issue on Intelligent Information Integration, Data & Knowledge Engineering, vol. 36, No. 1, Pages 215-249. Elsevier Science B.V. 2001.

15. [Berners-Lee et al., 2001] Tim Berners-Lee, James Hendler, Ora Lassila. The Semantic Web. Scientific American, vol. 284, no. 5, pp. 34-43. May 17, 2001.

16. [BPEL4WS Reference, 2003] Specification: Business Process Execution Language for Web Services Version 1.1. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/. May 5, 2003.

17. [Borgida et al., 1989] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnik. CLASSIC: A structural data model for objects. Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 59-67. 1989.

18. [Brachman et al., 1991] Ronald J. Brachman, Deborah L. McGuinness, Peter F. Patel-Schneider, and A. Borgida. Living with classic: When and how to use a kl-one-like language. J. F. Sowa, editor, Principles in Semantic Networks: Explorations in the Representation of Knowledge, pages 401--456. Morgan Kaufmann, San Mateo, California. 1991.

19. [Burnstein, 2003] The Many Faces of Mapping and Translation for Semantic Web Services. Fourth International Conference on Web Information Systems Engineering (WISE'03). Roma, Italy. December 10 - 12, 2003

20. [Castano et al., 2000] S. Castano, V. De Antonellis, S. De Capitani Di Vimercati. Global viewing of heterogeneous data sources. IEEE Transaction on Knowledge and Data Engineering. 2000.

21. [Chawathe et al., 1994] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. Proceedings of IPSJ Conference, pp. 7-18. Tokyo, Japan. October, 1994.

22. [DAML+OIL Reference, 2001]. Reference description of the DAML+OIL (March 2001) ontology markup language. http://www.daml.org/2001/03/reference. March, 2001.

23. [DAML-S, 2001] The DAML Services Coalition. DAML-S: Semantic Markup for Web

Services. Proceedings of the International Semantic Web Working Symposium (SWWS). July 30 - August 1, 2001.

24. [Davis and Smith, 1983] R. Davis, R. G. Smith. Negotiation as a Metaphor for Distributed Problem Solving. Artificial Intelligence, vol. 20, pp. 63-109. Elsevier Science Publishers. North. 1983.

25. [Decker et al., 1997] Keith Decker, Katia Sycara, Mike Williamson. Middle-Agents for the Internet. Proceedings of the 15th International Joint Conference on Artificial Intelligence. 1997

26. [Di Noia et al., 2003] Tommaso Di Noia, Eugenio Di Sciascio, Francesco M. Donini, and Marina Mongiello. A System for Principled Matchmaking in an Electronic Marketplace. Proceedings of the twelfth international conference on World Wide Web, pp. 321-330. Budapest, Hungary. 2003.

27. [DQL] DAML Query Language Development, hope page. http://ksl.stanford.edu/projects/dql/

28. [Duschka and Genesereth, 1997] O. M. Duschka and M. R. Genesereth. Answering Recursive Queries Using Views. Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 109-116. Tucson, Arizona. 1997.

29. [Friedman-Hill, 1995] Ernest Friedman-HIll. Jess: The rule engine for the Java platform. http://herzberg.ca.sandia.gov/jess/. 1995

30. [Fikes et al., 2002] Richard Fikes, Pat Hayes, and Ian Horroks. DQL - A Query Language for the Semantic Web. Knowledge Systems Laboratory, Stanford University, CA. 2002.

31. [Fikes et al., 2003] Richard Fikes, Jessika Jenkins, and Gleb Frank. JTP: A System Architecture and Component Library for Hybrid Reasoning. Proceedings of the Seventh Worl Multiconference on Systemics, Cybernetics, and Informatics. Orlando, Florida, USA. July 27 - 30, 2003

32. [Forgy, 1982] C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. Artificial Intelligence, vol. 19, vol. 1, pp. 17-37. 1982.

33. [Franklin and Graesser, 1996] Stan Franklin, Art Graesser. Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents. Proceeeding of the Third International Workshop on Agent Theories, Architectures, and Languages. Springer Verlag. 1996

34. [Garcia-Molina et al., 1997] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajararnan, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. Journal of Intelligent Information Systems, vol. 2. 1997.

