Intelligent Information Integration: 
The MOMIS Project

Integrazione Intelligente Di 
Informazioni:  
Il Progetto MOMIS

Tesi di:  
Dott. Ing. Alberto Corni

Coordinatore:  
Chiar.mo Prof. Ing. Fabio Filicori

Anno 1999 - 2000

Relatori:  
Chiar.mo Prof. Claudio Sartori  
Chiar.mo Prof. Sonia Bergamaschi
Key words:
Intelligent integration
MOMIS
Mediator
Lexicon derived knowledge
Affinity calculus
Foreword

This document describes the work done during my Ph.D studies in Computer Engineering. It is organized in two parts.

The first and main part describes the research project MOMIS for the Intelligent Integration of heterogeneous information. It outlines the theory for Intelligent Integration and the design and implementation of the prototype that implements the theoretical techniques.

During my Ph.D. studies I stayed at the Northeastern University in Boston, Mass. (USA). Subject of the second part of this document is the work I did with Professor Ken Baclawski in information retrieval on annotation of documents using ontologies, and retrieval of the annotated documents.
## Contents

1 Part I - Intelligent Integration 3

2 MOMIS: The Theory 7

2.1 Information extraction with ODL 18

2.1.1 The running example 8

2.1.2 Managing semistructured data 9

2.1.3 The ODL 13 language 10

2.1.4 The OLCD Description Logic 13

   Types and Schemas 13

   ODL 13 to OLCD translation 14

2.1.5 The WordNet lexical database 15

   Semantic relationships between schema terms 16

   Formalization of the WordNet database 16

   Extracting thesaurus relationships from the WordNet database 17

   Use case 17

2.2 Building the Common Thesaurus 18

2.2.1 Schema-derived relationships 18

2.2.2 Lexicon-derived relationships 19

2.2.3 Designer-supplied inter-schema relationships 20

2.2.4 Relationships validation 20

2.2.5 Checking consistency and inferring new relationships 21

2.2.6 Virtual schema creation 21

2.3 Semantic information integration 24

2.3.1 Affinity of ODL 13 classes 24

   Name Affinity coefficient 24

   Structural Affinity coefficient 25

   Global Affinity coefficient 26

   Clustering of ODL 13 classes 27

2.3.2 Synthesis into an integrated schema description 28

2.4 Base Extension and Extensional Hierarchy 33

2.5 The Object Fusion problem: the Join Maps 35
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6 Related works</td>
<td>36</td>
</tr>
<tr>
<td>Heterogeneous information integration</td>
<td>36</td>
</tr>
<tr>
<td>Semistructured data</td>
<td>37</td>
</tr>
<tr>
<td>Original contributions of MOMIS w.r.t. previous works</td>
<td>38</td>
</tr>
<tr>
<td>2.7 OLCD: Interpretations and database instances</td>
<td>38</td>
</tr>
</tbody>
</table>

## 3 The MOMIS Prototype: Architecture and Implementation 43

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>43</td>
</tr>
<tr>
<td>3.2 Overview of organisational infrastructures</td>
<td>44</td>
</tr>
<tr>
<td>3.3 Overview of how software is organised</td>
<td>45</td>
</tr>
<tr>
<td>3.4 The shared component directory</td>
<td>49</td>
</tr>
<tr>
<td>3.4.1 The ODL classes and the Parser</td>
<td>49</td>
</tr>
<tr>
<td>Introduction of MomisObject</td>
<td>51</td>
</tr>
<tr>
<td>Introduction of TypeContainer</td>
<td>51</td>
</tr>
<tr>
<td>Types name post parsing resolution</td>
<td>51</td>
</tr>
<tr>
<td>toOdl implementation</td>
<td>52</td>
</tr>
<tr>
<td>toOlcd implementation</td>
<td>52</td>
</tr>
<tr>
<td>toOlcdSimB implementation</td>
<td>53</td>
</tr>
<tr>
<td>Introduction of ThesaurusByObject</td>
<td>54</td>
</tr>
<tr>
<td>3.5 The MOMIS wrappers</td>
<td>54</td>
</tr>
<tr>
<td>3.5.1 The IDL interface for a wrapper</td>
<td>55</td>
</tr>
<tr>
<td>3.5.2 The dummy wrapper</td>
<td>56</td>
</tr>
<tr>
<td>3.5.3 The JDBC wrapper</td>
<td>56</td>
</tr>
<tr>
<td>3.5.4 Other wrappers</td>
<td>56</td>
</tr>
<tr>
<td>3.6 The MOMIS WordNet interface</td>
<td>57</td>
</tr>
<tr>
<td>3.7 The MOMIS ODB-Tools interface</td>
<td>59</td>
</tr>
<tr>
<td>3.8 The MOMIS Global Schema</td>
<td>61</td>
</tr>
</tbody>
</table>

## 4 The MOMIS SI-Designer 65

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 The SI-Designer architecture</td>
<td>65</td>
</tr>
<tr>
<td>4.2 An integration session</td>
<td>66</td>
</tr>
<tr>
<td>4.3 Methodological considerations</td>
<td>68</td>
</tr>
<tr>
<td>4.4 Implementation</td>
<td>71</td>
</tr>
<tr>
<td>4.4.1 The SIDPhase interface</td>
<td>71</td>
</tr>
<tr>
<td>4.4.2 Saving and exchanging knowledge in MOMIS</td>
<td>72</td>
</tr>
<tr>
<td>4.4.3 The Common Thesaurus Editor module</td>
<td>74</td>
</tr>
<tr>
<td>4.4.4 The ARTEMIS module</td>
<td>74</td>
</tr>
<tr>
<td>4.4.5 The test module</td>
<td>80</td>
</tr>
<tr>
<td>4.5 Implementation and Experimentation considerations</td>
<td>81</td>
</tr>
</tbody>
</table>

## 5 Conclusions 83

## 6 Part II - Information Retrieval with the Keynet system 85

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 The Keynet system</td>
<td>85</td>
</tr>
</tbody>
</table>
Chapter 1

Part I - Intelligent Integration

Developing intelligent tools for the integration of information extracted from multiple heterogeneous sources is a challenging issue to effectively exploit the numerous sources available on-line, for example, in global, Internet-based information systems. The problem we consider is the identification of semantically related information, that is, information describing the same real-world concepts in different sources having semantic heterogeneity. In fact, information sources to be integrated are usually pre-existing and have been developed independently. Consequently, semantic heterogeneity can arise for the aspects related to terminology, structure, and context of the information, and has to be properly dealt with during integration in order to effectively and correctly exploit the information available at the sources. Integration and reconciliation of data coming from heterogeneous sources is a research topic in databases [Hul97]. Several contributions have appeared in literature, including methods, techniques and tools for integrating and querying heterogeneous databases [CHS+94, GKD97, LRO96, PGMW95].

This document describes solutions to the integration issue developed in the MOMIS (Mediator envirOnment for Multiple Information Sources) [BCV99, BCVB00, BBC+00] project\(^1\).

The goal of information extraction and integration techniques developed in MOMIS is to construct synthesized, integrated descriptions (i.e., a global virtual view) of the information coming from multiple heterogeneous sources, to provide the user with a uniform query interface against the sources independent from their location and the level of heterogeneity of their data. Moreover, to meet the requirements of global, Internet-based information systems with a possibly high number of sources to be integrated, it is important to develop tool-assisted techniques in order to automate as much as possible information extraction and integration activities. This goal has been achieved with the MOMIS project by developing the SI-Designer [BBC+00].

Like other integration projects [AKH96, CHS+94, LRO96], MOMIS follows a “semantic approach” to information integration based on the conceptual schemas of the information sources, and on a mediator component. MOMIS implements a \(I^3\) [Age\(^2\)]

---

\(^1\)MOMIS is a joint project among the Università di Modena e Reggio Emilia, the Università di Milano, and the Università di Brescia started within the Italian research project INTERDATA, theme n.3 “Integration of Information over the Web”, coordinated by V. De Antonellis, Università di Brescia.

\(^2\)MOMIS will also financed by the MURST in 2000/2001 within the D2I: Integration, Warehousing, and Mining of Heterogeneous Data Sources project
architecture for integration and query optimization, the mediator component relies on several tools, namely ARTEMIS [CA99b], developed by University of Milano and University of Brescia, and ODB-Tools [BBSV97a], developed by University of Modena e Reggio Emilia and WordNet [GGV96] developed by the Cognitive Science Laboratory at Princeton University.

Most of the information for integration is provided by the wrappers, using ODL from the source description and stored in ODL data structure. By exploiting the WordNet lexical system [Mil95] and the OLCD Description Logic inference capabilities a Common Thesaurus is built. The Common Thesaurus is composed of relationships between schema elements extracted from the schemata descriptions, derived from the lexical relationships between the concepts in WordNet, and explicitly stated by the integration designer. Based on the relationships in the Common Thesaurus, affinity coefficients are computed. These coefficients give a measure of the level of matching between the elements of the different sources. MOMIS computes a set of candidates Global Classes by applying affinity-based clustering on the affinity coefficients. For each Global Class Mapping rules are defined that describe which local classes belong to the global class and how local attributes are mapped into the global attributes. Moreover, extensional relationships and join maps are defined to provide a solution to the object fusion problem. Such information are used by the Query Manager during the merge process of the local queries result.

The MOMIS project also faces the problems and challenges related to the integration of semistructured data sources. In this case, there is no strong distinction between meta-information and data, but meta-information is stored directly in the data. The significant growth of semi-structured data sources (e.g., Web documents) calls for the development of methods, techniques, and languages for this type of data [Bun97, BDHS96, CGL98]. Thus, the typical problems of integration should be addressed in the light of these new requirements.

Part of the project is the development of a prototype which is the software application based on the MOMIS theory. The prototype is written in Java, is based on the CORBA architecture and supports most of the MOMIS features like the Description Logics capabilities, the ARTEMIS affinity computation and clustering and the interaction with WordNet.

### Base terminology

In this document we argue about concepts like sources, classes, interface, which are briefly introduced, here.

A Local Source is a source of data, such as a database or a text file. Each local source is interfaced to MOMIS through a wrapper. A Local Class is an interface (or class) present in a local source. A Local Attribute is an attribute of a local class. The Integration Designer is a person that faces the integration problem using the MOMIS system. The Global View is a source schema presented by the mediator after the integration process that allows to access in a uniform way to the data in the local sources. A Global Class is an interface (or class) that is part of the Global View. A Global Attribute is an attribute of a global class.

---

3Since we integrate one schema at time during an integration session we will reference a single global views. Obviously it is possible to have several global view on the same or different local classes generated by several integration sessions built with different targets.
How this part is organized

This part of the document is organized as follows.
Chapter 2 describes the theory on which the MOMIS project is based. The integration approach is discussed from a theoretical point of view.
Chapter 3 discusses the software architecture of the MOMIS prototype. All its modules (except the SI-Designer) are outlined.
Chapter 4 gives a discussion of the main Graphical User Interface of the MOMIS prototype called SI-Designer. Such interface leads the integration designer through the integration phases from local sources to the integrated schema.
Chapter 5, we give our concluding remarks and an overview of future work about the MOMIS project.

Note

This document should also be a reference for the software I wrote. The description of some (I assume important) annoying technical details is targeted to people that will manage the software written during my studies.
Chapter 2

MOMIS: The Theory

This chapter discusses theoretical aspects of the MOMIS project. MOMIS is an Heterogeneous Source Integration project. The goal of the project is to study the problems related to integration and build a prototype that acts as a \( I^3 \) Semantic Integration and Transformation Service (see [Age, HK95] for a description of the \( I^3 \) architecture).

In MOMIS, theoretical solutions for several problems of integration are proposed. Most of the solutions discussed in this chapter have been implemented in the MOMIS prototype; others, like the semistructured meta-information extraction, have not.

![Diagram](image)

Figure 2.1: Integration phases in MOMIS

In MOMIS we divide the integration in the phases (showed in Figure 2.1): (1) description of a generic data source in ODL\(_{13}\), (2) generation of the common thesaurus, repository of integration knowledge, (3) computation of affinity between schema elements (local classes and local attributes) and generation of global classes as clusters of local classes, and (4) description of the local classes in terms of mapping of the global attributes on the corresponding local attributes.

This chapter is organized as follows:

In section 2.1, we describe the tools we use to automatically retrieve as much semantics as possible during the integration process. In particular we introduce the ODL\(_{13}\)
language, the OLCD Description Logic, and WordNet. In section 2.2, we describe how such tools are used to feed the integration knowledge base called Common Thesaurus. In section 2.3, we describe the integration techniques for building the mediator virtual global schema of considered sources using the knowledge in the Common Thesaurus. Sections 2.4 and 2.5 respectively present, techniques for optimizing query execution based on Base Extension and Extensional Hierarchy, and the way we faced the Object Fusion problem. Section 2.6, we make comparisons with related works. Finally, in section 2.7, the Interpretation and semantic of the OLCD language is briefly described.

2.1 Information extraction with ODL

An important goal of information extraction is the construction of a semantically rich representation of the information sources to be integrated by means of a common data model. In semantic approaches to integration, this task is performed by defining a model, ODL
3 in MOMIS and by wrapper tools developed for each kind of data source type, which translate the conceptual schema of the given sources into the common model. For conventional structured information sources (e.g., relational databases, object-oriented databases), schema descriptions are always available and can be directly translated into the selected common data model.

For example, for relational databases transformation rule-sets, as described in [FV95] for relational to ODMG schema conversion can be used. To show wrappers functionalities, let us introduce a running example, including a relational source and a semistructured source.

2.1.1 The running example

```
Restaurant(r_code, name, street, zip_code, pers_id,
special_dish, category, tourist_menu_price)
Bistro(r_code, type, pers_id)
Person(pers_id, first_name, last_name, qualification)
Brasserie(b_code, name, address)
```

Figure 2.2: Food Guide Database (FD)

We consider two sources in the Restaurant Guide domain, storing information about restaurants. The Eating Source guidebook (ED) is semistructured and contains information about fast foods of the West coast, their menus, quality, and so on. A portion of this source is shown in Figure 2.3. The Food Guide Database (FD) is a relational database containing information about USA restaurants from a wide variety of publications (e.g., newspaper reviews, regional guidebooks). The schema of this source is composed of four relations (see figure 2.2), namely, Restaurant, Bistro, Person, and Brasserie. Information related to restaurants is maintained in the Restaurant relation. Bistro instances are a subset of Restaurant instances and give information about the small informal restaurants that serve wine. Each Restaurant and Bistro is managed by a Person. Information about places where drinks and snacks are served are stored in the Brasserie relation.
2.1 Information extraction with ODL

2.1.2 Managing semistructured data

In this subsection we discuss how semi-structured data (e.g., Web data sources) can be handled in the MOMIS system where schema descriptions generally are not directly available. In fact, a basic characteristic of semistructured data is that they are “self-describing”. This means that the information generally associated with the schema is specified directly within data [Bun97].

One of the goals of information extraction for integration when semistructured information sources are involved is to derive and explicitly represent the schema of the source as well. For this purpose, we proceed as follows. According to the models proposed in literature for semistructured information sources [Bun97, PGMW95], a semistructured source is represented as a rooted, labeled graph where nodes contain data (e.g., an image or free-form text) and labeled edges describe the concept represented by data in the corresponding node.

Figure 2.3 shows an example of a graph-based representation of a semistructured source, called Eating Source, containing information related to local fast food. In the graph model, a semistructured object (object, for short) can be viewed as a triple of the form \( (id, label, value) \), where \( id \) is the object identifier, \( label \) is a string describing what the object represents, and \( value \) is the value, which can be atomic or complex. The atomic value can be integer, real, string, image, while the complex value is a set of pairs \((id, label)\), where \( id \) is an object identifier.

A complex object can be thought as the parent of all the objects that form its value (children objects). A given object can have one or more parents. We denote the fact that an object \( so \) is a child object of another object \( so \) by \( so \rightarrow so' \) and use notation \( label(so) \) to denote the label of \( so \). With reference to the source in Figure 2.3, there is one complex root object with four complex children objects that represent fast-foods. Each Fast_Food object has an atomic object name, category and specialty. Furthermore, some Fast_Food objects have an atomic address while some others a complex object address, an atomic phone, a complex object nearby (that specifies the nearest fast-food), and a complex object owner, specifying the name, the address and the job of the fast-food’s owner.

To extract schema information from a semistructured source \( S \), we introduce the notion of object pattern. In semistructured data models, labels are descriptive as much as possible. Furthermore, the same label is generally assigned to all objects describing the same concept in \( S \). All objects \( so \) of \( S \) are partitioned into disjoint sets, denoted \( set_l \), such that all objects belonging to the same set have the same label \( l \). An object
pattern is then extracted from each set to represent all the objects in the set. Formally, an object pattern is defined as follows:

**Definition 2.1.1 (Object pattern)** Let \( S \) be a set of objects in a semistructured source \( S \) having the same label \( l \). The object pattern of \( S \) is a pair of the form \( \langle l, A \rangle \), where \( l \) is the label of the objects belonging to \( S \), and \( A = \bigcup \text{label}(s \in S) \) such that there exists at least one object \( s \in S \) with \( s \rightarrow s' \).

From this definition, an object pattern is representative of all different objects that describe the same concept in \( S \). In particular, label \( l \) of an object pattern denotes the concept and set \( A \) of an object pattern denotes the properties (or attributes) characterising the concept in the source. Since semistructured objects can be heterogeneous, labels in \( A \) correspond to child objects that can be defined only for some of the objects in \( S \), but not for all. We call such kind of labels “optional” and denote them with symbol “?”.

\[
\text{Fast_Food-pattern} = \langle \text{Fast_Food}, \{\text{name, address, midprice? phone?, specialty, category, nearby?, owner?} \rangle \\
\text{Owner-pattern} = \langle \text{Owner, \{name, address, job\}} \rangle \\
\text{Address-pattern} = \langle \text{Address, \{street, city, zipcode\}} \rangle
\]

Figure 2.4: Object patterns for the ED source

With respect to the ED source of Figure 2.3, three object patterns are extracted (see Figure 2.4): Fast_Food, representing objects describing eating places; Owner representing objects describing people involved; Address, representing objects describing addresses. The extraction process produces also the Address pattern to take into account the different structure of the Address objects in the ED source (i.e., in semistructured objects 2, 4, and 5 address is atomic while in object 3 it is complex). An object pattern description follows an open world semantics typical of the Description Logic approach [WS89]. Objects conforming to a pattern share a common minimal structure represented by non-optional properties, but can have additional (i.e., optional) properties. In this way, objects in a semistructured data source can evolve and add new properties, but they will be retrieved as valid instances of the corresponding object pattern when processing a query.

### 2.1.3 The ODL* language

For a semantically rich representation of source schemas and object patterns associated with information sources to be integrated, we introduce an object-oriented description language, called ODL*. According to recommendations of ODMG and to the diffusion of ODL* [Age, ed.97], the object data model ODL* is very close to the ODL language. ODL* is a source independent language used for information extraction to describe heterogeneous schemas of structured and semistructured data sources in a common way. Refer to appendix A for the complete syntax of the ODL*.

ODL* introduces the following main extensions with respect to ODL:
2.1 Information extraction with ODL

Union constructor. The union constructor, denoted by `union`, is introduced to express alternative data structures in the definition of an ODL class, thus capturing requirements of semistructured data. An example of its use will be shown in the following.

Optional constructor. The optional constructor, denoted by the question mark `?`, is introduced for class attributes to specify that an attribute is optional for an instance (i.e., it could be not specified in the instance). This constructor too has been introduced to capture requirements of semistructured data. An example of its use will be shown in the following.

Integrity constraint rules. This kind of rule is introduced in ODL in order to express, in a declarative way, if then integrity constraint rules at both intra and inter-source level.

Intensional relationships. These are terminological relationships expressing inter/intra-schema knowledge for the source schemas. Intensional relationships are defined between classes and attributes, and are specified by considering class/attribute names, called terms. The following relationships can be specified in ODL:

- `SYN` (Synonym-of), defined between two terms \( t_1 \) and \( t_2 \), with \( t_1 \neq t_2 \), that are considered synonyms in every considered source (i.e., \( t_1 \) and \( t_2 \) can be indifferently used in every source to denote a certain concept).
- `BT` (Broader Terms), or hypernymy, defined between two terms \( t_1 \) and \( t_2 \) such as \( t_1 \) has a broader, more general meaning than \( t_2 \). `BT` relationship is not symmetric. The opposite of `BT` is `NT` (Narrower Terms), or hyponymy.
- `RT` (Related Terms), or positive association, defined between two terms \( t_1 \) and \( t_2 \) that are generally used together in the same context in the considered sources.

An intensional relationships is only a terminological relationship, with no implications on the extension/compatibility of the structure (domain) of the two involved classes (attributes).

Extensional relationships. To express knowledge about the source extensions the ODL provides the concept of extensional relationships:

- `C_1 \ SYN_{ext} C_2` : this means that the instances of \( C_1 \) are the same of \( C_2 \). A `SYN_{ext}` relationship implies an intensional `SYN` relationship.
- `C_1 \ BT_{ext} C_2` : this means that the instances of \( C_1 \) are a superset of the instances of \( C_2 \). A `BT_{ext}` relationship implies an intensional `BT` relationship.
- `C_1 \ NT_{ext} C_2` : this means that the instances of \( C_1 \) are a subset of the instances of \( C_2 \). A `NT_{ext}` relationship implies an intensional `NT` relationship.
- `C_1 \ DISI_{ext} C_2` : this means that \( C_1 \) and \( C_2 \) will never share instances - the instances are disjoint.

Note: defining extensional axioms `syn_{ext}`, `nt_{ext}` and `bt_{ext}` requires a structural compatibility between the involved classes in MOMIS.

Foreign keys. To preserve information between classes (tables) proper of the relational schema we support also Foreign keys descriptions. We use foreign keys information to feed the common thesaurus and enrich semantic knowledge of the sources.
Mapping Rules. This kind of rule is introduced in ODL₁₃ in order to express relationships existing between the integrated ODL₁₃ schema description of the information sources and the ODL₁₃ schema description of the original sources. These rules will be illustrated in detail in Section 2.3, together with examples of use.

The extraction process has the goal of translating object patterns and source schemas into ODL₁₃ descriptions. Translation is performed by a wrapper. Moreover, a wrapper can be asked for the source name and type (e.g., relational, semistructured).

The translation into ODL₁₃, on the basis of the ODL₁₃ syntax and of the schema definition, is performed by the wrapper as follows:

Given a relation of a relational source or a class or a pattern \((l, A)\), translation involves the following steps: i) an ODL₁₃ class name corresponds to the relation name or to \(l\), respectively, and ii) for each relation attribute or label \(l’ \in A\), an attribute is defined in the corresponding ODL₁₃ class. Furthermore, attribute domains are extracted. Structure extraction can be performed as proposed in [aSBDFS97, NAM98].

In figures 2.5 and 2.6 we report the ODL₁₃ representation of the ED.Fast_Food object pattern and of the FD.Restaurant relation. The complete ODL₁₃ schemas representation of the ED and FD sources are shown in figures 2.17 and 2.18.

interface Fast_Food
    ( source semistructured ED ) {  
        attribute string name;
        attribute Address address;
        attribute integer phone?;
        attribute set<string> specialty;
        attribute string category;
        attribute Fast_Food nearby?;
        attribute integer midprice?;
        attribute Owner owner?;
    };

Figure 2.5: ODL₁₃ representation of a Semistructured Class from object patterns from the ED source

interface Restaurant ( source relational FD
    key r_code
    foreign_key(pers_id) references Person ) {  
        attribute string r_code;
        attribute string name;
        attribute string street;
        attribute string zip_code;
        attribute integer pers_id;
        attribute string special_dish;
        attribute integer category;
        attribute integer tourist_menu_price;  
    };

Figure 2.6: ODL₁₃ representation of a Relational table from the FD source

To represent object patterns in ODL₁₃, union and optional constructors are used. In particular, the union constructor is used to represent object patterns describing het-
2.1 Information extraction with ODL

erogeneous objects in the source. An example of use of the union constructor in the ODL class representing the Address pattern of the ED source (see Figure 2.4) is shown in Figure 2.7. The semantics of the union constructor and of optional attributes in ODL will be discussed in the section 2.1.4, using the OLCD Description Logics.

interface Address
  { source semistructured Eating_Source } {
    attribute string city;
    attribute string street;
    attribute string zipcode;
  }
union
{ string; }

Figure 2.7: An example of union constructor in ODL

2.1.4 The OLCD Description Logic

ODL descriptions are translated into OLCD (Object Language with Complements allowing Descriptive cycles) descriptions in order to perform Description Logics inferences that will be useful for semantic integration.

In this section, we give the syntax of OLCD (the semantics is given in section 2.7); Readers interested in a formal account can refer to [BBLS98].

Types and Schemas  We assume a countable set of symbols A of attribute names (denoted by a, a1, a2,...) and we assume a countable set N of type names (denoted by N, N1, N2,...), which includes the set B = {Integer, String, Bool, Real} of base-type designators (which will be denoted by B) and the symbols ⊤, ⊥. A path p is either the symbol ε, or a dot-separated sequence of elements ei, ei2,..., en, where eι ∈ A ∪ {Δ, ∃} (i = 1, ..., n), ε denotes the unique path of length 0. Let W denote the set of all paths. S(A, N) denotes the set of all finite type descriptions (denoted by S, S1, S2,...), also briefly called types, over given A, N, obtained according to the following abstract syntax rule, where aι ∉ a j for i ≠ j (in the sequel p1, p1, p2, ..., denote a path, d denotes a base value, θ denotes a relational operator): S → N | S1 ⊔ S2 | S; ∩ S2 | ¬S | {S}v | {S}3 | [a1:S1,...,ak:Sk] | ΔS | pθd | p↓

⊤ denotes the top type, ⊥ denotes the empty type, { }v and [] denote the usual type constructors of set and record (tuple), respectively. The {S}3 construct is an existential set specification, where at least one element of the set must be of type S. The construct R stands for intersection, the construct ⊔ stands for union, the construct ¬ stands for complement, whereas Δ constructs class descriptions, i.e., is an object set forming constructor. pθd, p↓ represent atomic predicates: pθd is a range restriction and p↓ expresses path undefinedness.

Given a set of type descriptions S(A, N), a schema σ over S(A, N) is a total function σ: N \ (B ∪ {⊤, ⊥}) → S(A, N), which associates type names to descriptions. σ is partitioned into two functions: σR, which introduces the description of primitive type names whose extensions must be explicitly provided by the user; and σV, which introduces the description of virtual type names whose extensions can be recursively obtained from the extension of the types occurring in their description.
In OLCD cyclic type names are allowed: in fact, since a type name may appear in type descriptions, we can have *circular references*, that is, type names which make direct or indirect references to themselves.

Giving a *type as set* semantics to type descriptions, Description Logics, and thus OLCD, allows one to provide relevant reasoning techniques: computing *subsumption* relations between types (i.e., “is-a” relationships implied by type descriptions), deciding *equivalence* between types, and detecting *inconsistent* (i.e., always empty) types.

**ODLpb to OLCD translation**

In this section, we describe how ODLpb source schema descriptions are translated into OLCD descriptions.

**ODLpb classes.** In general, a ODLpb class is translated into a OLCD primitive class in a simple way: each attribute of the ODLpb class becomes an attribute of the corresponding OLCD class. For example, the Restaurant ODLpb class is translated as follows:

\[
\sigma_P(ES.Restaurant) = \Delta_{r\_code: String, name: String, street: String, zip\_code: String, pers\_id: Integer, special\_dish: String, category: Integer, tourist\_menu\_price : Integer}
\]

Some aspects of an ODLpb class declaration, such as key `r\_code` in the Restaurant ODLpb class, are not translated into OLCD, but will be used in the semantic information integration.

**Union constructor.** The union constructor of ODLpb is translated using the construct `∪` of OLCD; for example, the Address pattern of figure 2.7 is translated in OLCD as follows:

\[
\sigma_P(ES.Address) = \Delta \left( String \cup [city: String, street: String, zipcode: String] \right)
\]

**Optional constructor.** The construct `∪` is also used to translate *optional* attributes into OLCD. In fact, an optional attribute `att` specifies that a value may exist or not for a given instance. This fact is expressed in OLCD as the union between the attribute specification (with its domain) and attribute *undefinedness*, denoted by `∪` operator: `([att\_domain]∪att\_if)`. For example, in the Fast_Food interface, the optional attributes are translated as follows:

\[
\sigma_P(ES.Fast\_Food) = \Delta \left( [name: String, address: ES.Address, specialty: \{String\}, category: String] \cup \right.
\]

\[
\left( [phone: Integer] \cup \text{phone\_if} \right) \cup \left( [nearby: ES.Fast\_Food] \cup \text{nearby\_if} \right) \cup \left( [midprice: Integer] \cup \text{midprice\_if} \right) \cup \left( [owner: ES.Owner] \cup \text{owner\_if} \right)
\]

**Integrity constraint rules.** An *if then* integrity constraint rule is integrated into an OLCD class description, by using the `∧`, `∪` and `¬` constructs. For example, the rule:

\[
\text{rule Rule1 forall X in Restaurant :}
\]

\[
(X.category > 5) \text{ then X.tourist\_menu\_price} > 100;
\]
is added to the ES.Restaurant description as follows:

\[ \sigma_F(\text{ES.Restaurant}) = \Delta \left( \{ r\text{-code} : \text{String}, \text{name} : \text{String}, \right. \]
\[ \left. \text{street} : \text{String}, \right. \]
\[ \left. \text{zip\_code} : \text{String}, \text{pers\_id} : \text{Integer}, \text{special\_dish} : \text{String}, \right. \]
\[ \left. \text{category} : \text{Integer}, \text{tourist\_menu\_price} : \text{Integer} \} \right) \cap \]
\[ \left( \neg(\text{category} > 5) \cup (\text{tourist\_menu\_price} > 100) \right) \]

Then, in our framework, integrity constraints are statements about the world and not about the contents of the database as in Reiter’s approach [Rei88].

