Acquisition and Validation of Complex Object Database Schemata Supporting Multiple Inheritance*

Sonia Bergamaschi
Facoltá di Ingegneria
Universitá di Modena
CIOC-CNR
Italy
sonia@deis64.cineca.it

Bernhard Nebel
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-66123 Saarbrücken
Germany
nebel@dfki.uni-sb.de

Abstract

We present an intelligent tool for the acquisition of object oriented schemata supporting multiple inheritance, which preserves taxonomy coherence and performs taxonomic inferences. Its theoretical framework is based on *terminological logics*, which have been developed in the area of artificial intelligence. The framework includes a rigorous formalization of complex objects, which is able to express cyclic references on the schema and instance level; a *subsumption* algorithm, which computes all implied *specialization* relationships between types; and an algorithm to detect *incoherent* types, i.e., necessarily empty types. Using results from formal analyses of knowledge representation languages, we show that subsumption and incoherence detection are computationally intractable from a theoretical point of view. However, the problems appear to be feasible in almost all practical cases.

*This work was partially supported by the Italian project *Sistemi informatici e Calcolo Parallelo*, subproject 5, objective LOGIDATA$^+$, of the National Research Council (CNR) and by the German Ministry for Research and Technology (BMFT) under grant ITW 8901 8.

1 Introduction and Motivation

The organization of types in an inheritance taxonomy in order to describe an application domain constitutes a basic modeling principle in the database area and in artificial intelligence [2, 5, 17, 16, 20, 11, 8, 12]. In the database area, *the type taxonomy* is built by the designer: a type must be described by an explicit declaration of its *parent* types and its *differentiae* properties. The interpretation of such a type description is that it only gives the *necessary* conditions for objects to be instances of the described type. Much research effort has been devoted to formally define and guarantee inheritance consistency of database schemata, based on strict inheritance taxonomies [2, 20, 5]. In the O_2 object-oriented DBMS [21], for example, the introduction of new types which violate strict inheritance semantics is prevented by means of well-known type checking techniques [14] (i.e., A can only be a specialization of B if the *type* of A is a *refinement* of the *type* of B).

In the area of artificial intelligence, a class of knowledge representation systems, called *Terminological Logic Systems* [23] (henceforth *TL*) has been developed, which are based on the ideas developed in connection with the KL-ONE system [13].[1] These systems assign a more active role to type taxonomies. First, it is possible to specify necessary and *sufficient* conditions in a type description. Second, based on this, the task of creating the type hierarchy can be delegated to the system.

The knowledge-base designer gives a type description as a free composition of ancestor types (not necessarily parents) and of differentiae properties, and the system automatically *classifies* it, i.e., determines the "right" place of the new type in the already existing taxonomy, between its most specific generalizations and its most generalized specializations. Classification is performed by the so called *taxonomic reasoner* which finds all *specialization* relationships (also called *isa* relationships) between a new type description and the types in the taxonomy already given.

In *TL languages* is is possible to define types by specifying only necessary conditions resulting in what we call *base class-types* or by specifying necessary and sufficient conditions leading to what we call *virtual class-types*. The former kind of class type corresponds to ordinary class-types used in object-oriented database systems, while the latter are similar to relational database

[1]For a recent survey on different TL systems see [19].

views or to *virtual classes*, recently introduced by Abiteboul and Bonner [1]. With both base and virtual class-types, the taxonomic reasoner plays the passive role of an inheritance consistency checker and the more active role of an automatic classificator.

Applying taxonomic reasoning to traditional semantic data models led to a number of promising results for database schema design [17, 16, 11] and other relevant topics as query processing and data recognition [8, 12]. In particular, in [11] a very general theoretical framework (able to express the data semantics of the well-known conceptual models E/R [15], TAXIS [22], GALILEO [3], IFO [2]), is presented, which supports conceptual schema acquisition and organization by preserving *coherence* and *minimality* w.r.t. inheritance exploiting the framework of terminological reasoning.

Complex object data models, recently proposed in the areas of deductive databases [2] and object oriented databases [21, 20] are more expressive than implemented terminological logic languages in some aspects. For instance, most of the complex object data models introduce a distinction between *values* and *objects* with identity and, thus, between *value types* and *class types* (which are also briefly called *classes*). This distinction is not present in TL languages. Further, complex object models often support additional type constructors, such as *set* and *sequence*. Most importantly, complex object data models usually support the representation and management of *cyclic classes*, i.e., classes which directly or indirectly make references to themselves.

The aim of this paper, following the approach of [11], is to propose a theoretical framework based on taxonomic reasoning for complex object schema acquisition and organization, preserving inheritance consistency (in the following called *coherence*) and minimality w.r.t. inheritance. This framework serves as the theoretical kernel of an intelligent tool for advanced database schema design. The main extensions of this framework with respect to other complex object models are, firstly, the *conjunction* operator, which permits the expression of *multiple inheritance* as part of a class-type description. This avoids the specification of explicit redefinitions (cf. [21]) in case of multiple inheritance. Secondly, we introduce a distinction between *base class-types* and *virtual class-types*. While the set of objects that are instances of a base class has to be provided by the user, the set of instances of a virtual class is computed by the system. Finally, we allow circular references in class-type descriptions and extend the semantics using results from knowledge representation research [6, 25].

The model, which we call ODL, makes it possible to perform the passive coherence check of a database schema including multiple inheritance taxonomies of value-types, *base* classes and *virtual* classes. Furthermore, as a subsumption algorithm allows computation of the *minimal description* of a type with respect to the type specialization ordering, the more active role of building a minimal type taxonomy can be played.

The outline of the paper is as follows. In the next subsection we introduce the *company domain* example in order to illustrate our approach. In Section 2 we introduce the ODL formalism. It is a TL formalism which incorporates well-known notions such as value types, complex values, class types and objects with identity, in full analogy with recent complex object models proposed in the areas of deductive databases [2] and object oriented databases [21, 20]. Section 3 contains the algorithms to solve the *subsumption*[2] and *incoherence* problems. The approach is to firstly transform an ODL schema into a *canonical* form and then to define subsumption and incoherence algorithms for canonical schemata. The approach is demonstrated by some examples showing the effectiveness of the proposed algorithms. Section 4 contains an analysis of the computational complexity of the coherence and subsumption problems. In Section 5 we describe the implemented tool for advanced database schema design. Finally, in Section 6 we discuss our approach and relate it to other work.

1.1 The Company Domain Example

In order to illustrate our approach, let us consider the following description of part of the organizational structure of a company. Persons have a name; employees are *exactly* those persons who work in a branch earning a salary and have some skill level (`Level` = 1–10). Branches are described by a name. Managers are *exactly* those persons who work in and head a branch, earn a salary and have the highest level values (`AdvLevel` = 8–10). Sectors have *exactly* a name and a set of activities. Clerks are *exactly* those employees who work in a department, having a medium level (`MdmLevel` = 2–7). Departments, in turn, are *exactly* those branches that enroll only clerks. Sec-

[2] In [20] a relation called *refinement* is defined which is identical to our subsumption relation if we assume only virtual classes.

retaries are *exactly* those employees working in an office, having a medium level (`MdmLevel` = 2–7), and offices are *exactly* those particular branches and sectors that enroll only secretaries. Intuitively, descriptions without the keyword *exactly* introduce base classes, whereas those with the keyword *exactly* introduce virtual classes.

