Combining Description Logics and Object Oriented Models in an Information Integration Framework

Gergely Lukácsy and Péter Szeredi

Budapest University of Technology and Economics
Department of Computer Science and Information Theory
1117 Budapest, Magyar tudósok körútja 2., Hungary
Phone: +36 1 463-2585 Fax: +36 1 463-3157
`{lukacsy,szeredi}@cs.bme.hu`
Keywords: descripton logic, information integration, logic programming

Abstract. We present an information integration system called SINTAGMA which supports the semantic integration of heterogeneous information sources using a meta data driven approach. The main idea of SINTAGMA is to build a so called Model Warehouse, containing several layers of integrated models connected by mappings. At the bottom of this hierarchy there are the models representing the actual information sources. Higher level models represent virtual databases which can be queried, as the mappings provide a precise description of how to populate these virtual sources using the concrete ones.
The implementation of SINTAGMA uses constraints and logic programming, for example, the complex queries are translated into Prolog goals. This paper focuses on a recent development in SINTAGMA allowing the information expert to use Description Logic (DL) based ontologies in the development of high abstraction level conceptual models. Querying these models is performed using the Closed World Assumption as we argue that traditional Open World DL reasoning is less appropriate in the context of database oriented information integration environments.

1 Introduction

This paper presents the Description Logic modelling capabilities of the SINTAGMA Enterprise Information Integration system.

SINTAGMA is based on the SILK tool-set, developed within the EU FP5 project SILK (System Integration via Logic & Knowledge) [3]. SILK is a Prolog based, data centered, monolithic information integration system supporting semi-automatic integration on relational and semi-structured sources.

The SINTAGMA system extends the original framework in several directions. As opposed to the monolithic SILK structure, SINTAGMA is built from loosely coupled distributed components. The functionality has become richer as, among others, the system now deals with Web Services as information sources.

The present paper discusses a recent extension of the system which allows the integration expert to use Description Logic models in the integration process.

This paper is a revised and extended version of the paper presented at the ALPSWS '07 workshop in Porto [22]. It is structured as follows. Section 2 introduces description logic and logic programming. In Section 3 we give a general introduction to the SINTAGMA system, describing the main components, the SILan modelling language, and the query execution mechanism. In the next section we discuss the description logic extension of SILan: we introduce the syntactic constructs and the modelling methodology. Section 5 describes the execution mechanism used when querying Description Logic models. Section 6 presents a fairly complex example, demonstrating the tools and techniques we have discussed so far. In Section 7 we examine related work. Finally, we conclude with a summary of our results.

The examples we use in the upcoming discussions are part of the integration scenario described in detail in Section 6. This scenario represents a world where we attempt to integrate various information sources about writers, painters and their work (i.e. books, paintings, etc.) and present this information in the form of abstract views.

2 Background

Below we give a brief introduction to Description Logic and logic programming as these technologies form the basis of our work.

2.1 Description Logic

Description Logics (DL) [17] is a family of simple logic languages used for knowledge representation. DLs are used for describing the various kinds of knowledge for a selected field. The terminological system of a description logic knowledge base consists of *concepts*, which represent sets of *objects*, and *roles*, describing binary relations between concepts. Objects are the instances occurring in the modelled application field, and thus are also called *instances* or *individuals*.

A description logic knowledge base consists of two disjoint parts: the *TBox* and the *ABox*. The TBox (terminology box), in its simplest form, contains terminology axioms of form $C \sqsubseteq D$ (concept C is subsumed by D). The ABox (assertion box) stores knowledge about the individuals in the world: a concept assertion of form $C(i)$ denotes that i is an instance of concept C, while a role assertion $R(i, j)$ means that the objects i and j are related through role R.

Concepts and roles may either be *atomic* (referred to by a concept name or a role name) or *composite*. A composite concept is built from atomic concepts using *constructors*. The expressiveness of a DL language depends on the constructors allowed for building composite concepts or roles. Obviously there is a trade-off between expressiveness and inference complexity.

We use the language $\mathcal{ALCN}(\mathbf{D})$ in this paper. $\mathcal{ALCN}(\mathbf{D})$ concept expressions (often simply referred to as *concepts*) are built from role names, concept names (atomic concepts) and the top and bottom concepts ($\top$ and $\bot$) using the following constructors: intersection ($C \sqcap D$), union ($C \sqcup D$), negation ($\neg C$), value restriction ($\forall R.C$), existential restriction ($\exists R.C$) and number restrictions ($\geqslant n\, R$ and $\leqslant n\, R$). Here, C and D are concept expressions and R is a role name. The two kinds of number restrictions are jointly referred to as ($\bowtie n\, R$). In $\mathcal{ALCN}(\mathbf{D})$ we can also use concrete domains, such as integers or strings, when building concepts. For a detailed introduction to description logics we refer the reader to the first two chapters of [1].

2.2 Logic programming and Prolog

The main idea of Logic Programming is to use mathematical logic as a programming language. The execution of a logic program can be viewed as a reasoning process.

Prolog (Programming in Logic) [26] is the first and so far the most widely used logic programming language. Prolog uses Horn-clauses and SLD resolution [25] for reasoning. The basic elements of the Prolog execution process are procedure invocation based on unification and *backtracking* [28].

Prolog, and logic programming in general, is successfully used in several areas of computer science. These include natural language processing, planning, different kinds of reasoning systems, and information integration.

The notion of *term* is a principal concept of the Prolog language. It is either (a) a simple value (number, string) or (b) a variable or (c) a structure with a name and arbitrary number of arguments. These arguments are Prolog terms themselves. The name and the arity of a term together is referred as the *functor* of the term. A Prolog structure with three arguments can be seen below:

```
'Work:class:220'(DT, [A, B, C, D, E], _)                              (1)
```

Here the name of the structure is `'Work:class:220'`. The first and the third arguments are variables. These are denoted by identifiers starting with a capital or an underline. A single underline (`_`) is an *anonymous* variable, the value of which is of no interest. Multiple occurrences of such anonymous variables are considered different. The second argument of (1) is a structure in a special list notation. A list is actually a recursive structure `[Head|Tail]`, consisting of a `Head` (its first element) and a `Tail`, which is a list of the remaining elements. The list in the second argument contains five variables and is given in a simplified notation, i.e. `[A,B,C,D,E]`, in fact, corresponds to `[A|[B|[C|[D|[E|[]]]]]]`. Here `[]` represents an empty list (a list with no elements).

A Prolog program consists of a set of *clauses* of form *Head* `:-` *Body*, meaning *Head* is implied by *Body*. The *Head* is a term, while the *Body* is a term or a comma-separated sequence of terms. Here the comma denotes a conjunction. Clauses whose heads have the same functor are grouped together into *predicates*. The name of a predicate is the shared structure name of the heads of its clauses.

A Prolog *goal (query)* has the same form as a clause body. The execution of a goal wrt. a Prolog program succeeds if an instance of the goal can be deduced from the program. A goal can succeed multiple times, providing different variable substitutions as results. For example, let us consider the goal shown below.

```
'Writer:class:234'(ID), 'Painter:class:236'(ID)
```

This complex goal consists of two goals, separated by a comma. It succeeds if there is such an instantiation of variable `ID` under which both goals can be deduced from the given program (not shown here). The result of the execution is the enumeration of such `ID`s. Informally, this query enumerates those people who are writers and painters at the same time.