**Intensional relationships.** Are not translated.

**Extensional relationships.** An “isa” relationships \( C_1 \ ISA \ C_2 \) related to an Extensional relationships and expressed in ODL by the rule:

\[ \text{rule Rule2 for all } X \text{ in } C_1 \text{ then } X \text{ in } C_2 \]

is integrated in the \( C_1 \) class description, by using the \( \cap \) construct: \( \sigma_F(C_1) = C_2 \cap \ldots \)

**Foreign keys.** A Foreign key where the class \( c_1 \) contains a field that references the class \( c_2 \), is translated introducing one dummy attribute in both classes to keep track of the relationship.

For such foreign key we introduce in the definition of \( c_1 \) the attribute dummy1 of type \( c_2 \), and we introduce in \( c_2 \) the attribute dummy2 of type \( c_1 \).

**Mapping Rules.** Are not translated.

**Join Map.** Are not translated.

### 2.1.5 The WordNet lexical database

WordNet [GGV96, Mil95] is an on-line lexical reference system whose design is inspired by current psycholinguistic theories of human lexical memory. It is the most important resource for researcher in computational linguistic, text analysis, and other related fields.

In WordNet English nouns, verbs, and adjectives are organized into synonym sets, each representing one underlying lexical concept. Different relations link the synonym sets. WordNet presently contains approximately 95,600 different word forms organised into some 70,100 word meanings, or sets of synonyms (synset).

Lexical semantics begins with a recognition that a word is a conventional association between a lexicalized concept (the meaning) and an utterance (the written or pronounced word) that plays a syntactic role. This is a many-to-many association; associations can be distinguished in the following properties:

**Synonymy:** property of a meaning that has two or more words that express it. A group of synonyms is called (in WordNet) synset. Note that for each meaning/concept exists one and only one synset. We will denote a synset as \( s \), and \( \delta \) will denote the set of all synset.

**Polysemy:** property of a word to have two or more meanings.
Since the word *word* is commonly used to refer both to the utterance and to its associated concept, discussions of this lexical association are vulnerable to terminological confusion. In order to reduce ambiguity, therefore, *word form* will be used here to refer to the physical utterance or inscription and *word meaning* to refer to the lexicalized concept that a form can be used to express.

The correspondence between the *word form* and the *word meaning* is given by the Lexical Matrix $\mathcal{M}$, where by columns there are the word form and by rows the word meaning (one row represent a *synset*). If there is more than one entry in the same column, the word form is polysemous; if there are two entries in the same row, the two word forms are synonyms (relative to a context).

Every element of the matrix is an *entry definition* $e = (f, m)$, where $f$ is the word form and $m$ (meaning) is the meaning counter. For example, $(\text{address}, 2)$ refers to the place where a person or organisation can be found or communicated with, and $(\text{address}, 1)$ refers to a computer address.

In the following we will denote a *word form* and the meaning of a definition $e = (f, m)$ respectively with $e.f$ and $e.m$. An element of the lexical matrix $\mathcal{M}$ can be *null* or *undefined*.

Since a *synset* is associated with a single line of the $\mathcal{M}$, in the following we will denote $s \in \mathcal{S}$ as row index for $\mathcal{M}$. In other words, the not null elements of the row $\mathcal{M}[s]$, represent all and only the elements of $s$. In the same way, since a *word form* is associated to a single column or $\mathcal{M}$ in the following we will use the *word form* as column index of $\mathcal{M}$.

### Semantic relationships between schema terms

In this subsection we formally define what we mean by extracting *Thesaurus relation from the WordNet database*.

#### Formalization of the WordNet database

Let $\mathcal{T}_w$ the list of terms in WordNet and let $\mathcal{S}_w$ the list of *synset* in WordNet.

The lexical Matrix $\mathcal{M}$ can be expressed as a set:

$$\mathcal{M} \subseteq \mathcal{T}_w \times \mathcal{S}_w$$

We express the set of possible WordNet relationships between *synsets* as:

$$\mathcal{P}_w = \{\text{Synonymy, Hypernymy, Hyponymy, Co-relation}\}$$

The lexical database can then be expressed as a set of relationships:

$$\mathcal{R}_w \subseteq \mathcal{S}_w \times \mathcal{P}_w \times \mathcal{S}_w$$

The tuple $(\mathcal{T}_w, \mathcal{S}_w, \mathcal{M}, \mathcal{R}_w)$ represents the information we use from WordNet.

#### Formalization of the thesaurus relationships

Let $\mathcal{E}_e$ the set of element in a schema to be integrated. An element $e \in \mathcal{E}_e$ represents a local class or a local class attribute.

We define the set of thesaurus relations

$$\mathcal{F}_t = \{\text{SYN, BT, RT}\}$$
Our goal is to extract as much as possible correct relationships between elements in $E_a$, that is, we need to express a definition formula for the set:

\[ R_t \subseteq E_a \times P_t \times E_a \]

### Extracting thesaurus relationships from the WordNet database

Our process for extracting $R_t$ is the following:

1. **Definition of a mapping function between the relationships of WordNet and the thesaurus relationships:**

   \[ \phi_t : P_w \to P_t \]

   The definition of the function $\phi_t$ is given by the following table:
   - **Synonymy:** corresponds to a SYN relation.
   - **Hypernymy:** corresponds to a BT relation.
   - **Hyponymy:** corresponds to a RT relation.
   - **Correlation:** corresponds to a RT relation.

   The $\phi_t$ function is independent from the schema to be integrated.

2. **Definition of a partial Annotation function $\alpha : E_a \to 2^{\phi_t}$**.

   The $\phi_t$ function is strictly dependent from the schema to be integrated. This function must be defined by the designer during the integration process.

3. The set of interesting thesaurus relationships $R_t$ derived from WordNet is given by:

   \[ R_t = \{(e_1, r_t, e_2) \in E_a \times P_t \times E_a : \exists (a_1, \phi_t(r_t), a_2) \in P_w, \alpha(e_1), a_2 \in \alpha(e_2)\} \]

### Use case

To be used, the algorithm of thesaurus relationship extraction requires the designer to annotate the source schema. This means that for almost each element of the various sources the designer must associate to it a WordNet synset.

The method we propose to annotate a schema element is in two phases and consists of:

1. choosing the word form of the element. By default the given utterance is selected, but, such name is not always correct. For example, consider the attribute FD.Fast.Food.midprice the designer will have to map it into the word form *price* in order to get a proper meaning from WordNet.
2. when the word form has been selected, the designer must resolve the word Polysemny. This is done by choosing the right synset between the ones related to the selected word form.

The human contribution is necessary since the designer, relying on his experience and source knowledge, will choose which is the right word form and the right meaning for each entity.

The annotation is the most sensible phase of the integration in the MOMIS environment, since a good annotation will cause the extraction of good relationships and this will generate good clusters of similar concepts in different sources.

---

1. The attributes to be annotated are usually the ones the designer considers significant.
With annotation we state the association between elements and WordNet *synsets*; this allows us to transpose relationships between *synsets* on the *implied* relationships between schema elements.

## 2.2 Building the Common Thesaurus

To develop intelligent techniques for semantic integration, inter-schema knowledge between information sources in the considered domain has to be identified and properly represented. For this purpose, we construct a *Common Thesaurus* of intensional and extensional relationships, describing inter-schema knowledge about ODL classes and attributes of source schemas. The Common Thesaurus provides a reference on which to base the identification of ODL classes candidate to integration and subsequent derivation of their global representation. In the Common Thesaurus, we express inter-schema knowledge in form of relationships (SYN, BT, NT, and RT) and extensional relationships (SYN$_{ext}$, BT$_{ext}$, and NT$_{ext}$) between classes and/or attribute names. The Common Thesaurus is constructed through an incremental process during which the following type of relationships are added:

1. *schema-derived relationships*
2. *lexicon-derived relationships*
3. *designer-supplied relationships*
4. *inferred relationships*

Relationships present in the Common Thesaurus are used by the subsequent phase of semantic information integration (see section 2.3).

There are basically two types of relationships differing by strength: (1) intensional relationships, which are the weakest type of relationships, and (2) extensional relationships, which are the stronger type of relationships since they also have an extensional impact. The designer may at any time “strengthen” an intensional relationship and promote it to an extensional relationship. The specification of an extensional relationship implies compatibility between the related elements and enables subsumption computation (i.e., inferred relationships) and consistency check between the elements involved in the relationship.

### 2.2.1 Schema-derived relationships

In this step terminological and extensional relationships at intra-schema level are extracted by analysing each source schema description separately. Here are various cases for the different types of sources.

- From *object* sources
  
  ODB-Tools [BBSV97a] examines the given inheritance and aggregation hierarchies between classes and generates respectively NT$_{EXT}$ and RT relationships. NT$_{EXT}$ and RT may be explicit (direct inheritance, complex attributes) or computed by the subsumption algorithm.

- From *relational* sources
  
  an RT relationship is extracted for each foreign key.
2.2 Building the Common Thesaurus

a $NT_{EXT}$ relationship is extracted each time the foreign key is also a key (primary or candidate) of the class.

a $SYN_{EXT}$ relationship is extracted each time between two classes there exists a reciprocal relationship of $BT_{EXT}$ (or $NT_{EXT}$).

- From semistructured sources relationships are extracted applying techniques cited for the object and relational sources.

**Example 1** Consider the **ED** and **FD** sources. A subset of intra-schema relationships automatically extracted are the following:

- **ED.Fast_Food** rt **ED.Owner** since the type of the attribute **owner** of **ED.Fast_Food** is **Owner**.
- **ED.Fast_Food** rt **ED.Address** since the type of the attribute **address** of **ED.Fast_Food** is **Address**.
- **FD.Restaurant** rt **FD.Person** since exists a **foreign key** from **Restaurant** to **Person**.
- **FD.Restaurant** bt **FD.Bistro** since there is a **foreign key** from **Bistro** to **Restaurant** and such foreign key is also a key for **Bistro**.
- **ED.Address** rt **ED.Owner** since the type of the attribute **address** of **ED.Owner** is **Address**.

**2.2.2 Lexicon-derived relationships**

In this step, terminological and extensional relationships existing at inter-schema level are extracted by analyzing ODL schemas together. The extraction of these relationships is based upon the lexical relationships existing between classes and attributes names, deriving from the meaning associated to the schema element’s name (see section 2.1.5 on page 15). It is the designer’s task to assign descriptive/meaningful names or, at least, correctly interpretable names.

Knowledge carried by schema names is one of the most important information we can rely in the integration process. To better exploit such knowledge for extracting terminological relationship, WordNet [Mil95] lexical system has been used. This enables our system to extract a very high number of thersaurus relationships. It is very hard to carry out manually these relationships when the number and dimensions of schema grows. Relationships extracted by WordNet are then proposed to the the designer for validation (the designer can drop unwanted relationships).

**Example 2** Consider the **ED** and **FD** sources. The relationships derived using WordNet are the following:

| ED.Address | syn | ED.Fast_Food.address |
| ED.Address | syn | ED.Owner.address |
| ED.Address | syn | FD.Brasserie.address |
| ED.Address.street | syn | FD.Restaurant.street |
| ED.Fast_Food.address | syn | FD.Brasserie.address |
| ED.Fast_Food.category | syn | FD.Restaurant.category |
| ED.Fast_Food.midprice | syn | FD.Restaurant.tourist_menu_price |
| ED.Fast_Food.specialty | syn | FD.Restaurant.special_dish |
2.2.3 Designer-supplied inter-schema relationships

In this step, new relationships can be added directly by the designer to capture specific domain knowledge about the source schemas (e.g. new synonyms).

This is a crucial operation, because the new relationships are forced to belong to the Common Thesaurus and are thus used to generate the global integrated schema. This means that, if a nonsense or wrong relationship is inserted, the subsequent integration process can produce a wrong global schema. The following Relationship validation section shows how our system helps the designer in detecting wrong relationships.

Example 3 In our example, the designer supplies the following relationships for classes and attributes:

- ED.Fast_Food syn FD.Restaurant
- ED.Fast_Food.category bt FD.Bistro.type
- ED.Fast_Food.specialty bt FD.Bistro.special_dish

2.2.4 Relationships validation

In this step, ODB-Tools is employed to validate intensional relationships between attributes and extensional relationships between classes.

Intensional relationships between attributes

The validation of intensional relationships between attributes is based on the compatibility of the domains of the attributes. This way, valid and invalid intensional relationships are distinguished. In particular, let \( \alpha_1 = (n_1, d_1) \) and \( \alpha_2 = (n_2, d_2) \) be two attributes characterized by name and domain. The following checks are executed on intensional relationships defined on attributes in the Common Thesaurus:

- \( \langle \alpha_1 \text{ SYN } \alpha_2 \rangle \): the relationship is marked as valid if \( d_1 \) and \( d_2 \) are equivalent, or if one is a specialization of the other;
- \( \langle \alpha_1 \text{ BT } \alpha_2 \rangle \): the relationship is marked as valid if \( d_1 \) contains or is equivalent to \( d_2 \).

ED.Owner nt FD.Person
ED.Owner syn ED.Fast_Food.owner
ED.Owner.address syn ED.Fast_Food.address
ED.Owner.address syn FD.Brasserie.address
ED.Owner.name syn FD.Brasserie.name
ED.Owner.name syn FD.Restaurant.name
FD.Brasserie nt FD.Restaurant
FD.Brasserie rt FD.Bistro
FD.Brasserie.name syn FD.Restaurant.name
FD.Person bt ED.Fast_Food.owner
FD.Person.first_name nt ED.Owner.name
FD.Person.first_name nt FD.Brasserie.name
FD.Person.first_name nt FD.Restaurant.name
FD.Person.first_name rt FD.Person.last_name
FD.Person.last_name nt ED.Owner.name
FD.Person.last_name nt FD.Brasserie.name
FD.Person.last_name nt FD.Restaurant.name
2.2 Building the Common Thesaurus

- \(a_1 \text{ NT } a_2\): the relationship is marked as valid if \(d_1\) is contained in or is equivalent to \(d_2\).

When an attribute domain \(d_1\) (\(d_2\)) is defined using the \texttt{union} constructor, as in the Address example (see Figure 2.7 on page 13), a \textit{valid relationship} is recognised if at least one domain \(d_1\) (\(d_2\)) is compatible with \(d_2\) (\(d_1\)).

**Extensional relationships between two classes**

As an extensional relationship between two classes \(C_1\) and \(C_2\), the validation is performed checking the consistency of a \textit{virtual schema} described in OLCD exploiting ODB-Tools capabilities. The \textit{virtual schema} is generated by applying common thesaurus relationships to the ODL_{j2} schema. Extensional relationships affects the inheritance hierarchies of the schema.

For example, the extensional relationship:

\[
\text{FD.Restaurant btExt FD.Bistro}
\]

stated by the designer is expressed in the FD.Bistro class description as follows:

\[
\sigma_P(\text{FD.Bistro}) = \text{FD.Restaurant}\Pi
\]

\[
\Delta[\text{f_code: String, type: String, pers_id: Integer}]
\]

Since the FD.Bistro class description is consistent, the relationship between FD.Bistro and FD.Restaurant is validated. On the other hand, the extensional relationship

\[
\text{FD.Restaurant btExt ED.Fast_Food}
\]

is inconsistent (the attribute \texttt{category} is defined in both the classes but on disjoint domains). In the presence of integrity rules less intuitive incoherencies may arise. In this case the designer may choose to maintain only the terminological relationship

\[
\text{ED.Restaurant bt FD.Fast_Food}
\]

in the Common Thesaurus.

2.2.5 Checking consistency and inferring new relationships

In this step, inference capabilities of ODB-Tools are exploited to validate the common thesaurus relationships and to infer new relationships, in order to set up a rich common thesaurus to support the identification of semantically similar ODL_{j2} classes in different sources, as will be shown in next section. \textit{In the running example there are no inferred relations.} To perform this task it is necessary to build a \textit{virtual schema} taking in account all the common thesaurus relationships, then to run ODB-Tools on such virtual schema. ODB-Tools will compute incoherent relationships and will also infer all relationships involved by the \textit{virtual schema}; then, such new relationships will be added to the common thesaurus.

2.2.6 Virtual schema creation

The virtual schema passed to ODB-Tools description logic engine is described in OLCD (see section 2.1.4 on page 13) and is derived from the schema of the sources applying some modifications implied by the common thesaurus relationships.
Generating the virtual schema we exploit the ODB-Tools validation and inference capabilities to discover incoherences due to the common thesaurus relationships, validate relationships between attributes with respect to domain matching, and infer all possible relationship involved by the existent ones.

Since ODB-Tools is able to discover schema incoherences, during the virtual schema generation we transform the extensional relationships (the strongest relationship type) into structural declarations. If ODB-Tools does not signals any incoherence this means that extensional relationships are not conflicting.

Validation of the existing relationships is possible by defining, in the virtual schema, a virtual classes for each schema element involved in a relation. ODB-Tools will compute all relationships between such virtual classes based on relationships and schemata information and will produce a list of schema compliant relationships. If an old relationship appears in the list then this is marked as valid. Relationships that are in the list but do not exist in the common thesaurus are added as inferred relationships.

The various relationships types will produce the following modifications:

- **Extensional SYN**
  such relationship cause the creation of a double inheritance, that is
  
  \[ c_1 \text{SY} N_{ex} \text{c}_2 \]
  
  is converted into the following two OLCD declarations
  
  \[ \sigma_P(c_1) = c_2 \cap ... \]
  \[ \sigma_P(c_2) = c_1 \cap ... \]

- **Extensional NT and BT**

  \[ c_1 \text{NT}_{ex} c_2 \]

  is converted into the following OLCD declaration:

  \[ \sigma_P(c_1) = c_2 \cap ... \]

- **Intensional SYN**

  the idea is to find sets of synonym classes and make them homogeneous.
  A set of synonym classes is given by all the local classes directly related by a SYN relationship.
  To make a set of synonym classes “homogeneous” all class description are re-defined in order to share a common definition obtained by the union of the definitions of all the synonym classes. To do that we need to define inheritance, attributes and related classes (which are the classes in RT relation).

  Using the notation introduced in section 2.1.5 on page 16,
  Let \( \mathcal{C}_s \) the set of local classes in the schema.
  Let \( \mathcal{A}_s \) the set of attributes of all local classes in the schema.
  Let \( \gamma_s : \mathcal{A}_s \rightarrow \mathcal{C}_s \) the function that relates attributes and classes of the schema; \( \gamma_s(a) \) returns the class where \( a \) is defined as attribute.
  Let \( \mathbb{B}_s \) the set of common thesaurus relations.
  Let \( S_{syn} \subseteq \mathcal{C}_s \) a set of synonym classes.

  The set of inheritance shared by all the classes in the set, \( s_i \), can be calculated by the following iterative algorithm:
2.2 Building the Common Thesaurus

1. let \( s_i = S_{syn} \)
2. let \( s_i = s_i \cup \{ c \in C_a : \exists (c, r, c_1) \in R_t, c_1 \neq c, r = NT, c_1 \in s_i, c \in C_a \} \)
3. let \( s_i = s_i \cup \{ c \in C_a : \exists (c_1, r, c) \in R_t, c_1 \neq c, r = BT, c_1 \in s_i, c \in C_a \} \)
4. if \( s_i \) grew then goto point 2
5. done.

Let \( s_a \subseteq A_a \) the set of attributes shared by all the classes in the set
\[ s_a(S_{syn}) = \{ a \in A_a : \exists c = \gamma_a(a), c \in S_{syn} \} \]

Let \( s_r \subseteq C_a \) the set of classes related to the classes in the set
\[ s_r(S_{syn}) = \{ c \in C_a : \exists (c, r, c_1) \in R_t, r = RT, c_1 \in S_{syn}, c \in C_a, c_1 \neq c \} \cup \{ c \in C_a : \exists (c_1, r, c) \in R_t, r = RT, c_1 \in S_{syn}, c \in C_a, c_1 \neq c \} \]

Once calculated, the sets \( s_i, s_a \) and \( s_r \) in the definition of each class \( c \) of the set of synonym classes \( S_{syn} \) will be something like:
\[ \sigma_P(c) = c_0 \cap \ldots \cap c_m \cap \Delta \]
\[ a_{q_1}, a_{q_2}, \ldots, a_{n_m} : a_{t_m}, a_{r_1}, a_{r_2}, \ldots, a_{r_0} : c_{r_0} \]

where
- \( a_{q_1}, \ldots, a_{q_l} \) are attribute name for attributes in \( s_a \)
- \( a_{t_1}, \ldots, a_{t_m} \) are attribute type for attributes in \( s_a \)
- \( a_{r_1}, \ldots, a_{r_0} \) are dummy attributes name, and
- \( c_{r_1}, \ldots, c_{r_0} \) \( \in s_r \).

*Intensional NT and BT*

This is a case similar to the intensional SYN relationship. In this case we have to homogenize a class \( c_h \) with the classes \( c_b \) from which intensionally it inherits.

We need to compute:
- The set \( S_{bt} \) of intensional superclasses.
- The set \( s_a \) of attributes that represent the union of attribute of \( c_b \) and attributes of the classes in \( S_{bt} \).
- The set \( s_r \) of classes related to \( c_b \) or related to any of the classes in \( S_{bt} \).

The set \( S_{bt} \) can be computed by applying the following iterative algorithm:
1. \( S_{bt} = \{ c_h \} \)
2. let \( S_{bt} = S_{bt} \cup \{ c \in C_a : \exists (c, r, c_1) \in R_t, c_1 \neq c, r = NT, c_1 \in S_{bt}, c \in C_a \} \)
3. let \( S_{bt} = S_{bt} \cup \{ c \in C_a : \exists (c_1, r, c) \in R_t, c_1 \neq c, r = BT, c_1 \in S_{bt}, c \in C_a \} \)
4. if \( S_{bt} \) grew then go to point 2
5. done.

Both sets \( s_a \) and \( s_r \) can be computed using the algorithm described above for the case of SYN relationship considering \( S_{syn} = S_{bt} \)

*Terminological RT*

For each RT relationship involving a given class \( c \) and one other class \( c_1 \), a virtual complex attribute \( a_r \) of type \( c_1 \) is added to the definition of \( c \) in the virtual schema.

See section 3.4.1 on page 53 for how this feature has been implemented in the MOMIS prototype.
2.3 Semantic information integration

In this section, we describe the information integration process to construct the global integrated view of ODL$_p$ source schemas. The proposed process allows semi-automatic identification of ODL$_p$ classes candidate to integration by means of clustering procedures based on the concept of affinity and on the relationship knowledge in the Common Thesaurus. Moreover, the process supports a semi-automated synthesis of each selected cluster into an integrated global ODL$_p$ class, by handling semantic heterogeneity through the definition of suitable mapping rules for each global class.

2.3.1 Affinity of ODL$_p$ classes

To integrate the ODL$_p$ classes of the different sources into global ODL$_p$ classes, we employ hierarchical clustering techniques based on the concept of affinity. This way, we identify ODL$_p$ 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.

This activity is performed with the ARTEMIS tool environment. ARTEMIS has been conceived for a semi-automatic integration of heterogeneous structured databases [CAV00]. In the context of MOMIS, the ARTEMIS affinity framework has been extended and applied to the analysis of ODL$_p$ schema descriptions. In the following, we describe the extensions to the affinity-based clustering to cope with object patterns and semistructured data integration. ODL$_p$ classes are analyzed and compared by means of affinity coefficients which allow us to determine the level of similarity between classes in different source schemas.

In particular, ARTEMIS evaluates a Global Affinity coefficient as the linear combination of a Name Affinity coefficient and a Structural Affinity coefficient, respectively. Affinity coefficients for ODL$_p$ classes are evaluated by exploiting terminological relationships of the Common Thesaurus. To this end, a strength $\sigma_R$ is assigned to each type of terminological relationship $\mathcal{R}$ in the Common Thesaurus, with $\sigma_{\text{SYN}} \geq \sigma_{\text{BT/NT}} \geq \sigma_{\text{RT}}$. In the following, when necessary, we use notation $\sigma_{ij}$ to denote the strength of the terminological relationship $\mathcal{R}$ for terms $t_i$ and $t_j$ in the Thesaurus; furthermore, we use $\sigma_{\text{SYN}} = 1$, $\sigma_{\text{BT}} = \sigma_{\text{NT}} = 0.8$ and $\sigma_{\text{RT}} = 0.5$.

An affinity function $A()$ is defined on top of the Common Thesaurus to evaluate the affinity of two terms. The affinity $A(t, t')$ of two terms $t$ and $t'$ is equal to the highest-strength path of terminological relationships between them, if at least one path exists, and is otherwise zero. Given two terms $t$ and $t'$ and a path of terminological relationships between them, the strength of this path is computed by multiplying the strengths of all terminological relationships involved in it. $A(t, t')$ coincides with the strength of the highest-strength path between $t$ and $t'$, denoted by $\rightarrow^m$, that is, $A(t, t') = \sigma_{t_2t_3} \cdot \sigma_{t_3t_4} \cdot \ldots \cdot \sigma_{t_{m-1}t_m}$. In the following, we use the symbol $\sim$ to denote the fact that two terms have affinity in the Common Thesaurus. Let $c$ and $c'$ be two ODL$_p$ classes belonging to sources $S$ and $S'$ respectively. Let us now define how the affinity coefficients are computed.

Name Affinity coefficient

The Name Affinity coefficient of two ODL$_p$ classes $c$ and $c'$, denoted $NA(c, c')$, is the measure of the affinity between their names $n_c$ and $n_{c'}$, if this measure exceeds a specified threshold (see Table 2.1).
2.3 Semantic information integration

Table 2.1: Name Affinity coefficient

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA(c, c')</td>
<td>$A(n_c, n_{c'})$</td>
<td>if $A(n_c, n_{c'}) \geq \alpha$</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>if $A(n_c, n_{c'}) &lt; \alpha$</td>
</tr>
</tbody>
</table>

Legend:

$n_c, n_{c'}$ denote the name of $c$ and $c'$, respectively.

$\alpha$ is a threshold used to select high values of $NA(c, c')$.

$A(t, t')$ coincides with the strength of the highest-strength path between $t$ and $t'$.

For any pairs of classes, $NA(c, c') \in [0, 1]$. $NA(c, c')$ is equal to the strength of the path $\rightarrow m$ of terminological relationships in the Common Thesaurus originating the highest strength value, if this value exceeds a specified threshold $\alpha$ (e.g., $\alpha \in [0.4, 0.6]$). In this case, the Name Affinity value is proportional to the length of the path and to the type of relationships involved in this path. In particular, $NA(c, c')$ is 1 if only SYN relationships are involved in a path. Otherwise, $NA(c', c')$ is zero.

Structural Affinity coefficient

The Structural Affinity coefficient of two ODL classes $c$ and $c'$, denoted $SA(c, c')$, is the measure of the level of matching of $c$ and $c'$ based on attribute relationships in the Common Thesaurus (see Table 2.2).

Table 2.2: Structural Affinity coefficient

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SA(c, c')$</td>
<td>$\frac{</td>
<td>{\alpha \in A(c) : \alpha \notin A(c') \text{ and } \exists \beta \in A(c') : \alpha \sim \beta } +</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>if $</td>
</tr>
</tbody>
</table>

Legend:

$C = \{\alpha \in A(c) : \exists \alpha_{c} \in A(c') : \alpha \sim \alpha_{c}\}$, where $A(c)$ and $A(c')$ are the sets of attributes in $c$ and $c'$, respectively.

$F_c = \frac{|\{x \in C : flog(x) = 1\}|}{|C|}$, control factor where $flog(x) = 1$ stands for a valid terminological relationship in the Common Thesaurus

The Structural Affinity coefficient returns a value in the range $[0, 1]$ proportional to the number of attributes of the two classes whose names have affinity in the Common Thesaurus, refined by a control factor $F_c$. In particular, $F_c$ evaluates the percentage of attributes having affinity that have a valid relationship in the Common Thesaurus (see step Validation of relationships illustrated in Section 2.2.4).

The value 0 indicates the absence of attributes with affinity in the considered classes, while the value 1 indicates that all attributes defined in the two classes have affinity and are considered valid. The greater the number of attributes with affinity in the considered classes, and the greater the number of positive validity control results, the higher the value of $SA(c, c')$.