$$
\begin{array}{rcl}
\textbf{type}\ \texttt{Level} & = & 1\text{–}10 \\
\textbf{type}\ \texttt{AdvLevel} & = & 8\text{–}10 \\
\textbf{type}\ \texttt{MdmLevel} & = & 2\text{–}7 \\
\textbf{type}\ \texttt{Activities} & = & \{\texttt{String}\} \\
\textbf{class}\ \texttt{Person} & = & \Big[\texttt{name}\colon\texttt{String}\Big] \\
\textbf{class}\ \texttt{Branch} & = & \Big[\texttt{name}\colon[\texttt{bname}\colon\texttt{String}]\Big] \\
\textbf{virtual-class}\ \texttt{Sector} & = & \Big[\texttt{name}\colon[\texttt{sname}\colon\texttt{String}], \texttt{activity}\colon\texttt{Activities}\Big] \\
\textbf{virtual-class}\ \texttt{Employee} & = & \texttt{isa Person}\Big[\texttt{salary}\colon\texttt{Real}, \texttt{works-in}\colon\texttt{Branch} \\
 & & \qquad\qquad \texttt{level}\colon\texttt{Level}\Big] \\
\textbf{virtual-class}\ \texttt{Manager} & = & \texttt{isa Person}\Big[\texttt{salary}\colon\texttt{Real}, \texttt{works-in}\colon\texttt{Branch}, \\
 & & \qquad\qquad \texttt{head}\colon\texttt{Branch}, \texttt{level}\colon\texttt{AdvLevel}\Big] \\
\textbf{virtual-class}\ \texttt{Clerk} & = & \texttt{isa Employee}\Big[\texttt{works-in}\colon\texttt{Department}, \\
 & & \qquad\qquad \texttt{level}\colon\texttt{MdmLevel}\Big] \\
\textbf{virtual-class}\ \texttt{Department} & = & \texttt{isa Branch}\Big[\texttt{employs}\colon\{\texttt{Clerk}\}\Big] \\
\textbf{virtual-class}\ \texttt{Secretary} & = & \texttt{isa Employee}\Big[\texttt{works-in}\colon\texttt{Office}, \\
 & & \qquad\qquad \texttt{level}\colon\texttt{MdmLevel}\Big] \\
\textbf{virtual-class}\ \texttt{Office} & = & \texttt{isa Branch}, \texttt{Sector}\Big[\texttt{employs}\colon\{\texttt{Secretary}\}\Big]
\end{array}
$$

Table 1: The company domain schema in LOGIDATA^{+} syntax

Using LOGIDATA^{+} [4] syntax, the company domain example can be formalized as done in Table 1. Declarations prefixed with the keyword **type** introduce value-type declarations similar to types used in programming languages. In our case, three integer range types and one type having sets of strings as values are introduced. Declarations prefixed with **class** introduce base classes. Declarations prefixed by **virtual-class** introduce virtual classes.

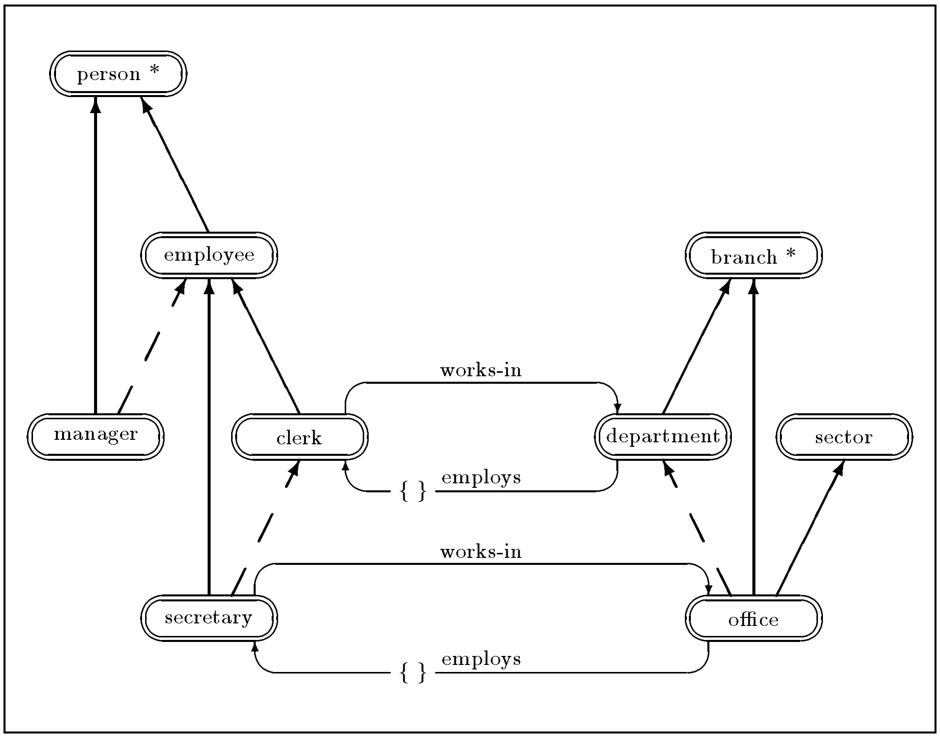

Figure 1: The company domain with computed specialization relations

The hierarchy of classes and some relationships between classes implied by the above descriptions are shown in Figure 1. Classes are denoted by ellipses, (base classes are indicated by an asterisk), explicitly given specialization relationships are denoted by solid arrows, while computed ones by dashed arrows. Furthermore, only the attributes which give rise to the cyclic virtual classes `Clerk` , `Department` , `Secretary` and `Office` (indicated by oriented arcs) are drawn. Note that the representation of the class hierarchy contains three new specialization relationships not mentioned in the schema. These follow from the fact that all the elements of `Manager` are elements of `Employee`. Furthermore, notice that `Secretary` is subsumed by `Clerk` and `Office` is subsumed by `Department` and `Sector`. This result, which is not obvious since `Clerk` and `Department`, `Office` and `Secretary` are cyclic virtual classes with mutual references, can be obtained by adopting a suitable fixed-point semantics (the greatest fixed-point); the difficulties connected with cyclic class declarations will be discussed in Section 2.4.

2 ODL: A Formalism for Complex Objects

The *object description language* ODL is a formalism similar to TL languages and complex object data models. In its specification we basically follow the specification of O_2 [20]. However, we assume a richer structure for the *system of atomic types*, the system of non-decomposable, basic value-types. Besides the atomic types *integer*, *boolean*, *string*, *real*, and *mono-valued types*, we consider also the possibility that subsets of these types are used, e.g., intervals of integers. The only restriction is that the system of atomic types is closed w.r.t. intersection and that the intersection of two atomic types can be computed in polynomial time.

Based on these atomic types, *tuple*, *sequence*, *set*, and *class types* can be created. Class types (also briefly called *classes*) denote sets of *objects with an identity and a value*, while the former three types – also called *value types* denote sets of *complex, finitely nested values* without object identity. Additionally, a conjunction operator can be used to create intersections of previously introduced types. This operator can be used to specify multiple inheritance. For instance, the class `Office` of the example of subsection 1.1 is defined as the conjunction of the class `Branch` and `Sector`. Finally, types can be given names. For value types, no circular references are permitted in order to guarantee that values are always only finitely nested. Class types, on the other hand, can be defined making circular references. Named class types come in two flavors. A named class type may be *base*, which means that the user has to specify the membership of an *object* in the interpretation of a class. Second, a named class can be a *virtual class*, in which case the *class interpretation* is computed.

2.1 Atomic Types

Let $\mathcal{D}$ be the countably infinite set of atomic values (which are denoted by $d_1, d_2, \ldots$), e.g., the union of the set of integers, the set of strings, and the booleans. We do not distinguish between atomic values and their encoding.

Let $\mathbf{B}$ be a countable set of designators for atomic types that contains $\mathcal{D}$ (i.e., all mono-valued types), and let $\mathcal{I}_{\mathbf{B}}$ be the (fixed) standard interpretation function from $\mathbf{B}$ to $2^{\mathcal{D}}$ such that for all $d \in \mathbf{D}$: $\mathcal{I}_{\mathbf{B}}[d] = \{d\}$. Let "$\sqcap$" be an operation on $\mathbf{B}$ defined by:

$$B' \sqcap B'' = B \text{ iff } \mathcal{I}_{\mathbf{B}}[B'] \cap \mathcal{I}_{\mathbf{B}}[B''] = \mathcal{I}_{\mathbf{B}}[B].$$

We say that $\mathbf{B}$ is a *system of atomic types* iff $\mathbf{B}$ is complete with respect to $\sqcap$. The special type that has an empty interpretation is called *empty type* and is denoted by $\perp$.[3] We say that $\mathbf{B}$ is a *PTIME system* iff $B' \sqcap B'' B$ can be decided in polynomial time. In the following we assume that the system of atomic types has this property.