35. [Gelati et al., 2001] G. Gelati, F. Guerra, and M. Vincini. Agents supporting information integration: the MIKS framework. Available at

http://dbgroup.unimo.it/Miks/publications/papers/woa2001.pdf

36. [Genesereth and Fikes, 1992] M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format Version 3 Reference Manual. Logic-92-1, Stanford University Logic Group, CA. 1992

37. ["Grounding" OWL ontology] http://www.daml.org/services/owl-s/1.0/Grounding.owl

38. [Guarino, 1998] Nicola Guarino. Formal Ontology and Information Systems. Proceedings of FOIS'98. Trento, Italy. June 6-8, 1998

39. [Haarlsev and Möller, 2003] Volker Haarslev and Ralf Möller. RACER User's Guide and Reference Manual Version 1.7.7. November 7, 2003.

40. [Hale, 1998]. Michael L. Mc Hale. A Comparison of WordNet and Roget's Taxonomy for Measuring Semantic Similarity. cmp-lg/9809003. September 14, 1998.

41. [Halevy, 2001] A. Y. Halevy. Answering queries using views: A survey. The VLDB Journal, 10, pp. 270-294. 2001

42. [Hammer et al, 1997] J. Hammer, M. Breunig, H. Garcia-Molina, S. Nestorov, V. Vassalos, and R. Yerneni. Template-Based Wrappers in the TSIMMIS System. Proceedings of ACM SIGMOD Conference, pp. 532-535. 1997

43. [Horroks, 1999] Ian Horroks. The FaCT system. http://www.cs.man.ac.uk/~horroks/FaCT/. 1999

44. [Horroks and Patel-Schneider, 1999] I. Horroks and P. F. Patel-Scheneider and Optimizing description logic subsumption. Journal and Computation, vol. 9, no. 3, pp. 267-293. 1999.

45. [Horrocks et al., 2000] Ian Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic SHIQ . Proceedings of the 17th International Conference on Automated Deduction (CADE-17), Lecture Notes in Computer Science. D. MacAllester editor. Springer Verlag. 2000.

46. [Horroks et al., 2003] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The Making of a Web Ontology Language. Journal of Web Semantics, vol. 1, no. 1. 2003

47. [Huhns et al., 1993] Michael N. Hunhs, Nigel Jacobs, Tomasz Ksiezyk, Wei-Min Shen, Munidar P. Singh, and Philip E. Cannata. Integrating Enterprise Information Models in Carnot. 1st International Conference on Intelligent and Cooperative Information Systems. Rotterdam. May, 1993.

48. [I^3 Reference, 1995] ARPA I^3 Reference Architecture. 1995 http://www.isse.gmu.edu/I3Arch/index.html.

49. [Java] Java, Sun Microsystems home page. http://java.sun.com/

50. [Jena] Jena Semantic Web Framework, home page. http://jena.sourceforge.net/index.html

51. [Jennings et al., 1998] Nicholas R. Jennings, Katia Sycara, Michael Wooldrige. A RoadMap of Agent Research and Development. Autonomous Agents and Multi-Agent Systems, 1, pp. 7-38. Kluwer Academic Publishers. Boston. 1998.

52. [Knoblock et al., 1998] C. A. Knoblock, S. Minton, J. L. Ambite, N. Ashish, P. J. Modi, I. Muslea, A. G. Philpot, and S. Tejada. Modeling Web Sources for Information Integration. Proceedings of the Fifteenth National Conference on Artificial Intelligence, pp. 211-218. Madison, WI. 1998.

53. [Koepena and Regli, 1992] Joe Koepena and William C. Regli. DAMLJessKB: A Tool for Reasoning with the Semantic Web. Available at http://edge.cs.drexel.edu/assemblies/software/damljesskb/articles/DAMLJessKB-2002.pdf. 1992.

54. [Lenat and Guha, 1990] Doug Lenat and R. V. Guha. Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project. Addison-Wesley Publishing Company Inc. Reading, MA. 1990.

55. [Levesque and Brachman, 1987] Hector. J. Levesque and Ronald. J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. Computational Intelligence journal, 3, pp. 78-93. 1987.

56. [Levy et al., 1995] Alon Y. Levy, Divesh Srivastava, and Thomas Kirk. Data model and query evaluation in global information systems. Journal of Intelligent Information Systems, 5, pp. 121-143. 1995.

57. [Levy et al., 1996] A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. Proceedings VLDB. 1996.

58. [MacGregor, 1990] Robert MacGregor. The evolving technology of classification-based knowledge representation systems. John Sowa (editor), Principles of Semantic Networks: Exploration in the Representation of Knowledge. Morgan Kaufmann. 1990.

59. [“mad cows” OWL ontology] http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/mad_cows.owl

60. [Mena et al., 1996] E. Mena, V. Kashyap, A. Sheth, A. Illarramendi. OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation across Pre-existing Ontologies. Conference on Cooperative Information Systems. 1996

61. [Miller et al., 1990] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Five Papers on WordNet. Special Issue of International Journal of Lexicography, vol. 3, no. 4. 1990.