In general, given two classes, an attribute of one class may have affinity with more than one attribute of the other class. In the evaluation of the $SA(c, c')$ coefficient, we consider these multiple affinities as a single affinity correspondence between one attribute and a set of attributes. For the evaluation of Structural Affinity, optional attributes of
ODL₁₃ classes representing object patterns must be considered properly. Depending on which attributes are taken into account, the following options are possible for the computation of the $SA()$ coefficient:

1. **All attribute-based.** With this option, *optional* attributes are treated as the other ones and are always taken into account when evaluating the affinity of ODL₁₃ classes describing object patterns.

2. **Common attribute-based.** With this option, *optional* attributes are not taken into account when evaluating the affinity.

3. **Threshold-based.** With this option, *optional* attributes are taken into account for affinity evaluation only if they are common to at least a certain number of objects (i.e., a threshold) of the considered object pattern.

The third option is difficult to apply, since it requires setting the value of a threshold, which can be dependent on the specific object pattern or on the source. As for the other two options, they have different implications on the affinity values produced. Given a pattern to be compared, the second option gives affinity values higher than the first one, in presence of the same number of attribute pairs with affinity. In fact, fewer attributes are considered at the denominator of the $SA()$ formula choosing option 2). On the other hand, if most attributes of an object pattern are optional, then option 1) is better for Structural Affinity evaluation. The choice between the first two options depends on the specific application under analysis. In our example, we applied both options and we discuss obtained values in the following, when presenting results of clustering.

**Example 4** Consider classes ED.Owner and FD.Person. By applying the all attribute-based option, we have that $SA(ED.Owner, FD.Person) = \frac{241}{354} \cdot 1 = 0.43$ due to the following affinities:

ED.Owner.name ≈ \{FD.Person.first_name, FD.Person.last_name\}.

**Global Affinity coefficient**

The Global Affinity coefficient of two ODL₁₃ classes $c$ and $c'$, denoted $GA(c, c')$, is the measure of their affinity computed as the weighted sum of the Name and Structural Affinity coefficients (see Table 2.3).

**Table 2.3: Global Affinity coefficient**

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GA(c, c')$</td>
<td>$w_{NA} \cdot NA(c, c') + w_{SA} \cdot SA(c, c')$</td>
<td>in all cases</td>
</tr>
</tbody>
</table>

Legend:

$w_{NA}$ and $w_{SA}$, with $w_{NA}, w_{SA} \in [0, 1]$ and $w_{NA} + w_{SA} = 1$, are introduced to assess the relevance of each coefficient computing the global affinity value.

Weights in $GA(c, c')$ allow the analyst to differently stress the impact of each coefficient in the evaluation of the global affinity value.

**Example 5** The Global Affinity coefficient of ED.Owner and FD.Person is computed as follows: $GA(ED.Owner, FD.Person) = 0.5 \cdot 0.8 + 0.5 \cdot 0.43 = 0.61$ using $w_{NA} = w_{SA} = 0.5$, since we consider both affinity coefficients equally relevant.
2.3 Semantic information integration

Clustering of ODL$_{f_3}$ classes

To identify all the ODL$_{f_3}$ classes having affinity in the considered source schemas, we employ a hierarchical clustering technique, which classifies classes into groups at different levels of affinity, forming a tree [Eve74]. The hierarchical clustering procedure uses a matrix $M$ of rank $K$ where $K$ is the total number of ODL$_{f_3}$ classes to be analysed. An entry $M[i,j]$ of the matrix represents the affinity coefficient $GA(c_h, c_k)$ between classes $c_h$ and $c_k$. Clustering is iterative and starts by placing each class in a cluster by itself. Then, at each iteration, the two clusters having the greatest affinity value in $M$ are merged. $M$ is updated at each merging operation by deleting the rows and the columns corresponding to the merged clusters, and by inserting a new row and a new column for the newly defined cluster $c_{hk}$. The affinity value between $c_{hk}$ and each remaining cluster $r$ in $M$ is computed. The new value between $c_{hk}$ and a remaining cluster $r$ is set to the maximum affinity value between the affinity values that $c_h$ and $c_k$ had with $r$ in $M$. The procedure terminates when only one cluster is left and produces as the output a tree, where leaves correspond to ODL$_{f_3}$ classes and intermediate nodes have an associated affinity value characterizing the underlying leaves.

<table>
<thead>
<tr>
<th>Initial Distance matrix:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements: 18</td>
</tr>
<tr>
<td>e0 Interface [ED.Address]</td>
</tr>
<tr>
<td>e1 Interface [ED.Fast_Food]</td>
</tr>
<tr>
<td>e2 Interface [ED.Owner]</td>
</tr>
<tr>
<td>e3 Interface [FD.Bistro]</td>
</tr>
<tr>
<td>e4 Interface [FD.Brasserie]</td>
</tr>
<tr>
<td>e5 Interface [FD.Person]</td>
</tr>
<tr>
<td>e6 Interface [FD.Restaurant]</td>
</tr>
<tr>
<td>e7 [odli3.SimpleAttribute: ED.Address.street type: string]</td>
</tr>
<tr>
<td>e8 [odli3.SimpleAttribute: ED.Fast_Food.address type: odli3.TypeToSolve@275a92]</td>
</tr>
<tr>
<td>e9 [odli3.SimpleAttribute: ED.Fast_Food.midprice type: unsigned long]</td>
</tr>
<tr>
<td>e10 [odli3.SimpleAttribute: ED.Owner.address type: odli3.TypeToSolve@2b5267]</td>
</tr>
<tr>
<td>e11 [odli3.SimpleAttribute: ED.Owner.name type: string]</td>
</tr>
<tr>
<td>e12 [odli3.SimpleAttribute: FD.Bistro.name type: string]</td>
</tr>
<tr>
<td>e13 [odli3.SimpleAttribute: FD.Person.first_name type: string]</td>
</tr>
<tr>
<td>e14 [odli3.SimpleAttribute: FD.Person.last_name type: string]</td>
</tr>
<tr>
<td>e15 [odli3.SimpleAttribute: FD.Restaurant.name type: string]</td>
</tr>
<tr>
<td>e16 [odli3.SimpleAttribute: FD.Restaurant.street type: string]</td>
</tr>
<tr>
<td>e17 [odli3.SimpleAttribute: FD.Restaurant.tourist_menu_price type: signed long]</td>
</tr>
<tr>
<td>src e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15 e16 e17</td>
</tr>
<tr>
<td>e0</td>
</tr>
<tr>
<td>e1</td>
</tr>
<tr>
<td>e2</td>
</tr>
<tr>
<td>e3</td>
</tr>
<tr>
<td>e4</td>
</tr>
<tr>
<td>e5</td>
</tr>
<tr>
<td>e6</td>
</tr>
<tr>
<td>e7</td>
</tr>
<tr>
<td>e8</td>
</tr>
<tr>
<td>e9</td>
</tr>
<tr>
<td>e10</td>
</tr>
<tr>
<td>e11</td>
</tr>
<tr>
<td>e12</td>
</tr>
<tr>
<td>e13</td>
</tr>
<tr>
<td>e14</td>
</tr>
<tr>
<td>e15</td>
</tr>
<tr>
<td>e16</td>
</tr>
<tr>
<td>e17</td>
</tr>
</tbody>
</table>

Figure 2.8: Initial affinity matrix for clustering

We can see a complete process of clustering starting from initial thesaurus relationships applying the relationships weights (for SYN, NT, RT respectively 1, 0.8, 0.5) is computed the Initial affinity matrix (Figure 2.8). In Figure 2.9 shows the corresponding Naming Affinity Matrix, Figure 2.10 shows the Structural Affinity Matrix, and last Figure 2.11 shows the Global Affinity Matrix.

Figure 2.12 shows the affinity tree resulting from clustering (using 0.4 as threshold value) our set of ODL$_{f_3}$ classes by using the all attribute-based method. Once the affinity tree has been constructed, an important issue is related to the selection of clus-
Making Affinity matrix:
Elements: 18
- e0 Interface [ED.Address]
- e1 Interface [ED.Fast_Food]
- e2 Interface [ED.Owner]
- e3 Interface [FD.Bistro]
- e4 Interface [FD.Brasserie]
- e5 Interface [FD.Person]
- e6 Interface [FD.Restaurant]
- e7 [odli3.SimpleAttribute: ED.Address.street type: string]
- e8 [odli3.SimpleAttribute: ED.Fast_Food.address type: odli3.TypeToSolve@275a92]
- e9 [odli3.SimpleAttribute: ED.Fast_Food.midprice type: signed long]
- e10 [odli3.SimpleAttribute: ED.Owner.address type: odli3.TypeToSolve@2b5267]
- e11 [odli3.SimpleAttribute: ED.Owner.name type: string]
- e12 [odli3.SimpleAttribute: FD.Brasserie.name type: string]
- e13 [odli3.SimpleAttribute: FD.Person.first_name type: string]
- e14 [odli3.SimpleAttribute: FD.Person.last_name type: string]
- e15 [odli3.SimpleAttribute: FD.Restaurant.name type: string]
- e16 [odli3.SimpleAttribute: FD.Restaurant.street type: string]
- e17 [odli3.SimpleAttribute: FD.Restaurant.tourist_menu_price type: signed long]

Figure 2.9: Naming Affinity Matrix for clustering

 ters to be integrated for the definition of the global ODL classes of the integrated schema. Cluster computation is interactive, based on the numerical affinity values in the affinity tree. In particular, ARTEMIS provides a threshold-based mechanism for cluster selection. The designer specifies a value for a threshold \( T \) and clusters characterised by an affinity value greater than or equal to \( T \) are selected and proposed to the designer. High values of threshold will generate small, highly homogeneous clusters. By decreasing \( T \)’s value, clusters containing more ODL classes can be selected (see cluster generated with \( T = 0.4 \) in figure 2.14). In the tool, the default value of \( T \) is set to 0.5. This default value can be refined dynamically, on the basis of the characteristics of retrieved clusters and of the specific application under analysis. Once clusters have been selected, ODL classes that have an extensional terminological relationship with at least one class in the cluster and not yet included in it (if any), are forced to belong to the cluster anyway, to define an integrated global ODL class that is representative of all possible semantically related source classes.

The designer can confirm this threshold-based cluster selection made by the tool for the subsequent synthesis activity.

2.3.2 Synthesis into an integrated schema description

The generation of ODL global classes out of selected clusters is a synthesis activity performed interactively by the designer. Synthesis of clusters of ODL classes requires taking into account semantic heterogeneity, which has to be treated properly to come up with an integrated and uniform representation at the global level. Let \( C_i \) be a selected cluster in the affinity tree and \( ge_i \) the global ODL class to be defined for \( C_i \).
2.3 Semantic information integration

Structural Affinity Matrix:

Elements: 6
- e0 Interface [ED.Address]
- e1 Interface [ED.Fast_Food]
- e2 Interface [ED.Owner]
- e3 Interface [FD.Brasserie]
- e4 Interface [FD.Person]
- e5 Interface [FD.Restaurant]

<table>
<thead>
<tr>
<th>src-</th>
<th>e0</th>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
<th>e5</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.08</td>
</tr>
<tr>
<td>e1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.09</td>
</tr>
<tr>
<td>e2</td>
<td>0.09</td>
<td></td>
<td>0.16</td>
<td>0.28</td>
<td></td>
<td>0.06</td>
</tr>
<tr>
<td>e3</td>
<td></td>
<td>0.16</td>
<td></td>
<td>0.28</td>
<td></td>
<td>0.09</td>
</tr>
<tr>
<td>e4</td>
<td></td>
<td></td>
<td>0.28</td>
<td>0.28</td>
<td></td>
<td>0.16</td>
</tr>
<tr>
<td>e5</td>
<td>0.08</td>
<td>0.06</td>
<td>0.09</td>
<td>0.09</td>
<td>0.16</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.10: Structural Affinity Matrix for clustering

First, we associate with \( gc_i \) a set of global attributes, corresponding to the union of the attributes of the classes belonging to \( Cl_i \). The attributes having a valid terminological relationship are unified into a unique global attribute in \( gc_i \). The attribute unification process is performed automatically for what concerns names according to the following rules:

- for attributes that have a SYN relationship, only one term is selected as the name for the corresponding global attribute in \( gc_i \);
- for attributes that have a BT/NT relationship, a name which is a broader term for all of them is selected and assigned to the corresponding global attribute in \( gc_i \).

For example, the attribute unification process for cluster \( Cl_2 \) of figure 2.13 on page 32 (that will be named in FoodPlace) automatically produces the following set of global attributes:

- name, address, phone?, specialty, category, nearby?, midprice?, owner?, special_dish, street, zip_code, type, r_code, b_code, pers_id, tourist_menu_price

The designer can add mapping rules to properly set the global class. A global class also includes mapping rules for global attributes. A mapping rule is defined for each global attribute \( a \) of \( gc_i \) and specifies:

- Attribute correspondences in the cluster: values of \( a \) depend on the attributes that have been unified into \( a \) during the construction of \( gc_i \). Mapping rules are defined to state for \( a \) which attributes of the ODL\(_{13}\) classes in the cluster under analysis correspond to \( a \). In specifying mapping rules for global attributes, the following correspondences can be specified:

  1. And correspondence: this specifies that a global attribute corresponds to the concatenation of two or more attributes of a class \( c_h \in Cl_i \). For example, by defining a mapping rule for the global attribute name of...
Global Affinity matrix:

<table>
<thead>
<tr>
<th>e0</th>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
<th>e5</th>
<th>e6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>0.25</td>
<td>0.16</td>
<td>0.16</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>e1</td>
<td>0.25</td>
<td>0.29</td>
<td>0.32</td>
<td>0.32</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>e2</td>
<td>0.25</td>
<td>0.29</td>
<td>0.24</td>
<td>0.24</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>e3</td>
<td>0.16</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>e4</td>
<td>0.16</td>
<td>0.32</td>
<td>0.24</td>
<td>0.24</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>e5</td>
<td>0.24</td>
<td>0.43</td>
<td>0.24</td>
<td>0.24</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>e6</td>
<td>0.24</td>
<td>0.43</td>
<td>0.24</td>
<td>0.44</td>
<td>0.33</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Figure 2.11: Global Affinity Matrix for clustering

C12, the designer specifies that a global attribute name corresponds to both first_name and last_name attributes of FD.Person class. By specifying the and correspondence between first_name and last_name for the global attribute name, the designer states that the values of both first_name and last_name attributes have to be considered as values of name when class FD.Person is considered.

2. Or correspondence: this specifies that a global attribute corresponds at most to one of the attributes of a class ci \( \in \mathcal{C} \). An or correspondence is useful when a global attribute is suitable for two or more local attributes of a source, depending on the value of another local attribute, called “tag attribute”. For example, let us suppose we have a cluster describing an automobile global class and that classes in the cluster have price values for cars in Italian Liras and US Dollars. Here, country is the tag attribute. In this example, it is possible to define an or correspondence between the attributes Italian_price and US_price by declaring the following mapping rule:

```java
attribute integer price
  mapping rule(S.car.Italian_price union S.car.US_price on Rule1),
  ...
rule Rule1 { case of S.car.country:
    "Italy" : S.car.Italian_price;
    "US"  : S.car.US_price; }
```

- Default/null values: these are possibly defined for local attributes corresponding to \( \alpha \), based on the knowledge of the single local source, if \( \alpha \) is not applicable in the considered source. For example, with reference to \( C1 \), the mapping rule defined for the global attribute zone specifies that the objects of the class ED.Fast_Food regard the “Pacific Area” while objects of FD.Restaurant
Cluster Tree:

```plaintext
| tools.Distance: 0.25  
+src:  
  | Interface [ED.Address]  
+dst:  
  | tools.Distance: 0.34285714285714286  
  +src:  
    | Interface [ED.Owner]  
    +dst:  
      | Interface [FD.Person]  
    +dst:  
      | tools.Distance: 0.4  
      +src:  
        | Interface [FD.Bistro]  
        +dst:  
          | tools.Distance: 0.43125  
          +src:  
            | Interface [ED.Fast_Food]  
            +dst:  
              | tools.Distance: 0.4454545454545455  
              +src:  
                | Interface [FD.Brasserie]  
                +dst:  
                  | Interface [FD.Restaurant]  
```

Figure 2.12: Cluster Tree in clustering

and FD.Bistro wherever in the USA.

For each global ODL\(_2\) class \(g_i\), a persistent mapping table storing all the mapping information is generated.

As an example, the mapping table for the Food.Place class, set by the mapping rules of figure 2.15, is shown in figure 2.16.

The mapping table in figure 2.15 describes the Food.Place global class. In the first row there is the list of local classes composing the global class. In the first column there is the list of the global attributes of the global class. Given a row (the global attribute \(a_g\)) and a column (the local class \(c_l\)) the corresponding mapping table cell \(m_{c_l}\) describes how combine the local attribute of \(c_l\) to compute the value for \(a_g\). The content of the cell \(m_{c_l}\) means as follows:

- **a single name:** means simple mapping, the local attribute is mapped as global attribute for the instance of the local class.
- **NULL:** the local class does not export any values for the given global attribute. It simply returns a `null` value.
Clusters:

cluster [0]
  [Interface [ED.Address]]

cluster [1]
  [Interface [ED.Owner]]
  [Interface [FD.Person]]

cluster [2]
  [Interface [FD.Bistro]]
  [Interface [FD.Fast_Food]]
  [Interface [FD.Brasserie]]
  [Interface [FD.Restaurant]]

Figure 2.13: Proposed clusters

Figure 2.14: Example of affinity tree

- an *and* between names: there is an *and* correspondence between attributes.
- an *or* between names: there is an *or* correspondence between attributes.
- a string between quotation marks: a constant (default) value is associated to this global attributes for each instance of the local class.

Integrity constraint rules can also be specified for global ODL classes to express semantic relationships existing among the different sources. Suppose that in our domain, a relationship exists between the category and the price of a food place. For example, the fact that all the food places with a ‘High’ category have a price higher than $100 can be expressed by the following integrity constraint rule in the global schema:

```plaintext
rule Rule2 forall X in Food Place :
  (X.category = 'High') then X.price > 100;
```
Base Extension and Extensional Hierarchy

Base Extension and Extensional Hierarchy are used by MOMIS to optimize the execution cost of a query.

A base extension data structure describes how data (extensions) stored in different local classes overlap. For each global class, a set of base extensions is built which describes how local classes data overlaps.

The information required for building base extensions is both supplied by the integration designer in the form of extensional axioms, and is extracted also from the local schemata.

In this section we will use the following terms with these meanings:

- **Instance** represents one of the data structure instance of any local classes.
- **Object** represents one object in the reality. For example, a restaurant $r$, which is an example of a real world object. The object $r$ is described in various data sources like the yellow pages database or the tax office database. In both such data sources there is at least one instance that refers to the same $r$ object.

The main property of a Base Extension is that an object always belongs to a single base extension, which further groups all local classes containing the instances that
The set of Base Extensions specific to each global class is then organized in such a way to optimize retrieving during global class querying. An Extensional Hierarchy is intuitively a specialisation hierarchy of basic extensions, based on the attributes common to the various basic extensions. The Extensional Hierarchy allows, given the attribute set of a query, to quickly retrieve the (minimal) set of basic extensions to query for obtaining the correct query answer with the minimal cost.

The algorithm for Base Extensions and Extensional Hierarchy computation is described in [SS98] starting from Extensional relationships (see section 2.1.3 on page 11) that we call also extensional axioms.

Base Extension and Extensional Hierarchy are Global class specific. Each global class has its own Base Extension and Extensional Hierarchy.

In the Masters thesis [Ven00] the theory related to the Base Extension and Extensional Hierarchy is described in detail, moreover it describes how such techniques are implemented in the MOMIS prototype.

Using Base Extension and Extensional Hierarchy

Here is a brief description of how Base Extension and Extensional Hierarchy technique are used in query execution optimization.

The execution plan is the following:

- generic queries containing joins are split in a set of basic queries. A basic query is a query on a single global class.
- each basic query is split into local queries to send to the source wrappers. This phase relies on Extensional Hierarchy and Base Extension in order to quickly choose local class involved by the basic query and generate the local queries.
- local queries are executed and the results are fused (using join maps, see section 2.5 on page 35) to obtain the basic query result set.
- last the join between the basic queries is computed in order to obtain the global query result set.
2.5 The Object Fusion problem: the Join Maps

The Object Fusion problem faces the problem of how to identify entries in different local classes (from different data sources) that refer to the same real world object. This is a problem that the query manager should resolve at each query execution during the phase of recomposition of data coming from the local classes, according to the information given by the designer during integration phase. Fusion ensures the correctness, completeness and minimality of the response but is not easy to implement. The following points basically exists:

- the mediator is not owner of the objects stored in local classes but only provides a global view. This mean that the mediator cannot impose a way to identify objects in the different sources. Each source may have its own techniques to retrieve objects, such as keys for relational sources and OID for object sources. Usually instance referring the same real word object are identified with different keys or OID, depending on the source such object is stored.
- it is necessary to study and use techniques to uniquely identify a given object across global schema and local sources.
- it is it is necessary to supply information to make such techniques work.

The MOMIS approach to object fusion is based on data structures called Join Maps. Join Maps are extensional relationships between local classes of the same global class; this fusion is performed on global classes object from local classes objects. Join Maps are based on the implicit assumption of the existence of keys for the local Interfaces. A single Join Map relationship between (local) classes \( I \) and \( N \) expresses how uniquely map instances of \( I \) and instances of \( N \) specifying something like a join predicate. This is possible expressing the correspondence between the keys attributes of the two interfaces.

In MOMIS we faced the problem of automatically extracting as much implied Join Maps as possible, and how to manage fusion based on semantically homogeneous and heterogeneous keys.

By exploiting information, such as: extensional thesaurus relationships, terminological thesaurus relationships, extensional axioms, mapping table, and keys (primary and candidate) of the local classes, it is possible to compute automatically a number of possible Join Maps\(^2\). For example, we automatically extract a Join Map between two attributes \( a_1 \) and \( a_2 \) semantically homogeneous, when (1) the are respectively key for two local classes \( c_1 \) and \( c_2 \) mapped in the same global class, (2) exists a terminological SYN relationship between \( a_1 \) and \( a_2 \) and (3) are both mapped in the same global attribute.

Joins between local classes having heterogeneous keys is possible using an appropriate conversion table. Such table is imported in momis as any other local classes from a source connected by a wrapper.

Refer to the thesis [Fer00] for more details on fusion solution adopted in the MOMIS project.

\(^2\) All Join Maps automatically computed must be validated by the designer
2.6 Related works

Works related to the issues discussed in this paper are in the area of semistructured data and heterogeneous information integration.

Heterogeneous information integration

In this area, many projects based on a mediator architecture have been developed. For example, the mediator-based TSIMMIS project [CMH94] follows a ‘structural’ approach and uses a self-describing model (OEM) to represent heterogeneous data sources and pattern matching techniques to perform a predefined set of queries based on a query template.

The semantic knowledge is effectively encoded in the MSL (Mediator Specification Language) rules enforcing source integration at the mediator level. Although the generality and conciseness of OEM and MSL make this approach a good candidate for the integration of widely heterogeneous and semistructured information sources, a major drawback in such an approach is that dynamically adding sources is an expensive task. In fact, new TSIMMIS sources must first be wrapped and the mediator rules have to be redefined to take into account new knowledge and their MSL definitions recompiled. The administrator of the system must figure out whether and how the new sources have to be assimilated in the mediator. In our case, this information is automatically discovered by means of clustering.

MOMIS is based on a mediator architecture and follows the ‘semantic approach’. Following the classification of integration systems proposed by Hull [Hul97], MOMIS is in the line of the “virtual approach” and is in the category of “read-only views”; that is, it is a system whose task is to support an integrated, read-only, view of data stored in multiple sources. The most similar projects are GARLIC, SIMS [AKH96], Information Manifold [LRO96] and Infomaster [GKD97].

The GARLIC project [CHS94] builds up on a complex wrapper architecture to describe the local sources with an OO language (GDL), and on the definition of Garlic Complex Objects to manually unify the local sources to define a global schema.

The SIMS project [AKH96] proposes to create a global schema definition by exploiting the use of Description Logics (i.e., the LOOM language) for describing information sources. The use of a global schema allows both GARLIC and SIMS projects to support every possible user queries on the schema instead of a predefined subset of them.

Information Manifold system [LRO96], as the MOMIS project, provides a source independent and query independent mediator. The input schema of Information Manifold is a set of descriptions of the sources. Given a query, the system will create a plan for answering the query using the underlying source descriptions. Algorithms to decide the useful information sources and to generate the query plan have been implemented. The integrated schema is defined mainly manually by the designer, while in our approach it is tool-supported.

Infomaster [GKD97] provides integrated access to multiple distributed heterogeneous information sources giving the illusion of a centralized, homogeneous information system. It is based on a global schema, completely modeled by the user, and a core system that dynamically determines an efficient plan to answer the user’s queries by using translation rules to harmonise possible heterogeneities across the sources.

An approach based on Description Logics and ontologies is taken in the OBSERVER
system to support semantic interoperability and formulation of rich queries over distributed information repositories where different vocabularies are used [MKS96]. Here the idea is that each repository has its own ontology. Inter-ontology relationships are specified in a declarative way (using Description Logics) in an inter-ontology manager module to handle vocabulary heterogeneities between ontologies of different information repositories for query processing. In this respect, our Common Thesaurus plays a similar role in that we specify inter-source terminological relationships. The focus here is more on representation of inter-ontology relationships to solve vocabulary problems at the intensional and extensional level for query processing rather than on using these relationships for deriving an integrated virtual view of the underlying information sources. Moreover, in our approach, we try to extract as much information as possible from source descriptions and from WordNet and we show how this information can be used for affinity evaluation and integration purposes.

The analysis, discovery, and representation of inter-schema properties is another critical aspect of the integration process and research proposals have appeared on this topic. In [PSU98a], semi-automatic techniques for discovering synonyms, homonyms and object inclusion relationships from database schemas are described and a semi-automatic algorithm for integrating and abstracting database schemes is presented in [PSU98b]. It is worth noticing that the design of systems for information gathering from multiple sources is also addressed in Artificial Intelligence through multi-agent systems [LHK98], concentrating mainly on high level tasks (e.g., o-operation, planning, belief revision) related to the extraction process [Dra97].

Object fusion

In this section, we will briefly discuss related works on Object fusion problem.

An interesting work on Object Fusion problem present in literature is [HST99] where a classification of heterogeneity in schema integration is presented. An architecture for schema mapping is also included. Object Fusion is solved by powerful rules stated by the BRITTY mapping language able to express conversion function and SQL where clauses.

One other interesting approach to the Object Fusion is the one implemented in the TSIMMIS mediator system [PAGM96, ea95]. In TSIMMIS it is assumed minimal knowledge of the structure and contents of the sources and have been developed optimisation techniques for data access to the fused objects. To represent such data, they use a schema-less object-oriented model, called Object Exchange Model (OEM). The approach to object fusion is based on semantic object identifiers specified by a set of declarative, logic rules. Each rule maps objects at a source that pertain to some identifiable real world entity, into a virtual object at the mediator. The virtual object is assigned a semantically meaningful object identifier. Mediator objects that have the same object-id are then fused together.

Semistructured data

The issue of modelling semistructured data has been investigated in the recent literature. In particular, a survey of problems concerning semistructured data modelling and querying is presented in [Bun97]. Two similar models for semistructured data have been proposed [BDHS96, PGMW95], based on rooted, labelled graph with the objects
as nodes and labels on edges. According to the model presented in [BDHS96], information resides at labels only, while according to the “Object Exchange Model” (OEM) proposed by Papakonstantinou et. al. in [PGMW95], information also resides at nodes. Very similar proposals for modeling semi-structured data come from the Artificial Intelligence area [CGL98], where in analogy with our approach, a Description Logic is adopted. The issue of adding structure to semistructured data, which is more directly concerned with our concept of object pattern, has also been investigated. In particular, in [GW97], the notion of *dataguide* has been proposed as a “loose description of the structure of the data” actually stored in an information source. A proposal to infer structure in semistructured data has been presented in [NAM98], where the authors use a graph-based data model derived from [BDHS96, PGMW95].

In [Bun97], a new notion of a graph schema appropriate for rooted, labelled graph databases has been proposed. The main usages of the structure extracted from a semistructured source have been presented for query optimization. In fact, the existence of a path in the structure simplifies query evaluation by limiting the query only to data that are relevant. In this paper, we are more concerned with usage of the structure in form of object patterns to support the integration of semistructured sources with structured databases. More recently, XML [Bos97] has emerged in the framework of information representation over the Web allowing the designer to explicitly point out the semantic role of data within a source. Here, the notion of Document Type Definition (DTD) is introduced to model the structure of a set of documents. DTDs play a role similar to our object patterns and can be directly used to derive the ODL description of the information associated with them.

**Original contributions of MOMIS w.r.t. previous works**

The original contribution of the work presented in this chapter is related to the availability of a set of techniques for the designer to face common problems that arise when integrating pre-existing information source, containing both semistructured and structured data.