Sometimes we will also talk about systems of atomic types with a particular simple structure, namely, systems such that for each subset $\mathbf{X} \subseteq \mathbf{B}$ with $\bigsqcap \mathbf{X} = B$, there are two elements $B', B'' \in \mathbf{X}$ such that $B' \sqcap B'' = B$. Such systems of atomic types are called *binary compact*.

Consider the following set of designators for atomic types, which we use in all examples:

$$\mathbf{B} = \{\mathtt{Int}, \mathtt{String}, \mathtt{Bool}, \mathtt{Real}, i_1\text{–}j_1, i_2\text{–}j_2, \ldots, d_1, d_2, \ldots\},$$

where the i_k–j_k's denote all possible ranges of integers, and the d_k's denote all the elements of $\mathtt{Int} \cup \mathtt{String} \cup \mathtt{Bool}$. Assuming the standard interpretation of the type designators, $\mathbf{B}$ is obviously a binary compact system of atomic types.

2.2 Values and Value Types, Objects and Class Types

We suppose a countable set $\mathbf{A}$ of *attributes* (denoted by $a_1, a_2, \ldots$) and a countable set $\mathbf{N}$ of type names (denoted by N, N') such that $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{N}$ are pairwise disjoint. $\mathbf{N}$ is partitioned into the sets $\mathbf{C}$, $\mathbf{D}$, and $\mathbf{V}$, where $\mathbf{C}$ consists of names for *base class-types* (denoted by $C, C' \ldots$), $\mathbf{D}$ consists of names for *virtual class-types* (denoted by $D, D' \ldots$), and $\mathbf{V}$ consists of names for *value-type* $(V, V', \ldots)$. Furthermore, $\mathbf{N}$ contains the special symbol $\top_C$ denoting the universal class.

$\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ denotes the set of all *finite type descriptions* (S, S', $\ldots$), also briefly called *types*, over given $\mathbf{A}, \mathbf{B}, \mathbf{N}$, that are built according to the

[3]This type must be part of $\mathbf{B}$ because the conjunction of different mono-valued types is empty.

following abstract syntax rule (assuming $a_i \neq a_j$ for $i \neq j$):

$$S \quad \rightarrow \quad B \mid N \mid \{S\} \mid \langle S \rangle \mid [a_1\colon S_1, \ldots, a_k\colon S_k] \mid S \sqcap S' \mid \; \triangle\, S.$$

The following are examples of well-formed type descriptions:

$$\begin{array}{l} \texttt{Int}, 5\text{–}10,\ \texttt{Department},\ \{\texttt{Secretary}\},\ \langle\texttt{Clerk}\rangle \\ [\texttt{street}\colon \texttt{String}, \texttt{number}\colon \texttt{Int}, \texttt{city}\colon \texttt{String}], \\ \texttt{Secretary} \sqcap \texttt{Employee},\ \triangle\texttt{Address}. \end{array}$$

We assume a countable set $\mathcal{O}$ of *object identifiers* (denoted by $o, o', \ldots$) disjoint from $\mathcal{D}$ and define the set $\mathcal{V}(\mathcal{O})$ of all *values over* $\mathcal{O}$ (denoted by v, v') as follows (assuming $p \geq 0$ and $a_i \neq a_j$ for $i \neq j$):

$$v \quad \rightarrow \quad d \mid o \mid \{v_1, \ldots, v_p\} \mid \langle v_1, \ldots, v_p \rangle \mid [a_1\colon v_1, \ldots, a_p\colon v_p],$$

where $[a_1\colon v_1, \ldots, a_i\colon v_i, \ldots, a_p\colon v_p]$ denotes the set of pairs (a_i, v_i). Using the above assumptions and $\mathcal{O} = \{o_1, o_2, \ldots\}$, the following expressions are possible values:

$$\begin{array}{l} \mathit{1}, \mathit{2},\ \mathit{true},\ \mathit{"Bologna"},\ o_{128},\ \{\mathit{1}, \mathit{2}, \mathit{"Bologna"}, o_{128}\},\ \emptyset,\ \langle \mathit{true}, \mathit{false} \rangle,\ \langle\rangle, \\ [\texttt{street}\colon \mathit{"Stuhlsatzenhausweg"}, \texttt{number}\colon \mathit{3}, \texttt{city}\colon \mathit{"Saarbrücken"}]. \end{array}$$

Object identifiers are assigned values by a *total value function* δ from $\mathcal{O}$ to $\mathcal{V}(\mathcal{O})$. For instance, we may have the following assignments of values to objects:

$$\delta\colon \begin{cases} o_1 & \mapsto \quad [\mathsf{a}\colon \mathit{"xyz"}, \mathsf{b}\colon \mathit{5}] \\ o_2 & \mapsto \quad \langle \mathit{true}, \mathit{false} \rangle \\ & \vdots \\ o_{128} & \mapsto \quad \{o_1, o_2\} \\ & \vdots \end{cases}$$

2.3 Database Schema

Given a system of atomic types $\mathbf{B}$ and finite sets of attributes $\mathbf{A}$ and names $\mathbf{N} = \mathbf{C} \cup \mathbf{D} \cup \mathbf{V}$, a *schema* σ over $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ is a total function from $\mathbf{N}$ to $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$. Intuitively, σ associates value-type names and class-type names to their descriptions. The company domain described in Section 1.1, for instance, would lead to the schema σ specified in Table 2.

$$
\begin{array}{rcl}
\mathbf{V} & = & \{\texttt{Activities}, \texttt{Level}, \texttt{AdvLevel}, \texttt{MdmLevel}\} \\
\mathbf{C} & = & \{\texttt{Person}, \texttt{Branch}\} \\
\mathbf{D} & = & \{\texttt{Clerk}, \texttt{Department}, \texttt{Sector}, \texttt{Employee}, \\
 & & \ \texttt{Secretary}, \texttt{Office}\} \\
\sigma(\texttt{Level}) & = & 1\text{–}10 \\
\sigma(\texttt{AdvLevel}) & = & 8\text{–}10 \\
\sigma(\texttt{MdmLevel}) & = & 2\text{–}7 \\
\sigma(\texttt{Activities}) & = & \{\texttt{String}\} \\
\sigma(\texttt{Person}) & = & \triangle[\texttt{name}{:}\,\texttt{String}] \\
\sigma(\texttt{Branch}) & = & \triangle[\texttt{name}{:}\,[\texttt{bname}{:}\,\texttt{String}]] \\
\sigma(\texttt{Sector}) & = & \triangle[\texttt{name}{:}\,[\texttt{sname}{:}\,\texttt{String}], \texttt{activity}{:}\,\texttt{Activities}] \\
\sigma(\texttt{Employee}) & = & \texttt{Person} \sqcap \triangle[\texttt{salary}{:}\,\texttt{Real}, \texttt{works-in}{:}\,\texttt{Branch}, \\
 & & \qquad\qquad \texttt{level}{:}\,\texttt{Level}] \\
\sigma(\texttt{Manager}) & = & \texttt{Person} \sqcap \triangle[\texttt{salary}{:}\,\texttt{Real}, \texttt{works-in}{:}\,\texttt{Branch}, \\
 & & \texttt{head}{:}\,\texttt{Branch}, \texttt{level}{:}\,\texttt{AdvLevel}] \\
\sigma(\texttt{Clerk}) & = & \texttt{Employee} \sqcap \triangle\Big[\texttt{level}{:}\,\texttt{MdmLevel}, \\
 & & \qquad\qquad\quad \texttt{works-in}{:}\,\texttt{Department}\Big] \\
\sigma(\texttt{Department}) & = & \texttt{Branch} \sqcap \triangle\Big[\texttt{employs}{:}\,\{\texttt{Clerk}\}\Big] \\
\sigma(\texttt{Secretary}) & = & \texttt{Employee} \sqcap \triangle\Big[\texttt{level}{:}\,\texttt{MdmLevel}, \texttt{works-in}{:}\,\texttt{Office}\Big] \\
\sigma(\texttt{Office}) & = & \texttt{Branch} \sqcap \texttt{Sector} \sqcap \triangle\Big[\texttt{employs}{:}\,\{\texttt{Secretary}\}\Big]
\end{array}
$$

Table 2: The company domain schema σ

We require some well-formedness conditions on a schema, namely, *well-formedness of value-type definitions* and of the *inheritance relation*. The former condition guarantees that all values are only finitely nested and the latter ensures that the inheritance relation is cycle-free.