Further control constructs such as disjunction `(` *Goal1* `;` *Goal2* `)` and negation `\+`*Goal* are also supported by Prolog. The latter is the so called "negation by failure", which is not capable of enumerating solutions, but just checks if the execution of *Goal* fails. There is a wide range of built-in predicates, including ones for collecting all solutions of a goal (e.g. `bagof`). For example,

```
bagof(ID, ('Writer:class:234'(ID), 'Painter:class:236'(ID)), IDs)
```

will collect the identifiers of all people who are writers and painters into the list `IDs`. An important property of `bagof` is that it can return multiple solutions if not all variables in its second argument appear in the first. For example, consider a predicate `edge` describing the edges of a directed graph:

```
edge(a,b).   edge(a,c).   edge(c,d).   edge(d,a).   edge(c,e).
```

By invoking the goal `bagof(End, edge(Start, End), EndPoints)` we collect the endpoints of the edges. This goal will produce three answers, one for each possible value of variable `Start`:

```
Start = a, EndPoints = [b,c]
Start = c, EndPoints = [d,e]
Start = d. EndPoints = [a]
```

More about the Prolog language can be read in the ISO standard for Prolog [26] and in textbooks, such as [28,10].

3 SINTAGMA System Architecture

The overall architecture of the SINTAGMA system can be seen in Figure 1. The main idea of the system is to collect and *manage meta-information* on the sources to be integrated. These pieces of information are stored in the *Model Warehouse*, in the form of UML-like models [12], constraints and mappings. This way we can represent structural as well as non-structural information, such as class invariants, etc. The Model Warehouse resides in and is handled by the *Model Manager* component.

We use the term *mediation* to refer to the process of querying SINTAGMA models. Mediation decomposes complex integrated queries to simple queries answerable by individual information sources, and, having obtained data from

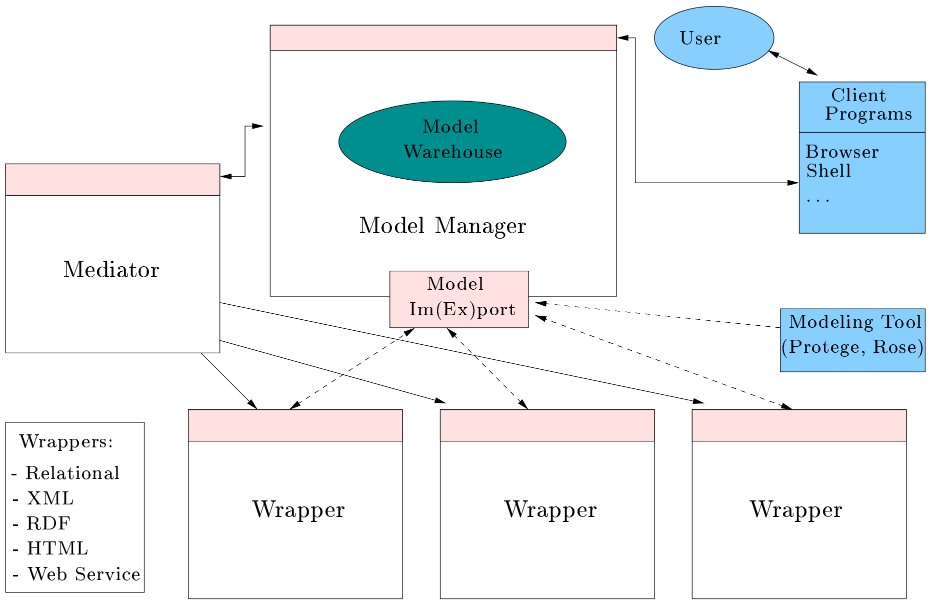


Fig. 1. The architecture of the SINTAGMA system

these, composes the results into an integrated form. Mediation is the task of the *Mediator* component.

Access to heterogeneous information sources is supported by *wrappers*. Wrappers hide the syntactic differences between the sources of different kinds, by presenting them to upper layers uniformly, as UML models. These models (called *interface models*) are entered into the Model Warehouse automatically. The following subsections give a brief description of the main SINTAGMA components.

3.1 The Model Manager

The Model Manager is responsible for managing the *Model Warehouse (MW)* and providing integration support, such as model comparison and verification (not covered in this paper). Here we focus on the role of the Model Warehouse.

The content of the MW is given in the language called *SILan* which is based on UML [12] and Description Logics [17]. The syntax of SILAN resembles IDL, the Interface Description Language of CORBA [19]. We demonstrate the knowledge representation facilities of SINTAGMA by a simple SILan example showing the relevant features of the meta-data repository (Figure 2).

The example describes the model `Art` containing two classes, `Artist` and `Work`. It also contains an association `hasWork` between artists and their works. We will explain the details of this example below.

```
 1 model Art {
 2     class Artist: BuiltIns::DLAny {
 3         attribute String name;
 4         attribute Integer birthDate;
 5         constraint self.creation.date > 1900;
 6     };
 7
 8     class Work: BuiltIns::DLAny {
 9         attribute String title;
10         attribute String author;
11         attribute Integer date;
12         attribute String type;
13         primary key title;
14     };
15
16     association hasWork {
17        connection Artist as creator;
18        connection Work as creation;
19     };
20 };
```

Fig. 2. SILan representation of the model `Art`

Semantics of SILan models The central elements of SILan models are classes and associations, since these are the carriers of information. A class denotes a set of entities called the *instances* of the class. Similarly, an n-ary association denotes a set of n-ary tuples of class instances called *links*.

Classes can have *attributes* which are defined as functions mapping the class to a subset of values allowed by the type of the attribute. Classes can inherit from other classes. All instances of the descendant class are instances of the ancestor class, as well. In our example both `Artist` and `Work` inherit from the built-in class `BuiltIns::DLAny`[1] (cf. lines 2 and 8). See Section 4.3 for more details.

Associations have *connections*, an n-ary association has n connections. In an association some of the connections can be named, providing intuitive navigation. For example, the connections of association `hasWork`, corresponding to classes `Artist` and `Work`, are called `creator` and `creation`, respectively (lines 17–18).

Classes can have a primary key, composed of one or more attributes. This specifies that the given subset of the attributes uniquely identifies an instance of the class. In our example, as a gross simplification, attribute `title` serves as a key in class `Work`, i.e. there cannot be two works (books, for example) with the same title.

[1] In SILan double colons (::) separate the model name from the name of its constituent (class, association, etc.).

Finally, invariants can be specified for classes and associations using the object constraint extension of UML, the OCL language [9]. Invariants give statements about instances of classes (and links of associations) that hold for each of them. The constraint in the declaration of `Artist` (line 5) is an invariant stating that the publication date of each work of an artist is greater than `1900`[2]. The identifier `self` refers to an arbitrary instance of the context, in this case the class `Artist`. Then two *navigation* steps follow. In the first step we navigate through the association `hasWork` to an *arbitrary* piece of work of the artist, while in the second step we go from the work to its publication date, and finally state that this date is always greater than `1900`.

In addition to the object oriented modelling paradigm, the SILan language also supports constructs from the Description Logic (DL) world [17]. This recently added feature of SINTAGMA is discussed in Section 4.

Abstractions For mediation, we need mappings between the different sources and the integrated model. These mappings are called *abstractions* because they often provide a more abstract view of the notions present in the lower level models. An example abstraction called `w0` can be seen in Figure 3.

```
 1  abstraction w0 (m0: Interface::Product,
 2                  m1: Interface::Description
 3               -> m2: Art::Work) {
 4
 5    constraint
 6      m1.id = m0.id and
 7      m1.category = "artwork"
 8    implies
 9      m2.title = m0.name and
10      m2.author = m0.creator and
11      m2.date = m0.creation_date and
12      m2.type = m1.subcategory and
13      m2.DL_ID = m0.name;
14  };
```

Fig. 3. SILan representation of the abstraction populating class `Work`

This abstraction populates the class `Work` (cf. Figure 2) in the model `Art` using classes `Product` and `Description`, both from the model `Interface` (lines 1–3). This means that the abstraction specifies how to create a "virtual" instance of class `Work`, given that the other two classes are already populated (e.g. they correspond to real information sources). In lines 1–3 the identifiers `m0`, `m1` and

[2] This may be so because the underlying information sources are known to be dealing with works of art of 20th century or later.

m2 are declared, and these will be used throughout the abstraction specification to denote instances of the appropriate classes.
The abstraction describes that given an instance of class Product called m0 and an instance of class Description called m1, for which the conditions in lines 6–7 hold, there exists an instance m2 of class Work with attribute values specified by lines 9–13[3]. Note that line 6 specifies that the id attributes of the two instances have to be the same, and thus corresponds to a relational *join* operation. In our integration scenario (see Section 6) Product and Description actually correspond to real-world Oracle tables containing various products and their descriptions, including books and paintings.