The idea of combining reasoning capabilities of Description Logics with affinity-based clustering techniques is new and allows both the validation of the inter-source knowledge used for the integration and the identification of candidates to integration in a way automated as much as possible.

The interactive exploitation of WordNet from within our tools combined with subsequent affinity analysis is also a novel capability for an integration tool. In this way, we can take into account and interactively revise semantic knowledge related to the meaning of names in the considered source description, as well as the structure of classes in source schema descriptions and their level of matching to come up with as much information as possible for the extraction of global integrated classes.

Furthermore, we provide the capability of explicitly introducing many kinds of knowledge such as: integrity constraints, extensional relationships and to check the global consistency of implicit and designer provided knowledge.

### 2.7 OLCD : Interpretations and database instances

We assume the union of the integers, the strings, the booleans, and the reals as the set $D$ of base values. To build complex values, we further assume a countable, set disjoint
interface Fast_Food
(s source semistructured ED)
{ attribute string name;
attribute Address address;
attribute integer phone?;
attribute set<string> specialty;
attribute string category;
attribute Fast_Food nearby?;
attribute integer midprice?;
attribute Owner owner?;};

interface Address
(s source semistructured ED)
{ attribute string city;
attribute string street;
attribute string zipcode;
}; union {
string;
};

interface Owner (source semistructured ED)
{ attribute string name;
attribute Address address;
attribute string job;};

from $D$, of object identifiers (denoted by $o, o_1, o_2, \ldots$). The set $V$ of all values over $O$

is defined as the smallest set containing $D$ and $O$, such that, if $v_1, \ldots, v_p$ are values, then the set \{ $v_1, \ldots, v_p$ \} is a value, and a partial function $t: A \rightarrow \{ v_1, \ldots, v_p \}$ is a value. The function $t$ is the usual tuple value; the standard notation $[a_1: v_1, \ldots, a_p: v_p]$ will be henceforth used. Let $\in, \ne, <, \ge, \le$ be the equality, inequality and total order relations, denoted by $\theta$, defined as usual on $D$. Equality and inequality can be extended from $D$ to all $V$: the equality operator ($\equiv$) has the meaning of identity, i.e., two objects are equal if they have the same identifier, two sets are equal iff they have equal elements, two tuples, say $v_a = [a_1: v_1, \ldots, a_p: v_p]$, $v_b = [a_1': v_1', \ldots, a_p': v_p']$, are equal if they have the same attributes and equal attribute labels are mapped to equal values. Object identifiers are assigned values by a total value function $\delta$ from $O$ to $V$. Let $W$ denote the set of all paths. Given a set of object identifiers $O$ and a value function $\delta$, let $J: W \rightarrow 2^{V \times V}$ a function defined as follows:

- empty path: $J[e] = \{(v, v) \in V \times V\}$
- single element path:
  \[
  J[a] = \{(v_1, v_2) \in V \times V \mid v_1 = [\ldots, a: v_2, \ldots]\}
  \]
  \[
  J[\Delta] = \{(o, v) \in O \times V \mid \delta(o) = v\}
  \]
- multiple element path: $J[e_1, e_2, \ldots, e_n] = J[e_1] \circ J[e_2] \circ \cdots \circ J[e_n]$
  where $\circ$ is the symbol of function composition.
Notice that, for all $d$, $\hat{\hat{d}}$ is undefined on set values. Let $v$ be a value and $d$ be a path. By $\mathcal{J}[p](v)$ we mean the unique value (when it exists) reachable from $v$ following $p$, that is the value of the partial function $\mathcal{J}[p]$ in $v$. Let $\mathcal{I}_B$ be the (fixed) standard interpretation function from $B$ to $2^D$. For a given object assignment $\delta$, each type expression $S$ is mapped to a set of values (its interpretation). An interpretation function is a function $\mathcal{I}$ from $S$ to $2^\nu$ satisfying the following equations:

\[
\begin{align*}
\mathcal{I}[\top] &= \nu \\
\mathcal{I}[\bot] &= \emptyset \\
\mathcal{I}[B] &= \mathcal{I}_B[B] \\
\mathcal{I}[\{S\}_\nu] &= \{ M \mid M \subseteq \mathcal{I}[S] \} \\
\mathcal{I}[\{S\}_\exists] &= \{ M \mid M \cap \mathcal{I}[S] \neq \emptyset \} \\
\mathcal{I}[\{a_1 : S_1, \ldots, a_p : S_p\}] &= \{ t : A \rightarrow \nu \mid t(a_i) \in \mathcal{I}[S_i], 1 \leq i \leq p \} \\
\mathcal{I}[\{S_1 \cap S_2\}] &= \mathcal{I}[S_1] \cap \mathcal{I}[S_2] \\
\mathcal{I}[\{S_1 \cup S_2\}] &= \mathcal{I}[S_1] \cup \mathcal{I}[S_2] \\
\mathcal{I}[^{-}S] &= \nu \setminus \mathcal{I}[S] \\
\mathcal{I}[\Delta S] &= \{ o \in O \mid \delta(o) \in \mathcal{I}[S] \} \\
\mathcal{I}[(p@d)] &= \{ v \in \nu \mid \mathcal{J}[p](v) \} \\
\mathcal{I}[(p\#)] &= \{ v \in \nu \mid v \notin \text{dom}\mathcal{J}[p]\}
\end{align*}
\]

Note that the interpretation of tuples implies an open world semantics for tuple types similar to the one adopted by Cardelli [Car84], and that $(p\#)$ selects objects which do not have the path $p$.

It should be noted that an interpretation does not necessarily imply that the extension of a named type is identical to the type description associated with the type name via the schema $\sigma$. For this purpose, we have to further constrain the interpretation function: An interpretation function $\mathcal{I}$ is a legal instance of a schema $\sigma$ iff the set $O$ is finite, and for all $N \in \mathcal{N}$:

- $\mathcal{I}[N] \subseteq \mathcal{I}[^{\sigma_p}(N)]$ if $N \in \text{dom}\sigma_p$ From the above definition, we see that the interpretation of a primitive type name is included in the interpretation of its description, while the interpretation of a virtual type is the interpretation of its description. In other words, the interpretation of a primitive type name has to be provided by the user, according to the given description, while the interpretation of a virtual type name is drawn from its definition and from the interpretation of primitive type names, thus corresponding to a view in database context. Given a type $S$ of a schema $\sigma$, we say that $S$ is consistent if and only if there is a legal instance $\mathcal{I}$ of $\sigma$ such that $\mathcal{I}[S] \neq \emptyset$. Given two types $S_1$, $S_2$ of a schema $\sigma$, we say that $S_1$ subsumes $S_2$ iff $\mathcal{I}[S_1] \supseteq \mathcal{I}[S_2]$ for all legal instances $\mathcal{I}$ of $\sigma$. 


interface Restaurant
{
  source relational FD
  key (r_code)
  foreign_key(pers_id) references Person

  attribute string r_code;
  attribute string name;
  attribute string street;
  attribute string zip_code;
  attribute integer pers_id;
  attribute string special_dish;
  attribute integer category;
  attribute integer tourist_menu_price;
};

interface Person
{
  source relational FD
  key (pers_id)

  attribute integer pers_id;
  attribute string first_name;
  attribute string last_name;
  attribute integer qualification;
};

interface Bistro
{
  source relational FD
  key (r_code)
  foreign_key (r_code) references Restaurant
  foreign_key (pers_id) references Person

  attribute string r_code;
  attribute set<string> type;
  attribute integer pers_id;
};

interface Brasserie
{
  source relational FD
  key (b_code)

  attribute string b_code;
  attribute string name;
  attribute string address;
};

Figure 2.18: Example source Food_Guide_Source (FD)
Chapter 3

The MOMIS Prototype: Architecture and Implementation

The MOMIS prototype is a software of about 100 thousands lines of Java code and was (and will be) developed by several people. To develop a coherent system, reusing code as much as possible, we needed an effective and clean architecture and a set of small tools (like makefiles or start script) to manage the system.

In this chapter, the MOMIS architecture and software organization are described. During my Ph.D. studies, I set up most of the architecture described in this chapter, both software (organization of software development directories and tools) and organizational (a website for the development coordination).

3.1 Introduction

The MOMIS system is designed for schema integration. Figure 3.1 shows the architecture of the system in terms of functional modules. The system is composed by several objects that communicate using the CORBA [Groa] standard.

The object are:

**Wrappers** Each data source (relational, object, XML, ...) must be presented to MOMIS in a standard way, and this is what the wrappers do. Each object represents a data structure (see section 3.5 for a detailed description).

**SI-Designer** SI-Designer is the GUI (Graphic User Interface) for the MOMIS Global Schema Builder. Its goal is to lead designer from the schemata acquisition of sources to the tuning of the mapping table through the various steps of the integration. This is the client object that uses other server objects (see chapter 4 for a detailed description).

---

1 We describe many details; this is written to be a quick reference for people who will be part of the MOMIS development team.
44 The MOMIS Prototype: Architecture and Implementation

WordNet This is the server whose object provides the access to the WordNet, and that is able to extract lexicon derived intensional relationships between attributes and classes, exploiting the lexical relationships contained in the WordNet lexical system (see section 3.6 for a detailed description).

ODB-Tools This object provides access to the ODB-Tools prototype. ODB-Tool is a framework for object-oriented database (OODB) schema validation, preserving taxonomy coherence and performing taxonomic inference, and semantic query optimization (see section 3.7 for a detailed description).

Global Schema A Global Schema object contains all information for querying the resulting global schema by a queries manager object. Such object is produced by the SI-Designer GUI and contains the sources schemata, the thesaurus, and the mapping table (see section 3.8 for a detailed description).

Query Manager A Global Schema object uses information produced during the integration steps to query a global schema. It is able to divide a query on the global schema in a set of query for the local sources and to merge results (for further information see the Silvia Zanni’s thesis [Zan00] in Italian).

This chapter describes module implementation and all the organizational infrastructures to support the software development.

3.2 Overview of organizational infrastructures

In developing the MOMIS prototype we had to spread development work between several developers, assigning to each person the responsibility of a part of the prototype.
3.3 Overview of software organization

To coordinate the development team we periodically have group meetings and set up a development private (password protected) web site and one mailing list. The development site is the official media for group communication. All meetings, news and choices are reported through the site.

Each group member can modify both the source code directory and the development site directory, has the responsibility to develop part of the MOMIS software and maintain updated the corresponding documentation. Such documentation is related both to theoretical aspects and implementation architecture choices.

It is mandatory for each group member to present his work at meetings using documents written in HTML stored on the development web site. This guarantees that the development site content is constantly updated.

Other information present on the coordination site includes:

- *The group agenda*, document that contains a synthesis of all group meetings.
- *The to-do document*, containing constantly updated future work divided by Theoretical aspects, Product re-engineering and Implementation aspects.
- *The conventions document*, describing guidelines and standards shared by the group members.
- *The how-to document*, a few practical how-to to solve most common problems.
- *The know bugs document*.

Nowadays we have a 10 Mega bytes of on-line hand made documentation on modules. Moreover, other services provided through the development site are the access to all manuals (such as CORBA specification, and Java API and tutorial) and MOMIS prototype source code documentation generated by javadoc (roughly 7 Mega bytes).

We found meetings, mailing list and development web site very useful tools for knowledge sharing and coordination.

3.3 Overview of software organization

Software development directories are divided in different packages; this allows several people to work together on the prototype development. The development documentation is a collection of HTML pages where the documentation structure is cloned by the software structure.

Each development member is responsible for its assigned project module and should maintain aligned both software and documentation.

Following a macro distinction between the MOMIS prototype, components can be divided into the following directories:

- *shared components* Contains code (libraries) that is shared by all MOMIS modules like the package odli3, the parser for ODL, and several tools classes.
- *server* Contains the implementation of CORBA server objects, main momis servant and factory objects. These are the main interface for accessing the MOMIS distributed integration services. Basically it contains the implementation for the Momis Factory, the Global Schema and the Query Manager servers.
- *modules* Every code that is not a server or a shared library, every application that is related or developed for the MOMIS project is stored in this directory.
Before examining the content of each one of the cited directories, let me say a few words on the other service directories and files present in the root directory of the MOMIS prototype.

Prototype directories structure

- **applications** contains all demonstration application developed for or using the MOMIS prototype.
- **declarationsIDL** Contains the IDL CORBA declaration for all the MOMIS CORBA interfaces.
- **doc** contains source code documentation produced by JavaDoc for the whole MOMIS prototype code.
- **lib** contains Java libraries used by several modules. It contains, for example, the libraries for XML parsing and management.
- **utilities** contains miscellaneous Java tools.
- **var** contains log and Pid (Process identifier) files for the running MOMIS server objects (such as wrappers or WordNet interface).
- **bat** contains all script to start the MOMIS prototype on MS Windows systems.

Important files

- **setVars** Script (for both csh and bash) for setting environment variable for the MOMIS prototype.
- **momis.conf** is the configuration file for all the MOMIS server objects. Each server, starting, reads configuration from this file.
- **startMomis** is a bash script to start and stop

General conventions for directories  In the important directory there are special proposal files or directories with the following functions:

- **Onote.txt:** file used to add development notes and documentation for the current dir. Usually contains a brief description of what the directory contains, instructions for using the software implemented in the directory, and a (optional) file by file description of the directory content.
  
  I wrote a small perl program called ldir, to help to manage the Onote.txt file.

- **Makefile:** this file describes how to do the following actions: compile the code contained in the current directory from source code, clean the directory from compiled classes and all useless files generated during development, and generate the documentation for the code contained in the current directory in the directory doc.

- **doc:** this dir contains the documentation generated from the source code.

The rest of this section will be devoted to the most important directories and files.

The starting script **startMomis**

As shown in Figure 3.2, the script startMomis in the momis prototype development main directory is able to start or stop all MOMIS services or to start or stop the single service, including the most commonly used wrappers.
Start or stop MOMIS components

Syntax: 

dir ./startMomis {start|stop} [option]

where "option" are:

n name server  
m MOMIS factory server  
o ODB-Tools  
w CORBA_WordNet  
wd1 Wrapper dummy Univers  
wd2 Wrapper dummy University

Example:

to start ODB-Tools use:  

dir ./startMomis start o

to start the whole MOMIS:  

dir ./startMomis start

to stop the whole MOMIS:  

dir ./startMomis stop

Figure 3.2: The hints from the startMomis script

This script maintains the process number and the log file for each managed server object generating files with extensions respectively .pid and .log in the directory var, the file name without extension identifies the server.

The pid file is created each time a server is started and is removed when the server is stopped and contains the Pid to kill to stop the server.

Starting MOMIS under Windows

We brought the MOMIS prototype on the MS Windows platform. We did not have to recompile a single file, but we had to write a set of scripts to start servers in Windows. Such scripts are the following:

- setVars.bat called from the other scripts, sets the common environment variables
- startMomis_naming.bat Starts naming server
- startMomis_services.bat Starts basic Momis services, MomisFactory and WordNetServer
- SIDesigner.bat Start a new session of SI-Designer

The configuration file momis.conf

In figure 3.3 the momis.conf file currently used is shown. With this file we can specify several configuration parameters such as the CORBA naming server address, the server time out period, where to write temporary files and where are located ODB-Tools commands in the system.

The modules component directory

This directory contains the modules used by the MOMIS prototype such as:
# Configuration file for the MOMIS prototype

# orbPort=1050
orbServer=sparc20.dsi.unimo.it
# Time out in seconds
timeOutGracePeriod=28800
timeOutCheckPeriod=3600

# OdbTools parameters

# name for the registration in the naming server
odbt_namingName=odbt
# dir where write the tmp files
odbt_dirTmp=/tmp/
# command used to limit the system resources at each run
odbt_limitaCommand=/export/home/progetti.comuni/Momis/prototype/modules/ODBTools/_sorgentiC/odbtools/limita
odbt_odlTrasl=/export/home/progetti.comuni/Momis/prototype/modules/ODBTools/_sorgentiC/odbtools/odl_trasl
odbt_olcdDesigner=/export/home/progetti.comuni/Momis/prototype/modules/ODBTools/_sorgentiC/odbtools/ocdl-designer
odbt_odbqo=/export/home/progetti.comuni/Momis/prototype/modules/ODBTools/_sorgentiC/odbtools/odbqo

# Global Schema parameters

# next: Momis Extensional knowledge management
next_namingName=nextServer

---

Figure 3.3: MOMIS main configuration file momis.conf

- **ODBTools**: This module implements a CORBA interface of the Description logics based Object Schema optimization tool.
- **SIDesigner**: Main MOMIS Graphical User Interface. This module leads the Designer in integration from local schema acquisition to the definition of the mapping tables.
- **WordNet**: Implementation of the CORBA interface for the WordNet ontology. The server directly accesses the database files of WordNet and implements Thesaurus relations extraction capabilities.
- **wrapperAccess**: This is a MOMIS Wrapper for MS Access.
- **wrapperClient**: MOMIS simple MOMIS Wrapper client. Allows access to all wrappers functionalities through a command line text interface.
- **wrapperDummy**: MOMIS Wrapper for files, the only action is to return as description the content of a given file.
- **wrapperJDBC**: MOMIS Wrapper for JDBC data sources.
- **wrapperXMLg**: MOMIS Wrapper for XML documents.
3.4 The shared component directory

This directory contains all java code shared by all servers and MOMIS modules. This directory must be included in the CLASSPATH to run almost all MOMIS tools. This directory contains the following files:

- **Parser.java**: generated from odli3.y by byacc/j [Jam], is the parser for the ODL language. From an ODL stream it generates in memory object representation using the classes of the odli3 package.
- **GSStatus.java**: nowadays is a serializable object that stores all java objects produced during the MOMIS integration phase using SI-Designer.
- **GlobalSchemaProxy.java**: this class supplies facilities to access, serialize, save and load a GSStatus object including the communication with GlobalSchema CORBA objects.

This directory contains packages that implement Java Stubs and Skeleton for the CORBA interfaces generated using idlj command starting from the IDL definition in the declarationsIDL directory.

In addition, the following fundamental packages are implemented:

- **odli3**: Contains classes to manage the ODL language
- **globalschema**: Contains classes to manage integration steps like mapping table or join maps.
- **tools**: Contains several classes that implements useful features like the management of time out or read a configuration file.

For further information about classes and implementation see the documentation generated by javadoc, it can be request from the webmaster of the MOMIS web site [Grob].

3.4.1 The ODL classes and the Parser

During my thesis I modified the parser and some data structure in the ODL package created by Alberto Zanoli (see [Zan99]).

This package contains all data structures used for the local classes description.

Figure 3.4 shows the list of the classes in the odli3 package preserving ordered to show the inheritance hierarchy. For example, it contains Java classes that describe types where odli3.Type is the ancestor class and odli3.StructType is a leaf class that inherits from odli3.ConstrType and then from odli3.Type.

Major changes I made to the package are described in the following subsections.

**Introduction of MomisObject**

All the objects of the package odli3 inherit from the MomisObject. This was done to eventually add features shared by all odli3 object.

This feature is used by the SI-Designer to save the status of its component modules. Each component module of SI-Designer saves a single object in the additional info of the current schema using as key a name what is unique for the module.
class java.lang.Object
class odli3.Error (implements java.io.Serializable)
class odli3.MomisObject (implements java.io.Serializable)
class odli3.Attribute (implements java.io.Serializable)
class odli3.GlobalAttribute (implements java.io.Serializable)
class odli3.Relationship (implements java.io.Serializable)
class odli3.SimpleAttribute (implements java.io.Serializable)
class odli3.CandidateKey (implements java.io.Serializable)
class odli3.Case (implements java.io.Serializable)
class odli3.Constant (implements java.io.Serializable)
class odli3.ForeignKey (implements java.io.Serializable)
class odli3.KeyList (implements java.io.Serializable)
class odli3.MappingRule (implements java.io.Serializable)
class odli3.AndList (implements java.io.Serializable)
class odli3.AttributeOnly (implements java.io.Serializable)
class odli3.DefaultValue (implements java.io.Serializable)
class odli3.UnionList (implements java.io.Serializable)
class odli3.Operation (implements java.io.Serializable)
class odli3.OpVar (implements java.io.Serializable)
class odli3.Rule (implements java.io.Serializable)
class odli3.CaseRule (implements java.io.Serializable)
class odli3.ExtRule
class odli3.ForallRule (implements java.io.Serializable)
class odli3.RuleBody (implements java.io.Serializable)
class odli3.Compare (implements java.io.Serializable)
class odli3.ForallExist (implements java.io.Serializable)
class odli3.Exists (implements java.io.Serializable)
class odli3.Forall (implements java.io.Serializable)
class odli3.In (implements java.io.Serializable)
class odli3.RuleOperation (implements java.io.Serializable)
class odli3.RuleOpArg (implements java.io.Serializable)
class odli3.DotNameArg (implements java.io.Serializable)
class odli3.ValueArg (implements java.io.Serializable)
class odli3.StructVar (implements java.io.Serializable)
class odli3.ThesRelation (implements java.io.Serializable)
class odli3.AttributeRel (implements java.io.Serializable)
class odli3.AttrIntRel (implements java.io.Serializable)
class odli3.InterfaceRel (implements java.io.Serializable)
class odli3.Type (implements java.io.Serializable)
class odli3.TypeToSolve (implements java.io.Serializable)
class odli3.ValueType (implements java.io.Serializable)
class odli3.ConstrType (implements java.io.Serializable)
class odli3.EnumType (implements java.io.Serializable)
class odli3.StructType (implements java.io.Serializable)
class odli3.UnionType (implements java.io.Serializable)
class odli3.SimpleType (implements java.io.Serializable)
class odli3.BaseType (implements java.io.Serializable)
class odli3.AnyType (implements java.io.Serializable)
class odli3.BooleanType (implements java.io.Serializable)
class odli3.CharType (implements java.io.Serializable)
class odli3.FloatingType (implements java.io.Serializable)
class odli3.IntegerType (implements java.io.Serializable)
class odli3.OctetType (implements java.io.Serializable)
class odli3.RangeType (implements java.io.Serializable)
class odli3.StringType (implements java.io.Serializable)
class odli3.DefinedType (implements java.io.Serializable)
class odli3.TemplateType (implements java.io.Serializable)
class odli3.ArraySequence (implements java.io.Serializable)
class odli3.BagType (implements java.io.Serializable)
class odli3.ListType (implements java.io.Serializable)
class odli3.SetType (implements java.io.Serializable)
class odli3.TypeContainer (implements java.io.Serializable)
class odli3.IntBody (implements java.io.Serializable)
class odli3.Interface (implements java.io.Serializable)
class odli3.Module (implements java.io.Serializable)
class odli3.Schema (implements java.io.Serializable)
class java.lang.Throwable (implements java.io.Serializable)
class java.lang.Exception
class odli3.OdlException
Figure 3.4: Hierarchy tree of the classes of the odli3 package
3.4 The shared component directory

When the schema that contains the whole integration status object is serialized and saved, the status objects of the component modules are saved too.

Introduction of TypeContainer

To allow a nested definition, we needed a uniform way to store types. Now the data type containers (IntBody, Interface, Module, Schema, Source) inherited types management facilities from this object. In addition, this enables a way to resolve types identified by identifier in a post-processing phase.

Types managed by this object are: constants, types defined through typedef, enums, interfaces, modules, rules, structs, thesaurus relations, types to be solved in a post-processing phase and unions.

Each container also knows his parent, and this allows the inherited data types to be found.

Methods defined for each data type by this Object, for example, for the struct type are:

```java
public StructType getStruct(java.lang.String name)
public void addStruct(StructType c) throws OdlException
public void addStructs(java.lang.Object[] c)
    throws OdlException
public java.lang.Object[] getStructs()
```

The search method `getStruct` performs the recursive search back from parent to parent up to the top element of the schema tree. Similar methods are defined for the other kinds of data managed by the TypeContainer objects.

Types name post parsing resolution

The Parser has been modified to accept recursive definitions and ODL modules.

During parsing all types referenced by identifier (not basic types) or foreign keys are used without any control of existence because the definition could be after that point of elaboration. These types are accepted as TypeToSolve and for each TypeContainer there is trace of all such TypeToSolve.

In post parsing processing, all references left to be resolved must be checked and re-solved. This process is performed by two methods defined in the class TypeContainer:

- `solveUnsolvedForeignKeys`: This method tries to solve unsolved references for: Foreign Keys, Keylist and Thesaurus relations. It also verifies that there are no duplicated Thesaurus relations. If any reference is not correct, an exception OdlException is thrown.

- `solveUnsolved`: Tries to solve types defined by typedef, for each interfaces an attribute list is built since (because of the `union` clause) the same name of attribute could be associated with different attributes (differing in type).

  Since types can be defined in schema, interfaces and modules, these procedures are recursive through the object tree that describes the ODL schema.

  Type solved are: Interfaces, types defined by TypeDef and ExtRule.

  These are recursively solved in the following sub-type-container: Interfaces, Modules, IntBody.
toOdl implementation

This feature exports the content of the odli3 classes into a ODL format. This feature has been implemented defining the toOdl method for each odli3 class. Calling such method on the (top level) Schema object will produce the ODL description of the schema and of all its sub-components like typedefs interfaces, attributes, const, etc. recursively calling the toOdl method for each sub-component. The produced ODL code will reflect the in memory data structure where are allowed nested module definitions.

toOlcd implementation

This features exports the content of the odli3 classes into a olcd format ready to be used by ODB-Tool for schema validation. Olcd is the input data format recognized by ODB-Tools. Since ODL is richer than Olcd, information is lost during the conversion and the process is not reversible.

This feature has been implemented defining the toOlcd method for (almost) each class in the package odli3. The method invocation on high level objects, causes the recursive call of the method on every sub-component.

Tables 3.1, 3.2 and 3.3 shows how this function maps the types. Figure 3.5 shows an example of mapping, since in olcd yet the concept of union or or does not exist. All attributes of all InterfaceBodies of an interface are mapped as attributes in olcd. Attributes with the same name are duplicated using unique names.

```plaintext
interface Course
{ source object Univers
  extent Courses
  key (course_name) }
{ attribute string course_name;
  attribute Professor taught_by; }
union Course_1
{ attribute string course_name;
  attribute string course_description;
  attribute Professor taught_by; }

is mapped as:

prim Course = "[ 
  course_name : string ,
  taught_by : Professor ,
  course_desc : string ,
  course_name0 : string ,
  taught_by0 : Professor ] ;
```

Figure 3.5: Example of /odli3/ to olcd mapping

toOlcdSimB implementation

toOlcdSimB is a feature very similar to toOlcd (see section 3.4.1 on page 52) but is used to implement Thesaurus Relationships validation and inference (by the module SIMb).

This method will create a virtual schema (see section 2.2.6 on page 21 for more information) described in OLCD to be passed to ODB-Tools for Thesaurus Relationships
3.4 The shared component directory

<table>
<thead>
<tr>
<th>ODL</th>
<th>OLCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>top</td>
</tr>
<tr>
<td>boolean</td>
<td>bool</td>
</tr>
<tr>
<td>char</td>
<td>string</td>
</tr>
<tr>
<td>double</td>
<td>real</td>
</tr>
<tr>
<td>float</td>
<td>real</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>long</td>
<td>integer</td>
</tr>
<tr>
<td>short</td>
<td>integer</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>unsigned short</td>
<td>integer</td>
</tr>
<tr>
<td>unsigned long</td>
<td>integer</td>
</tr>
</tbody>
</table>

Table 3.1: Mapping between ODL₁₃ and olcd simple data types

<table>
<thead>
<tr>
<th>ODL</th>
<th>OLCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>type []</td>
<td>&lt; type &gt; (list of)</td>
</tr>
<tr>
<td>list</td>
<td>&lt; &gt; (list of)</td>
</tr>
<tr>
<td>bag</td>
<td>{ } (set of)</td>
</tr>
<tr>
<td>set</td>
<td>{ } (set of)</td>
</tr>
</tbody>
</table>

Table 3.2: Mapping between ODL₁₃ and olcd of collections

validation and inference.

A main characteristic of this feature is that Thesaurus Relationships contribute in Interface definition.

**NT** lexical relationships cause the Interface to inherit from the Broader Interface.

**RT** interface cause the definition of dummy attributes that links to the related Interface.

**SYN** relationships cause the Interface to be grouped in a set of Synonym Interface. These interfaces share the same definition in terms of Parent Interfaces and attributes. The parent Interfaces of each of the synonym Interfaces is given by the union of the Interface each Interface inherits from or takes part in a NT or BT relationship. The set of attribute is given by the union of the attributes each interface inherits from. RT relationship that give rise to the RT dummy attributes is given by the union of the RT relationship that involves at least one Interface in the SYN group.

**Introduction of ThesaurusByObject**

This class speeds up relationships retrieving from a Schema object. All the ThesRelation objects are organized in Map objects where keys are the element as ODL₁₃ Objects of the ThesRelation objects.

This object is constructed over a schema and indexes all thesaurus relations contained by such schema giving a set of method for fast access to set of relations.