Let τ_T be a function from $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ to $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ such that

$$
\begin{array}{rcl}
\tau_T(S) & = & S \text{ if } S \in \mathbf{B} \cup \mathbf{C} \cup \mathbf{D} \ \text{ or } \ S = \triangle S' \\
\tau_T(S) & = & \sigma(S) \text{ if } S \in \mathbf{V} \\
\tau_T(\{S\}) & = & \{\tau_T(S)\} \\
\tau_T(\langle S \rangle) & = & \langle \tau_T(S) \rangle \\
\tau_T([a_1{:}\, S_1, \ldots, a_k{:}\, S_k]) & = & [a_1{:}\, \tau_T(S_1), \ldots, a_k{:}\, \tau_T(S_k)]
\end{array}
$$

$$\tau_T(S \sqcap S') = \tau_T(S) \sqcap \tau_T(S').$$

We say that the schema σ is *type well-formed* iff there is a natural number n such that $\tau_T^n = \tau_T^{n+1}$. Note that τ_T does not expand class names and therefore its iteration is not influenced by the presence of the cyclic classes: `Department`, `Office`, `Clerk`, `Secretary`.

Similarly to value-type well-formedness, we require that the inheritance relation expressed by conjunctions in a value-type or class-type definition is cycle-free. Let ρ be a function from

$\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ to $2^{\mathbf{N}}$ defined as follows:

$$\rho(S) = \begin{cases} \{S\} & \text{if } S \in \mathbf{N} \\ \rho(S') \cup \rho(S'') & \text{if } S = S' \sqcap S'' \\ \{\} & \text{otherwise.} \end{cases}$$

We say that $N \in \mathbf{C} \cup \mathbf{D} \cup \mathbf{V}$ *inherits directly from* N', written $N \prec\!\!\mid_\sigma N'$ iff $N' \in \rho(\sigma(N))$. A schema is *inheritance well-formed* iff the transitive closure of $\prec\!\!\mid_\sigma$, which is denoted by $\prec_\sigma$, is a strict partial order. The reflexive and transitive closure is denoted by $\preceq_\sigma$. If a schema is type well-formed and inheritance well-formed we call it *well-formed*. Reconsidering our company example, it can be easily verified that this schema is type and inheritance well-formed.

2.4 Interpretation of a Schema

In the following, we specify the legal database states by defining the notions of *possible instance* and *legal instance* of a schema. To this purpose, let us define firstly an *interpretation function*, which associates a set of values to every value-type and a set of objects to every class type.

For a given set of object identifiers $\mathcal{O}$ and a value function δ, the *interpretation function* $\mathcal{I}$ is a function from $\mathbf{N}$ to $2^{\mathcal{V}(\mathcal{O})}$ such that:

$$\mathcal{I}[B] = \mathcal{I}_{\mathbf{B}}[B], \;\; \mathcal{I}[C] \subseteq \mathcal{O}, \;\; \mathcal{I}[D] \subseteq \mathcal{O}, \;\; \mathcal{I}[V] \subseteq \mathcal{V}(\mathcal{O}) - \mathcal{O}.$$

The interpretation of type descriptions is defined inductively for all $S, S' \in \mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ by:

$$\mathcal{I}[\{S\}] = \Big\{\{v_1, \ldots, v_p\} \Big| v_i \in \mathcal{I}[S], 0 \le i \le p\Big\}$$

$$
\begin{aligned}
\mathcal{I}[\langle S\rangle] &= \Big\{\langle v_1,\ldots,v_p\rangle\Big| v_i\in\mathcal{I}[S], 0\le i\le p\Big\}\\
\mathcal{I}\Big[[a_1{:}\,S_1,\ldots,a_p{:}\,S_p]\Big] &= \Big\{[a_1{:}\,v_1,\ldots,a_q{:}\,v_q]\Big|\; p\le q, v_i\in\mathcal{I}[S_i], 0\le i\le p,\\
&\qquad\qquad\qquad\qquad v_j\in\mathcal{V}(\mathcal{O}), p+1\le j\le q\Big\}\\
\mathcal{I}[S\sqcap S'] &= \mathcal{I}[S]\cap\mathcal{I}[S']\\
\mathcal{I}[\triangle S] &= \Big\{o\in\mathcal{O}\Big|\delta(o)\in\mathcal{I}[S]\Big\}\\
\mathcal{I}[\top_C] &= \mathcal{O}.
\end{aligned}
$$

Note that the interpretation of tuples implies an open world semantics for tuple types similar to the one adopted by Cardelli [14]. For the example above, it follows that

$$
\begin{aligned}
o_1 &\in \mathcal{I}\Big[\triangle[\mathtt{a}{:}\,\mathtt{String}]\Big], & o_1 &\in \mathcal{I}\Big[\triangle[\mathtt{a}{:}\,\mathtt{String},\mathtt{b}:\mathtt{Int}]\Big],\\
o_2 &\in \mathcal{I}[\triangle\langle\mathtt{Bool}\rangle], & o_{128} &\in \mathcal{I}[\{\triangle\top_C\}].
\end{aligned}
$$

It should be noted that an interpretation does not necessarily imply that the extension of a named type is identical to the type description associated with the type name via the schema σ. For this purpose, we have to further constrain the interpretation. We say that an interpretation function $\mathcal{I}$ as defined above is a *possible instance* of a schema σ iff the set $\mathcal{O}$ is *finite*, and for all $C\in\mathbf{C}, D\in\mathbf{D}, V\in\mathbf{V}$:

$$
\begin{aligned}
\mathcal{I}[V] &= \mathcal{I}[\sigma(V)]\\
\mathcal{I}[C] &\subseteq \mathcal{I}[\sigma(C)]\\
\mathcal{I}[D] &= \mathcal{I}[\sigma(D)].
\end{aligned}
$$

Possible instances are *legal instances* of a database, provided the schema is cycle-free. In fact, given a schema σ over $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$, a set of object identifiers $\mathcal{O}$, a value function δ, and a *partial interpretation*, i.e., an assignment defined over $\mathbf{C}$, there is at most one possible instance, provided the schema is cycle-free. In the general case, however, there are many possible instances. In fact, let us consider the object identifiers and value function as specified in Table 3.