These two sources share the key id (line 6). While the first one supplies four fields to Work objects (title, author, date and DL_ID), the contribution of the second one is a single field (type). However, this second table has information to ensure that only relevant products (works of art) are included in class Work, through the condition in line 7.

We note that other abstractions can also populate class Work. In this case the set of instances of Work will be the union of the instances produced by the appropriate abstractions. Note that if a new information source is added, we only have to specify a new abstraction corresponding to this source, while the existing abstractions do not have to be modified.

Notice that the abstraction in Figure 3 takes the form of an implication describing how the given sources can contribute to populating the high level class Art::Work. This is characteristic of the Local as View integration approach [6].

3.2 The Wrappers

Wrappers provide a common interface for accessing various information source types, such as relational and object-oriented databases, semi-structured sources (e.g. XML or RDF), as well as Web-services.

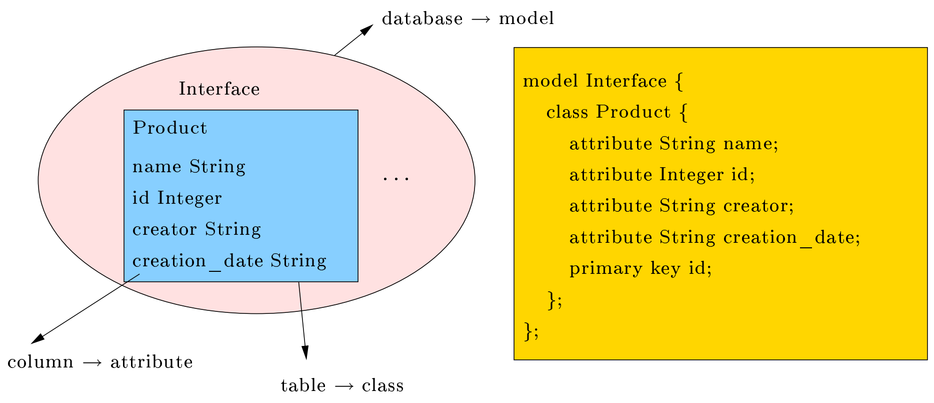
A wrapper has two main tasks. First, it extracts meta-data from the information source and delivers these to the Model Manager in the form of SILan models. For example, in case of relational sources, databases correspond to models, tables to classes, columns to attributes, as shown in Figure 4.

The other principal task of a wrapper is to transform queries, formulated in terms of this interface model, into the format required by the underlying information source, and thus allow for running queries on the sources.

3.3 The Mediator

The Mediator [2] supports queries on high level model elements by decomposing them into interface model specific questions. This is performed by creating a query plan satisfying the data flow requirements of the sources. During the execution of this query plan the data transformations described in the abstractions

[3] Attribute DL_ID comes from the class DLAny, of which class Work is a descendant. It has a special role, as explained in Section 4.3.



```
model Interface {
  class Product {
    attribute String name;
    attribute Integer id;
    attribute String creator;
    attribute String creation_date;
    primary key id;
  };
};
```

Fig. 4. Modelling relational sources in SILan

are carried out. Whenever we query a model element in SINTAGMA, the Model Manager provides the following two kinds of information to the Mediator:

1. the query goal itself, i.e. a Prolog term representing what to query;
2. set of mediator rules, using which the Mediator can decompose the complex query into primitive ones (i.e. queries that refer only to interface models).

For example, let us consider the query shown below involving class `Work`.

```
query RecentWork
  select *
  from w: Art::Work
  where w.date > 2000;
```

This query is looking for recent works, namely those instances of the class `Art::Work` that were created after 2000[4]. In this case, the query goal is similar to the following simple Prolog expression:

```
:- 'Work:class:220'(DT, [A, B, C, D, E], DA), C > 2000.          (2)
```

Here, the first Prolog goal retrieves an instance of `Art::Work`. The variables in this term will be instantiated during query execution. The predicate name `'Work:class:220'` is a concatenation of three strings: the kind of the model element (`class`) and its unique internal identifier (220), preceded by the unqualified—and thus non-unique—SILan name (`Work`), provided for readability. Model elements are often referred to by *handles* of form *Kind(Id)*, e.g.

[4] We could have created a class named `RecentWork` and populated it by an appropriate abstraction. Then, instead of formulating a SILan query, we could have simply directly asked for the instances of this class. The question whether to use a query or an abstraction is a modelling decision.

`class(220)`. Note that the above predicate name represents the *static type* of the instances queried for, as opposed to the *dynamic type* which can be different, if the returned object belongs to a descendant class of `Work`.

The dynamic type of the queried instance, i.e. the handle of the *most specific* class it actually belongs to, is returned in the first argument of the goal. The second argument contains the values of the static attributes, in this case we have five such variables (cf. declaration of class `Work` in Figure 2). For example, `C` denotes the value of the attribute `date`. The third and last argument of the query term carries the values of the dynamic attributes. These represent the additional attributes (not known at query time) of the instance if it happens to belong to a *descendant* class of `Art::Work`.

The second part of the query goal corresponds to a simple arithmetic OCL constraint, which uses variable `C` representing the `date` attribute of the work in question.

The mediator rules representing the abstraction `w0` shown in Figure 3 take the following form:

```
'Product:class:190'(_,[Title,Id,Author,Date],_),
'Description:class:191'(_,["artwork",Id,Type],_) --->
     'Work:class:220'(class(220),[Title,Title,Author,Date,Type],[])
```

The specific rule above describes how to create an instance of the class `Work` whenever we have two appropriate instances of classes `Product` and `Description` available. If there were more abstractions, the Mediator would get more rules as there would be more than one possible way to populate the given class.

Note that the mediator rules are also used to describe inheritance between model elements. In such a case the dynamic type of the model element on the right hand side of the rule is a variable (as opposed to the constant `class(220)` above). This variable is the same as the dynamic type of the model element on the left hand side. The dynamic attributes are propagated similarly.

Finally, let us state that an n-ary association is implemented as an n-ary relation, each argument of which is a ternary structure corresponding to a class instance, similar to the first goal of (2). For example, a query goal for the association `hasWork` (cf. Figure 2) has the following form:

```
:- 'hasWork:association:227'(
        'Artist:class:218'(DT1,[DL_ID1,Name,Birthdate],DA1),       (3)
        'Work:class:220'(DT2,[DL_ID2,Title,Author,Date,Type],DA2)
    ).
```

4 DL modelling in SINTAGMA

Let us now introduce the new DL modelling capabilities of the SINTAGMA system. First we discuss why we need Description Logic models during the integration process and provide an introductory example. Then we present the DL constructs supported by our system and discuss the restrictions we place on their usage. Finally, we summarise the tasks of the integration expert when using DL elements during integration.

4.1 An introductory example

In the Model Warehouse we handle models of different kinds. We distinguish between *application* and *conceptual models*. The application models represent existing or virtual information sources and because of this they are fairly elaborate and precise. Conceptual models, however, represent mental models of user groups, therefore they are vaguer than the application models.