To index relations a HashMap is used, which allows information (objects) to be related to a given key object, thus providing a fast way to retrieve such information given the key object.
This object provides methods like:

- **Set getRelations(java.lang.Object element)** that returns a set of ThesRelations related to a given element.
- **Set getRelations(java.lang.Object element1, java.lang.Object element2)** that returns a set of ThesRelations between the given elements.
- **Set getSynRelations(java.lang.Object element)** Returns all the synonymy relation for a given Element
- **Set getInterfaceParentRelations(Interface element)** Given an interface, returns all InterfaceRel relations where the given Interface is narrow term NT to another Interface. Note that MOMIS stores all InterfaceRel as NT relations (broad term - BT relations are transformed in NT by the constructors). So it is enough to look for the relations where the given interface appears as FIRST element.

And so on. All interesting and used queries at the thesaurus are coded in this object and are usable by all MOMIS components.

### 3.5 The MOMIS wrappers

The wrappers in **MOMIS** are the access point for the data sources. Each data source (relational, object, XML, ...) must be presented to the **MOMIS** system in a standard way, and this is what the wrapper does. A wrapper is a CORBA object that is connected to a source and is able to describe the source structure using the ODL language and supplies a way to query to source using the OQL language.

This section shows the interface for accessing wrapper features and an overview of the wrappers I implemented for my Ph.d. thesis.

#### 3.5.1 The IDL interface for a wrapper

Figure 3.6 shows the CORBA interface implemented by the Wrapper Objects. The functions available are:

- **get Type** returns the type of the wrapper.
- **getSourceName** returns the name of the data source. The returned name is only the name the designer of the wrapper assigned to the data source, it is not a universal name.
3.5 The MOMIS wrappers

interface Wrapper {
    string getType() raises (momisOqlException);
    string getDescription() raises (momisOqlException);
    MomisResultSet runQuery(in string oql)
                                                     raises (momisOqlException);
    string getSourceName() raises (momisOqlException);
};

Figure 3.6: The MOMIS Wrapper IDL interface

description This function asks the source the meta-information about the schema
and produces a ODL description of the source schema and returns it to the CORBA
client as a string.

runQuery This function causes the source to evaluate a OQL query and generates a
MomisResultSet CORBA object that (like a cursor) allow to access to the resulting
data set.

interface MomisResultSet {
    long getColumnCount() raises (momisOqlException);
    string getColumnName(in long column) raises (momisOqlException);
    long getColumnType(in long column) raises (momisOqlException);
    boolean next() raises (momisOqlException);
    void close() raises (momisOqlException);
    long getColumnByName(in string column) raises (momisOqlException);
    long getLong(in long column) raises (momisOqlException);
    char getChar(in long column) raises (momisOqlException);
    double getDouble(in long column) raises (momisOqlException);
    float getFloat(in long column) raises (momisOqlException);
    string getString(in long column) raises (momisOqlException);
    string stringSet();
    const long TYPE_Unsupported = -1; /* Unsupported Type */
    const long TYPE_Long = 1;
    const long TYPE_Char = 2;
    const long TYPE_Double = 3;
    const long TYPE_Float = 4;
    const long TYPE_String = 5;
};

Figure 3.7: The MomisResultset IDL interface

Figure 3.7 shows the MomisResultset CORBA interface. It is a very simple cursor
implementation. We can also see that MOMIS data types recognized through wrappers
is very poor related to the data types supported by ODL.

3.5.2 The dummy wrapper

The dummy wrapper is the simplest wrapper we can implement. It is not associated
with any data source and, it reads the schema description directly from a file containing
a ODL description. Obviously it is not able to reply to query requests.
This wrapper is parametrized to select in which naming server register itself as CORBA
object. Other parameters are also the registration name and the name of the file containing the ODL source description.

3.5.3 The JDBC wrapper

This is quite a real wrapper which means that it has both source introspection and source querying capabilities. Both these features are provided through standard JDBC calls. This ensures that this wrapper can be used for all data management systems that support JDBC.

On starting the wrapper it reads a configuration file that contains parameters for the JDBC connection like JDBC driver to use and the connection string, as well as other CORBA parameters that specify where is the naming server and the name to use for registering in the naming server.

The main object is a factory object because it generates a new MomisResultSet object each time the method runQuery is called. In the configuration file parameters are also specified for managing time-out of unused MomisResultSet objects.

The introspection of the source is performed using the ResultSetMetaData objects. Using JDBC-1 does not make certain information available like foreign or primary key definitions or constraints, so the description returned by this wrapper contains only the list of tables that are mapped into /odli3/ classes and, for each class, a list of typed attributes.

The querying feature is implemented without applying any filter to the /oqli3/ query and is sent as-is to the underlying data source management system. This implies that the wrapper works fine only in cases of extremely simple queries where /oqli3/ queries are SQL queries. This is not a strong assumption because the /momis/ Query manager is also able to formulate only basic (very simple) queries towards local data sources.

3.5.4 Other wrappers

During the project development other wrappers were developed and others are in developing phase. Among the developed ones we should cite the XML wrapper, see [Gue00], which does not have the querying capabilities yet because the XML-QL is not yet a stable standard.

Other wrappers in developing are the wrapper based on JDBC 2.0 that will make it possible to obtain richer description of the source or ad-hoc wrapper like wrappers for DBMS like MS Access, IBM DB2 and Oracle.

3.6 The MOMIS WordNet interface

WordNet is a lexical database introduced in section 2.1.5 on page 15.

The MOMIS WordNet module was developed by Giovanni Malvezzi [Mal00] and implements a technique to find intensional relations inter-schema. The goal is to discover affinities/similitude between classes from different data sources. The modules extract lexical relations between schema element names (classes and attributes are the elements) and works on the meaning of the names used to describe the classes and attributes content.
This module provides access to the WordNet ontology by accessing the database files directly, it has been designed for use by the SLIM module of the SI-Designer GUI. It is possible to extract from WordNet the following types of lexical relations:

- **Synonymy**, **Hypernymy**, **Hyponymy**, **Holonomy**, **Meronomy**, **Correlation**

Where Hyponymy and Meronomy are respectively inverse relations of Hypernymy and Holonomy.

The relations from WordNet are proposed as semantic relations to be added to the Common Thesaurus according to the following mapping:

- **Synonymy**: corresponds to a SYN relation.
- **Hypernymy**: corresponds to a BT relation.
- **Holonomy**: corresponds to a RT relation.
- **Correlation**: corresponds to a RT relation.

The Common Thesaurus must contain only semantic relations, so the WordNet module only proposes relations to the designer, who must validate them manually to promote lexical relations to semantic relations.

The modules implement an algorithm that, given the schema elements annotated (manually by the designer through a GUI) with the right meaning, produces all semantic relations present in the ontology.

Starting from the schema to be integrated, the designer must declare a relation between each schema elements names (class names or attribute names) and the WordNet meaning used in the context. That is, given a name, the designer must choose one or more of the name meanings.

This is a two step process.

1. **Word form choice.** In this step, the WordNet morphologic processor should aid the designer. A word form is the word without any suffixes due to declination of conjugation.

   For example, in Figure 3.8, by selecting the address attribute the morphologic processor returns the base form address.

   If such word form is not found or there is ambiguity, or it is not satisfactory, the designer can set a custom word form.

2. **Sense choice.** The designer can choose to set the mapping of an element with zero, one or more senses. For example, in figure 3.8 WordNet has 15 meanings for the word form address from which the most appropriate ones are chosen.

Figure 3.9 shows the interface to access the MOMIS WordNet service. A short description of the available methods follows:

---

2. Generalization relation
3. Specialization relation
4. Aggregation relation, part
5. Aggregation relation, all
6. The correlation is a relation between two terms in two synonym sets that shares the same father in hypernymy sense.
7. E.g. axe has 3 word forms: ax (1 sense), axi (5 senses), axe (2 senses).
module WordNetApp {
  interface CORBA_WordNet {
    string trovaSensi( in string word);
    string trovaParenti(in string formBase,
                        in long synset_offset, in long pos);
    string creaThes( in string namePathTaged);
    string trovaHashObject( in string word);
    long killObject();
  };
  interface WordNetFactory {
    CORBA_WordNet newCORBA_WordNet();
  };
}

• trovaSensi means search for meanings. When given a word, it runs the morphological engine to find the word form. It returns a string that contains the serialization or a Java Vector of objects (wn2slim.Meaning) that represents WordNet meanings.

• trovaParenti means search for Parents. Given a word in a normalized form, the reference position of the synset and number that identifies the type of WordNet element (noun, verb, adjective, adverb), it calculates the sense’s expansion to his near senses. It returns a Vector of serialized meanings.

• creaThes means create thesaurus. When given a string that contains the serialized form of a wn2slim.NodeTag vector, it returns a set of lexical relations to be added to the thesaurus. This set is given by a string that contains the serialized form of wn2slim.ThesaurusEntry vector.

• trovaHashObject searches for hypernymy tree for the noun syntax category. When given a word form, it returns all its hypernyms.

Because of the complexity of the data structure used in the WordNet interaction, the
3.7 The MOMIS ODB-Tools interface

This section describes the module that allows access to the ODB-Tools [BBSV97b, BBSV97c] services through CORBA.

ODB-Tools is an integrated environment for object-oriented database (OODB) schemata validation and semantic query optimization, based on a description logic kernel [BN94], preserving taxonomy coherence and performing taxonomic inferences.

```java
module OdbToolsApplic {
    interface OdbTools {
        string translate_Odl_Olcd ( in string odl, out string olcd );
        string translate_Odl_Olcd_vf ( in string odl, out string olcd, out string vf );
        string validate_Odl ( in string odl, out string fc );
        string validate_Odl_XML ( in string odl, out string xmlf );
        string validate_OdISS ( out string fc );
        string validate_OdISS_vf ( out string fc, out string vf );
        string optimize_Oql ( in string odl, in string oql, out string stdOut );
        string optimize_OqlISS ( in string oql );
        string optimize_OqlISS_vf ( in string oql, out string vf );
        long killObject();
    }
    interface OdbToolsFactory {
        OdbTools newServant ( in string description );
    }
};
```

Figure 3.10: The ODB-Tools IDL interface

We developed a CORBA application to use ODB-TOOLS services through CORBA. Such application is made of a server side and a client side. The server part provides standard CORBA services accessible by CORBA clients.

To access the server we wrote two client applications, a textual client and a graphical client. The Textual client is simple and can be invoked from a command line or in batch scripts. The Graphical client has a user friendly interface (figure 3.11) to access ODB-Tools services. It can be executed as a Java application or applet, and contains ScVisualTool tool to visualize optimizations graphically.

Moreover, in order to access ODB-Tools from another CORBA application, the only things to know are (1) the idl interface and (2) the address of the computer on which CORBA objects resides (at present, sparc20.dsi.unimo.it on port 1050).

The server side services are described by the IDL description shown in figure 3.10 and...
correspond to the following ODB-Tools services:

**OdbTool** describes the *servant* object supplying ODB-Tools services invoked by the following operations:

- **translate** *Odl* _Olcd_ translates a ODBMS schema from ODL (an ODL ODMG extension) to OLCD [BBLS98].
- **translate** *Odl* _Olcd_ *vf* Translates a schema described in ODL into a OLCD description returning also a *visual form* description used by the ScVisual GUI.
- **validate** *Odl* Translates and validates schema described in ODL. ODB-Tools internally, translates the schema into OLCD and runs the Validator. The validator optimizes the schema and, if it exists, signals incoherences.
- **validate** *Odl* _XML_ Works as **validate** *Odl* but output is returned also in XML.
- **validate** _Olcd_ _XML_ Validates (optimize) a schema described in OLCD and returns a XML file. ODB-Tools runs the Validator directly on the OLCD schema and signals any incoherences. Output is returned in XML. This is the method that used by MOMIS.
- **validate** _OdlSS_ Activates the validation and optimization of a previously translated schema. The **validate** *Odl* translates and validates and optimizes a schema described in ODL.
- **validate** _OdlSS_ *vf* Works as **validate** _OdlSS_ and a *visual form* is also returned.
- **optimize** _Oql_ Optimizes a OQL query on a ODL schema. The ODB-Tools query optimizer optimizes a OQL query on a given schema.
- **optimize** _OqlSS_ Optimizes an OQL (ODMG) query on a previously validated schema. **optimize** _Oql_ optimizes a query on a ODL schema by performing schema translation and validation and query optimization steps.
- **killServer** De-allocates the connection *servant* object. This operation should be invoked by a *client* before closing connection; if not, the *servant* object will also be de-allocated after a time-out period.

**idlOT_factory** Describes a *factory* object, i.e., the CORBA object to refer to in order

---

8*translate** _Odl* _Olcd_ *vf*, **validate** _OdlSS_ *vf* and **optimize** _OqlSS_ *vf* supply the same services of the three methods above, but generate a schema description in *visual form* (see [Cor97]).
to open a connection.

**newServer** Opens a new connection, i.e., creates a new *servant* object instance whose reference is posted to the *client*.

![Diagram of request of a new client connection]

Figure 3.12: Request of a new client connection

In the implementation, the main ODB-Tools object is a *factory object* (interface `idlOT_factory`); for each client connection request a new *servant object* (interface `idlOdbTool`) is created and its reference is returned to the client (see figure 3.12).

**The XML representation**

The SIM module of the SI-Designer uses ODB-Tools to perform Thesaurus relation validation and inference. To provide a *standard* way to get ODB-Tools results we modified the ODB-Tools so to generate an XML output as well. Such XML adheres to the DTD showed in figure 3.13:

The corresponding client should use a XML parser to get ODB-Tools data.

### 3.8 The MOMIS Global Schema

The role of a Global Schema object is to represent an integrated schema. It contains all information about integration such as the defined global classes, the mapping tables, the thesaurus and the description of the schema of the integrated sources. This information will allow any *query manager* to perform queries on the represented integrated schema.

Nowadays, the Global Schema is an extremely simple component. Figure 3.14 shows the IDL interface to access a *GlobalSchema* object.

The method simply allows a client to access the schema name (*getName* and *setName*) or the schema status (*getStatus* and *setStatus*).

The status is exchanged as a serialized *GSStatus* Java object. This object is defined in the `shared` directory and basically contains a `odli3.Schema` object, i.e., the root
Figure 3.13: The ODB-Tools XML output Document Type Definition

of the tree of objects that maintains information about sources, thesaurus and global classes.

A CORBA GlobalSchema object is accessible through an instance of the class GlobalSchemaProxy. This manages CORBA connection and status serialization (see [Gui00]).
interface GlobalSchema {
    void setName(in string name);
    string getName();
    string getStatus();
    void setStatus(in string status);
    long killObject();
};

Figure 3.14: The Query Manager IDL Interface
Chapter 4

The MOMIS SI-Designer

This chapter presents the MOMIS module SI-Designer, architecture and user's manual. SI-Designer is the GUI (Graphic User Interface) for the MOMIS Global Schema Builder. SI-Designer leads the designer through the steps of the integration process, from the acquisition of schemata sources to the tuning of the mapping table. This is the most important component of the MOMIS prototype; it is a modular container of tools implemented by different people.

In this chapter, we focus on the strategic choices and concepts related to the SI-Designer software. Most of the implementation choices are described in the source code documentation, so for further information refer to the documentation extracted from the source code by javadoc.

4.1 The SI-Designer architecture

Most of SI-Designer modules have been developed by several students during computer engineering Laurea thesis. Their goal was to analyze, design and implement one of the integrations phases described in the theory.

My role was to define and project the modular architecture to allow people to work together, establish set of guidelines for development and coordinate the students.

Figure 4.1 shows the main objects SI-Designer is composed of. For an overview of the MOMIS architecture, see chapter 3.

- SI-Designer, the main container, is a GUI object. It is responsible for CORBA connections (through a proxy) towards MOMIS servers and manage all other sub-modules. It also maintains also the schema object that holds all integration information and is responsible for its initialization and serialization, it also allows saving it and allows saving it on disk.

- GlobalSchemaProxy is the object responsible for the creation of the Global Schema CORBA object. This maintains aligned the Global Schema and a local working copy of the integration knowledge (i.e. all integration information like the local schemata description, the thesaurus, etc.).

- SIDPhase Each panel that implements a different function inherits from this
class the features to communicate with the SI-Designer.

- The objects ARM, ARTEMIS, EXTM, SAM, SIM, joinMap, testPackage, thesRelationEditor, SLIM and TUNIM are the modules that compose the SI-Designer interface.

![SI-Designer run-time architecture](image)

**Figure 4.1: SI-Designer run-time architecture**

### 4.2 An integration session

In this section, a whole integration session, from source wrapping to the tuning of the mapping tables, using SI-Designer is shown.

In the following *macro* steps for the integration in an ideal sequential order are described. Each step is highly interactive, the designer usually acquires better and better knowledge of the source to be integrated during integration, so, in a real integration session, the designer will move back and forth through the integration phases.

**Wrapping of the sources**

The first step of the integration using MOMIS is to prepare data sources to be accessed by MOMIS. Each source should be wrapped using the most appropriate Wrapper. We developed a very simple and generic JDBC wrapper (see section 3.5.3) but it may be necessary to develop an ad hoc wrapper for a specific data source.

Whoever runs the Wrapper must choose the source name and register the running Wrapper into a naming server accessible by SI-Designer. Wrappers for a given source should be initialized and run by the responsible of the source.

**Choosing a significant name for the global view**

(Figure 4.2) The SI-Designer integration session starts choosing a significant name for the global schema we will build.
4.2 An integration session

Source connection/acquisition

Each source, or better, the wrapper of each source to be integrated, is reached by the CORBA naming service: one need only to state the registration name and the naming service address where the wrapper is registered.

If there are two (or more) sources with the same name, SI-Designer will ask you to internally rename sources with the same name. This will be the name of the source in SI-Designer.

![Figure 4.2: SI-Designer source acquisition module](image)

For each source, SI-Designer reads the source description and shows the sources in terms of local classes and attributes on the left of the SI-Designer window (see Figure 4.2).

Computation of intra-schema relationships

Figure 4.3 shows the panel that enables the designer to browse and edit Thesaurus Relationships. Clicking the button run SIM the ODB-Tools CORBA service is called and intra schemata relations (see section 2.2.1 on page 18) extracted.

WordNet annotation and Lexical relationships extraction

Figure 4.4 shows the panel that allows the designer to annotate the schema to be integrated (see section 2.1.5 on page 16 and 3.6 on page 57). Figure 4.5 shows Common Thesaurus terminological relationships extracted from WordNet.
Validation of lexical relationships and inference of new relationships

After the terminological relationship extraction, clicking on the button run SIMb on the Common Thesaurus relationship editor panel, ODB-Tools service is called again to validate and infer new relationships (see section 2.2.4 on page 20 and 2.2.5 on page 21). Figure 4.6 shows again relationships in the common thesaurus, after the validation phase.

Creation of proposed Global Classes

Figure 4.7 shows a screen shot of the interface for running and tuning the ARTEMIS Clustering/affinity algorithm (see section 2.3.1 on page 24). The clustering tool can be tuned by working on clustering thresholds and relation weights. In the lower part of the window, a text area shows the trace of the intermediate results during the run of the ARTEMIS algorithm.

In addition Figures 4.8 and 4.9 show the panel for tuning the proposed global classes. Cluster of local classes computed by ARTEMIS are proposed as Global Classes, and the designer can rename or modify the composition of such Global Classes.

Creation of the global class attributes and mapping table

The last step of integration (with the current version of SI-Designer) is the Creation and tuning of the global class attribute mapping table (see section 2.3.2). After the simple mapping (Figure 4.10) an attribute aggregation is proposed, and the designer can then refine the mapping (Figure 4.11).

4.3 Methodological considerations

This section gives a few hints for the correct use of SI-Designer derived from our experience.

Relationships editor hints

The module shown in Figure 4.3 is designed to manage the Common Thesaurus relationships.

Through this module it is possible to run the extraction from schemata, validation of relationships and new relationship inferential algorithms on the schemata.

The designer can also choose to manually add or remove relationships.

We suggest running the inferential algorithm after each new thesaurus relationships is added in order to detect immediately if the new relationship generates inconsistencies.

It is possible to iteratively refine the clustering adding or removing relationships. To find the relationships to add or remove to improve clustering, try to look at the cluster coefficients.

The displayed relationships are the ones used by the ARTEMIS algorithm to compute global classes by clustering.
4.3 Methodological considerations

Lexical annotation hints

The SLIM module shown in Figure 4.4 allows interaction with WordNet. It graphically allows the designer to associate WordNet meanings to every schema element, then retrieve from WordNet lexical relationships between such meanings and convert them into (not yet validated) common thesaurus relationships.

This is the most sensible point in integration, if a schema is well when annotated the integration time and effort can dramatically be reduced.

Main rules to follow annotating are:

- Annotate every local class, taking care to choose the WordNet meaning that best corresponds with the reality the local schema describes. Context is very important.
- If the word form for the element is not significant, replace it with a meaningful word and choose the right meaning.
- Annotate only significant attributes.
  Attribute present in all interfaces like name, code or identifier could lead to the generation of wrong relationships.
  For example, name has the right meaning associated to the Interface name. It could mean person name, city name, or the name of anything. This mean that (wrong) relationships between the interface Person and City, or Person and anything could be generated, and so on.
- Annotate only elements you are sure about the meaning.
  If you are not sure, you should study the data source to be integrated better, and then annotate the elements.
Clustering tuning hints

Remember that clustering is strictly implied by the relationships present in the thesaurus. If the relationships are wrong it is not possible to automatically extract a good cluster.

For a better understanding of the clustering algorithm (the ARTEMIS tool) and concepts like naming affinity and structural affinity, see section 2.3.1 on page 24;

The tuning parameters for the clustering are the following (see Figure 4.7).

- The relationship weights. When working on these parameters we can to give more or less importance to *this* or *that* kind of relationship.
  
  For example, if we have a schema with a lot of RT relationship and the number of clusters is too high, we can try to lower the weight of the RT relationship and recompute the clusters.

- The affinity threshold. This parameter gives an index of reliability to the naming affinity. The clustering will be affected by the naming affinity only if the naming affinity coefficient is higher than the threshold.

- The clustering threshold. This parameter that influences the width of clusters. The higher the threshold, the smaller the computed clusters are because a high affinity is needed between Interfaces to be part of the same cluster.
  
  In case of good clustering, this parameter chooses the number of clusters vs the number of local classes in each cluster.

  To have one idea of the tree clustering quality, look at the *Clustering Tree* (like...
the ones showed in Figure 4.14). From this tree you can see the distances in terms of affinity values between Interfaces.
If the instances the designer knows to be strictly related are represented as near tree leaves, the tree structure is good, and by increasing or decreasing the clustering threshold we can to obtain good clusters.
We can also try to modify other tuning parameters or study the relationships present in the thesaurus.
- The name affinity and the structural affinity coefficients.
  These parameters simply give more importance to the name or the structural affinity while weighing the affinity between Interfaces. The clustering algorithm will then use such weights to compute clusters.
  The higher the name affinity parameter, the higher the importance of the name affinity. The same goes for structural affinity.

Global classes tuning hints

(Figure 4.9) This is very important step.
In this phase the designer chooses the composition of the global classes. No one can guarantee that the semiautomatic model proposed by MOMIS is correct (usually indeed there are a few errors) and it is the responsibility of the designer to perform the right tuning.
An important step is to assign the right name to the global classes. This will help the designer gather local classes by meaning.
The designer is the judge. Right mapping relies on the designer’s experience and knowledge of the sources acquired during the previous phases.
It is correct to leave some useless local classes unmapped (even if it would be better if these useless local classes where not exported by the source wrapper).

Mapping table tuning hints

In this phase the designer chooses for each global class how each attribute of each local class that is part of the global class is mapped into the global class global attributes.
Like for the Global classes tuning, here the experience and knowledge of the designer are the basis to obtain the right mapping.

4.4 Implementation

This section describes some concepts, data structures and techniques implemented in SI-Designer, and modules I implemented.

4.4.1 The SIDPhase interface

Each module of SI-Designer must implement this interface. Through the methods defined in this interface we can to share integration knowledge through SI-Designer modules and provide a simple way for coordination between modules.
The methods implemented by this interface include:
Figure 4.5: SI-Designer Part of the relationship extracted from WordNet

- *GlobalSchemaProxy* `getProxy()` Returns the Global schema proxy object. This is the way to retrieve and modify schema information. The `GlobalSchemaProxy` object is in fact responsible for maintaining the GlobalSchema data repository aligned with the local copy.

- `void update()` throws `SIDException` This method is called by SI-Designer each time the module is displayed.

- `void saveStatus()` throws `SIDException` This method is called by SI-Designer each time the module loses focus.

The strategy for module coordination implemented by the SIDPhase interface is the following:

- When the user decides to get into the module, the module using the method `getProxy` is responsible for loading the module status from the main `Schema` object (that contains all information related to the integration).
- When the user decides to change module, the module is responsible for saving its status and all module information into the `Schema` object.

### 4.4.2 Saving and exchanging knowledge in MOMIS

Here we discuss various features adopted to save, retrieve and share Integration knowledge.

Problems related to status in MOMIS are:

- how to interrupt and resume an SI-Designer integration session
Figure 4.6: SI-Designer SIM: relationship are validated and new relationships inferred

- how to allow users to run query managers using the integration structures created by SI-Designer.

We needed a fast, reliable and easy solution to save and communicate the integration knowledge that is carried by many complex memory data structures.

Our choice was to use Java Object Serialization, sparing ourselves the worry of defining description languages or data interchange specifications.

We supported serialization in the main code of SI-Designer (i.e., the class SI-Designer) allowing the designer to save, whenever he wants, the status of the session. Once saved, it is possible to start a new SI-Designer session specifying a serialized status using the option:

```
java SI-Designer -l /path/to/the/serializedStatus.mms
```

We also used serialization to exchange data using the CORBA connections. The idea was to use strings containing serialized Java Objects. We did it using a base 64 coding technique to overcome problems due to different charset representation. Each time we serialize an object we transform the resulting binary stream in a base 64 coding string. Deserializing, at first we decode the base 64 string in a binary stream that can be deserialized. We use this technique when saving SI-Designer status and when communicating between the SLIM module and the WordNet CORBA interface.

The main disadvantage in using java serialization is that, especially in developing phases, once a class is modified, all old serialized status containing the class become incompatible with the new recompiled java code. This mean that all existing integration must be redone.

We are currently studying and developing tools for automatically serializing any seri-
alizable java Object in XML (even if in Sun there is an analogous project). Using XML we expect the following advantages: less of a problem passing through different versions of serialized java classes, human readability of the serialized code (if needed we can manually modify or create a status) and, above all, better portability for integrating information. Using XML easy parsing capabilities, and XML translation languages like XSL, it will be easy to exchange integration knowledge with applications requiring them like other mediators.

### 4.4.3 The Common Thesaurus Editor module

This module, an example of which is given in Figure 4.6, allows the designer to browse and edit relationships defined in the Common Thesaurus. It shows relationships in different colors according to the module producer of the relationships. For example, relationships extracted from schema are shown in dark gray; terminological relationships from WordNet in blue, inferred relationships in red, and designer manually added relationships in green. Moreover, the thesaurus editor make it possible to show the relation ordered by source, by type (SYN, NT, BT, RT), by destination or by producer. One need only to click on the corresponding header to sort the relationships on the chosen column.

Edit capabilities allow the designer to remove any relationship from the thesaurus, or, add explicitly new relationships.
4.4 Implementation

4.4.4 The ARTEMIS module

In this section implementation related problems and solutions are presented. The theory related to ARTEMIS was presented in section 2.3.1.

A lot of information about implementation like the detailed description of all the methods or the description of the objects private variable is also omitted; these details are available in the documentation extracted from the source code by javadoc.

In designing the ARTEMIS I adhered to the following guidelines:

- ARTEMIS must be a package,
- all the algorithms must be located in a well interfaced object,
- development should be as fast as possible,
- the source code should be as readable as possible.

To build a package was not a problem. Moreover, the architecture of SI-Designer requires that modules be developed in packages. To develop the prototype quickly and get the code readable I relied on a strongly Object Oriented architecture and developed a few tools for managing distance graphs.

Object part of the ARTEMIS package

The main class implemented has been called Artemis and contains all ARTEMIS computation algorithms. Its interface is the following:

- public Artemis(GlobalSchemaProxy schemaProxy) is the constructor. An ARTEMIS object contains a reference to the GlobalSchemaProxy. In this way it is always
able to access to the current schema to get/save the ARTEMIS status or to get thesaurus relationships.

