Combining Description Logics and object oriented models in an information integration framework

Gergely Lukácsy / Péter Szeredi

Received 2007-10-16, revised 2008-03-13, accepted 2009-03-16

1 Introduction

This paper presents the Description Logic modeling 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 centred, monolithic information integration system supporting semi-automatic integration of 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 modeling 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 modeling methodology. Section 5 describes the execution mechanism used when querying Description Logic models. Section 6 presents a fairly complex example, demonstrating the use of the tools and techniques discussed previously. 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 Logics and logic programming as these technologies form the basis of our work.

2.1 Description Logics

Description Logics (DL) \[17\] is a family of simple logic languages used for knowledge representation. DLs are used for describing 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 the form \( C \subseteq D \) (concept \( C \) is subsumed by \( D \)). The ABox (assertion box) stores knowledge about the individuals in the world: a concept assertion of the 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 \( ALCN(D) \) in this paper. \( ALCN(D) \) concept expressions (often simply referred to as concepts) are built from role names, concept names, and the top and bottom concepts (\( T \) and \( \bot \)) using the following constructors: intersection (\( C \cap D \)), union (\( C \cup D \)), negation (\( \neg C \)), value restriction (\( \forall R.C \)), existential restriction (\( \exists R.C \)) and number restrictions (\( \geq n \) and \( \leq n \)). Here, \( C \) and \( D \) are concept expressions and \( R \) is a role name. The two kinds of number restrictions are jointly referred to as (\( \equiv n \)). In \( ALCN(D) \) 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 invocations 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 to as the functor of the term. A Prolog structure with three arguments can be seen below:

\[
\text{'Work:}\text{class:}220'(\text{DT}, \{A, B, C, D, E\}, \_)
\]

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 letter 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 \([\text{Head}|\text{Tail}]\), 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. \([A,B,C,D,E]\), which, in fact, corresponds to \([A\mid B\mid C\mid D\mid E\mid[]]\). Here [] represents an empty list (a list with no elements).

A Prolog program consists of a set of clauses of the 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:

\[
\text{'Writer:}\text{class:}234'(\text{ID}), \text{'Painter:}\text{class:}236'(\text{ID})
\]

This complex goal consists of two goals, separated by a comma. It succeeds if there is 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 IDs. Informally, this query enumerates those people who are writers and painters at the same time.

Further control constructs such as disjunction ( \(+\text{Goal1} ; \text{Goal2} \) ) and negation \(\neg\text{Goal} \) are also supported by Prolog. The latter is the so called “negation by failure”, which is not capable of enumerating solutions, just checks if the execution of Goal fails.

The standard for the Prolog language \[26\] defines a wide range of built-in predicates. Of these we mention the predicate bagof, which plays an important role in the execution of DL
models. This predicate serves for collecting all solutions of a goal. For example,

\[
\text{bagof}(\text{ID}, (\text{`Writer::class:234'}(\text{ID}), \text{`Painter::class:236'}(\text{ID})))
\]

collects the identifiers of all people who are writers and painters, and returns these in the list IDs. An important property of \text{bagof} is that it can return multiple solutions, if there are variables in its second argument which do not appear in the first. For example, consider a predicate \text{edge} describing the edges of a directed graph:

\[
\text{edge}(a, b). \quad \text{edge}(a, c). \quad \text{edge}(c, d). \quad \text{edge}(d, a). \quad \text{edge}(c, e).
\]

By invoking the goal \text{bagof}(\text{End}, \text{edge}(\text{Start}, \text{End}), \text{EndPoints}) we collect the endpoints of the edges. This goal produces three answers, one for each possible value of variable \text{Start}:

\[
\begin{align*}
\text{Start} &= a, \text{EndPoints} = [b, c] \\
\text{Start} &= c, \text{EndPoints} = [d, e] \\
\text{Start} &= d, \text{EndPoints} = [a]
\end{align*}
\]

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

### 3 SINTAGMA System Architecture

The overall architecture of the SINTAGMA system can be seen in Fig. 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 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 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 Model Warehouse is given in the language called \textit{SILan} which is based on UML [12] and Description Logics [17]. The syntax of \text{SILan} resembles IDL, the Interface Description Language of CORBA [19]. We demonstrate the knowledge representation facilities of SINTAGMA by a simple \text{SILan} example showing the relevant features of the meta-data repository (Fig. 2).

```
model Art {
  class Artist: BuiltIns::DLAny {
    attribute String name;
    attribute Integer birthDate;
    constraint self.creation.date > 1900;
  };

  class Work: BuiltIns::DLAny {
    attribute String title;
    attribute String author;
    attribute Integer date;
    attribute String type;
    primary key title;
  };

  association hasWork {
    connection Artist as creator;
    connection Work as creation;
  };
}
```

Fig. 2. \textit{SILan} representation of the model \textit{Art}

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

#### 3.1.1 Semantics of \textit{SILan} models

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

Classes can have \textit{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 \textit{Artist} and \textit{Work} inherit from the built-in class \text{BuiltIns::DLAny} (cf. lines 2 and 8). See Section 4.3 for more details.