Our experience shows that to construct such models it is more appropriate to use some kind of ontological formalism instead of the relatively rigid object oriented paradigm. Accordingly, we have extended our modelling language to incorporate several description logic constructs, in addition to the UML-like ones described earlier. In the envisioned scenario, the high-level models of the users are formulated in description logic and via appropriate definitions they are connected to lower-level models. Mediation for a conceptual model follows the same idea we use for any other model: the query is decomposed, following the definitions and abstractions, until we reach the interface models (in general, through some further intermediate models) which can be queried directly.

Before going into the details, we show an example to illustrate the way how DL descriptions are represented in SILan (note that `Writer` and `Painter` are both descendants of class `Artist`, but otherwise they are normal UML classes; we will present more details about these classes in Section 6).

```
model Conceptual {
  class WriterAndPainter {};
  constraint equivalent {                                    (4)
         WriterAndPainter,
         Unified::Writer and Unified::Painter};
};
```

Here we define the class `WriterAndPainter` by providing a SILan constraint. This constraint can be placed anywhere in the Model Warehouse: in the example above we simply put it in the very model that declares the class `WriterAnd Painter` itself. The constraint actually corresponds to a DL *concept definition axiom*: $\mathsf{WriterAndPainter} \equiv \mathsf{Writer} \sqcap \mathsf{Painter}$. Namely, it states that the instances of class `WriterAndPainter` are those (and only those) who belong to the unnamed class containing the individuals who are both writers and painters. Thus, DL concepts are defined using the Global as View approach [6], as opposed to the Local as View techniques applied in populating high-level classes using abstractions (cf. Section 3.1).

Note that the class `WriterAndPainter` could be created without DL support. However, in that case the integration expert would have to go through a much more elaborate process of creating the high level class `WriterAndPainter`, specifying all its attributes and populating it with an appropriate abstraction. This abstraction would have to implement the constraint (4), through an appropriate join-like operation.

Now, with DL support, the expert simply formulates a very short and intuitive DL axiom. We argue that this is easier for the expert to do, and it also makes the content of the Model Warehouse more readable to others.

4.2 DL elements in SILan

From the DL point of view, SINTAGMA supports acyclic Description Logic TBoxes containing only concept *definition axioms*, which are formulated in an extension of the $\mathcal{ALCN}(\mathbf{D})$ language (see more below about the extension). Only single atomic concepts, so called *named symbols* can appear on the left hand side of the axioms, such as `WriterAndPainter` in example (4). The remaining atomic concepts, not appearing on the left hand side are called *base symbols*. Such a TBox is definitorial, i.e. the meaning of the base symbols unambiguously defines the meaning of the named symbols. The base symbols, in our case, correspond to normal SINTAGMA classes and associations, e.g. `Writer` and `Painter` in the example (4). The ABox is a set of concept and role assertions, as determined by the instances of the classes which correspond to the base symbols participating in the TBox.

The DL concept constructors supported by SINTAGMA and their SILan equivalents are summarised in Table 1. Note that this table actually describes the possible concept formats on the right hand side of a definition axiom, assuming that we have *expanded* the TBox[5].

The only non-classical DL element in Table 1 is the *concrete domain restriction* (the last line in the table). Such a restriction specifies a subset of instances of the base concept A for which the given OCL constraint holds. This is a generalisation of the idea of concrete domains in the Description Logics world. Below we show an example of a concrete SILan restriction describing those works whose type (i.e. the value of the attribute `type`) is "`painting`".

```
class constraint Art::Work satisfies self.type="painting"
```

The reason we allow only concept *definition* axioms is that we aim to use DL concepts to describe executable high-level views of information sources. In this sense a DL concept is actually a syntactic variant of a SILan query or a SILan class populated by an abstraction.

Note that this also implies that we use the Closed World Assumption (CWA) in DL query execution. We argue that this is appropriate because of the following three reasons. First, CWA automatically ensures that our DL constructs are

[5] The expanded version of an acyclic TBox is obtained by repeatedly replacing every named symbol on the right hand side of an axiom by its definition. This process is repeated until no further named symbols are left on the right hand side. The fact that the TBox is acyclic ensures the termination of this process.

Name	**DL Syntax**	**SILan equivalent**
Base concept	A	UML class
Atomic role	R	UML association
Intersection	$C \sqcap D$	`C and D`
Union	$C \sqcup D$	`C or D`
Negation	$\neg C$	`not C`
Value restriction	$\forall R.C$	`slot constraint R all values C`
Existential restriction	$\exists R.C$	`slot constraint R some value C`
Number restriction	$\bowtie nR$	`slot constraint R cardinality i..j`
Top	$\top$	`DLAny`
Bottom	$\perp$	`DLEmpty`
Concrete restriction	—	`class constraint A satisfies OCL`

Table 1. DL-related constructs supported in SILan

semantically compatible with other constructs in the SINTAGMA system. Second, we argue that the Open World Assumption(OWA) is applicable when we have only partial knowledge and would like to determine the consequences of this knowledge, true in every universe in which the axioms of this partial knowledge hold. In contrast with this, in the context of information integration, our users would like to consider a single universe, in which a base concept or a role denotes *exactly* those individuals (or pairs of individuals) which are present in the corresponding database. To illustrate this issue, let us consider the following example: the concept of novice painter is defined to contain painters having at most 5 paintings (for example, being a novice painter may be a precondition for a government grant). To model this situation, the integration expert creates the DL axiom shown below.

$$\mathsf{NovicePainter} \equiv \mathsf{Painter} \sqcap (\leqslant 5\, \mathsf{hasPainting})$$

However, querying this concept, using OWA, will provide no results in general, as an open world reasoner would return an individual only if it is *provable* that it has no more than 5 paintings. Practically, this is not what the information expert wants.

The third reason why we decided to use the closed world assumption is the fact that we envisage handling huge amounts of data in the underlying databases. Traditional, tableau based DL reasoners do not cope well with large ABoxes [15].

Resolution based DL proving techniques [18] do much better, but they are either still not fast or not expressive enough [24]. By using CWA we can implement DL queries using the well researched, efficient database technology.

4.3 Modeling methodology and tasks of the integration expert

The integration expert is responsible for creating the DL axioms. Although these are represented in SILan within the SINTAGMA system, the expert can use any available OWL editor to create OWL descriptions. These descriptions then can be loaded by the OWL importer of the SINTAGMA system that basically realises an OWL-SILan translation (cf. the "Model Im(Ex)port" box in Figure 1).

One thing the expert should take care of is to match the names of the base symbols and the corresponding SINTAGMA classes and associations. This is often done in two steps: first the integration expert creates concept definition axioms using the widely accepted terminology of the domain, not paying attention to the names of the model elements in the Model Warehouse. Next, the expert provides additional definition axioms for each base symbol connecting it with the proper model element. For example, we could use names `A` and `B` instead of `Writer` and `Painter` in (4), provided that we also encode in SILan the equivalents of the following DL axioms:

$$\mathtt{A} \equiv \mathtt{Writer}$$
$$\mathtt{B} \equiv \mathtt{Painter}$$

A further crucial issue is to decide how to identify the instances of the base concepts, e.g. the instances of the class `Writer` and class `Painter`. Without this, it is not possible to determine the instances of class `WriterAndPainter`.

In a traditional DL ABox, an instance has a name that unambiguously identifies it. In SINTAGMA, similarly to databases, an instance is identified by the subset of its attribute values. For example, two writers could be considered to be the same if their names match, assuming that `name` is a key in class `Writer`.

The problem is that such keys are fairly useless when we compare instances of different data sources. This is because, in general, we cannot draw any direct conclusion from the relation of the keys belonging to instances from different classes. For example, databases containing employees often use numeric IDs as keys. Having two employees from different companies with the same ID does not mean that we are talking about the same person. Similarly, if the IDs of the employees do not match, they are not necessarily different persons.