Supposing the following partial interpretation

$$
\begin{aligned}
\mathcal{I}[\mathtt{Person}] &= \{o_1,o_3,o_5,o_7\}\\
\mathcal{I}[\mathtt{Branch}] &= \{o_2,o_4,o_6\},
\end{aligned}
$$

$$
\begin{array}{rcl}
\mathcal{O} & = & \{o_1, o_2, o_3, o_4, o_5, o_6\} \\
\delta(o_1) & = & [\texttt{name} : \text{“ \textit{Mark}”}, \texttt{salary} : 8000, \texttt{works-in} : o_2, \texttt{level} : 3] \\
\delta(o_2) & = & [\texttt{name} : [\texttt{bname} : \text{“\textit{Administration}”}], \texttt{employs} : \{o_1, o_7\}] \\
\delta(o_3) & = & [\texttt{name} : \text{“\textit{Robert}”}, \texttt{salary} : 9000, \texttt{works-in} o_4, \texttt{level} : 3] \\
\delta(o_4) & = & [\texttt{name} : [\texttt{bname} : \text{“\textit{Development}”}], \texttt{employs} : \{\}] \\
\delta(o_5) & = & [\texttt{name} : \text{“\textit{Andy}”}, \texttt{salary} : 8500, \texttt{works-in} :_6, \texttt{level} : 4] \\
\delta(o_6) & = & [\texttt{name} : [\texttt{bname} : \text{“\textit{Research}”}, \texttt{sname} : \text{‘\textit{Database}”}], \\
 & & \ \texttt{activity} : \{\text{“\textit{Databases, UserInterfaces}”}\}, \texttt{employs} : \{o_5\}] \\
\delta(o_7) & = & [\texttt{name} : \text{“ \textit{Peter}”}, \texttt{salary} : 10000, \texttt{works-in} : o_2, \texttt{head} : o_2, \texttt{level} : 8]
\end{array}
$$

Table 3: A possible database state

we can have many possible instances for virtual classes, some of them are shown in Table 4.

	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$
`Employee`	$\{o_1, o_3, o_5, o_7\}$	$\{o_1, o_3, o_5, o_7\}$	$\{o_1, o_3, o_5, o_7\}$
`Manager`	$\{o_7\}$	$\{o_7\}$	$\{o_7\}$
`Sector`	$\{o_6\}$	$\{o_6\}$	$\{o_6\}$
`Clerk`	$\{o_1, o_3, o_5\}$	$\{o_3, o_5\}$	$\{o_3\}$
`Department`	$\{o_2, o_4, o_6\}$	$\{o_4, o_6\}$	$\{o_4\}$
`Secretary`	$\{o_5\}$	$\{o_5\}$	$\{\}$
`Office`	$\{o_6\}$	$\{o_6\}$	$\{\}$

Table 4: Examples of possible instances

Since the interpretation of virtual classes shall be uniquely computable for a given assignment of object identifiers to interpretations of base classes, we have to single out one particular possible instance. The natural candidate for such a "canonical" interpretation would be the smallest possible interpretation (corresponding to the least fixpoint of an operator on interpretations). There are, however, drawbacks to such an approach. The least possible instance is often "too small", namely, empty. For instance, in our case the smallest possible instance is $\mathcal{I}_3$ in Table 4, and we have `Secretary` , `Office` empty; `Department` is not empty as we have made a particular assignment (the empty set) of employees to o_4. Furthermore, virtual classes with circular references using type constructor different from sequence and set would always be empty.[4] For this reason, we choose the *largest possible instance* as the "right one" ($\mathcal{I}_1$ in Table 4). Let $\mathbf{P}$ be the *set of possible instances with identical* $\mathcal{O}$ *and* δ such that for all $\mathcal{I}, \mathcal{I}' \in \mathbf{P}$:

$$\mathcal{I}[C] \quad = \quad \mathcal{I}'[C] \text{ for all } C \in \mathbf{C}.$$

Further, let "$\stackrel{P}{\trianglelefteq}$" be a relation over $\mathbf{P}$ such that for all $\mathcal{I}, \mathcal{I}' \in \mathbf{P}$:

$$\mathcal{I} \stackrel{P}{\trianglelefteq} \mathcal{I}' \quad \text{iff} \quad \mathcal{I}[N] \subseteq \mathcal{I}'[N] \text{ for all } N \in \mathbf{N}.$$

Then $(\mathbf{P}, \stackrel{P}{\trianglelefteq})$ forms a partial ordering. We say that $\mathcal{I}$ is a *legal instance* of a schema σ iff it is the unique *greatest* instance of the set $\mathbf{P}$ w.r.t. $\stackrel{P}{\trianglelefteq}$.

Theorem 1 *If $\mathcal{I}$ is a possible instance, then a legal instance $\mathcal{I}'$ exists such that $\mathcal{I} \stackrel{P}{\trianglelefteq} \mathcal{I}'$.*

It should be noted that in the ODL framework multiple inheritance is realized as a semantic property via the conjunction operator. If we take, for example, the class `Office` from Table 2, we note that it "inherits" from `Sector` and `Branch` since these are conjuncts in the defining type description of `Office`, i.e., the `Office` objects have values that satisfy the restrictions spelled out in `Sector` and `Branch`. This means that for the `name` attribute, which is defined in both `Sector` and `Branch`, the following restriction is met

`name`: [`bname`: `String`, `sname`: `String`],

i.e., the restrictions on the `name` attribute are simply conjunctively combined. In other words, it is not necessary to "resolve inheritance conflicts" on attributes that inherit different value ranges from multiple classes as in O_2 [21], but the value range of an attribute is simply the intersection over the value ranges of this attribute in all parent classes.

[4] See [6, 25, 9] for a more detailed description of the various semantics.

2.5 Inheritance, Subsumption, and Coherence

Based on the definitions above, we now give formal definitions for the notions of *subsumption* and *incoherence*. Given a schema σ, there is the question for the semantic relationship between types. For example, we would expect that a named type N that inherits from another named type N', i.e., $N \preceq_\sigma N'$, is always interpreted as a subset of the interpretation of N'. The converse, however, does not hold necessarily.

In order to capture this formally, let us define the *subsumption relation*, written $S \sqsubseteq_\sigma S'$ for $S, S' \in \mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ of a schema σ:

$$S \sqsubseteq_\sigma S' \quad \text{iff} \quad \mathcal{I}[S] \subseteq \mathcal{I}[S'] \text{ for all legal instances } \mathcal{I} \text{ of } \sigma.$$

It follows immediately that $\sqsubseteq_\sigma$ is a pre-order (i.e. transitive and reflexive) that induces an equivalence relation $\doteq_\sigma$ on types

$$S \doteq_\sigma S' \quad \text{iff} \quad S \sqsubseteq_\sigma S' \text{ and } S' \sqsubseteq_\sigma S$$

Proposition 1 *For a given well-formed schema σ and $N, N' \in \mathbf{N}$:*

$$\textit{If } N \preceq_\sigma N' \textit{ then } N \sqsubseteq_\sigma N'.$$

Reconsidering our example in Section 1.1, it is obvious that `Secretary` $\sqsubseteq_\sigma$ `Employee` and `Office` $\sqsubseteq_\sigma$ `Branch`. However, as we already pointed out, we also have `Office` $\sqsubseteq_\sigma$ `Department` and `Secretary` $\sqsubseteq_\sigma$ `Clerk`. Thus, the converse does not generally hold. One reason is the presence of virtual classes. Another reason is the fact that a type can be equivalent to $\perp$, the empty type. Class types and value types equivalent to $\perp$ are called *incoherent*. We require that all named class types and value types in a schema are *coherent* (i.e. $\not\doteq_\sigma \perp$), in which case the schema is called *coherent*. With this definition we can give a partial converse of the above proposition.

Proposition 2 *For a given well-formed and coherent schema σ, for all base classes $C, C' \in \mathbf{C}$: $C \sqsubseteq_\sigma C'$ iff $C \preceq_\sigma C'$.*

3 Algorithms

It is quite easy to check whether a schema is well-formed. It amounts to checking the graphs induced by value-type declarations and the inheritance relation for cycle-freeness. Testing for the coherence of a schema and computing subsumption between types is, however, more difficult. Instead of directly specifying algorithms for these problems, a detour will be taken, defining an algorithm on *canonical extensions* of a given schema.