Associations have \textit{connections}, an \textit{n}-ary association has \textit{n} connections. In an association some of the connections can be named, providing intuitive navigation. For example, the connections of association \textit{hasWork}, corresponding to classes

\footnote{In \textit{SILan} double colons (::) separate the model name from the name of its constituent (class, association, etc.).}
**Fig. 1.** The architecture of the SINTAGMA system

**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.

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. 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, by creation, we navigate through the association hasWork to an arbitrary piece of work of the artist, while in the second step, by date, we go from the work to its publication date. Finally we state that this date is always greater than 1900.

In addition to the object oriented modeling 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.

---

3.1.2 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 Fig. 3.

```
abstraction w0 (m0: Interface::Product, 
m1: Interface::Description 
-> m2: Art::Work) {

constraint
m1.id = m0.id and
m1.category = "artwork"
implies
m2.DL_ID = m0.name and
m2.title = m0.name and
m2.author = m0.creator and
m2.date = m0.creation_date and
m2.type = m1.subcategory;
}
```

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

This abstraction populates the class Work (cf. Fig. 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 m2 are declared, and these are used throughout the abstraction specification to denote instances of the appropri-
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 9–11.\footnote{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 modeling decision.} Note that line 6 specifies that the id attributes of the two instances have to be the same, and thus corresponds to a relational \textit{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 type), the contribution of the second one is a single field (id). However, this second source 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 Fig. 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 \footnote{Attribute DL_ID comes from the class DLAny, of which class Work is a descendant. It has a special role, as explained in Section 3.3}.  

### 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 Fig. 2.

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 \footnote{3Attribute DL_ID comes from the class DLAny, of which class Work is a descendant. It has a special role, as explained in Section 3.3} supports the execution of 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 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. a 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.

```prolog
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\footnote{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 modeling decision.}. In this case, the query goal is similar to the following simple Prolog expression:

```prolog
:~'Work:class:220'(DT,[ID,Ti,Au,Date,Ty],DA),
  Date > 2000.
```

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 the form `Kind(Id)`, e.g. `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 happens to belong to a descendant 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 a list of the static attributes. In the example we have five such variables, the first of which is inherited from the class DLAny, while the remaining four are the attributes of the class Work, see lines 9–12 in Fig. 2. The last but one of these, `Date`, denotes the value of the attribute date. The third argument of the query term, `DA`, 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 of Art::Work.

The second part of the query goal corresponds to a simple arithmetic OCL constraint, which uses the variable `Date`.

The mediator rules representing the abstraction w0, shown in Fig. 3 take the following form (note that in w0 we stated that
both m2.title and m2.DL_ID are populated by the same value, denoted by Title below):

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

---

'Work class:220'(class(220),
[Title,Title,Author,Date,Type],[1])

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 (3). For example, a query goal for the association hasWork (Fig. 2) has the following form:

\[ \text{:- 'hasWork:association:227'(}
\text{'Artist:association:218'(}
\text{DT1,[DL_ID1,Name,Birthdate],DA1),}
\text{Work:association:220'(}
\text{DT2,[DL_ID2,Title,Author,Date,Type],DA2})\]

\] (3)

4 DL modeling in SINTAGMA

We now introduce the new DL modeling 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 modeling 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 {
    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 WriterAndPainter itself. The constraint actually corresponds to a DL concept definition axiom: WriterAndPainter ≡ Writer ⊓ 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 3.1.2).

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 (??), 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 \( ALC\mathcal{N}(D) \) 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 (??). 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>DL Syntax</th>
<th>SILan equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base concept</td>
<td>( A )</td>
<td>UML class</td>
</tr>
<tr>
<td>Atomic role</td>
<td>( R )</td>
<td>UML association</td>
</tr>
<tr>
<td>Intersection</td>
<td>( C \cap D )</td>
<td>C and D</td>
</tr>
<tr>
<td>Union</td>
<td>( C \cup D )</td>
<td>C or D</td>
</tr>
<tr>
<td>Negation</td>
<td>( \neg C )</td>
<td>not C</td>
</tr>
<tr>
<td>Value restriction</td>
<td>( \forall R.C )</td>
<td>slot constraint R all values C</td>
</tr>
<tr>
<td>Example restriction</td>
<td>( \exists R.C )</td>
<td>slot constraint R some value C</td>
</tr>
<tr>
<td>Number restriction</td>
<td>[nR]</td>
<td>slot constraint R cardinality i..j</td>
</tr>
<tr>
<td>Top</td>
<td>( \top )</td>
<td>DLAny</td>
</tr>
<tr>
<td>Bottom</td>
<td>( \bot )</td>
<td>DLEmpty</td>
</tr>
</tbody>
</table>

Concrete restriction — class constraint A satisfies OCL

The only non-classical DL element in Table 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”.

```java
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 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.

\[
\text{NovicePainter} \equiv \text{Painter} \land (\leq 5 \text{hasPainting})
\]

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 Import/Export” box in Fig. 1).