What we need is some kind of shared key that uniquely identifies the instances of the classes participating in DL concept definitions. Luckily, the object-oriented paradigm we use in SINTAGMA provides a nice way to have such identifiers.

We have mentioned earlier that in SINTAGMA the notion of DL concept is a syntactic variant of SINTAGMA class. This also means that the result of a DL query is an ordinary instance that has to belong to some class(es). For example, when we are looking for the instances that are elements of both classes `Writer` and `Painter` we are actually interested in an *artist* instance belonging to these classes simultaneously. This is true in general: whatever DL concept constructs we use to describe a DL concept the result must belong to some class that is a common ancestor (in terms of inheritance) of the classes involved.

Instead of asking the integration expert to define such common ancestor classes in an ad hoc way, we introduce the built-in class `DLAny`. This class corresponds to the DL concept top ($\top$) and it has only one attribute called `DL_ID`, which is a key. We require that all the classes participating in DL concept definitions are the descendants of `DLAny`[6] (cf. lines 2 and 8 of Figure 2). Because of the properties of inheritance, attribute `DL_ID` will be a key in all of the descendant classes, i.e. it will exactly serve as the global identifier we were looking for.

Now, the task of the integration expert is to assign appropriate values to the `DL_ID` attributes: she needs to extend the existing abstractions populating the base symbols (classes) to also consider the attribute `DL_ID`. By appropriate values we mean that the `DL_ID`s of two instances should match if these instances are the same, and should differ otherwise. An example for this can be seen in Figure 5 populating the class `Writer`, which is part of a bigger integration scenario to be shown later in Section 6.

```
 1 abstraction ap (m0: Interface::Member ->
 2                 m1: Unified::Writer) {
 3
 4    constraint let n = m0.fname.concat(" ").concat(m0.lname) in
 5      m1.name = n and
 6      m1.birthDate = m0.date and
 7      m1.member_id = m0.iwa_id and
 8      m1.style = m0.style and
 9      m1.DL_ID = n;
10 };
```

Fig. 5. Populating the `DL_ID` attribute of a base concept

This abstraction populates the class `Writer` from an interface class called `Member` (lines 1–2), which represents a membership database of an imaginary "International Writer Association" (IWA). Let us assume that the members of this association have some kind of a unique identifier, such as the membership number, present in the underlying database. It may be worth bringing this key to the class `Writer` (line 7) as it makes possible to find writers efficiently if they happen to be IWA members. However, the unique identifier from the DL point of view has to be different: in fact it is the concatenation of the first and last name of the writer, with a space in between (lines 4 and 9).

[6] Note that this is a necessary condition. As for any concept C, $C \sqsubseteq \top$ holds, any DL instance has to belong to the class corresponding to $\top$, i.e. to `DLAny`.

This is because the class `Writer` can also be populated from other sources (e.g. `Person`, see Figure 8) where the IWA number makes no sense and so the `member_id` attribute is set to `"n/a"`. Furthermore, we may want class `Writer` to be a descendant of class `Artist` (cf. Figure 8), together with some other classes, such as `Painter`. This requires a key that can be computed from all the underlying sources, such as the name of the artist[7].

To summarise, the integration expert has to perform the following tasks when DL modelling is used during the integration process:

1. declare DL classes and for each provide corresponding definition axioms;
2. ensure that each base concept appearing in the definition axioms is:
   (a) inherited from class `DLAny`,
   (b) populated properly, i.e. its `DL_ID` attribute is filled appropriately.

5 Querying DL models in SINTAGMA

Now we turn our attention to querying DL concepts in SINTAGMA. As described in Section 3.3 our task is to create a *query goal* and a set of *mediator rules*. When we query a DL class, mediator rules are only generated for the base symbols. As these are ordinary classes and associations, this process is exactly the same as the one we use for cases without any DL construct involved. This means that we can now focus on the construction of the query goal.

Recall that a SINTAGMA instance is characterised by three properties, as exemplified by (2) on page 9: its dynamic type `DT`, its static attributes `SA` and its dynamic attributes `DAs`. Below we will use the variable name `As` to denote the full attribute list of an instance, i.e. the concatenation of the static and dynamic attribute values, with the exclusion of `DL_ID`.

A DL class has only a single static attribute, the `DL_ID` key. However, in contrast with an object oriented query, a DL query may return an answer that has *multiple dynamic types*. For example, when we enumerate the class `WriterAndPainter` we get instances that belong to both classes `Writer` and `Painter` (something which is not possible in the standard UML modelling). Accordingly, an answer to a DL query takes the form of a pair `(ID, DTA)`, where `ID` is the `DL_ID`[8] containing the unique name of the DL instances (see Section 4.3), while `DTA` is a Prolog structure containing the dynamic types of the answer, each paired with the corresponding full attribute list. The `DTA` structure is thus either a single `DT-As` pair, or recursively, two `DTA` structures joined using the comma operator: (DTA$_1$, DTA$_2$).

Figure 6 describes the mapping from an arbitrary DL concept expression to the corresponding query goal. Here we define a function Φ_C which, given an arbitrary concept expression C, returns the corresponding query goal with two arguments, `ID` and `DTA`. We define this function by considering the DL concept constructors, as listed in Table 1.

[7] This is also a simplification. More realistically, the key could be the name together with the birth date.

[8] We use the name `ID` instead of `DL_ID` for conciseness.

$$
\begin{aligned}
\Phi_A(\texttt{ID, DTA}) &= A^N\texttt{(DT, [ID|SAs], DAs), DTA = DT-(SAs } \oplus \texttt{ DAs)}\\
\Phi_{C\sqcap D}(\texttt{ID, DTA}) &= \Phi_C(\texttt{ID, DTA}_1\texttt{), }\Phi_D(\texttt{ID, DTA}_2\texttt{), DTA = (DTA}_1\texttt{, DTA}_2\texttt{)}\\
\Phi_{C\sqcup D}(\texttt{ID, DTA}) &= (\Phi_C(\texttt{ID, DTA) ; }\Phi_D(\texttt{ID, DTA}))\\
\Phi_{\neg C}(\texttt{ID, \_}) &= \texttt{\textbackslash+ }\Phi_C(\texttt{ID, \_})\\
\Phi_{\exists R.C}(\texttt{ID, DTA}) &= R^N\texttt{(}R^N_D\texttt{(DT, [ID|SAs], DAs), }R^N_R\texttt{(\_, [ID}_2\texttt{|\_], \_)),}\\
&\qquad \Phi_C(\texttt{ID}_2\texttt{, \_), DTA = DT-(SAs } \oplus \texttt{ DAs)}\\
\Phi_{\forall R.C}(\texttt{ID, DTA}) &= \Phi_{R_D}(\texttt{ID, DTA}),\\
&\qquad \texttt{\textbackslash+ (}R^N\texttt{(}R^N_D\texttt{(\_, [ID|\_], \_), }R^N_R\texttt{(\_, [ID}_2\texttt{|\_], \_)),}\\
&\qquad\quad \Phi_{\neg C}(\texttt{ID}_2\texttt{, \_))}\\
\Phi_{\bowtie nR}(\texttt{ID, DTA}) &= \texttt{bagof(Y, }R^N\texttt{(X, Y), Ys),length(Ys, S),condition}_{\bowtie}\texttt{(n, S),}\\
&\qquad \texttt{X = }R^N_D\texttt{(DT, [ID|SAs], DAs), DTA = DT-(SAs } \oplus \texttt{ DAs)}\\
\Phi_{\top}(\texttt{\_, \_}) &= \texttt{true}\\
\Phi_{\bot}(\texttt{\_, \_}) &= \texttt{false}
\end{aligned}
$$

Fig. 6. Transforming DL constructs into query goals

Let us consider the cases one by one. If we have a base class, we simply create a query term representing the instances of the class, similar to the one in goal (2) and then convert the attributes retrieved to the required form (`DTA`). Here operation $\oplus$ denotes the compile time concatenation of lists[9], while A^N stands for the predicate name corresponding to concept A. For example, $\texttt{Work}^N$ = `'Work:class:220'`, cf. (2) on page 9. Note that in the second argument of the query goal A^N we make use of the fact that the `DL_ID` attributes are always placed first in the static attribute list of an instance.