- `void saveStatusInSchema()` Save the status of this Artemis Object into the schema reachable by `schemaProxy`;
- `void doClustering()` Call all computing routines getting the clustering. This is only one shortcut to perform the steps provide by calling the methods in sequence: `computeInitialDistanceMatrix`, `computeNameAffinity`, `computeStructuralAffinity()`, `computeGlobalAffinity()`, `computeClustering()` and `computeClustersFromClusterTree()`.
- `void computeInitialDistanceMatrix()` Computes the Initial Distance Matrix. This method simply translates the relationship present in the thesaurus in affinity, according to the coefficient chosen by the designer.
- `void computeNameAffinity()` Computes the Name Affinity using an interactive algorithm. This algorithm will find the maximum affinity between each element trying any possible path (see below for a better description).
- `void computeStructuralAffinity()` Computes the Structural Affinity from the Schema structure and the naming affinity matrix (see section 2.3.1 for a theoretical description of this algorithm).
- `void computeGlobalAffinity()` Computes the Global Affinity from the Structural affinity matrix and the Naming affinity matrix.
- `void computeClustering()` Computes the affinity tree from the Global affinity matrix.
- `void computeClustersFromClusterTree()` From the affinity tree produced by `computeClustering` this method divides local classes in clusters.
- `static void populateGlobalClassesFromClusters()` From clusters (vector of vec-
4.4 Implementation

Figure 4.10: SI-Designer the TUNIM module, a first simple attribute mapping

- java.util.Vector getClusters() Returns a Vector of Vectors of Interfaces.
- ArtemisStatus getStatus() Returns the Status object.
- public String toString()

The object ArtemisStatus simply contains the configuration parameters for ARTEMIS such as the clustering threshold or the weight to use for each kind of relationship (SYN, NT, BT, RT). An object of the class ArtemisStatus is saved in the schema additional information to store/retrieve the ARTEMIS status.

All ARTEMIS computation algorithm are based on distances between elements, I defined the Distance Object, which simply stores a generic distance (represented as a floating point values) between two Java objects. In addition, I created the DistanceMap class and the DistanceMapSymmetrical that inherits from DistanceMap. Each instance of the DistanceMap class can manage a set of Distance objects and is suitable to represent directed graph. Instances of the DistanceMapSymmetrical manages distances considering them as symmetrical and is suitable to represent undirected graph. The method of the class DistanceMapSymmetrical are the same of the DistanceMap but overridden for managing symmetrical connections.

The first step in the ARTEMIS computations is to transform all thesaurus relationship between schema element into distances (according to the status settings). By doing this we can utilize algorithms for symmetrical distance graph.

The following section describes the most significant techniques implemented in the ARTEMIS module.
The algorithm to compute naming affinity

The naming affinity computation requires the computation of the path that maximizes the affinity between all nodes of the graph. Maximum affinity is the SYN affinity and has value 1; minimum affinity is absence of affinity and is associated to the value 0. For example, given the elements $e_1$, $e_2$ and $e_3$ where the only affinity between them are $A(e_1,e_2) = 0.8$ and $A(e_2,e_3) = 0.5$, the affinity $A(e_1,e_3)$ has value 0.4 given by the product of $A(e_1,e_2) \cdot A(e_2,e_3)$. In the implementation, the affinity is called distance (since it is most usual to work with distances in graphs) and a single affinity value between two elements is represented by a Distance object.

The problem of computing all maximal affinity between all nodes is the same of finding the minimum distance (following any path) between all nodes of a distance graph. There are only two differences:

- the goal is to find maximum distance
- the distance on a path is computed making the product of the distances instead of the sum.

As an algorithm to compute such distance, I chose an iterative algorithm that, instead of recursively exploring the graph searching for the minimum path, tries to find the shortest local distance considering ever 3 node at a time. At each step we consider all possible combinations of 3 nodes A,B and C, if the distance $D(A,C)$ passing by B, computed as $D(A,B) \cdot D(B,C)$ is better than the distance $D(A,C)$ the distance $D(A,C)$ is replaced with the new one. The algorithm iterates these steps until there are no changes in the distance graph.

The demonstration that this works can be done by absurdity. Imagine that the algorithm
4.4 Implementation

has finished and that a path \( p_1 \) exists from A to C passing by the point B that gives a distance better than the distance \( D(A, C) \). The absurdity is that the algorithm is terminated! In fact, evaluating the points A,B and C the algorithm finds a better distance and will replace as \( D(A, C) \), this graph modification require that the algorithm performs a new iteration on the graph and cannot be terminated.

Tree representation as composition of Distance objects

The algorithm that builds the clustering tree uses the Distance objects to represent the built tree. In fact, a Distance object relates two tree branches indicates the distance between them. Leaf nodes of the tree are graph elements, while non leaf nodes of the graph are Distance objects. For example, consider the graph in Figure 4.12; the tree is composed by 3 leaf schema elements connected by distance elements that carry the affinity information.

```
distance2 - 0.5
    +-- distance1 - 0.8
        |    +-- element1
        |    +-- element2
    +-- element3
```

Figure 4.12: An example of Artemis tree expressed using Distances

This graph means that between element1 and element2 there is affinity 0.8 and that between these and element3 there is affinity 0.5.

The toString method and the DistanceMapSymmetrical class

In Figure 4.13 and Figure 4.14 part of the output of the Artemis.toString method output is shown. This method simply calls the toString method for each local object of Artemis. We can therefore see the ArtemisStatus status; the initial distance matrix (which simply reports the thesaurus knowledge); the Affinity matrix, which has a lot of computed distances; the Structural Affinity matrix; the Global Affinity matrix; the generated cluster tree; and the proposed clusters. The toString representation of a Artemis shows all computation steps and allows the computation steps to be traced. Distance and DistanceMapSymmetrical classes manages distances between any kind of Java object. The result of the toString method of a Distance object contains the toString representation of both the related objects.

Other objects in the ARTEMIS package - the graphical interface

The other objects contained in the ARTEMIS package are GUI specific objects. Figure 4.7 shows the panel that allows the designer to tune the ARTEMIS module. Through this panel we can modify the ARTEMIS computation parameters, and at bottom the results calculated step-by-step are indicated.

In figure 4.8 the panel for global classes/cluster tuning is shown. Such panel allows the designer to modify the name and the content of each single global class, adding
Global Affinity matrix:

<table>
<thead>
<tr>
<th>Elements: 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0 Interface [Computer_Science.CS_Person]</td>
</tr>
<tr>
<td>e1 Interface [Computer_Science.Course]</td>
</tr>
<tr>
<td>e2 Interface [Computer_Science.Location]</td>
</tr>
<tr>
<td>e3 Interface [Computer_Science.Office]</td>
</tr>
<tr>
<td>e4 Interface [Computer_Science.Professor]</td>
</tr>
<tr>
<td>e5 Interface [Computer_Science.Student]</td>
</tr>
<tr>
<td>e6 Interface [University.Department]</td>
</tr>
<tr>
<td>e7 Interface [University.Research_SStaff]</td>
</tr>
<tr>
<td>e8 Interface [University.Room]</td>
</tr>
<tr>
<td>e9 Interface [University.School_Member]</td>
</tr>
<tr>
<td>e10 Interface [University.Section]</td>
</tr>
<tr>
<td>e11 Interface [tax_position_xml.ListOfStudent]</td>
</tr>
<tr>
<td>e12 Interface [tax_position_xml.Student]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>src-</th>
<th>e0</th>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
<th>e5</th>
<th>e6</th>
<th>e7</th>
<th>e8</th>
<th>e9</th>
<th>e10</th>
<th>e11</th>
<th>e12</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.73</td>
<td>0.68</td>
<td>0.2</td>
<td>0.56</td>
<td>0.2</td>
<td>0.60</td>
<td>0.2</td>
<td>0.2</td>
<td>0.56</td>
<td></td>
</tr>
<tr>
<td>e1</td>
<td>0.2</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.58</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>e2</td>
<td>0.1</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>e3</td>
<td>0.2</td>
<td>0.12</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>e4</td>
<td>0.73</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>e5</td>
<td>0.68</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>e6</td>
<td>0.2</td>
<td>0.12</td>
<td>0.06</td>
<td>0.08</td>
<td>0.16</td>
<td>0.16</td>
<td>0.08</td>
<td>0.16</td>
<td>0.08</td>
<td>0.16</td>
<td>0.08</td>
<td>0.16</td>
<td></td>
</tr>
<tr>
<td>e7</td>
<td>0.56</td>
<td>0.25</td>
<td>0.56</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td></td>
</tr>
<tr>
<td>e8</td>
<td>0.1</td>
<td>0.25</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td></td>
</tr>
<tr>
<td>e9</td>
<td>0.40</td>
<td>0.25</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td></td>
</tr>
<tr>
<td>e10</td>
<td>0.2</td>
<td>0.12</td>
<td>0.06</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td></td>
</tr>
<tr>
<td>e11</td>
<td>0.1</td>
<td>0.25</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td></td>
</tr>
<tr>
<td>e12</td>
<td>0.56</td>
<td>0.25</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.13: An example from Artemis.toString() output, the Global Affinity matrix

or removing local classes. By using this graphical interface the designer can build his own global classes from scratch or tune the ones proposed by MOMIS.

The main objects used are two DynamicTree, which contains as main nodes, respectively, NodeSource and NodeGlobalClasses objects. Both NodeSource and NodeGlobalClasses are DefaultMutableTreeNode objects (this means that are nodes managed by a JTree Swing component) that contain as children NodeInterface objects. A NodeInterface represent a local classes and can be either in the Tree of the classes to be mapped (organized by sources) or in the mapped interface tree (organized by global classes). All routines for mapping or un-mapping interface simply move NodeInterface nodes from one tree to the other. At the end of the Designer tuning, the Global Class tree is parsed and new Global Classes data structure are created.

Actually the two sub-panels of the clustering module, the one for Global classes tuning and the one for the ARTEMIS algorithm tuning, are designed to be two independent modules and could appear as main SI-Designer pads since they are totally independent from each other. Both read status through the GSProxy and both generate Global Classes. I developed the ClusteringPanel specifically to unify these two panels. This object acts as a sub SI-Designer panel propagating the saveStatus and Update method to the single modules.

### 4.4.5 The test module

The test module has been developed for testing routines during the execution of SI-Designer. Since it is a SI-Designer module it can access all integration information. The GUI is rather simple and is composed only of a big text area (used to display the output of the test routines) and a tool-bar that contains the buttons connected to the test routines.

Two functions are permanently placed in this panel:

- the Schema.toOlc method. The method is called on the main schema object and
4.4 Implementation

ClusterTree:
  tools.Distance: 0.25
  +src:
    | tools.Distance: 0.25
    | +src:
    |   | Interface [Computer_Science.Office]
    | +dst:
    |   | tools.Distance: 0.314285714285714
    |   | +src:
    |   |   | tools.Distance: 0.5833333333333334
    |   |   | +src:
    |   |   |   | Interface [Computer_Science.Course]
    |   |   | +dst:
    |   |   |   | Interface [University.Year]
    | +dst:
    |   | tools.Distance: 0.5
    |   | +src:
    |   |   | Interface [Computer_Science.Location]
    |   | +dst:
    |   |   | Interface [University.Room]

Figure 4.14: An example from Artemis.toString() output, the Clustering Tree

returns a olcd description of the global schema. This description is ready to be
sent to ODB-Tools to perform schema validation or relationship inference.

- the Schema.toOdl method. The method is called on the main schema object and
returns an ODL\textsubscript{12} description of the integrated schema containing the complete
description of the local sources, the thesaurus (expressed as a list of relationships), and the generated mapping tables.

This description could (but does not work yet) be passed as input to the ODL\textsubscript{12} parser that should reallocate almost totally the data structures that describe the integration. This cannot be a total reconstruction since information like the SI-Designer module status (like the SLIM element annotation) cannot be expressed in ODL\textsubscript{12} and will be lost.
4.5 Implementation and experimentation considerations

Some activities performed in MOMIS for information source integration have an element of subjectivity, being an intuitive process, because the knowledge and experience of the designer can be a requirement to ensure that the integration is proceeding correctly. In particular, the specification of inter-source relationships, both intensional and extensional, and the various tuning activities are of this kind.

Our effort has been concentrated in helping the designer by providing interactive functionalities and allowing the designer to have easy access each time to as much information possible about source and integration process.

In the following, we give the main feedbacks of the experimentation of MOMIS tools on practical integration examples. The good quality of affinity evaluation relies on both the correctness of the intensional relationships and on the parameters (i.e., strengths, weights, thresholds) used in the calculation of the coefficients and their relative values.

ARTEMIS has been experimented on different sets of conceptual database schemas to select a set of default values (i.e., the ones working satisfactorily in most cases) for the various parameters (strengths, thresholds, weights) intervening in the affinity and clustering stages. The values of terminological relationship strengths in the Common Thesaurus and of affinity weights and thresholds used in the examples of this paper correspond to these selected default values. Default values, however, can be dynamically varied by the designer when necessary in order to tailor the affinity calculation to the specific integration contexts.

As a consequence, most difficult tasks involved in the integration process result simplified, in that semantically related information is automatically identified and the designer is asked for a validation of the proposed results or, for ambiguous situations, for a selection among a set of pre-defined choices.

Moreover, the possibility of interacting with the tool to vary default parameter configurations and compare their results allows the tuning of the integration process. The experimented interaction with WordNet was satisfactory in sources integration when the schemata to be integrated have “meaningful names”; in this case, most of the terminological inter-schema relationships are obtained, thus avoiding a lot of boring work for the designer. However, for many legacy applications adopted names are not meaningful. In this case, the automatic extraction of intra-schema relationships and the aid of the system in checking consistency of explicitly given relationships are good aids for the designer.

A deeper discussion of the experimentation results of the use of strengthened terminological relationships and affinity-based clustering of ARTEMIS for the integration of heterogeneous data schemas in the Italian Public Administration domain, is given in [CAV00, CA99a].

With respect to the MOMIS prototype, we implemented it in Java, relying on the portability of such language. We obtained a portable software application. The MOMIS prototype runs on Unix machine as well as Windows machine without having to recompile a single class. Also communications (CORBA connection) are quite transparent, and it works well regardless of the execution platform. Using Java we obtained portability but we lost performances. The most resource consuming algorithms in MOMIS are the search for lexical relationships in WordNet and the validation/inferential algorithm (SimB). Such algorithms, with a schema of 37 interfaces in 3 sources and 247 attributes, in a distributed environment (SI-Designer running on a Windows Ma-
chine Pentium III 600Mhz, servers, running on a Solaris Sparc 433Mhz) takes about 70 seconds to extract 1016 new lexicon-derived relationships, while the execution of the SimB algorithm takes about 240 seconds to execute and infers 40 new relationships. ARTEMIS (running on the Pentium) takes about 15 seconds proposing 23 global classes.
Chapter 5

Conclusions

In this document, we have presented a semi-automated approach to information extraction and integration of heterogeneous information sources. The ODL$_{13}$ language is introduced for information extraction and integration, by taking into account also semistructured information sources. A Description Logic module (ODB-Tools engine) provides inference capabilities to construct a Common Thesaurus of inter-source relationships, the WordNet lexical database is used as knowledge base to add information for the integration in form of lexicon derived relationships, and a cluster generator module (ARTEMIS tool environment) provides capabilities to identify candidates global classes of the integrated global view. The proposed approach has been implemented in the MOMIS prototype following a conventional wrapper/mediator architecture. MOMIS provides a set of tools and associated techniques for performing semantic integration. MOMIS provides several techniques and associated tools for helping the designer in the integration of heterogeneous information sources by means of an application with a graphical interface, called SI-Designer. SI-Designer interfaces all the employed tools with the goal of allowing an interactive and customized use of MOMIS techniques by the designer based on the specific requirements of a given integration process.

Future research will be devoted to the development of the Query Manager component of MOMIS with query optimization and “answer composition” functionalities, based on definition of extensional axioms and integrity constraints defined on global ODL$_{13}$ classes. One of the original aspects of the Query Manager will consist of employing Description Logics-based components (i.e., ODB-Tools) to perform semantic optimization steps both on global and local queries, to minimize the number of accessed sources and the volume of data to be integrated as the result of sub-query execution.

Research in MOMIS will also face the problem of the Automatic Integration, where we will study techniques to eliminate (at least in some applications) the role of the integration designer. To automate the integration we will move the annotation capabilities on Wrappers and we will extend the ODL$_{13}$ (language and data structures) to support annotations. Furthermore, we will consider using Mobile Agents as tool to discover interesting data sources.

The MOMIS project will be financed for years 2001/2002 by the D2I: Integration, Warehousing, and Mining of Heterogeneous Data Sources project of the Italian MURST ministry.
Chapter 6

Part II - Information Retrieval with the Keynet system

This part presents the results obtained in the information retrieval (IR) research activity at the Northeastern University (Boston, MA - USA) under the supervision of Professor Kenneth Baclawski during my six-month stay in the USA.

In particular, a set of algorithms to efficiently compare Keynet system objects in order to speed-up the information retrieval are presented.

Also presented is a technique to classify documents with respect to a given query and a technique to aid the user in refining the query.

To this end we developed:

- an intersection algorithm for graphs
- an indexing and classification technique based on the intersection with the query graph
- a tool for superquery classification calculus of a set of documents that satisfy the given query. This study does not present any result about precision and recall (these are indexes used to measure the information retrieval system performances) because we did not have any already annotated collection on which to measure these indexes (for example, an annotated collection is the Tipster’s collections).

6.1 The Keynet system

The Keynet system is designed for IR from a corpus of information objects in a single subject area. It is well suited for non-textual information objects, such as scientific data files, satellite images and videotapes. For example, the literal content of a satellite image does not include the geographic coordinates of the boundaries of the image or other cartographic abstractions. Some kinds of textual document, such as research papers in a single discipline, can also be supported. With current technology, Keynet can support very high-performance IR from a corpus having up to several million objects at approximately the same level of performance as smaller corpora.

A Keynet system requires the development of a subject-specific concepts ontology that is understandable to a literate practitioner of the field. A Keynet ontology represents
knowledge using a directed graph of conceptual categories and relationships among them. **Keynet**, further, assumes that each information object (*document*) in its corpus has been annotated with a content label that indicates what portion of the subject-specific ontology relates to the content of the object (*keynet structure*).

Tools have been developed and tested working in a biomedical context. The Unified Medical Language System (UMLS) developed by the National Library of Medicine is the reference ontology \[HL93\] for this context.

Both content labels and queries have the same data structure called the *keynet structure*. A *keynet* may be regarded as a kind of semantic network \[Lev92\] , although in practice it is semantically intermediate between keywords and semantic networks. The keynet framework generalizes many commonly used mechanisms for information retrieval, such as: subject classification schemes, keywords, document abstracts, reviews, content labels for non-textual information objects, properties such as author or date of publication, ranges of text strings, such as “wild card” match strings, and ranges of quantities. The **Keynet** system allows a uniform treatment of these disparate techniques in a system that permits a great deal of flexibility compared to traditional database and information retrieval systems. For example, one can combine all of the above mechanisms in a single system and easily add new features to the ontology, such as new attributes and keywords. In addition, the **Keynet** framework allows for sequences of concepts linked by relationships and expressed in natural language using phrases, clauses, sentences and paragraphs.

A content label is similar to an abstract or review of a document both in size and in being separately accessible from its corresponding information object. Using a tool such as M&M-Query System \[BF93\], a content label can be generated by the author of the information object with no more effort than is now taken to write the abstract or to select the keywords.

### 6.2 Graphs in Keynet system

Both queries and content labels are represented as graphs of concepts. To represent nodes of each graph in main memory we use the following data structure, called the *keynet structure*. This structure is used to store keynet information ad for communication in the **Keynet** cluster of computers. This is the structure of the contents:

- **Ontology Identifier**. Each ontology is assigned a unique integer identifier.
- **Major version number**. Since ontologies can evolve over time, a version number is used to distinguish both major and minor versions of an ontology. Minor versions differ from one another only by the addition of new concepts, conceptual categories and relationship types. Major versions may differ in more substantial ways, including the splitting of categories, merging of categories, as well as more complex alterations in the ontology.
- **Minor version number**.
- **Count**. Since the **Keynet** system is distributed, keynets have to be transmitted between nodes of the network. During the transmission the count field specifies that the keynet has been split into a number of pieces as specified in this field. Normally the value of this field is 1.
- **Sequence number** When a keynet is in several pieces, this field will have the sequence number of this piece. The values range from 0 to one less than the count field above.
6.2 Graphs in Keynet system

- **Vertex count** This is the count of the number of vertices that will be specified in the list immediately following this field.
- **Vertex list** This is a set of zero or more vertex specifications. Each vertex specification consists of three integers as follows:
  - **Vertex id** Each vertex of a keynet has a unique identifier within the keynet.
  - **Type id** Each vertex has a type or conceptual category. The types and their identifiers are listed in the ontology.
  - **Instance id** A vertex may be instantiated using one of the instances given in the ontology. Instance identifiers are nonzero integers. If this field has value 0, then the vertex has not been instantiated.
  - **Outgoing Edge list** This is a set of all the edges with this vertex as source.
  - **Incoming Edge list** This is a set of all the edges with this vertex as destination.

- **Edge count** This is the count of the number of edges that will be specified in the list immediately following this field.
- **Edge list** This is a set of zero or more edge specifications. Each edge specification consists of three integers as follows:
  - **Source vertex** This is the vertex identifier of the source of this edge in the keynet.
  - **Destination vertex** This is the vertex identifier of the destination of this edge in the keynet.
  - **Edge type id** Each edge has a type. The edge or relationship types and

A keynet structure \( k \) is basically a graph \( (V, E) \) where the couple of vertices and edges is associated respectively to the couple of concepts and relationships among concepts of the ontology.

![Figure 6.1: Example of keynet graphs](image_url)

In figure 6.1 two examples of keynet structure are shown. Keynet (a) represent the text “Monkeys can suffer from hernias incarcerate.” and Keynet (b) represents the text “The right ventricle is contained in the heart, both of which primarily consist of muscle tissue. Lymphocytes are associated with both the right ventricle and the heart.”

In figure 6.2 two examples of keynet structure for representing query are shown. Keynet (q1) derives from the text “What antibodies lyse (lysis) lymphocytes in vivo?” and Keynet (q2) derives from the text “What marrow transplant protocols use monoclonal antibodies to deplete recipient lymphocytes?”
6.3 The Keynet search engine architecture

The purpose of Keynet\textsuperscript{1} is to assist in retrieving information objects from a corpus of them. These information objects need not be textual and may be physically located anywhere in the network. Retrieval is accomplished by means of a content label for each information object. These content labels are stored in a repository at the Keynet site. The structure of the content labels is specified by an information model or ontology. The content labels are indexed by means of a distributed hash table stored in the main memories of a collection of processors at the Keynet site. These processors form the search engine. Each content label contains information about locating and acquiring the information object. The Keynet system is only concerned with finding information objects; users are responsible for actually acquiring (and presumably paying for) information objects.

To see more precisely where all of these components reside, and how they are connected to one another, refer to Figure 6.3. The user’s computer is in the upper left position. A copy of the ontology is kept locally at the user site. As this will require from several hundred megabytes to a few gigabytes of memory, it would generally be stored on a CD-ROM. The ontology is also the basis for the user interface to the search engine. Queries must conform to the format specified by the ontology, and are sent over the network to a front-end processor at the Keynet site. Responses are sent back over the network to the user’s site, where they are presented to the user using the ontology. The prototype system uses a connectionless communication protocol so that no connection is required for making a query, and also so that the responses need not be sent back from the same computer that originally received the query.

At the Keynet site, the front-end computer is responsible for relaying query requests to one of the search engine computers. The reason for having a front-end computer is mainly for distributing the workload but it also helps to simplify the protocol for making queries. The search engine itself is a collection of processors (or more precisely server processes) joined by a high-speed local area network. The search engine processors will be called nodes. The repository of content labels is distributed on disks attached to some of the nodes. The index to the content labels is distributed among the main memories of the nodes. The prototype differs from the Keynet architecture only in that it randomly generates the repository as well as queries sent to it.

Since a connectionless communication protocol is unreliable, it is necessary for the user computer to re-send the query if there is no response after a timeout period. The Keynet protocol is stateless and idempotent, and so it works well with a connection-

\textsuperscript{1}from [BS94d]
6.4 Graph comparison functions

Introduction

In the theory of graphs, a graph G consist of a finite nonempty set V of elements that are called vertices or points, together with a set E of distinct ordered (for the directed graphs) or unordered (not directed graphs) pair of points that belong to V. Each element e of the set E is said edge.

A vertex is said labeled when each vertex can be be identified uniquely by a label lv. In the same way, we can say an edge is labeled when the edge can be uniquely identified by a label le.

The keynet structure is a directed graph with labeled vertices and labeled edges. In a keynet structure it is not required that labels uniquely identifies vertices or edges. This means that vertices or edges that shares the same label are allowed. This is due to the nature of the concepts described in a document, i.e., in keynet structure a vertex represents a concept in a document.

A concept may be introduced and referred in a document more than once. This produces a single keynet vertex. But, in a document the same concept can also be introduced in more than one different context and this requires more equals vertices to be represented.

Comparison functions

To categorize graphs and use efficient algorithm we need to define a comparison function. Since our algorithms are based on comparing sorted list of vertices or edges, such function must define an order relationship among objects.

For example, in the Keynet system the label for a vertex is given by the concept identifier\(^2\). If we assume that the concept identifier for a vertex is an integer number, the

\(^2\)The concept identifier for a vertex is the identifier of the ontology concept represented by the vertex.
comparison function between two vertices can be easily defined as the function “algebraic difference between the concept identifier of the two vertices” $f_{cv}(v_1, v_2) = v_1.c_{id} - v_2.c_{id}$.

Two edges are equals when the edge label, the source vertex and the destination vertex are equals.

Using the comparison function among vertices and a function that compares the edges label we can define the comparison function among edges, for example, evaluating the three attributes in the following order: (1) the edge label, (2) the source vertex and (3) the destination vertex.

$$f_{ce}(e_1, e_2) = \begin{cases} 
0 & \text{if } e_1.src.l \equiv e_2.src.l \\
& e_1.dst.l \equiv e_2.dst.l \\
& e_1.l \equiv e_2.l \\
& e_1.src.l < e_2.src.l \text{ or } ( \\
& e_1.dst.l \equiv e_2.dst.l \text{ and } \\
& e_1.l < e_2.l)) \\
< 0 & \text{if } ( \\
& e_1.dst.l < e_2.dst.l \text{ or } ( \\
& e_1.dst.l \equiv e_2.dst.l \text{ and } \\
& e_1.l < e_2.l)) \\
& e_1.src.l > e_2.src.l \text{ or } ( \\
& e_1.src.l \equiv e_2.src.l \text{ and } ( \\
& e_1.dst.l \equiv e_2.dst.l \text{ and } \\
& e_1.l > e_2.l)) ) \\
> 0 & \text{if } ( \\
& e_1.dst.l > e_2.dst.l \text{ or } ( \\
& e_1.dst.l \equiv e_2.dst.l \text{ and } \\
& e_1.l > e_2.l) ) \\
\end{cases}$$

Matching algorithm

The matching\(^3\) algorithm compares two graphs and creates the intersection graph. Since we defined comparison functions for vertices and edges we can assume that the lists of vertices and edges are sorted. So we can apply the algorithm to find equals elements in two ordered list (that is quite fast), the list of vertices and the list of the edges.

This algorithm performs, in the worst case, $n_1 + n_2$ comparison where $n_1$ and $n_2$ are the length of the the two sorted list (see Figure 6.4).

6.5 Classification and indexing of a query result set

Introduction

We developed this tool to categorize query results. The goal is to build a tool that quickly give an idea to the users of the contents of the query result set. Since the graphical User Interface shows the keynet graphical representation of the query, our proposal is to show to the user all documents that match the query categorized by the “type of matching”.

The Keynet system, using its distributed hash table, is able to retrieve from the database all the documents that roughly match the query. The tool presented in this section

\(^3\)In the graph theory a matching is a subset of the edges in which no vertex appears more than once. We use another meaning for the the term match. For us, two graphs match when they carry the same information.
categorize these documents according to the following rules:

1. Let the *representative graph* for a document be the intersection between the *query* and the document himself. We assume the *representative graph* expresses the *type of matching*.
2. One category is identified uniquely by a "*representative graph*" that is itself a *Keynet* structure.
3. One category contains a list of *all and only* the documents that share the same "*representative graph*" characterizing the category.

This implies that given a document and a query, each document belongs to a uniquely identified category.

Categorizing the document and showing for each category the *representative graph* and other statistical information such as the number of documents in each category, the average ranking weight\(^4\) of the documents gives the user an overall idea of the kind of documents has been retrieved by the query.

**The classifier**

A *category* is a couple (an object) \(c : (k, L_k)\) that relates a keynet structure \(k\) with a list of other keynet structures \(L_k\).

A *categorizer* object must manage a list of *categories*. The main operations for a *categorizer* are:

1. insert a document \(d\) in the *categorizer* specifying its representative graph \(g\).
2. provides a way to access to the categories and to the document list of each category.

Now we describe the easiest way to insert a new document \(d\) characterized by the graph \(g\) in a *categorizer*. Let us call this process the *naive approach*. We search a graph that is equal to \(g\) within the characteristic graphs of the categories. If found, let \(c\) be the category with the characteristic graph equals to \(g\). The category \(c\)

---

\(^4\)At present, the ranking of documents is performed by using the *cosine* weighting technique [Sal89],
is the category for $d$ and we add $d$ to the list of document of $c$.

If not found, we create a new category $c$ with $g$ as characteristic graph and $d$ in the list of documents. Then we add $c$ to the list of categories of the categorizer.