A *canonical schema* ν over $\mathbf{S}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$, where $\overline{\mathbf{N}} = \overline{\mathbf{C}} \cup \overline{\mathbf{D}} \cup \overline{\mathbf{V}}$, has the following form (for all $C \in \overline{\mathbf{C}}, D \in \overline{\mathbf{D}}, V \in \overline{\mathbf{V}}$):

$$
\begin{array}{rcl}
\nu(C) & = & C \\
\nu(D) & = & \bigsqcap C_i \sqcap \triangle N \text{ where } C_i \in \overline{\mathbf{C}}, 0 \le i, N \in \overline{\mathbf{D}} \cup \overline{\mathbf{V}} \\
\nu(V) & = & \left\{ \begin{array}{ll}
\bot & \text{or} \\
B & \text{where } B \in \mathbf{B}, \text{ or} \\
\{N\} & \text{where } N \in \overline{\mathbf{N}}, \text{ or} \\
\langle N \rangle & \text{where } N \in \overline{\mathbf{N}}, \text{ or} \\
[a_1 \colon N_1, \ldots, a_p \colon N_p] & \text{where } p \ge 0, N_i \in \overline{\mathbf{N}}, 1 \le i \le p.
\end{array} \right.
\end{array}
$$

A schema ν over $\mathbf{S}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ is a *conservative extension* of a schema σ over $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$ iff $\mathbf{N} \subseteq \overline{\mathbf{N}}$ and for any legal instance $\mathcal{I}$ of σ a legal instance $\mathcal{I}'$ of ν exists and *vice versa* such that

$$\mathcal{I}[N] \;=\; \mathcal{I}'[N] \;\text{ for all } N \in \mathbf{N}.$$

Note that this implies that the subsumption relations on $\mathbf{N}$ are identical. A schema ν is called a *canonical extension* of σ iff ν is canonical and ν is a conservative extension of σ.

Theorem 2 *Any schema σ over $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$ can be effectively transformed into a schema ν over $\mathbf{S}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ that is a canonical extension of σ.*

The algorithm for transforming a given schema into an equivalent canonical schema is given in [10]. The main points are: every base class C is transformed into a virtual class (expressing it as the conjunction of an imaginary atomic class $\overline{C}$, which belongs to the set $\overline{\mathbf{C}}$, and its description $\sigma(C)$); the description of every class is transformed by substituting only the names of classes from which it inherits with their descriptions, expanding the type names and applying suitable conjunction rules together with the denomination of new types obtained as conjunctions. Therefore, at first $\overline{\mathbf{D}} = \mathbf{C} \cup \mathbf{D}$, $\overline{\mathbf{V}} = \mathbf{V}$ but these sets of names grow with the transformations made. The examples provided below should give a rough idea.

Using canonical schemata, checking coherence of a schema and computing subsumption is easy. As a matter of fact, both problems can be solved in time polynomial in the size of the canonical schema. For the purpose of coherence checking, let us define the notion of *conditional incoherence* of a type name N in a canonical schema ν based on a set of type names $I \subseteq \overline{\mathbf{N}}$. N is *conditionally incoherent* in ν based on $I \subseteq \overline{\mathbf{N}}$ iff[5]

$$\nu(N) \quad = \quad \begin{cases} \perp & \text{or} \\ [\ldots, a' \colon N', \ldots] & \text{where } N' \in I, \text{ or} \\ \bigsqcap C_i \sqcap \triangle N' & \text{where } N' \in I. \end{cases}$$

Algorithm 1 (Incoherence) *Let $I^0 = \emptyset$ and define*

$$I^{i+1} = I^i \cup \{N \in \overline{\mathbf{N}} | N \text{ is incoherent in } \nu \text{ based on } I^i\}.$$

Compute $I^0, \ldots, I^i, \ldots$ until $I^i = I^{i+1}$ and set $\widetilde{I_\nu} = I^i$.

Since I^i grows monotonically with increasing i and $\overline{\mathbf{N}}$ is finite, there exists a natural number n such that $I^n = I^{n+1}$, i.e, the algorithm terminates. Further note that the maximal n is $|\overline{\mathbf{N}}|$, i.e., the algorithm is polynomial in ν.

Theorem 3 *If ν is a canonical schema on $\mathbf{S}(\mathbf{A}, \mathbf{B}, \overline{\mathbf{N}})$, then $\nu(N) \doteq_\sigma \perp$ iff $N \in \widetilde{I_\nu}$.*

Corollary 1 *Let σ be a schema over $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, and let ν be a schema over $\mathbf{S}(\mathbf{A}, \mathbf{B}, \overline{\mathbf{N}})$ that is a canonical extension of σ. Then σ is coherent iff $\widetilde{I_\nu} \cap \mathbf{N} = \emptyset$.*

To give a simple example, let us define a new class `Typist`, which is a specialization of `Secretary`, with a level attribute value equal to 1 and a new class `TypeOffice` which is a branch employing only typists:

$$\begin{aligned} \sigma(\texttt{Typist}) &= \texttt{Secretary} \sqcap \triangle\Big[\texttt{level}\colon 1\Big] \\ \sigma(\texttt{TypeOffice}) &= \texttt{Branch} \sqcap \triangle\Big[\texttt{employs}\colon \texttt{Typist}\Big] \end{aligned}$$

[5] Note that $\{\perp\}$ and $\langle\perp\rangle$ are coherent types denoting the empty set and the empty sequence, respectively.

$\texttt{Typist}$ is immediately incoherent as from the canonical transformation its attribute $\texttt{level}$ has values in the type $\texttt{Level} \sqcap \texttt{MdmLevel} \sqcap 1$, which is equal to $\bot$, i.e., we have

$$\nu(\texttt{Typist}) = \bot.$$

The incoherence of $\texttt{TypeOffice}$ derives from the incoherence of the generated value-type $T13 = [\texttt{name}: [\texttt{bname}: \texttt{String}], \texttt{employs}: \texttt{Typist}]$, which is part of the canonical description of $\texttt{TypeOffice}$. Therefore:

$$\begin{array}{rcl} I^1 & = & \{\texttt{Typist}\} \\ I^2 & = & \{\texttt{Typist}, T13\} \\ I^3 & = & \{\texttt{Typist}, T13, \texttt{TypeOffice}\}, \end{array}$$

and thus $\texttt{Typist}$ and $\texttt{TypeOffice}$ are detected as incoherent.

Turning now to subsumption computation, we define *conditional subsumption* based on a relation $\leq \subseteq \overline{\mathbf{N}} \times \overline{\mathbf{N}}$. We say that S is *conditionally subsumed* by S'

based on $\leq$, written $S \sqsubseteq_{\leq} S'$ under the following conditions:

$$\begin{array}{rcl} B \sqsubseteq_{\leq} B' & \text{iff} & B \sqcap B' = B \\ \{N\} \sqsubseteq_{\leq} \{N'\} & \text{iff} & N \leq N' \\ \langle N \rangle \sqsubseteq_{\leq} \langle N' \rangle & \text{iff} & N \leq N' \\ [\ldots, a_i : N_i, \ldots] \sqsubseteq_{\leq} [\ldots, a'_j : N'_j, \ldots] & \text{iff} & \forall j \exists i : (a_i = a'_j \wedge N_i \leq N'_j) \\ \bigsqcap_i C_i \sqcap \triangle N \sqsubseteq_{\leq} \bigsqcap_j C'_j \sqcap \triangle N' & \text{iff} & N \leq N' \wedge \forall j \exists i : C_i = C'_j. \end{array}$$

We can now give the *subsumption* algorithm.

Algorithm 2 (Subsumption) *Let* $\leq^0 = \overline{\mathbf{D}}^2 \cup \overline{\mathbf{V}}^2 \cup \{(C, C) \mid C \in \overline{\mathbf{C}}\}$ *and define* $\leq^{i+1}$ *in the following way:*

$$\leq^{i+1} = \left\{ (N, N') \in \overline{\mathbf{N}} \times \overline{\mathbf{N}} \,\middle|\, \nu(N) \sqsubseteq_{\leq^i} \nu(N') \vee N \in \widetilde{I_\nu} \right\}.$$

Compute $\leq^0, \ldots, \leq^i, \ldots$ *until* $\leq^i \; = \; \leq^{i+1}$ *and set* $\lesssim_\nu \; = \; \leq^i$.