If we have the intersection of two concepts C and D, we recursively transform concepts C and D and put them in a Prolog conjunction. The `DTA` structure is built from the structures recursively obtained from the execution of the transformations of concepts C and D. Note that the resulting structure may contain duplicates, i.e. the same `DT-As` pair may be found in `DTA` more than once. These duplicates are only removed at the top level, i.e. when the final result of a query is presented. The transformation of union concepts is similar to the intersection: we create a Prolog disjunction.

Negation $\neg C$ is implemented by using the Prolog negation-as-failure. This translation is only capable of *checking* whether a given instance with `ID` belongs to concept C or not. As usual in the database context, we restrict the use of negation to cases where negated queries appear only in conjunction with at least

[9] The $\oplus$ operator is used only with a static attribute list (`SAs`). For any given base class, the length of the corresponding `SAs` is fixed (the number of static attributes excluding the `DL_ID`). Therefore, the `SAs` $\oplus$ `DAs` concatenation can be carried out at compile time.

one non-negated query. In terms of DL concept expressions this means that negated concepts have to appear either in the scope of a quantifier, or in an intersection together with at least one non-negated concept. It is the task of the Mediator to find an appropriate order in the final query plan where negation appears in a place where `ID` is instantiated [5]. The Mediator refuses to execute the query if such an order does not exist.

The next two cases involve associations. On the right hand side of these formulas R^N denotes the predicate name corresponding to the association itself. R_D (R_R) denotes the base class that is the domain (range) of association R. Correspondingly, R_D^N and R_R^N stand for the predicate names of the classes R_D and R_R, respectively[10]. Recall that a binary association is represented by a binary relation with ternary structures as arguments, as in (3).

The existential restriction $\exists R.C$ is simply transformed to a query of the association R and the concept C.

The goal corresponding to a value restriction $\forall R.C$ first enumerates the domain of R and then uses double negation to ensure that the given instance has no R-values which do not belong to C. Note that $\Phi_{\neg C}$`(ID2, _)` is invoked only when `ID2` is already instantiated.

A number restriction ($\bowtie nR$) is transformed into a goal which uses the Prolog built-in predicate `bagof` (cf. Section 2.2, page 4) to enumerate the instances in the domain of R together with the number of R-values connected to them, and then simply applies the appropriate arithmetic comparison.

The last two lines of Figure 6 define the transformation of the top and bottom concepts. $\top$ is mapped into `true`, while $\bot$ to `false`. Querying these concepts on their own does not make sense, but these mappings are useful when transforming DL concepts such as $\exists R.\top$ or $\forall R.\bot$.

Having described the transformation of DL concepts to query goals, we now deal with the only remaining construct: the concrete restriction. A concrete restriction involving a base concept A and an OCL constraint O is transformed in a straightforward way into the query goal as shown below[11]:

Φ_A`(ID, DTA), DTA = DT-AT,` Ψ_O`(ID, AT)`

To illustrate the general algorithm, two example transformations are presented in Figure 7. The first one shows the translation of the `WriterAndPainter` class described in (4) on page 11. The query goal is a conjunction that consists of three goals. The first two goals enumerate the instances of classes `Writer` and `Painter` with a condition that their `ID` attributes match. At this point we have identified those instances who are writers and painters at the same time. The last goal constructs the structure `DTA`, describing the dynamic types and the corresponding attribute values of the given instances.

[10] For example, if $R =$ `hasWork`, cf. Figure 2, then $R^N =$`'hasWork:association:227'`, $R_D^N =$`'Artist:class:218'` and $R_R^N =$`'Work:class:220'`.

[11] Ψ_O`(ID, AT)` denotes the Prolog translation of the OCL constraint O. This is a feature which has already been present in earlier versions of SINTAGMA, before the introduction of the DL extensions, see [3].

In the second example we look for a writer who has at least one piece of modern work. This DL concept involves the association `hasWork` and a class `Modern` (representing, say, contemporary pieces of art). The query goal becomes a bit more complex than in the first example: now it consists of four goals. The first goal enumerates the instances of class `Writer`. The second and the third goals filter out those writers that do not have any modern works. Here we have used the facts that the domain of `hasWork` is the class `Artist` and the range is the class `Work` (cf. Figure 2). Finally, the last goal builds the structure `DTA`.

```
Class to query: WriterAndPainter
DL definition:  Writer ⊓ Painter
Query goal:     'Writer:class:234'(DT1,[ID,Name1,Birth1,IWA,Style],DA1),
                'Painter:class:236'(DT2,[ID,Name2,Birth2,Colour],DA2),
                DTA  = (DT1-[Name1,Birth1,IWA,Style|DA1],
                        DT2-[Name2,Birth2,Colour|DA2])

Class to query: ModernWriter
DL definition:  Writer ⊓ ∃hasWork.Modern
Query goal:     'Writer:class:234'(DT1,[ID,Name1,Birth1,IWA,Style],DA1),
                'hasWork:association:227'(
                       'Artist:class:218'(DT2,[ID,Name2,Birth2],DA2),
                       'Work:class:220(_,[ID2|_],_)),
                'Modern:class:237'(_,[ID2|_],_),
                DTA  = (DT1-[Name1,Birth1,IWA,Style|DA1],
                        DT2-[Name2,Birth2|DA2])
```

Fig. 7. Transformation examples

Note that if a writer has more than one piece of modern work, the transformation in Figure 7 enumerates the writer multiple times. This is because the second goal can succeed more than once, leaving a *choice point* [26]. In the present version of SINTAGMA these duplicates are removed at the top level only, before the query results are presented to the user. In future, we will consider a more efficient solution, utilising the Prolog pruning operators (conditionals or cuts) to eliminate the unnecessary choices.

Also note that in our example scenario attributes `Name1`, `Name2` and `Birth1`, `Birth2` will be instantiated to the same values, i.e. to the name and birth date of the modern writer. This is the consequence of the data representation we use in SINTAGMA, i.e. if an instance has multiple dynamic types, for each of them we supply all the attribute values.

6 A case study: artists

In this section we present a simple use case, where we focus on illustrating the DL extension of SINTAGMA. More complex traditional integration problems solved using SINTAGMA are discussed in other papers, for example in [21].

Figure 8 shows the content of our example Model Warehouse. Here we have four models on different abstraction levels.