The main problem of the naive categorizer is that it is too slow. Each time we have to insert a new document we perform an average of $n_c/R$ graph comparison, where $n_c$ is the number of categories. Moreover, comparing two graphs is a expensive operation because we have to compare all edges and vertices of both graphs.

An indexed classifier

We want a classifier that reduces the number of vertex and edge comparison in order to speed up the classification process.

It is possible to substitute the comparison among vertices (or edges) with the comparison among references to categories (for example a pointer to a category object or a category identifier) and we will show how to do it.

Let us introduce a simplified representation of the Keynet graph model.

Let $\mathcal{O}$ the reference ontology for a Keynet system. Here is described a simplified representation for the ontology:

$\mathcal{O} = (\mathcal{V}_o, \mathcal{E}_o, \mathcal{R}_o)$

Where $\mathcal{V}_o$ is the set of nodes (or concepts) of the ontology $\mathcal{R}_o$ are the type of relationships allowed by the ontology among ontology nodes and $\mathcal{E}_o$ is the set of edges (relationships among nodes) composing the ontology. Note that $\mathcal{E}_o \subseteq \{(v_1, r, v_2) : v_1, v_2 \in \mathcal{V}_o, r \in \mathcal{R}_o\}$

A Keynet graph $\mathcal{G}$ can be represented as a subset with repetition of the ontology elements. It can be represented by the following couple:

$\mathcal{G} = (\mathcal{V}_t, \mathcal{E}_t)$

$\mathcal{V}_t \subseteq \{(v, n) : v \in \mathcal{V}_o, n \in \{1, 2, \ldots\}\}$

$\mathcal{E}_t \subseteq \{(v_1, r, v_2, n) : v_1, v_2 \in \mathcal{V}_o, r \in \mathcal{R}_o, n \in \{1, 2, \ldots\}\}$

The number $n$ is necessary since vertices and edges might appear more than once in a Keynet graph.

This kind of representation relies on the index technique used to index graphs or categories (in the categorizer we index the categories describing the characteristic graphs of the category).

By definition, two graphs are equals when they have exactly (the same number of) the same vertices and (the same number of) the same edges. The equality of two vertices or edges of different graphs is given by the comparison functions described in section 6.4 on page 89.

Data structure

The above representation is obtained by maintaining a list of vertices and for each of such vertex $v$, storing a list of couple $c_n : (e, n)$. In this couple $c$ is an identifier of a category (for example a pointer) and $n$ is the number of times the vertex $v$ appears in the characteristic graph of the category $c$. To save memory space and speedup comparisons we assume that if a category does not appear in the list of a vertex $v$ it means that its representative graph does not have the vertex $v$ (this assumption implies that the empty graph, no vertices and no edges, must be treated in a particular manner).

The same representation is maintained for the edges (for each edge is maintained a list
of couples \((c, n)\) category, number). Figure 6.5 are shows the data structures maintained by the classifier.

This kind of representation is useful when we try to find a category given its representative graph (this is the first step of the \textit{insert a new document} operation).

To retrieve the category for a given representative graph \(g\) we follow the algorithm below according to the following hypothesis:

1. lists of vertices are kept ordered both in the \(g\) graph and in the categorizer
2. lists of edges are kept sorted both in the \(g\) graph and in the categorizer

The algorithm

The following is the algorithm implemented by the indexed categorizer.

\begin{verbatim}
if (\(g\) is a empty graph)
    handle the empty category
else
    search the category using data structure
    compare the edge lists
    get the first categorizer \(e_c\) edge
    get the first graph edge \(e_g\) counting the occurrences
    do
        if (no more edge to compare)
            exit
        endif
        compare the two edges
        if (edges match)
            if (is not initialized the \(l_e\) category list)
                initialize the \(l_e\) category list from \(e_c\)
            else
                intersect \(l_e\) with the \(e_c\) category list
            endif
            if (\(l_e\) is empty \(\text{(no categories)}\))
                the category for \(g\) does not exists
            endif
        else
            exit
        endif
    do
end
\end{verbatim
endif

counting the occurrences

g has an edge not present in the categorizer

the category for g does not exists

exit_routine #2

done

if(there are other edges in g)

g has an edge not present in the categorizer

the category for g does not exists

exit_routine #3

done

compare the the vertices lists

g has a vertex not present in the categorizer

the category for g does not exists

exit_routine #4

done

if(there are other vertices in g)

g has a vertex not present in the categorizer

the category for g does not exists

exit_routine #5

done

if(there are other edges in g)

g has an edge not present in the categorizer

the category for g does not exists

exit_routine #6
6.5 Classification and indexing of a query result set

. endif
. label #2
. in \( l_g \) there is only one category
. and is the category for \( g \)
endif

Basically the category for \( g \) is selected intersecting all the list of categories that match all the edges and all the vertices of the graph \( g \). The intersection is performed by initializing the intersection list \( l_c \) on the first edge or vertex matching, and intersecting this list with the list of the other edge or vertex matching.

To prove the correctness of the algorithm, we have yet to describe the two following steps:

1. **initialize the \( l_c \) category list from \( e_c \) (or \( v_c \))**

   This procedure initializes a list of category \( l_c \) used to select the category of a given graph \( g \). The same procedure is used to initialize \( l_c \) from a list of categories associated to both an edge of the categorizer \( e_c \) or to a vertex of the categorizer \( v_c \).

   The list \( l_c \) is initialized by copying the categories from the list associated with the edge \( e_c \) (or the vertex \( v_c \)) according to the following rules:
   
   1. remember that \( e_c \) matches \( e_g \) and we call this vertex \( e \).
   2. the number of occurrences of the edge \( e_g \) (or the vertex \( v_g \)) in \( g \) must match the number associated the category. This means that the graphs of each selected category must have exactly the same number of occurrences of the edges \( e \) (or the vertex \( v \)) as the graph \( g \).
   3. the number of edges and the number of vertices of \( g \) must match the respective number of edges and number of vertices of the graph of each selected category.

2. **intersect \( l_c \) with the \( v_c \) category list**

   In the list \( l_c \) only those classes that are present also in the list associated with the edge \( e_c \) (or with the vertex \( v_c \)) are maintained. Moreover, the number of occurrences of the edge \( e_g \) (or the vertex \( v_g \)) in \( g \) must match with the number associated with the category. This means that the graph of each maintained category must have exactly the same number of occurrences of the edges \( e \) (or the vertex \( v \)) as the graph \( g \).

**Proof of correctness**

We prove that our indexing algorithm finds for \( g \) all and only the categories that have as representative graph \( g_c \) a graph that matches \( g \).

We assume that two graphs \( g_c \) and \( g \) match when they have exactly the same vertices and edges and with the same number of occurrences. Two vertices or two edges match according to the comparison functions described on page 89. The data structures described on page 92 must be correctly maintained as well.

**Find only those categories:**

We can have different situations that differ from the case where \( g \) is the empty graph. This case is handled as an exception so we make sure that if the empty category is already present, then we will find it. In the following list we will examine all possible ways to select or unselect a category in the algorithm on page 93:

- from the initialization of the list \( l_c \) all categories where \( g_c \) has a number of edges
or of vertices different than \( g \) are dropped.

- **exit #1** \( g \) can differ from a \( g_c \) because has a different number of occurrences of a given edge. This means that the representative graph \( g_c \) of the selected categories has the same number of occurrences of each matching edge for each matching edge.

- **exit #2 and exit #3** if we find an edge in \( g \) that is not present in any \( g_c \), we do not select any category. This means that all selected categories must have at least the same edges of \( g \).

- **label #1** in this point in the list \( l_c \) all the categories with \( g_c \) with the same number of edges of \( g \) are present, the edges are the same and the occurrence is the same for each matching edge.

- **exit #4** \( g \) can differ from a \( g_c \) because it has a different number of occurrences of a given vertex. This means that the representative graph \( g_c \) of the selected categories has the same number of occurrences of each matching vertex, for each matching vertex.

- **exit #5 and exit #6** if we find a vertex in \( g \) that is not present in any \( g_c \), we do not select any category. This means that all selected categories must have at least the same vertices of \( g \).

- **label #2** at this point in the list \( l_c \) all categories with \( g_c \) are present with the same number of edges of \( g \), the edges are the same and the occurrence is the same for each matching edge and also with the same number of vertices of \( g \), the vertices are the same and the occurrence is the same for each matching vertices. This is the definition of matching graphs.

**Find all those categories:**

The routine of initialization initialize the list \( l_c \) with all the categories present in the lists associated with a categorizer edge or vertex. Since these categorizer list are correctly maintained by the categorizer, then, in each list, all the categories with the given edge or vertex are present.

Suppose there are two categories that match the given graph \( g \). Both categories, for construction of the data structures, will appear in all the categorizer lists in such a way to describe the graph \( g \).

This means that both categories will be selected by the initialization procedure of the list \( l_c \) and will also remain in the list \( l_c \) after each category intersection step. Since both of them match the graph \( g \), both will be selected by the algorithm (each of the exit points will be skipped).

But in the list \( l_c \) there will be only one category. We can prove this by supposing the existence of two different categories \( c_1 \) and \( c_2 \). If \( c_1 \) and \( c_2 \) will be selected by the algorithm, it means that both the respective characteristic graphs \( g_1 \) and \( g_2 \) matches \( g \) and this implies that \( g_1 \) match \( g_2 \). This is impossible for the definition of categorizer, which requires that the representative graphs of each category must all be different from the others.

**Complexity**

To study the algorithm complexity we need to parametrize the data to be processed and to make several assumptions. Our goal is not to express the exact formula, however we want to give an approximation of the algorithm complexity.

We represent our set of graphs to be classified by these characteristic parameters:
6.5 Classification and indexing of a query result set

- \( n_g \) number of graphs to categorize
- \( n_c \) number of categories in which such graphs will be classified
- \( n_v \) average number of vertices for the graphs in the set
- \( n_e \) average number of edges for the graphs in the set

We assume a random, uniform distribution of edges and vertices in each graph and of graphs in the set. We also assume that the cases of repeated vertices (e.g., more than one vertex with the same label) or edges in a graph are rare (the indexing algorithm will run faster with a high number of vertices or edges repetitions). We consider as unit of measure for the calculus of the complexity the comparison between two identifiers and assign the following weight:

- 1 compare two identifiers, by assumption
- 2 compare two graph vertices, because in our case a vertex is identified by a couple of other identifiers. This is an optimistic assumption; a vertex may be identified in a much complex way
- 3 compare two graph edges, because an edge is characterized by two vertices and some other information. An average comparison matches half of these data.

To compare two lists of \( n \) elements (of edges or vertices) we assume they are ordered so that this operation can be performed with \( n \) comparisons.

**Naive algorithm**

To search a category we assume it employs average \( \frac{n_e}{n_c} \) graph comparisons. The cost for a graph comparison is \( 2n_v \cdot 3n_v \).

The complexity for the naive algorithm is given by:

\[
3 \cdot n_g \cdot n_c \cdot n_v \cdot n_e
\]

**Indexing algorithm**

To search a category we assume it performs, in the worst case \( n_e \) edge comparisons, \( n_v \) vertex comparisons and \( \frac{n_e}{n_c} \cdot (n_v + n_e) \) category comparisons.

To insert a category we have to compare \( n_v \) vertices, \( n_e \) edges and sort \( n_c \cdot (n_v + n_e), n_v, n_e \) lists of identifiers.

The complexity for the indexing algorithm is given by:

\[
\begin{align*}
n_g \cdot (3n_e + 2n_v + \frac{n_e}{n_c} \cdot (n_v + n_e)) + \\
n_c \cdot (3n_v + 2n_v + n_v \cdot n_e) \cdot \log_2(n_c \cdot (n_v + n_e)) + \\
n_e \cdot \log_2(n_e) + n_e \cdot \log_2(n_e)
\end{align*}
\]

The most important operation (i.e., find a class function) can be performed in a \( O(n) \) of the number of the graph.

**Tests**

The test were computed on a linux machine; the time has been measured using the \( C \) function \texttt{times}.

We ran two kinds of test sets using random generated graphs with different topologies.
1. A first test set (abc) is composed by graphs with only 3 vertices (labels 'a', 'b' and 'c') repeated several times. For the edges, only one kind of label exists, so the number of edges is very small $3^3$. In these tests the most characterizing data
of a graph is the number of repetitions of a given vertex or edge.
These tests correspond to the graph in Figure 6.7 and 6.8.
Figure 6.7 is obtained by categorizing this set of graphs using a query with 6
vertices and 6 edges. This produces categories with average 6 vertices and 3 edges matched. Figure 6.8 is obtained by categorizing this set of graphs using a query with 15 vertices and 25 edges. This produces categories with average 10 vertices and 6 edges matched.

2. The second test set (a..z) is composed of graphs with a more realistic distribution of vertices (the vertices allowed are 26 and repetitions are allowed, too). Also, the number of allowed labels for the relationships are 3 so the number of allowed edges is $26^2 \cdot 3$.

These tests correspond to the graph in Figure 6.9 and 6.10. Figure 6.9 is obtained by categorizing this test set of graphs using a query with 6 vertices and 5 edges. This produces categories with an average of 2 vertices and 0 edges matched. Figure 6.10 is obtained by categorizing this test set of graphs using a query with 15 vertices and 42 edges. This produces categories with an average of 5 vertices and 0 edges matched.

Figure 6.6 summarizes the number of categories extracted by classifying the test set for each, varying the test type and the number of graph of the test set.

The estimated time for the indexed algorithm has been computed using the formula of page 97 without adding the insertion times of the news categories.

### 6.6 Query refinement

Refine the query $Q$ means to write a more selective query $Q_1$ the results of $Q_1$ are a subset of the results of $Q$. In the following we say extend a query to mean refine the query. This is because the refinement of a Keynet query is done by adding vertices or
Our goal is to design a tool that assists the user in query refinement. We propose a tool that given a query and a list of documents that exactly matches the query, the tool behaves like a categorizer that classifies documents according to all possible ways the query can be refined. This tool is useful when the number of documents returned from the query is high (for example more than 150 documents).

We define a supergraph graph of a given query computed on a given document as the union of the query graph and an edge that is not part of the query but is part of the document. In Figure 6.11 an example of supergraph extraction comparing a query and a document is shown.

Imagine that the user submits a query that is associated to too many documents. In this case, the GUI of the system will show the keynet graph of the query and allow the user to explore ways to refine the query. Clicking on a concept (a vertex) of such graph will pop up a list of all relationships that extend such concept (with all related information such as number of document, average weight, ...). The user can refine the query choosing one concept, then a relationship and then a concept again.

The process can obviously be iterated and the new query extension can be recomputed using the document in the document list of the chosen relationship).

This way to browse the database starting from the given query allows the user to learn about the distribution of the documents in the database.

Figure 6.12 the steps for keynet query refinement are shown.

Contents: In this section we present two procedures: one to extract all possible extensions of a query from a document, and one to handle the association among all possible extension of a query and the associated document.
Edge matching for the supergraph problem

Here we describe how to perform the matching between a query graph and a document in order to extract all new edges that extends the query.

The problem

Let $Q(V_q, E_q)$ the query graph and $D(V_d, E_d)$ the document graph we want to use to extend the query $Q$.

Let $e$ a generic edge and, let $e.src$ and $e.dst$ the source and destination vertices of $e$.

We want to extract the set of output edges $O \subset E_d$, $o \in O$ where:

1. $\exists v \in V_q : o.src == v$ or $o.dst == v$

Where the relationship $==$ is calculated using the comparison function of page 89.

The algorithm

If for each vertex $v \in V$ of a graph $G(V, E)$ are maintained the list of ingoing and outgoing edges we can apply the following algorithm.

1. compare the vertices of the document $(d)$ with the vertices of the query ($(d)$)
2. for each matching vertex copy the reference (the pointer) of the ingoing and outgoing edges in a list of duplicated edges.
3. sort the extracted edges and delete the duplicated edges.

Complexity

We make the following assumptions:

- $n_{vd}$ number of vertices of the document graph
- $n_{vq}$ number of vertices of the query graph
- $n_{ed}$ number of edges of the document graph
- $n_{eq}$ number of edges of the query graph

We can distinguish the following steps and corresponding costs:

- sorting vertices of document and query: $n_{vd} \log(n_{vd}) + n_{vq} \log(n_{vq})$
- extracting edges with duplicates: $n_{vd} + n_{vd} + 2 \alpha n_{ed}$
- sorting and deleting duplicated edges: $2 \alpha n_{ed} \log(2 \alpha n_{ed})$

Where $\alpha \in [0, 1]$ is a parameter of the graph to compare.

$\alpha$ is the probability for a document to have a vertex in common with the query. If we suppose uniform distribution in edges correlation, and given that the document and the query shares the query vertices, then $\alpha = \frac{n_{vd}}{n_{vd}} + \frac{n_{vd} - n_{vd}}{\text{number of corpus vertices}}$.

When $\alpha = 1$ this means that all the edges of any document will be selected by the current query.

Data structure for supergraph categorizer

In this section we describe a procedure for supergraph extraction that fulfills the following requirements:

1. It processes several documents comparing them with a unique given query.
2. For each document, we want to handle (generate and memorize) all possible supergraphs of the document according to the given query.
3. Given a supergraph, we want to know which documents share the given supergraph.
4. Given a vertex of the query we want to know what the possible supergraphs are that involve such vertex. Therefore, given a vertex, it is possible to select a way to extend (refine) the query, and then the list of document with the selected extension will be available.

We assume that the following statements are true:

- a graph is a couple $G : (V, E)$ where $V$ is a list of vertices and $E$ is a list of edges
- for each vertex $v \in V$ the list of edges ingoing and the list of edges outgoing from the vertex $v$ are maintained
- functions to compare edges and vertices have been defined. Such functions make it possible to uniquely define an order relationship among edges and among vertices.

Observe that any supergraph of a given query can be described as the union of the query graph and an edge that extends the query but does not belongs to the query. This means that a subgraph for a given query is completely defined by an extension edge. This allows us to save memory and cpu time in handling supergraphs.

The proposed data structure to store and classify query supergraphs is implemented memorizing:

**superquery** is a graph that contains the query graph and all (not duplicated) edges that extend the query in the processed documents.

**categories** is a list of category object. A category object associates an edge with a list of documents and knows if the vertex belongs to the query. In the list categories there will be a category for each superquery edge. And in each category's list of documents, there will be a list of all documents that contains the category edge.

By maintaining this data structure it is easy to extract information about the extension of the query starting from a given vertex of the query. All the edges that extend the category are given by the union of the ingoing and outgoing edges from the selected vertex in the superquery graph. The list of documents associated to each single selected edge is stored in the category characterized by that edge.

### 6.7 Related work

We worked in a specific branch of Information Retrieval, we developed tools to handle knowledge in the Keynet system. For more information about Keynet system see [BS94a, BS94b, BS94d, BF93, BS94c]. The Keynet system uses the cosine weighting technique to rank documents described in [Sal89].

Basically we work with labeled directed graphs and these are treated in the graph theory, but our tools are designed for retrieving information. Since the graphs are labeled, we defined a order relationship on vertices and edges and used such order relationships for comparing graphs. To handle the graph isomorphism problem, we looked at the algorithm presented in [Ull76].

The architecture of the graph categorizer is a simplification of the architecture of the Keynet system [BS94c][BS94d].
Query refinement

[VWSG97] introduces the concept recall. This is an experimental measure of an algorithm’s ability to suggest terms that are semantically related to the user’s information need; it also presents a fast algorithm for terms suggestions with the goal of improving the precision. Such algorithm is based on term weighting among the terms of the documents that match the query.

[LCHH97] presents a visual query interface for querying multimedia databases that allows users to pose queries using icons and menu in a drag-and-drop fashion. This interface translates the visual query in CSQL, a query language with additional predicates for image matching and semantic-based query condition. This is a tool to help users query a multimedia database in an “exploratory” fashion.

[JFS98] presents an improvement of the unsafe (or approximate) query algorithm of [Per94]. It modifies the query evaluation based on the current contents of the buffers, and this speeds up the retrieval in query refinements.

In contrast with [LH99] and [VWSG97], whose motivation is to enhance information retrieval by developing models with the ability to diagnose a user’s informational goals, our approach is not based on probability but relies on the structure of the Keynet system.

6.8 A graphical user interface for Keynet visualization

This section describes a tool (KNEditor) developed by myself for Keynet visualization over text documents. The goal of this tool is to let the user perceive by intuition how concepts are correlated in the showed text.

The description of this graphical interface should aid to explain, through example, what a Keynet is.

Guideline: in designing this application I tried to reduce the redundancy of information stored in the memory data structure. The application was also designed to provide text editing functions (text insert or delete).

The application is a now keynet viewer application. It can load a Keynet and it allows the user to navigate interactively through relationships.

6.8.1 Specifications

Graphical layout

The GUI is composed of four windows that display the following information:

- list of concepts
- list of relationships
- properties of the selected items
- the document viewer, shows the text and the keynet graph of the document.

In the document viewer window we adopt the following conventions:

- a word related to a concept is displayed in blue.
  A concept is represented by a small blue dot over the text at the beginning of the word, all relationships arrows starts and ends on that point.
6.8 A graphical user interface for Keynet visualization

- a relationship is shown in red
  Relationships are represented with arrows from the starting concept to the end concepts.
  Unselected relationships are visualized as translucent arrows; selected ones are displayed as filled red arrows (highlighted).
- plain text is shown as black text over white background
- highlighted text is represented with a light gray background.

The KNEditor allows the user to (1) select a concept. This will highlight the concept and related relationships and show the concept properties. (2) selecting a relationship, will highlight the relationship and related concepts and show the relationship properties. (3) selecting a region of text will highlight the region.

6.8.2 The ontology

Ontology access

We need to access the ontology (1) in order to extract the names of the category/terms/relationship given the ID; (2) to extract the ID of the category/terms/relationship given the name; (3) to extract all possible relationships among two concepts given the concepts, and let the user choose the preferred one.

I need an accurate description of the ontology.

Since the ontology database is 1.6 gigabytes, we need a fast and efficient way to access to the ontology (e.g., using a database or preparing reverse indexes).

For the development proposal we do not have such kind of ontology interaction, but the temporary solution is to read categories/terms/relationships from the file containing the keynet annotation files and store them in three files:
Such files act as a (very simple) ontology proxy. The ontology interaction and the user’s ability to edit a Keynet description are the main features to be implemented.

### 6.8.3 Implementation notes

Figure 6.14 the object structure of the tool is shown. It is fully written in Java. This section adresses the different problems faced during the implementation.

Text representation

The text (KNEDocument) is a double chained list of characters (KNEDocCharacter). Each character has several properties, such as: the value (the character to display), the text color and the background color, the font, then the width, height and ascent sizes. I wrote a routine for formatting text that divides the text in the several lines used only for visualization, according to the display area available the height and width of each single character and the line separators (EOL characters). Lines are stored in a double-chained list of lines. A line contains at least one word (obviously only if the line has at least one word before the EOL). The line layer is used also to speed up character identification from the windows coordinates when the user clicks over the text.

**Saving format vs runtime format**

(1) The text in the saving format can be seen as a list of characters, and in such text a character could be identified by absolute position. The text associated with a concept or with a relationship, can be represented as a couple

---

5A Document could be represented as a list of pages and each page as a list of lines. But this page layer is not implemented at moment because we assume only small documents will be handled.
of integer values \((\text{position}, \text{length})\), where \text{position} is the number of the first char in the text and \text{length} expresses the number of char of the text.

(2) For the KNEditor, this kind of representation is too rigid. We need a more flexible representation that allows characters and concepts to be added without worrying about maintaining coherence between the position and length. So, for the editor we use the object pointers to double link concepts and characters. A character can be related to a single concept, a relationship or nothing. Moreover and a concept (but also a relationship) maintains a list of characters that represents the text associated with the concept.

\textit{Translation}

For translating among these two representation (\text{loading} translate \((1)\rightarrow(2)\) and \text{saving} translate \((2)\rightarrow(1)\)) we require two functions:

1. \text{\textit{loading}} retrieves a char given the position
2. \text{\textit{saving}} calculates the relative position of a given a char as if the text was a list of characters.

To speed up this process we need a \text{dirty} flag and data structure that is refreshed after loading and before saving (if the \text{dirty} flag is true), which memorize the starting position of the \text{Lines} in the document.

\textbf{Graphical layout management}

The graphical layout is generated in an image (with the size of the display area) stored in memory. This image is generated each time something changes in the text, such as window resizing, highlighting or un-highlighting of any object, text scrolling and \text{Text inserting or deleting} (not yet implemented).

This image is displayed in the \text{document display area} each time that something changes in the text or each time the window requires repainting.

\textbf{Concepts, vertices and relationships} A \textit{keynet} is a list of \textit{concepts} and a list of \textit{relationships} (called also \textit{edges}) among concepts. I extended this model allowing a concept to be represented by more \textit{text area} in the text. A concept can be recalled by more \textit{text areas}, but a text area can recall a single concept. So I called such text area as \textit{vertex}. A concept thus contains a list of vertices and is identified by a \text{category}._\text{id} and a \text{term}._\text{id} from the ontology. To maintain the spatial meanings of the relationship \text{i} defined \textit{edges} to be relationships between vertices. Obviously, the concepts inherit relationships from the vertices.

\textbf{DocConcepts} Since both vertices and edges have references to text, I introduced the \textit{DocConcepts} class that is used to manage the mapping between the keynet items and the text in the document. A \textit{DocConcepts} object can be a vertex or an edge. Each character of the document has a reference to \textit{DocConcept} object (is null if the character is not related to any keynet item). For this reason, I introduced the \text{non overlapping rule} among concepts in the text.

\footnote{\text{A text area} is a contiguous sequence of characters in the text of the document}
How a region of text is identified

The following items require a clever method to identify a region of text:

- current selection of text
- text associated with a concept
- text associated with a relationship

There are different ways to identify a region: (a) specifying the first and the last character pointer. The sequentiality of the selection is guaranteed. I do not have to worry about the number of characters in the selection, but I may have some trouble when I delete or move the last char of the selection. (b) specifying the first character and the length of the text. The sequentiality of the selection is guaranteed. Each time I add or delete a char of the selection I have to increase or decrease the counter of chars (this is not a computationally cost full operation because each char of the selection knows to that it is part of the selection, which makes the operation go fast). (c) memorizing in a vector all pointers to text characters. The sequentiality of the selection is not guaranteed. Each time I add or delete a char I have to align the whole vector.

Solution

I chose the solution b: a starting character plus length of the selection.

During the selection of text, the text associated with a concept or a relationship can be either all selected or all unselected. It is not possible select only a piece piece of text related to a concept or a relationship.

Current cursor Position. The cursor position identify the position of the insertion point. This can be represented by a character pointer. The portion where to insert is after the pointed character. The cursorPosition can have null value, null value means insert new characters at the beginning of the document.

6.8.4 Implemented functionalities

This is a row listing implemented functionalities written during the implementation.

- Speed up the image generation. To print each char, I have to set the following attributes: (1) color (2) font (3) background color.
  In order to speed up the character writing, I can cluster a sequence of character (with the same attributes) and draw a background rectangle and a single string once without setting many times these properties.
- Arrows for the relationships. I also tried arcs but I don’t like them.
- Selection vs highlighting policies.
  We have to handle two different kind of selections:
  - text selection.
    A text selection can be described as a list of consecutive characters.
  - concept (edge/vertices) selection. This is useful in text navigation

Solution

Only unassigned text (text not related to any concept) can be selected while concepts can only be highlighted. But this will not allow cutting and pasting of portion of text containing concepts.

To facilitate cutting and pasting I must define an approach that allows the selection of the text related to some concept. The most important thing to do is not to
allow partial selection of a assigned portion of text; that is, the text related to a
concept can be only or all selected or all unselected.

- **Highlight:** How to highlight:
  - A **vertex** is highlighted by:
    - clicking on the text related to the vertex.
    - selecting the concept that contains the vertex.
    - selecting a relationship that recall the vertex.
  - A **concept** can be highlighted by:
    - clicking on the concept on the concepts list.
    - selecting a vertex.
  - A **relationship** can be highlighted by:
    - clicking on an (highlighted) arrow or
    - clicking on the text related to the relationship.
    - selecting a vertex that this relationship is related.
    - clicking on the relationship on the edges list.

- **Events to handle:**
  - Click on a **vertex** text. **Action:** Highlight all the relationships connected
to such vertex (and the text of the relationship) and destination related vertices.
  - Highlight the concept in the concept list.
  - Click on a **edge** text or click on a **edge arrow** or click on a edge in the
    **edge list. Action:** Highlight both vertices of such relationship and the text
    related to the relationship. Each time I highlight a relationship I will move this
to the last position in the vector of the relationships. This means that
  - this will be the last to be displayed and the first to be selected.
  - Click on a **concept** in the concept list. **Action:** Highlight all vertices of
    such concept.

**Implementation:** I can use a vector of the highlighted concepts that maintains
the list of the currently highlighted items. This speeds up the un-highlighting
process.
I will implement as many functions as the events to handle.