62. [MySql] Microsoft MySql, home page. http://www.mysql.com/

63. [ODS, 1997] The Object Data Standard: ODMG 2.0. R. G. G. Cattel et al. editor. Morgan Kauffmann, San Mateo, CA. 1997

64. [OIL Technical Report] I. Horrock, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. van Harmelen, M. Klein, S. Staab, R. Studer, and E. Motta. The Ontology Inference Layer OIL. http://www.ontoknowledge.org/oil/TR/oil.long.html.

65. [Ouksel and Sheth, 1999] Aris M. Ouksel and Amit Sheth. Semantic Interoperability in Global Information Systems: A Brief Introduction to the Research Area and the Special Section. SIGMOD Record, vol. 28, no. 1, pp 5-12. March, 1999.

66. [OWL Abstract Syntax and Semantics, 2002] Web Ontology Language (OWL) Abstrract Syntax and Semantics, W3C Working Draft. http://www.w3.org/TR/2003/CR-owl-ref-20030818/. November 8, 2002.

67. [OWL Reference, 2003] OWL Web Ontology Language, Reference, W3C Candidate Recommendation. http://www.w3.org/TR/2003/CR-owl-ref-20030818/. August 18, 2003.

68. [Paolucci and Sycara, 2003] Massimo Paolucci and Katia Sycara. Autonomous Semantic Web Services. IEEE Internet Computing. September/Octorber, 2003.

69. [Paolucci et al., 2002] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic Matching of Web Services Capabilities. Proceedings of the first International Semantic Web Conference (ISWC), LNCS 2342, pp. 333-347. Springer Verlag. 2002

70. [Paolucci et al., 2003] M. Paolucci, A. Ankolekar, N. Srinivasan, and K. Sycara. The DAML-S Virtual Machine. To appear in proceedings of the Second International Semantic Web Conference (ISWC). 2003.

71. [Papakonstantinou et al., 1995] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information souces. Proceeding International Conference on Data Engineering. March, 1995.

72. [Papakonstantinou et al., 1996] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. Proceedings PDIS Conference, pp. 170-181. 1996

73. [Papakonstantinou et al., 1996, bis] MedMaker: A Mediation System Based on Declarative Specifications. Proceedings International Conference on Data Engineering. March, 1996.

74. [Patel-Schneider, 1998] P. F. Patel-Scheneider. DLP system description. Proceedings of DL'98, pp. 87-89. 1998

75. [Patel-Schneider and Swartout, 1993] P.F. Patel-Schneider, B. Swartout. Description Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knowledge Sharing Effort. November, 1993.

76. ["Process" OWL ontology] http://www.daml.org/services/owl-s/1.0/Process.owl

77. ["Profile" OWL ontology] http://www.daml.org/services/owl-s/1.0/Profile.owl

78. [Quillian, 1968] M. Ross Quillian. Semantic memory. M. Minsky editor, Semantic Information Processing. MIT Press, Cambridge, MA. 1968.

79. [Rada et al., 1989] Roy Rada, Hafedh Mili, Ellen Bicknell, and maria Blettner. Development adn application of a metric on semantic nets. IEEE Transaction on Systems, Man, Cybernetics, 19(1), pp. 17-30. February, 1989.

80. [RDF Reference, 2003]. Resource Description Framework (RDF), Concepts and AbstractSyntax, W3C Proposed Recommendation. http://www.w3.org/TR/2003/PR-rdf-concepts-20031215/. December 15, 2003.

81. [RDF Semantics, 2003] RDF Semantics, W3C Working Draft. http://www.w3.org/TR/2003/WD-rdf-mt-20030123/. January 23, 2003.

82. [RDFS Reference, 2003] RDF Vocabulary Description Language 1.0: RDF Schema. W3C proposed Recommendation. http://www.w3.org/TR/2003/PR-rdf-schema-20031215/. December 15, 2003.

83. [Resnik, 1995] Philip Resnik. Using Information Content to Evaluate Semantic Similarity in a Taxonomy. Proceedings of the 14th International Joint Conference on Artificial Intelligence. vol. 1, pp. 448-453. Montreal, Canada. August, 1995.

84. ["Service" OWL ontology] http://www.daml.org/services/owl-s/1.0/Service.owl

85. [SEWASIE] SEWASIE home page. http://www.sewasie.org/

86. [SEWASIE D2.1] Specification of the general framework for the multilingual semantic enrichment processes and of the semantically enriched data stores. SEWASIE Deliverable D2.1. February 23, 2003.