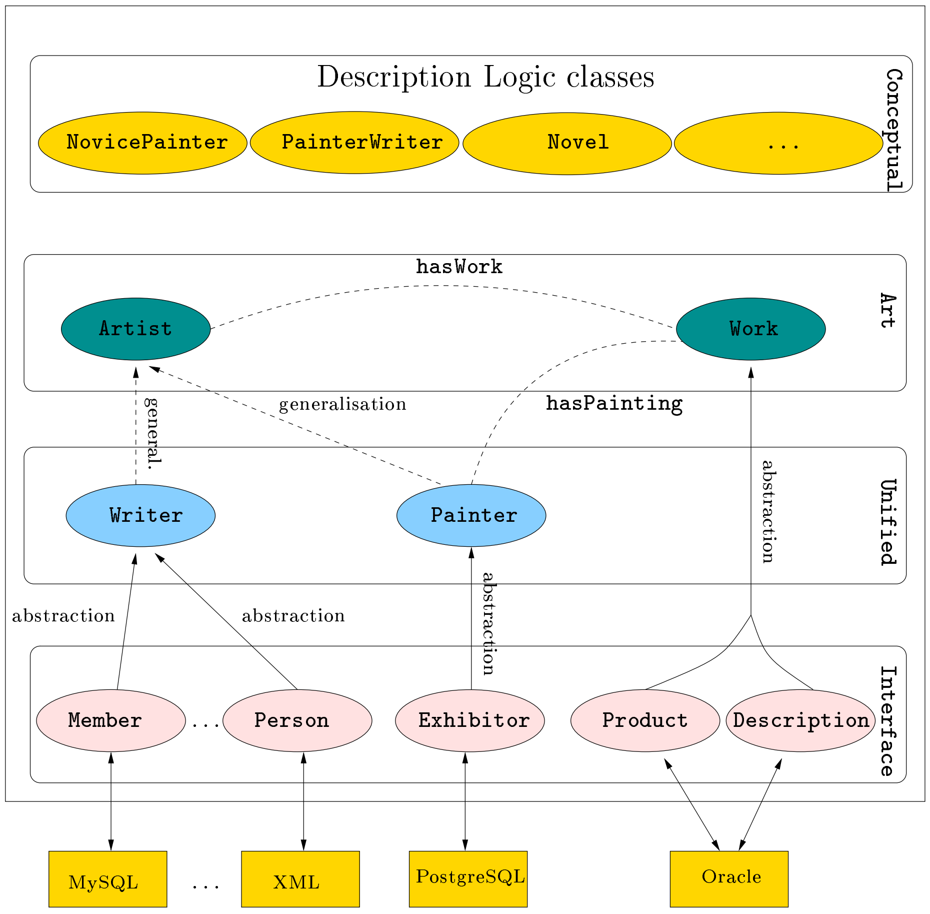


Fig. 8. Content of the Model Warehouse

The lowest one, `Interface`, contains classes directly corresponding to the information sources we aim to integrate. Class `Member` corresponds to some database table containing information about writers (members of a certain writers association), `Person` is the model of an XML source describing people (some of whom are possibly writers). We also have here class `Exhibitor` containing people some of whom are painters, and class `Product` containing art works among other products together with class `Description` which provides some information on products. These models are constructed automatically by different wrappers of the SINTAGMA system.

The next, more abstract model, called `Unified`, contains two classes `Writer` and `Painter`, their SILan descriptions are shown in Figure 9 (referring to class `Artist` introduced in Figure 2 in page 6). These classes provide a unified view of writers and painters over our heterogeneous information sources, i.e. querying `Writer` and `Painter` gives us all the known writers and painters respectively. These classes are populated by SILan abstractions: `Writer` by two, while `Painter` by only one. We can later extend our Model Warehouse to include more information sources on painters. This way `Painter` would also be populated by several abstractions. Please note how flexible this approach is: whenever we would like to add a new information source, all we have to do is to provide a new abstraction. This is fundamentally different from the way views are created in traditional database systems.

```
model Unified {

     class Writer: Art::Artist {
         attribute Integer member_id;
         attribute String style;
     };

     class Painter: Art::Artist {
          attribute String favourite_colour;
     };
};
```

Fig. 9. SILan description of classes `Writer` and `Painter`

The third model `Art` describes an even higher view of the underlying information sources. It contains two classes connected by an association. Class `Artist` is declared to be the generalisation of classes `Writer` and `Painter`, i.e. `Artist` is a common "parent" of `Writer` and `Painter`, in terms of inheritance. Accordingly, it contains the union of the instances of these classes. Class `Work` incorporates works (books and paintings). In the example, class `Work` is populated by only one abstraction. Association `hasWork` connects instances in class `Artist` with those in class `Work`, i.e. it allows us to navigate from an artist to her works. This association is populated by an abstraction (not shown in Figure 8) by creating virtual pairs from those instances of classes `Artist` and `Work` where the author of the work matches name of the artist.

Note that there is one more association in the Model Warehouse, called `hasPainting`. This association connects painters with their paintings and goes between different models. Similarly to `hasWork` this association is also populated by an abstraction, not shown here. Association `hasPainting` is used in the definition of `PainterWriter` (see below).

Up until now we have used the traditional features of SINTAGMA: classes, associations, generalisations, abstractions. Now we turn to the most abstract model, named `Conceptual`, which provides an even higher-level view of the information than the previous model.

The model `Conceptual` represents the knowledge of our specific example domain, in the form of DL concept definition axioms. These axioms form a simple ontology, a part of which is shown in Figure 10. This ontology talks about special types of artists, painters and writers. It states that a novice painter is a painter who has only painted no more than 5 paintings (axiom 1). Somebody is mostly writer if she is an artist who has produced at least 3 works, but has at most one painting (axiom 2). A productive writer has created at least 10 works (axiom 3). Somebody is painter-writer is she is a writer who has some paintings (axiom 4). Finally, a novelist is somebody who is only writing novels (axiom 5).

$$\mathsf{NovicePainter} \equiv \mathsf{Painter} \sqcap (\leqslant 5\, \mathsf{hasPainting}.\top) \tag{1}$$
$$\mathsf{MostlyWriter} \equiv \mathsf{Artist} \sqcap (\geqslant 3\, \mathsf{hasWork}.\top) \sqcap (\leqslant 1\, \mathsf{hasPainting}.\top) \tag{2}$$
$$\mathsf{ProductiveWriter} \equiv \mathsf{MostlyWriter} \sqcap (\geqslant 10\, \mathsf{hasWork}.\top) \tag{3}$$
$$\mathsf{PainterWriter} \equiv \mathsf{Writer} \sqcap (\exists \mathsf{hasPainting}.\top) \tag{4}$$
$$\mathsf{Novelist} \equiv \forall \mathsf{hasWork}.\mathsf{Novel} \tag{5}$$
$$\ldots \equiv \ldots$$

Fig. 10. An ontology describing artists, painters and writers.

In practice, such an ontology can be created by the information expert manually or can be imported from an existing ontology using the OWL importer component of the SINTAGMA system. In SINTAGMA this ontology is represented by a model containing classes with no attributes, together with the corresponding SILan constraints as shown below:

```
model Conceptual {
   class NovicePainter {};     class MostlyWriter {};
   class ProductiveWriter {};  class PainterWriter {};
   class Novelist {};          ...

   constraint equivalent {
       NovicePainter,
       Painter and {slot constraint hasPainting cardinality 0..5}
   };
   ...
};
```

Let us consider the base concepts used in our concept definitions in Figure 10. Most of these (i.e `Painter`, `Writer` and `Artist`), appear in the underlying UML models. However, there is the concept of `Novel`, which has no direct UML counterpart. This concept can be defined using a *concrete restriction* of SILan, as shown below.

```
constraint equivalent {
   Novel,
   {class constraint Art::Work satisfies self.type="novel"}
};
```

This concludes the description of our example models. Having encoded our DL axioms in terms of SILan constraints, we can now execute DL queries. For example, we can ask SINTAGMA to enumerate the instances of class `ProductiveWriter`. This query will produce instances similar to the following:

```
1 ('Lisa James',
2      [
3       'Writer'-['Lisa James', 1965, 42, 'fantasy'],
4       'Painter'-['Lisa James', 1965, 'red']
5      ]
6 )
```

Here, the string `'Lisa James'`, appearing in line 1, corresponds to the `ID` of Figure 6, i.e. the shared DL identifier. Lines 3–4 contain the list of the dynamic types and corresponding attributes of the instance. This specific instance has two dynamic types: she is a writer and a painter at the same time (lines 3 and 4). As a writer, she has a name, birth date, her membership ID and a style attribute. As a painter we also know her favourite colour.