### 6.9 Conclusions and future work

In this document we presented a set of tools for Information Retrieval relying on graph
of concepts. We presented high performance **Keynet** (or, in general, graph) classifier
providing description, proof of correctness and a set of tests performed on an imple-
mentation. In studying how to comparing graphs with duplicated nodes, the problem
arises of **graph isomorphism** in distinguishing among the **equals** vertices relying on
graph structure. An open problem is to evaluate if it is worth distinguishing among
**equivalent** vertices in order to obtain more knowledge useful for the document re-
trieval.

We also presented an analysis on how **Keynet** features can be exploited in query refin-
ing or database browsing starting from a query. Query refinement by classification is
a query classification tool, but it is also an instrument to learn from the search engine,
since the document database can be browsed starting from any query. It should be in-
teresting to study how to bring this feature on a term-based search engine (**Keynet** is
based on concept graphs).
During my stay in the USA a theory for document ranking specifically for the Keynet system was in development stage. This theory is necessary because in the Keynet system the independence of terms is not guaranteed (the Keynet system deals with edges and vertices and, edges and vertices in a graph are not independent, e.g., an edge requires the presence of two vertices) and this is one of the main assumptions in the Information Retrieval Theory.

Research on Keynet is proceeding developing a module for automating the extraction of a content label from a textual document. It is based on a machine-learning engine able to disambiguate the mapping among terms of the document and the concepts of the ontology.
Appendix A

ODO

syntax

Here is the full syntax of the ODL syntax. This syntax has been automatically generated from the yacc definition of the parser of the MOMIS prototype.

\[
\begin{align*}
\text{(OdlSpecification)} &::= \text{(Definition)} . \\
&| \text{(Definition)} ; \\
&| \text{(Definition)} ; \text{(OdlSpecification)} \\
\text{(Definition)} &::= \text{(TypeDcl)} \\
&| \text{(ConstDcl)} \\
&| \text{(ExceptDcl)} \\
&| \text{(Interface)} \\
&| \text{(RuleDcl)} \\
&| \text{(ExtRuleDcl)} \\
&| \text{(ThesaurusRelation)} \\
&| \text{(Module)} \\
&| \text{(error)} \\
\text{(TypeDcl)} &::= \text{typedef} \ (\text{TypeDeclarator}) \\
&| \ (\text{ConstrTypeSpec}) \\
\text{(TypeDeclarator)} &::= \ (\text{TypeSpec}) \ (\text{Declarators}) \\
\text{(Declarators)} &::= \ (\text{Declarator}) \\
&| \ (\text{Declarators}) , \ (\text{Declarator}) \\
\text{(Declarator)} &::= \ (\text{Identifier}) \\
&| \ (\text{Identifier}) \ (\text{ArraySizeList}) \\
\text{(ArraySizeList)} &::= \ (\text{FixedSize}) \\
&| \ (\text{FixedSize}) \ (\text{FixedSize}) \\
\text{(TypeSpec)} &::= \ (\text{SimpleTypeSpec}) \\
&| \ (\text{ConstrTypeSpec})
\end{align*}
\]
\langle\text{SimpleTypeSpec}\rangle \ ::= \langle\text{BaseTypeSpec}\rangle \\
| \langle\text{TemplateTypeSpec}\rangle \\
| \langle\text{Identifier}\rangle \\
\langle\text{BaseTypeSpec}\rangle \ ::= \langle\text{FloatingPtType}\rangle \\
| \langle\text{IntegerType}\rangle \\
| \langle\text{CharType}\rangle \\
| \langle\text{BooleanType}\rangle \\
| \langle\text{OctetType}\rangle \\
| \langle\text{RangeType}\rangle \\
| \langle\text{AnyType}\rangle \\
\langle\text{FloatingPtType}\rangle \ ::= \ 	ext{float} \\
| \text{double} \\
\langle\text{IntegerType}\rangle \ ::= \langle\text{LongIntType}\rangle \\
| \text{short} \\
| \text{unsigned long} \\
| \text{unsigned short} \\
\langle\text{LongIntType}\rangle \ ::= \ 	ext{integer} \\
| \text{int} \\
| \text{long} \\
\langle\text{CharType}\rangle \ ::= \ 	ext{char} \\
\langle\text{BooleanType}\rangle \ ::= \ 	ext{boolean} \\
\langle\text{OctetType}\rangle \ ::= \ 	ext{octet} \\
\langle\text{RangeType}\rangle \ ::= \ 	ext{range} ( \langle\text{RangeSpecifier}\rangle ) \\
\langle\text{RangeSpecifier}\rangle \ ::= \langle\text{SignedIntegerLiteral}\rangle, \langle\text{SignedIntegerLiteral}\rangle \\
| \langle\text{SignedIntegerLiteral}\rangle, + \text{infinite} \\
| \text{- infinite}, \langle\text{SignedIntegerLiteral}\rangle \\
\langle\text{AnyType}\rangle \ ::= \ 	ext{any} \\
\langle\text{TemplateTypeSpec}\rangle \ ::= \langle\text{ArrayType}\rangle \\
| \langle\text{StringType}\rangle \\
| \langle\text{CollectionType}\rangle \\
\langle\text{ArrayType}\rangle \ ::= \ 	ext{array} < \langle\text{SimpleTypeSpec}\rangle, \langle\text{IntegerLiteral}\rangle > \\
\langle\text{StringType}\rangle \ ::= \ 	ext{string} < \langle\text{IntegerLiteral}\rangle > \\
\langle\text{CollectionType}\rangle \ ::= \langle\text{AttrCollectionSpecifier}\rangle < \langle\text{SimpleTypeSpec}\rangle > \\
\langle\text{AttrCollectionSpecifier}\rangle \ ::= \ 	ext{set} \\
| \text{list} \\
| \text{bag}
\{ConstrTypeSpec\} ::= \{StructType\}  \\
| \{UnionType\}  \\
| \{EnumType\}  \\
\{StructType\} ::= struct \{Identifier\} \{\{MemberList\}\}  \\
\{MemberList\} ::= \{Member\}  \\
| \{MemberList\} \{Member\}  \\
\{Member\} ::= \{TypeSpec\} \{Declarators\} ;  \\
\{UnionType\} ::= union \{Identifier\} switch \{\{SwitchTypeSpec\}\} \{\{SwitchBody\}\}  \\
\{SwitchTypeSpec\} ::= \{IntegerType\}  \\
| \{CharType\}  \\
| \{BooleanType\}  \\
| \{EnumType\}  \\
| \{ScopedName\}  \\
| \{RangeType\}  \\
\{SwitchBody\} ::= \{Case\}  \\
| \{Case\} \{SwitchBody\}  \\
\{Case\} ::= \{CaseLabelList\} \{ElementSpec\} ;  \\
\{CaseLabelList\} ::= \{CaseLabel\}  \\
| \{CaseLabel\} \{CaseLabelList\}  \\
\{CaseLabel\} ::= case \{ConstExp\} :  \\
| default :  \\
\{ElementSpec\} ::= \{TypeSpec\} \{Declarator\}  \\
\{EnumType\} ::= enum \{Identifier\} \{\{EnumeratorList\}\}  \\
\{EnumeratorList\} ::= \{Enumerator\}  \\
| \{EnumeratorList\}, \{Enumerator\}  \\
\{Enumerator\} ::= \{Identifier\}  \\
\{ConstExp\} ::= \{OrExpr\}  \\
\{OrExpr\} ::= \{XOrExpr\}  \\
| \{OrExpr\} | \{XOrExpr\}  \\
\{XOrExpr\} ::= \{AndExpr\}  \\
| \{XOrExpr\} \^\{AndExpr\}  \\
\{AndExpr\} ::= \{ShiftExpr\}  \\
| \{AndExpr\} \& \{ShiftExpr\}  \\
\{ShiftExpr\} ::= \{AddExpr\}  \\
| \{ShiftExpr\} >> \{AddExpr\}  \\
| \{ShiftExpr\} << \{AddExpr\}  \\
\{AddExpr\} ::= \{MultExpr\}  \\
| \{AddExpr\} + \{MultExpr\}  \\
| \{AddExpr\} - \{MultExpr\}
\[\text{MultExpr} \quad ::= \quad (\text{UnaryExpr})
\quad | \quad (\text{MultExpr}) \ast (\text{UnaryExpr})
\quad | \quad (\text{MultExpr}) / (\text{UnaryExpr})
\quad | \quad (\text{MultExpr}) \div (\text{UnaryExpr})\]

\[\text{UnaryExpr} \quad ::= \quad (\text{Signes}) (\text{PrimaryExpr})
\quad | \quad (\text{PrimaryExpr})\]

\[\text{PrimaryExpr} \quad ::= \quad (\text{Identifier})
\quad | \quad (\text{IntegerLiteral})
\quad | \quad (\text{FloatingPtLiteral})
\quad | \quad (\text{OrExpr})
\quad | \quad (\text{error})\]

\[\text{ConstDcl} \quad ::= \quad \text{const} (\text{StringType}) (\text{Identifier}) = (\text{StringLiteral})
\quad | \quad \text{const} (\text{CharType}) (\text{Identifier}) = (\text{CharacterLiteral})
\quad | \quad \text{const} (\text{IntegerType}) (\text{Identifier}) = (\text{ConstExp})
\quad | \quad \text{const} (\text{FloatingPtType}) (\text{Identifier}) = (\text{ConstExp})\]

\[\text{SignedIntegerLiteral} \quad ::= \quad (\text{IntegerLiteral})
\quad | \quad (\text{Signes}) (\text{IntegerLiteral})\]

\[\text{SignedFloatingPtLiteral} \quad ::= \quad (\text{Signes}) (\text{FloatingPtLiteral})
\quad | \quad (\text{FloatingPtLiteral})\]

\[\text{Signes} \quad ::= \quad -
\quad | \quad +\]

\[\text{ExceptDcl} \quad ::= \quad \text{exception} (\text{Identifier}) \{ (\text{OptMemberList}) \}
\]

\[\text{OptMemberList} \quad ::= \quad (\text{MemberList})\]

\[\text{ScopedName} \quad ::= \quad (\text{Identifier})
\quad | \quad :: (\text{Identifier})
\quad | \quad (\text{ScopedName}) :: (\text{Identifier})\]

\[\text{Interface} \quad ::= \quad (\text{InterfaceDcl})\]

\[\text{IntView} \quad ::= \quad \text{interface}
\quad | \quad \text{view}\]

\[\text{InterfaceDcl} \quad ::= \quad (\text{IntView}) (\text{Identifier}) : (\text{InheritanceSpec})
\quad (\text{OptTypePropertyList}) (\text{OptPersistenceDcl})
\quad (\text{SingleInterfaceBody})\]

\[\text{OptTypePropertyList} \quad ::= \quad (\text{IntView}) (\text{Identifier}) : (\text{InheritanceSpec})
\quad (\text{OptTypePropertyList}) (\text{OptPersistenceDcl})
\quad (\text{SingleInterfaceBody}) (\text{InterfaceBodyUnionList})\]

\[\text{OptPersistenceDcl} \quad ::= \quad (\text{IntView}) (\text{Identifier}) (\text{OptTypePropertyList})
\quad (\text{OptPersistenceDcl}) (\text{SingleInterfaceBody}) (\text{InterfaceBodyUnionList})\]
\( \text{InheritanceSpec} :: = \langle \text{Identifier} \rangle \)
\( | \langle \text{InheritanceSpec} \rangle , \langle \text{Identifier} \rangle \)

\( \text{OptTypePropertyList} :: = \)
\( | \langle \text{OptSourceSpec} \rangle \langle \text{ExtentSpec} \rangle \langle \text{OptKeySpec} \rangle \langle \text{OptCandKeySpec} \rangle \langle \text{OptForKeySpec} \rangle ) \)

\( \text{OptSourceSpec} :: = \)
\( | \text{source} \langle \text{SourceType} \rangle \langle \text{Identifier} \rangle \)

\( \text{SourceType} :: = \)
\( | \text{relational} \)
\( | \text{nfrelational} \)
\( | \text{object} \)
\( | \text{file} \)
\( | \text{semistructured} \)

\( \text{OptExtentSpec} :: = \)
\( | \text{extent} \langle \text{ListExtent} \rangle \)

\( \text{ListExtent} :: = \langle \text{Identifier} \rangle \)
\( | \langle \text{ListExtent} \rangle , \langle \text{Identifier} \rangle \)

\( \text{OptKeySpec} :: = \)
\( | \text{key} \langle \text{Key} \rangle \)

\( \text{OptCandKeySpec} :: = \)
\( | \langle \text{CandKeySpecList} \rangle \)

\( \text{CandKeySpecList} :: = \langle \text{CandKeySpec} \rangle \)
\( | \langle \text{CandKeySpecList} \rangle , \langle \text{CandKeySpec} \rangle \)

\( \text{CandKeySpec} :: = \langle \text{candidate_key} \rangle \langle \text{Identifier} \rangle \langle \text{Key} \rangle \)

\( \text{OptForKeySpec} :: = \)
\( | \langle \text{ForKeySpecList} \rangle \)

\( \text{ForKeySpecList} :: = \langle \text{ForKeySpec} \rangle \)
\( | \langle \text{ForKeySpecList} \rangle , \langle \text{ForKeySpec} \rangle \)

\( \text{ForKeySpec} :: = \langle \text{foreign_key} \rangle \langle \text{ForeignKeyList} \rangle \langle \text{references} \rangle \langle \text{Identifier} \rangle \langle \text{OptRefKeyList} \rangle \)

\( \text{OptRefKeyList} :: = \)
\( | \langle \text{ForeignKeyList} \rangle \)

\( \text{ForeignKeyList} :: = \langle \text{Identifier} \rangle \)
\( | \langle \text{ForeignKeyList} \rangle , \langle \text{Identifier} \rangle \)

\( \text{Key} :: = \langle \text{PropertyList} \rangle \)

\( \text{PropertyList} :: = \langle \text{PropertyName} \rangle \)
\( | \langle \text{PropertyList} \rangle , \langle \text{PropertyName} \rangle \)

\( \text{PropertyName} :: = \langle \text{Identifier} \rangle \)

\( \text{OptPersistenceDcl} :: = \langle \text{persistent} \rangle \)
\( \langle \text{transient} \rangle \)

\( \text{InterfaceBodyUnionList} :: = \langle \text{InterfaceBodyUnion} \rangle \)
\( | \langle \text{InterfaceBodyUnion} \rangle \langle \text{InterfaceBodyUnionList} \rangle \langle \text{InterfaceBodyUnion} \rangle \)
\langle \text{InterfaceBodyUnion} \rangle ::= \text{union} \langle \text{Identifier} \rangle \langle \text{SingleInterfaceBody} \rangle

\langle \text{SingleInterfaceBody} \rangle ::= \{ \langle \text{InterfaceBody} \rangle \}

\langle \text{InterfaceBody} \rangle ::= \langle \text{Export} \rangle ;
| \langle \text{Export} \rangle ; \langle \text{InterfaceBody} \rangle

\langle \text{Export} \rangle ::= \langle \text{TypeDcl} \rangle
| \langle \text{ConstDcl} \rangle
| \langle \text{ExceptDcl} \rangle
| \langle \text{AttrDcl} \rangle
| \langle \text{RelDcl} \rangle
| \langle \text{OpDcl} \rangle

\langle \text{AttrDcl} \rangle ::= \langle \text{OptReadonly} \rangle \text{attribute} \langle \text{DomainType} \rangle \langle \text{AttributeName} \rangle \langle \text{OptFixedArraySize} \rangle \langle \text{OptMappingRuleDcl} \rangle

\langle \text{OptReadonly} \rangle ::= \text{readonly}
| \langle \text{Identifier} \rangle
| \langle \text{Identifier} \rangle ?

\langle \text{DomainType} \rangle ::= \langle \text{SimpleTypeSpec} \rangle
| \langle \text{StructType} \rangle
| \langle \text{EnumType} \rangle

\langle \text{OptFixedArraySize} \rangle ::= \langle \text{FixedArraySize} \rangle
| \langle \text{IntegerLiteral} \rangle

\langle \text{OptMappingRuleDcl} \rangle ::= \langle \text{MappingRuleDcl} \rangle

\langle \text{MappingRuleDcl} \rangle ::= \text{mapping rule} \langle \text{MapRuleList} \rangle

\langle \text{MapRuleList} \rangle ::= \langle \text{MapRule} \rangle
| \langle \text{MapRuleList} \rangle , \langle \text{MapRule} \rangle

\langle \text{MapRule} \rangle ::= \langle \text{LocalAttributeName} \rangle
| \langle \text{DefaultValue} \rangle
| \langle \text{MapAndExpression} \rangle
| \langle \text{MapUnionExpression} \rangle

\langle \text{LocalAttributeName} \rangle ::= \langle \text{LocalClassName} \rangle . \langle \text{Identifier} \rangle

\langle \text{LocalClassName} \rangle ::= \langle \text{Identifier} \rangle . \langle \text{Identifier} \rangle

\langle \text{DefaultValue} \rangle ::= \langle \text{Identifier} \rangle . \langle \text{Identifier} \rangle = \langle \text{StringLiteral} \rangle

\langle \text{MapAndExpression} \rangle ::= \langle \text{MapAndList} \rangle \text{and} \langle \text{LocalAttributeName} \rangle

\langle \text{MapAndList} \rangle ::= \langle \text{LocalAttributeName} \rangle
| \langle \text{MapAndList} \rangle \text{and} \langle \text{LocalAttributeName} \rangle

\langle \text{MapUnionExpression} \rangle ::= \langle \text{MapUnionList} \rangle \text{union} \langle \text{LocalAttributeName} \rangle \text{on} \langle \text{Identifier} \rangle

\langle \text{MapUnionList} \rangle ::= \langle \text{LocalAttributeName} \rangle
| \langle \text{MapUnionList} \rangle \text{union} \langle \text{LocalAttributeName} \rangle
\langle \text{RelDcl} \rangle ::= \text{relationship} \langle \text{TargetOfPath} \rangle \langle \text{Identifier} \rangle \text{inverse} \langle \text{InverseTraversalPath} \rangle \langle \text{OptOrderBy} \rangle

\langle \text{TargetOfPath} \rangle ::= \langle \text{Identifier} \rangle
| \langle \text{RelCollectionType} \rangle < \langle \text{Identifier} \rangle >

\langle \text{RelCollectionType} \rangle ::= \text{set}
| \text{list}

\langle \text{InverseTraversalPath} \rangle ::= \langle \text{Identifier} \rangle :: \langle \text{Identifier} \rangle

\langle \text{OptOrderBy} \rangle ::= \langle \text{Identifier} \rangle :: \langle \text{Identifier} \rangle
| \{ \text{order_by} \langle \text{ScopedNameList} \rangle \}

\langle \text{ScopedNameList} \rangle ::= \langle \text{ScopedName} \rangle
| \langle \text{ScopedNameList} \rangle , \langle \text{ScopedName} \rangle

\langle \text{OpDcl} \rangle ::= \langle \text{OptOneway} \rangle \langle \text{OperTypeSpec} \rangle \langle \text{Identifier} \rangle \langle \text{ParameterDcls} \rangle \langle \text{OptRaisesExpr} \rangle \langle \text{OptContextExpr} \rangle

\langle \text{OptOneway} \rangle ::= \text{oneway}
| \text{oneway}

\langle \text{OperTypeSpec} \rangle ::= \langle \text{SimpleTypeSpec} \rangle
| \text{void}

\langle \text{ParameterDcls} \rangle ::= \langle \text{ParamDclList} \rangle
| \langle \text{ParamDcl} \rangle

\langle \text{ParamDclList} \rangle ::= \langle \text{ParamDcl} \rangle
| \langle \text{ParamDclList} \rangle , \langle \text{ParamDcl} \rangle

\langle \text{ParamDcl} \rangle ::= \langle \text{ParamAttribute} \rangle \langle \text{SimpleTypeSpec} \rangle \langle \text{Declarator} \rangle

\langle \text{ParamAttribute} \rangle ::= \text{in}
| \text{out}
| \text{inout}

\langle \text{OptRaisesExpr} \rangle ::= \text{raises} \langle \text{ScopedNameList} \rangle

\langle \text{OptContextExpr} \rangle ::= \text{context} \langle \text{StringLiteralList} \rangle

\langle \text{StringLiteralList} \rangle ::= \langle \text{StringLiteral} \rangle
| \langle \text{StringLiteralList} \rangle , \langle \text{StringLiteral} \rangle

\langle \text{ExtRuleDcl} \rangle ::= \text{extrule} \langle \text{Identifier} \rangle \langle \text{ExtRuleSpec} \rangle

\langle \text{ExtRuleSpec} \rangle ::= \langle \text{ForAll} \rangle \langle \text{Identifier} \rangle \text{in} \langle \text{ExtRuleType} \rangle

\langle \text{ExtRuleType} \rangle ::= \langle \text{ExtRuleIsa} \rangle
| \langle \text{ExtRuleBottom} \rangle

\langle \text{ExtRuleBottom} \rangle ::= \langle \text{LocalClassName} \rangle \text{and} \langle \text{LocalClassName} \rangle \text{then} \langle \text{Identifier} \rangle \text{in bottom}

\langle \text{ExtRuleIsa} \rangle ::= \langle \text{LocalClassName} \rangle \text{then} \langle \text{Identifier} \rangle \text{in} \langle \text{LocalClassName} \rangle

\langle \text{RuleDcl} \rangle ::= \text{rule} \langle \text{Identifier} \rangle \langle \text{RuleSpec} \rangle
\langle \text{RuleSpec} \rangle \ ::= \langle \text{ForAll} \rangle \langle \text{Identifier} \rangle \text{ in } \langle \text{Identifier} \rangle : \langle \text{RuleBodyList} \rangle \text{ then } \langle \text{RuleBodyList} \rangle \\
| \langle \text{ForAll} \rangle : \langle \text{Identifier} \rangle \text{ in } \langle \text{Identifier} \rangle \} \\
| \langle \text{forall} \rangle \\
\langle \text{RuleBodyList} \rangle \ ::= ( \langle \text{RuleBodyList} \rangle ) \\
| \langle \text{RuleBody} \rangle \\
| \langle \text{RuleBodyList} \rangle \text{ and } \langle \text{RuleBody} \rangle \\
\langle \text{RuleBody} \rangle \ ::= \langle \text{DottedName} \rangle \langle \text{RuleConstOp} \rangle \langle \text{OptRuleCast} \rangle \langle \text{LiteralValue} \rangle \langle \text{DottedName} \rangle \langle \text{RuleConstOp} \rangle \langle \text{OptRuleCast} \rangle \langle \text{DottedName} \rangle \\
| \langle \text{DottedName} \rangle \text{ in } \langle \text{DottedName} \rangle \\
| \langle \text{ForAll} \rangle \langle \text{Identifier} \rangle \text{ in } \langle \text{DottedName} \rangle : \langle \text{RuleBodyList} \rangle \\
| \langle \exists \rangle \langle \text{Identifier} \rangle \text{ in } \langle \text{DottedName} \rangle : \langle \text{RuleBodyList} \rangle \\
| \langle \text{DottedName} \rangle = \langle \text{SimpleTypeSpec} \rangle \langle \text{Identifier} \rangle ( \langle \text{DottedLiteralList} \rangle ) \\
\langle \text{DottedLiteralList} \rangle \ ::= \langle \text{DottedLiteral} \rangle \\
| \langle \text{DottedLiteralList} \rangle , \langle \text{DottedLiteral} \rangle \\
\langle \text{DottedLiteral} \rangle \ ::= \langle \text{OptSimpleTypeSpec} \rangle \langle \text{DottedName} \rangle \\
| \langle \text{OptSimpleTypeSpec} \rangle \langle \text{LiteralValue} \rangle \\
\langle \text{OptSimpleTypeSpec} \rangle \ ::= \langle \text{SimpleTypeSpec} \rangle \\
| \langle \text{RuleConstOp} \rangle \ ::= = \\
| > = \\
| <= \\
| < \\
| > \\
\langle \text{LiteralValue} \rangle \ ::= \langle \text{SignedFloatingPtLiteral} \rangle \\
| \langle \text{SignedIntegerLiteral} \rangle \\
| \langle \text{CharacterLiteral} \rangle \\
| \langle \text{StringLiteral} \rangle \\
\langle \text{DottedName} \rangle \ ::= \langle \text{Identifier} \rangle \\
| \langle \text{DottedName} \rangle \cdot \langle \text{Identifier} \rangle \\
\langle \text{OptRuleCast} \rangle \ ::= ( \langle \text{SimpleTypeSpec} \rangle ) \\
\langle \text{CaseList} \rangle \ ::= \langle \text{CaseSpec} \rangle \\
| \langle \text{CaseList} \rangle \langle \text{CaseSpec} \rangle \\
\langle \text{CaseSpec} \rangle \ ::= \langle \text{LiteralValue} \rangle : \langle \text{DottedName} \rangle
\(\text{ThesaurusRelation} \ ::= \ \langle \text{LocalAttributeName} \rangle \ (\text{ThesRelType}) \\
\quad \langle \text{LocalAttributeName} \rangle \\
\ | \ \langle \text{LocalClassName} \rangle \ (\text{ThesRelType}) \ (\text{LocalClassName}) \\
\ | \ \langle \text{LocalClassName} \rangle \ (\text{ThesRelType}) \ (\text{LocalAttributeName}) \\
\ | \ \langle \text{LocalAttributeName} \rangle \ (\text{ThesRelType}) \ (\text{LocalClassName})
\)

\(\text{ThesRelType} \ ::= \ \text{syn} \\
\ | \ \text{bt} \\
\ | \ \text{nt} \\
\ | \ \text{rt}
\)

\(\text{Module} \ ::= \ \text{module} \ \langle \text{Identifier} \rangle \ \{ \ \langle \text{OdlSpecification} \rangle \ \} \)
Bibliography


[LCHH97] Wen-Syan Li, K. Selçuk Candan, Kyoji Hirata, and Yoshinori Hara. Facilitating multimedia database exploration through visual interfaces and perpetual query reformulations. In Matthias Jarke, Michael J. Carey,


Index

OLCD, 13, 38
0note.txt, 46

affinity
  global, 24
  name, 24
  structural, 25

architecture
  MOMIS, 43
ARTEMIS, 4
  Implementation, 74
  theory, 24

Baclawski, 85
base 64, 73
Base Extension, 33
Base Extensions, 33

category
  keynets, 91
classification
  keynets, 90
CLASSPATH, 49
Comparison Functions
  edges, 90
Comparison Functions
  definition, 89
  edges, 90
content label, 86

declarationsIDL, 46, 49
description logic, 13
Designer-supplied inter-schema relationships, 20

Extensional Hierarchy, 33
Extensional Relationships, 11

global affinity, 24
Global Attribute
  definition, 4
Global Attributes, 28

Global Class
  definition, 4
Global Schema, 61
Global View
  definition, 4
GlobalSchemaProxy, 49
Graph comparison, 89

information retrieval, 85
Integration Designer
  definition, 4

Join Maps, 35

Keynet
  Comparison Functions, 89
  Matching algorithm, 90
  The Keynet system, 86
  The Keynet system, 85
keynet
  Graph comparison, 89
  graphs, 86
  search engine, 88
keynet structure, 86
Keynets, 85
  definition, 85

ldir, 46
Lexical Matrix, 16
Lexicon-derived inter-schema relationships, 19
Local Attribute
  definition, 4
Local Attributes, 22
Local Class
  definition, 4
Local Classes, 22
Local Source
  definition, 4
log file, 47

Mapping Tables, 28
Matching algorithm, 90
modules
directory, 48
MOMIS
architecture, 43
GUI, 65
the momis system, 43
MomisObject, 51

name
resolution, 51
name affinity, 24

Object fusion
related works, 37
Object Fusion problem, 35
objects
Artemis, 75
ArtemisStatus, 76
Distance, 77
DistanceMap, 77
Symmetrical, 77

ODB-Tools, 4, 18, 20, 21, 59
CORBA interface, 59
XML, 61

odli3
package shared, 49

parser
odli3, 49
pid file, 47
Polysemy, 15
precision, 85

query refinement in keynets, 100
recall, 85

Relationships
Extensional, 11
relationships
Designer-supplied, 20
Lexicon-derived, 19
Schema-derived, 18
Validation, 20
relationships check, 21
relationships inference, 21
resolution post parsing, 51

save SI-Designer statues, 65, 72
Schema-derived relationships, 18
search engine for keynet, 88

Semantic network, 86
semistructured data, 9
serialisation
saving integration status, 73
shared
directory, 49
odli3, 49
parser, 49
SI-Designer, 65
MomisObject, 51
strategy, 72
test module, 80
similarity, 24
skeleton, 49
starting MOMIS
unix, 47
Windows, 47
status
SI-Designer, 65, 72
structural affinity, 25
stubs, 49
supergraph, 101
Synonymy, 15
test module
SI-Designer, 80
type
SI-Designer, 13
theory of graphs, 89
Tipster’s collections, 85

Union constructor, 13
Union constructor, 10, 12, 14

Validation
Relationships, 20
Virtual Schema
creation, 21, 53
virtual schema, 21

Word Form, 16
Word Meaning, 16

WordNet
interface, 57
server, 57
type, 15
wrappers, 54