It is the task of the expert to map the names of the base symbols onto 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 (??), provided that we also encode in SILan the equivalents of the following DL axioms:

\[
A \equiv \text{Writer} \\
B \equiv \text{Painter}
\]

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 which 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 which 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 (⊤) 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 [cf. lines 2 and 8 of Fig. 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: he needs to extend the existing abstractions populating the base symbols (classes) by considering also the attribute DL_ID. By appropriate values we mean that the DL_IDs of two instances should match if these instances are the same, and should differ otherwise. An example for this can be seen in Fig. 5 populating the class Writer, which is part of a bigger integration scenario to be shown later in Section 6.

```plaintext
abstraction ap (m0: Interface::Member -> m1: Unified::Writer) {
  m1.name = n and m1: Unified::Writer) {
  m1.DL_ID = n;
  m1.birthDate = m0.date and m1.style = m0.style and
  constraint let n = m0.fname.concat(" ").concat(m0.lname) in
  m1.member_id = m0.iwa_id and
  m1.style = m0.style and
  m1.DL_ID = n;
};
```

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

This abstraction populates the class Writer from an inter-
face 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).

This is because the class Writer can also be populated from other sources (e.g. Person, see Fig. 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. Fig. 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.

To summarise, the integration expert has to perform the following tasks when DL modeling 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 5.3 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 on page 21 its dynamic type DT, its static attributes SAs 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 modeling). Accordingly, an answer to a DL query takes the form of a pair (ID, DTA), where ID is the DL_ID containing the unique name of the DL instance (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 standard Prolog comma operator: (DTA1, DTA2).

Fig. 6 describes the mapping from an arbitrary DL concept expression to the corresponding query goal. Here we define a function $\Phi_C$ 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.

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 the operator $\oplus$ denotes the compile time concatenation of lists while $A_N$ stands for the predicate name corresponding to concept $A$. For example, $Work_N = 'Work: class:220'$, cf. 2 on page 21. 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 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 the 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 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. 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 formulae $R_N$ denotes the predicate name corresponding to the association itself. $R_D$ ($R_R$) denotes the base

---

3This is also a simplification. More realistically, the key could be the name together with the birth date.

4We use the name ID instead of DL_ID for conciseness.

5The $\oplus$ 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 concatenation SAs $\oplus$ DAs can be calculated at compile time.
class which is the domain (range) of association $R$. Correspondingly, $R^N_D$ and $R^N_R$ stand for the predicate names of the classes $R_D$ and $R_R$, respectively. 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 $\Phi_{\neg\forall}(ID_2, \_)$ is invoked only when $ID_2$ is already instantiated.

A number restriction ($\ni n R$) is transformed into a goal which uses the Prolog built-in predicate `bagof` (cf. Section 2.2, page [19]) 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 Fig. 6 define the transformation of the top and bottom concepts. $\top$ is mapped into `true`, while $\bot$ into `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:

$$\Phi_A(ID, DTA), DTA = DT-AT, \Psi_O(ID, AT)$$

To illustrate the general algorithm, two example transformations are presented in Fig. 7. The first one shows the translation of the `WriterAndPainter` class described in (??) on page ???. The query goal is a conjunction consisting 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.

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 classes `Writer` and `Reference` with a condition that their ID attributes match. The second and the third goals ensure that the writer in question is also an artist, who does have some modern works. Here we use the facts that the domain of `hasWork` is the class `Artist` and the range is the class `Modern` (cf. Fig. 2). Finally, the last goal builds the structure `DTA`.

Note that if a writer has more than one piece of modern work, the transformation in Fig. 7 enumerates the writer several 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].

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

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). Class **Exhibitor** contains people some of whom are painters, class **Product** contains products including art works, and class **Description** 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 Fig. 9 (referring to class **Artist** introduced in Fig. 2 on page 19). 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.

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 Fig. 8, by creating virtual pairs from those instances of classes **Artist** and **Work** where the author of the work matches the 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...
Fig. 8. Content of the Model Warehouse

Fig. 8. Content of the Model Warehouse

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 Fig. [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 he 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 if he is a writer who has some paintings (axiom 4). Finally, a novelist is somebody who is only writing novels (axiom 5).

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:

```plaintext
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 Fig. [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.
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:

```java
{"Lisa James",
['Writer'=>['Lisa James', 1965, 'fantasy'],
'Painter'=>['Lisa James', 1965, 'red']
}
```

Here, the string 'Lisa James', appearing in line 1, corresponds to the ID of Fig. 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) [5]. 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 [15, 18, 24]. 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 5). 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 Wraper, which provides a uniform view over the heterogeneous 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 modeling elements the integration expert can use when building conceptual models and we have also discussed the modeling methodology he 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 modeling problems, some kind of hybrid techniques are necessary. We argue that our solution for combining DL and UML modeling 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 modeling efficiency and maintenance.

References
19 Interface Definition Language. ISO International Standard, number 14750.
23 Mena E, Kashyap V, Sheth A P, Illarramendi A. OBSERVER: An approach for query processing in global information systems based on interoperability across pre-existing ontologies, Conference on cooperative information systems, 1996, pp. 14–25, citeseer.ist.psu.edu/article/ mena00observer.html
25 Nguyen L A. A fixpoint semantics and an SLD-resolution calculus for modal logic programs, Fundam. Inf. 55 (2003), no. 1, 63–100.