As above, a natural number n always exists such that $\leq^n = \leq^{n+1}$. This time, the maximal n is $|\overline{\mathbf{N}}|^2$, i.e., also the subsumption algorithm is polynomial in ν.

Theorem 4 *Given a canonical schema ν over $\mathbf{S}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$, for all $N \in \overline{\mathbf{N}}$:*

$$N \sqsubseteq_\nu N' \quad \textit{iff} \quad N \lesssim_\nu N'.$$

Further, using the property that a canonical extension of a schema is a conservative extension, it follows immediately that subsumption in an arbitrary schema can be computed by computing $\lesssim_\nu$ on its canonical extension.

Corollary 2 *Let σ be a schema over $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$, and let ν be a schema over $\mathbf{S}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ that is a canonical extension of σ. Then for all $N, N' \in \mathbf{N}$:*

$$N \sqsubseteq_\sigma N' \quad \textit{iff} \quad N \lesssim_\nu N'.$$

3.1 Subsumption for the Company Domain

Let us briefly show subsumption computation for the company domain, emphasising the problem that arise from the cyclic virtual classes (`Clerk`, `Department`), (`Secretary`, `Office`). Subsumption computation for a schema without cyclic classes can be performed with exactly three iterations of the subsumption algorithm, whereas cyclic references increase the number of iterations. At step $\leq_0$, we have that each class subsumes all the other classes and each type subsumes all types. At step $\leq_1$, we can drop many subsumption relationships in addition to those between non-homogeneous types. For instance, it is detected that the classes `Person` and `Branch` and the value type `Activities` are not subsumed by any other class or value type. At step $\leq_2$ we detect that `Department` $\not\sqsubseteq_\sigma$ `Office` supposing that the canonical extension ν of σ looks as follows:

$$\begin{aligned}
\nu(\texttt{Department}) &= \overline{\texttt{Branch}} \sqcap \triangle T9 \\
\nu(T9) &= [\texttt{name} : T2, \texttt{employs} : D2] \\
\nu(T2) &= [\texttt{bname} : \texttt{String}] \\
\nu(D2) &= \{\texttt{Clerk}\} \\
\nu(\texttt{Office}) &= \overline{\texttt{Branch}} \sqcap \triangle T12 \\
\nu(T12) &= [\texttt{name} : T11, \texttt{activity} : \texttt{Activities}, \texttt{employs} : D4] \\
\nu(T11) &= [\texttt{bname} : \texttt{String}, \texttt{sname} : \texttt{String}] \\
\nu(D4) &= \{\texttt{Secretary}\}
\end{aligned}$$

Since $T9 \not\lesssim_1 T12$, as the attribute $\mathtt{name}$ in $T12$ is defined on a strict subtype ($T11$) of the corresponding attribute $\mathtt{name}$ of $T9$ and $T12$ has the further attribute $\mathtt{activity}$, it follows that $\mathtt{Department} \not\lesssim_i \mathtt{Office}$, for all $i \geq 2$, hence $\mathtt{Department} \not\sqsubseteq_\sigma \mathtt{Office}$.

Assuming that ν contains the following additional type definitions

$$
\begin{array}{rcl}
\nu(\mathtt{Secretary}) & = & \overline{\mathtt{Person}} \sqcap \triangle T10 \\
\nu(T10) & = & [\mathtt{name} : \mathtt{String}, \\
 & & \mathtt{salary} : \mathtt{Real}, \mathtt{works\text{-}in} : D3, \mathtt{level} : \mathtt{MdmLevel}] \\
\nu(\mathtt{Clerk}) & = & \overline{\mathtt{Person}} \sqcap \triangle T8 \\
\nu(T8) & = & [\mathtt{name} : \mathtt{String}, \mathtt{salary} : \mathtt{Real}, \\
 & & \mathtt{works\text{-}in} : D1, \mathtt{level} : \mathtt{MdmLevel}],
\end{array}
$$

$\mathtt{Clerk} \not\sqsubseteq_\sigma \mathtt{Secretary}$ can be detected at step 4, as $\mathtt{Department} \not\lesssim_2 \mathtt{Office}$; $D1 \not\lesssim_2 D3$ and $T8 \not\lesssim_3 T10$. At step 5 we obtain $D2 \not\lesssim_5 D4$ and we can stop at step 6 as we have the same subsumption relations as at step 5:

$$
\begin{array}{rcl}
\mathtt{Secretary} & \leq_5 & \mathtt{Clerk} \\
\mathtt{Office} & \leq_5 & \mathtt{Department} \\
D4 & \leq_5 & D2 \\
 & \vdots &
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{Secretary} & \leq_6 & \mathtt{Clerk} \\
\mathtt{Office} & \leq_6 & \mathtt{Department} \\
D4 & \leq_6 & D2 \\
 & \vdots &
\end{array}
$$

and thus $\lesssim_\nu = \leq_5$.

At the end of the subsumption computation we obtain for each type name N the set of all its subsumers (i.e., its generalizations $GS(N)$) and subsumees (i.e., its specializations) with respect to the overall taxonomy. Elimination of redundancies is very easy, as we can easily compute for each type name N its most specific generalizations set $MSG(N)$ as follows:

$$MSG(N) = \{N' \in GS(N) \mid \not\exists N'' \in GS(N) \colon N'' \sqsubseteq_\sigma N' \wedge N' \not\sqsubseteq_\sigma N''\}$$

Figure 2 displays the class taxonomy induced by the company schema after MSG computation.

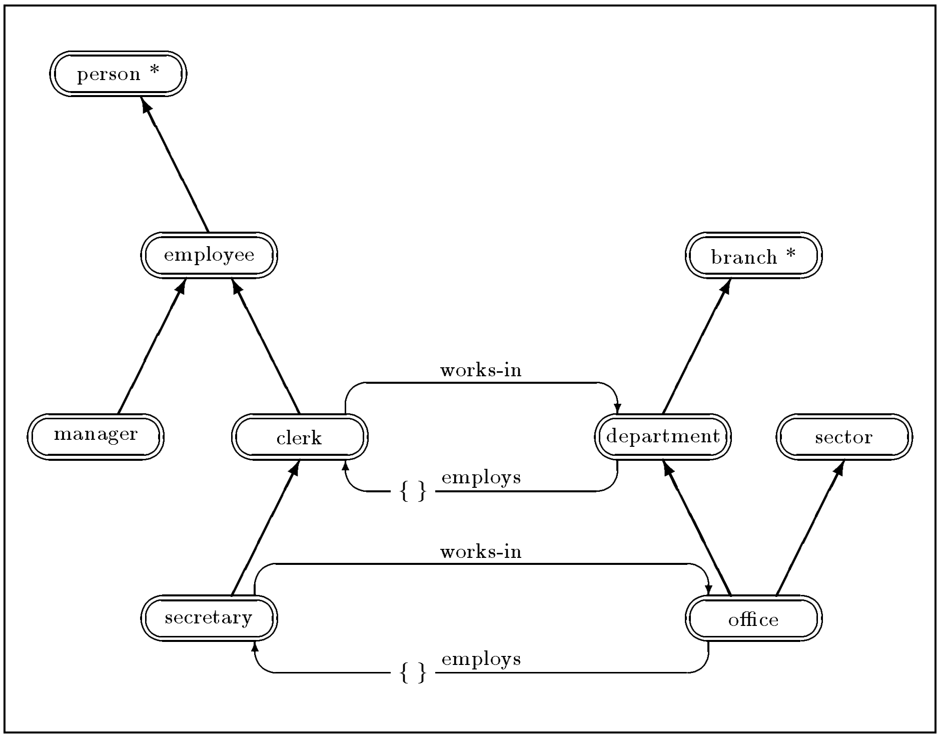

Figure 2: The minimal taxonomy for the company domain example

4 Computational Complexity

Although coherence checking and subsumption computation can easily be performed on canonical schemata, there are probably no efficient algorithms for the general case. Considering coherence checking first, it can be shown that binary compactness of the system of atomic types (see Section 2.1) is a crucial condition for efficiency.