7 Related work

The two main approaches in information integration are the Local as View (LAV) and the Global as View (GAV) [6]. In the former, sources are defined in terms of the global schema, while in the latter, the global schema is defined in terms of the sources (similarly to the classical views in database systems). Information Manifold [20] is a good example for a LAV system. Examples for the GAV approach include the Stanford-IBM integration system TSIMMIS [8], and the DL based integration system called Observer [23].

In SINTAGMA we apply a hybrid approach, i.e. we use both LAV and GAV. When using abstractions to populate high-level classes we employ the LAV principle, while in case of DL class definitions we use the GAV approach.

There are several completed and ongoing research projects in the area of using description logic-based approaches for both Enterprise Application Integration (EAI) and Enterprise Information Integration (EII).

The generic EAI research stresses the importance of the Service Oriented Architecture, and the provision of new capabilities within the framework of Semantic Web Services. Examples for such research projects include DIP [16] and INFRAWEBS [13]. These projects aim at the semantic integration of Web Services, in most cases using Description Logic based ontologies and Semantic Web technologies. Here, however, DL is used mostly for service discovery and design-time workflow validation, but not during query execution.

On the other hand, several logic-based EII tools use DL and take a similar approach as we did in SINTAGMA. That is, they create a DL model as a view over the information sources to be integrated. The basic framework of this solution is described e.g. in [7,4]. The fundamental difference with our approach is that these applications deal with the classical Open World Assumption, as already discussed in Section 4.2. We argue that existing DL reasoners are not usable when large amounts of data and complex DL queries are involved [15,18,24].

On the theoretical side an interesting description logic is the $\mathcal{ALCK}$ [11] which adds a non-monotonic K operator to the $\mathcal{ALC}$ language to provide the ability to use both the CWA and the OWA, when needed. $\mathcal{ALCK}$ has several implementation, the Pellet reasoner [27], for example, supports this logic. However, $\mathcal{ALCK}$ lacks the ability to express cardinality constraints, which is a feature frequently used in information integration scenarios.

Finally, we mention that the Description Logic Programming (DLP) approach, first introduced in [14], also employs the idea of translating DL axioms into Prolog goals (cf. the approach summarised in Table 6). In contrast with our approach DLP uses the Open World Assumption and does not deal with negation and cardinality restrictions.

8 Conclusions

In this paper we have presented the DL extension of the information integration system SINTAGMA. This extension allows the information expert to use Description Logic based ontologies in the development of high abstraction level conceptual models. Querying these models is performed using the Closed World Assumption over the underlying information sources.

We have presented the main components of the SINTAGMA system: the Model Manager which is responsible for maintaining the Model Warehouse repository, the Wrapper, which provides a uniform view over the heterogenous information sources and the Mediator, which decomposes complex high-level queries into primitive ones answerable by the individual information sources.

Next, we have described the newly introduced DL modelling elements the integration expert can use when building conceptual models and we have also discussed the modelling methodology she has to follow. We have defined a transformation of DL queries to Prolog goals, used in the SINTAGMA system for DL query execution. We have also illustrated our approach by providing a use case about artists and their works.

We believe that because Description Logics are not expressive enough to be used alone for solving complex modelling problems, some kind of hybrid techniques are necessary. We argue that our solution for combining DL and UML modelling in a unified integration framework provides a viable alternative to existing systems. The usage of DL constructs in building high-level conceptual models has substantial benefits, both in terms of modelling efficiency and maintenance.

Acknowledgements

The authors acknowledge the support of the Hungarian NKFP programme for the SINTAGMA project under grant no. 2/052/2004. We thank all the people participating in this project, especially Tamás Benkő, the lead architect.

We are also grateful to the anonymous reviewers of a preliminary version of this paper [22], for their insightful comments.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.
2. Liviu Badea and Doina Tilivea. Query Planning for Intelligent Information Integration using Constraint Handling Rules, 2001. IJCAI-2001 Workshop on Modeling and Solving Problems with Constraints.
3. T. Benkő, P. Krauth, and P. Szeredi. A logic based system for application integration. In *Proceedings of the 18th International Conference on Logic Programming, ICLP 2002*. Springer, LNCS, 2002.
4. A. Borgida, M. Lenzerini, and R. Rosati. Description logics for databases. In *Description Logic Handbook*, pages 462–484, 2003.
5. András G. Békés and Péter Szeredi. Optimizing Queries in a Logic-based Information Integration System. In Wim Vanhoof Patricia Hill, editor, *Proceedings of the 17th Workshop on Logic-based methods in Programming Environments (WLPE 2007)*, pages 1–15, Porto, Portugal, 2007.
6. D. Calvanese, D. Lembo, and M. Lenerini. Survey on methods for query rewriting and query answering using views. Tech. report, University of Rome, April 2001.
7. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Daniele Nardi, and Riccardo Rosati. Description logic framework for information integration. In *Principles of Knowledge Representation and Reasoning*, pages 2–13, 1998.
8. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.
9. T. Clark and J. Warmer, editors. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.
10. William F. Clocksin and C. S. Mellish. *Programming in PROLOG.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1994.

11. F. M. Donini, M. Lenzerini, D. Nardi, W. Nutt, and A. Schaerf. An epistemic operator for description logics. *Artif. Intell.*, 100(1-2):225–274, 1998.
12. Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.
13. Vladislava Grigorova. Semantic description of web services and possibilities of BPEL4WS. *Information Theories and Applications*, 13(2):183–187, 2006.
14. Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: Combining logic programs with description logic. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 48–57. ACM, 2003.
15. V. Haarslev and R. Möller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In *Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5*, pages 163–173, 2004.
16. M. Hepp, F. Leymann, J. Domingue, A. Wahler, and D. Fensel. Semantic business process management: A vision towards using semantic web services for business process management, 2005.
17. Ian Horrocks. Reasoning with expressive description logics: Theory and practice. In *Proc. of the 18th Int. Conf. on Automated Deduction (CADE 2002)*, number 2392 in Lecture Notes in Artificial Intelligence, pages 1–15. Springer, 2002.
18. U. Hustadt, B. Motik, and U. Sattler. Reasoning for description logics around $\mathcal{SHIQ}$ in a resolution framework. Technical Report 3-8-04/04, June 2004.
19. Interface Definition Language. ISO International Standard, number 14750.
20. T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In C. Knoblock and A. Levy, editors, *AAAI Spring Symposium ob Information Gathering from Heterogeneous, Distributed Environments*, 1995.
21. Gergely Lukácsy, Tamás Benkő, and Péter Szeredi. Towards automatic semantic integration. In *Enterprise Interoperability II, New Challenges and Approaches, Proceedings of the I-ESA 2007*, pages 795–806, Funchal, Portugal, 2007. Springer.
22. Gergely Lukácsy and Péter Szeredi. Ontology based information integration using logic programming. In Edna Ruckhaus, editor, *Proceedings of the 2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS2007)*, pages 39–54, Porto, Portugal, 2007.
23. Eduardo Mena, Vipul Kashyap, Amit P. Sheth, and Arantza Illarramendi. OBSERVER: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. In *Conference on Cooperative Information Systems*, pages 14–25, 1996.
24. Zsolt Nagy, Gergely Lukácsy, and Péter Szeredi. Translating description logic queries to Prolog. In *Proc. of PADL, Springer LNCS 3819*, pages 168–182, 2006.
25. Linh Anh Nguyen. A fixpoint semantics and an sld-resolution calculus for modal logic programs. *Fundam. Inf.*, 55(1):63–100, 2003.
26. ISO Prolog standard, 1995. ISO/IEC 13211-1.
27. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Web Semant.*, 5(2):51–53, 2007.
28. Leon Sterling and Ehud Shapiro. *The art of Prolog: advanced programming techniques.* MIT Press, Cambridge, MA, USA, 1986.