87. [SEWASIE D3.2] Definition of the techniques for query reformulation and information reconciliation. SEWASIE Deliverable D3.2. June, 2003.

88. [SOAP Reference 1, 2003] SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation. June 24, 2003.

89. [SOAP Reference 2, 2003] SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation. http://www.w3.org/TR/2003/REC-soap12-part2-20030624/. June 24, 2003.

90. [SIG] Special Interest Group on Intelligent Information Agents, home page. http://www.dbgroup.unimo.it/IIA/index.html

91. [Stickel, 1985] MArk Stickel. Automated deduction by theory resolution. IJCAI-85, pp. 455-458. 1985.

92. [“storage” ODL_{I3} schema] http://www.dbgroup.unimo.it/prototipo/slide/sc_odl1.html.

93. [Sterling and Shapiro, 1986] L. Sterling and E. Shapiro. The Art of Prolog: Advanced Programming Techniques. Cambridge, Mass.: MIT Press. 1986.

94. [Stevens and Pooley, 2000] P. Stevens, and R. Pooley. Using UML: Software Engineering with Objects and Components Addison Wesley. 2000.

95. [UDDI Reference, 2003] UDDI Version 3.0.1, UDDI Spec Technical Committee Specification. http://uddi.org/pubs/uddi-v3.0.1-20031014.htm. October 14, 2003.

96. [Sycara et al., 2001] K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. The RETSINA MAS Infrastructure. Technical Report CMU-RI-TR-01-05, Robotics Institute Technical Report, Carnegie Mellon. 2001.

97. [Vassalos and Papakonstantinou, 1997] Vasilis Vassalos and Yannis Papakonstantinou. Describing and Using Query Capabilities of Heterogeneous Sources (Extended Version). VLDB Conference, 1997.

98. [Vassalos and Papakonstantinou, 2000] V. Vassalos and Y. Papakonstantinou. Expressive Capabilities Description Languages and Query Rewriting Algorithms. Journal of Logic Programming, vol. 43, no. 1, pp. 75-122. 2000.

99. [Vassalos, 1996] Wrapper specification and query processing in the TSIMMIS project. Unpublished memorandum.

100. [Vianu, 1997] Victor Vianu. Rule-Based Languages. Annuals of Mathematics and Artificial Intelligence, vol. 19, no. 1-2, pp. 215-259. 1997.

101. [Web Services Architecture, 2003] Web Services Architecture, W3C Working Draft. http://www.w3.org/TR/2003/WD-ws-arch-20030808/. August 8, 2003.

102. [“wines” DAML+OIL ontology] http://ontologia.stanford.edu/doc/chimaera/ontologies/wines.daml

103. [Winston, 1992] P. H. Winston. Artificial Intelligence (3rd Edition. Addison Wesley. 1992.

104. [Woelk et al., 1992] Darrell Woelk, Wei-Min Shen, Michael N. Huhns, and Philip E. Cannata. Model-Driven Enterprise Information Management in Carnot. Enterprise Integration Modeling: Proceedings of the First International Conference, Charles J. Petrie Jr editor. MIT Press, Cambridge, MA. 1992.

105. [Wong and Sycara, 2000] H. Chi Wong and Katia Sycara. A Taxonomy of MIddle-Agents for the Internet. Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS2000). 2000

106. [WSCI Reference, 2003] Web Services Choreography Requirements 1.0. W3C Working Draft. http://www.w3.org/TR/2003/WD-ws-chor-reqs-20030812/. August 12, 2003.

107. [WSDL Reference 1, 2003] Web Service Description Language (WSDL) Version 2.0, Part 1: Core Language. W3C Working Draft.

http://www.w3.org/TR/2003/WD-wsdl20-20031110/. November 10, 2003.

108. [WSDL Reference 2, 2003] Web Service Description Language (WSDL) Version 2.0, Part 2: Message Pattern. W3C Working Draft. http://www.w3.org/TR/2003/WD-wsdl20-patterns-20031110/. November 10, 2003.

109. [XML Reference, 2000] Extensible Markup Language (XML) 1.0 (Second Edition), W3C Reccommendation. http://www.w3.org/TR/2000/REC-xml-20001006. October 6, 2000.

110. [XMLS Reference 1, 2001] XML Schema Part 1: Structures. W3C Recommendation. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/. May 2, 2001.

111. [XMLS Reference 2, 2001] XML Schema Part 2: Datatypes. W3C Recommendation. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/. May 2, 2001.

112. [Zaniolo, 1991] C. Zaniolo. The Logical Data Language (LDL): An Integrated Approach to Logic and Databases. MCC Technical Report STP-LD-328-91. 1991.