Theorem 5 *Checking coherence of a schema can be done in polynomial time provided the system of atomic types is binary compact, and is NP-hard in the general case.*

The NP-hardness result follows by a reduction from the emptiness of intersection problem for finite state automata, which is NP-hard [18].

Subsumption in ODL is even more difficult, as the next theorem shows.

Theorem 6 *Subsumption is PSPACE-hard.*

Even a restriction to binary compact systems of atomic types does not help in this case because the language inclusion problem for non-deterministic finite state automata can be reduced to subsumption [24, 6, 25].

Since the coherence problem and subsumption problem on canonical schemata both can be decided in polynomial time, the above results imply that computing the canonical extension of a schema is difficult or that the canonical extension of a schema has a worst-case size that is exponential in the size of the original schema. As can be shown, there are indeed schemata that lead to exponentially sized canonical extensions. A worst-case example is given in [24].

Although these results may suggest that subsumption computation may not be feasible in our model, it turns out that the intractability of the problem does not very often show up in practice—an observation also made in the area of knowledge representation [24], where we have to deal with quite similar problems. The reason is probably that schemata are usually formulated in such a way that they are almost canonical, and, as is obvious from the description of the algorithms, coherence checking and subsumption computation on such schemata can be done efficiently.

More evidence for the fact that subsumption computation can be done efficiently in most practical cases can be found in related data models. In the data model O_2 [21], for instance, a relation called *refinement* is computed. This relation is identical to our subsumption relation if we assume that there are no base classes. Multiple inheritance in O_2 always leads to a request for a "conflict resolution" by the user [21]. Thus, instead of automatically using conjunctions of types—as we do—the user has to provide a type that refines the inherited types. The computation of the refinement relation is then performed on an internal normalized description, where no further inheritance is needed [20]. It is easy to see that this can be done in polynomial time.[6] Since the internal form can be handled efficiently, it must be the case that schemata exist that would lead to exponentially many conflict-resolution requests. However, such schemata are not very likely to appear in practice. Hence, we conclude that transforming a schema to its canonical extension is

[6]Polynomiality is guaranteed only if the disjunction operator discussed in that paper is omitted, a point that has been missed by Lecluse and Richard [20].

feasible for most cases that appear in practice. In fact, our experience with the ODL-DESIGNER tool, which is described below, supports this hypothesis.

5 The ODL-DESIGNER Tool

The ODL-DESIGNER prototype [7], which has been developed at CIOC-CNR, Bologna, as part of the LOGIDATA$^+$ project [4], implements taxonomic reasoning methods and techniques for advanced database management systems handling complex objects data models. It is an active tool which supports automatic building of type taxonomies for complex object database systems, preserving coherence and minimality with respect to inheritance. It implements the theoretical framework of Sections 2 and 3. Due to the generality of the ODL formalism, this tool can be used as a kernel component for schema acquisition for a large range of available research and commercial object-oriented database system.

The user interface of the current version of ODL-DESIGNER is quite rudimentary, supporting only keyboard interaction with the programming system Quintus Prolog, in which ODL-DESIGNER is implemented. This lack in emphasis on the user interface has two reasons. First of all, we needed a quick prototype implementation in order to test the practical feasibility of the theoretical framework of Sections 2 and 3. Second, within the LOGIDATA$^+$ project, ODL-DESIGNER is a kernel component which will be connected in the second phase of the project with the graphical user interface supporting class taxonomies and with the object store system developed by other research groups [7].

Figure 3 shows the functional architecture of ODL-DESIGNER. The program is divided into two main functional modules:

F1: allows the creation of an ODL schema, the addition of new classes to a pre-existing ODL schema, and the modification of a pre-existing ODL schema. It performs coherence control and produces the canonical form of the ODL schema. The output of this module is the canonical form of the ODL schema and the list of incoherent classes and types (if any).

F2: performs subsumption and MSG computations having as input the canonical form of the ODL schema. The output of this module is the minimal ODL schema and the list of computed *isa* relations.

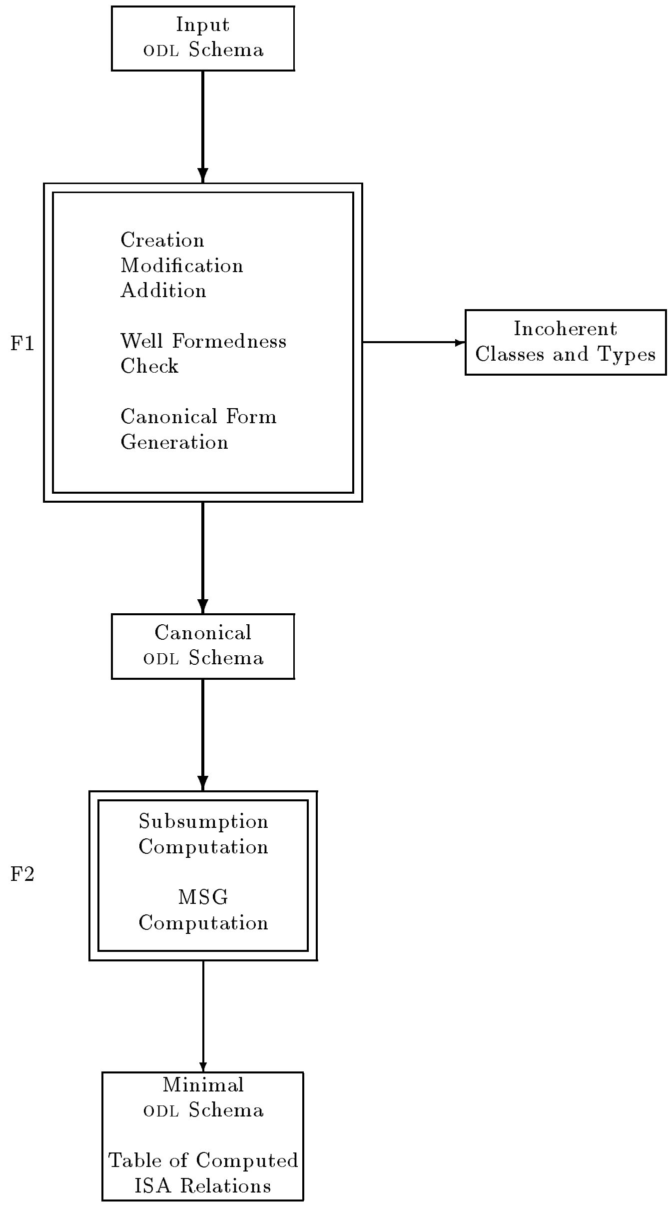

Figure 3: ODL-DESIGNER's functional architecture

The tool is able to guarantee coherence for a taxonomy consisting of only base classes by activating the **F1** module. Class descriptions of a given schema can be compared and *incoherent* classes are detected. Thus, the **F1** module can constitute a type-checking module for an object-oriented database management system.

A more active role is played if the taxonomy includes also virtual classes and value-types. In this case it is necessary to activate also the **F2** module. This module computes a *minimal description* (i.e., a rewritten description on the basis of its most specific generalizations) for each class and value-type and puts them into the right places in the taxonomy. Semantic equivalence of classes is recognized and redundancies with respect to inheritance are removed.

In general, database schema design can be subdivided in two phases. The first one is devoted to the application domain requirements analysis and its formal mapping into an ODL schema. This phase is supported by the CREATE/MODIFY function of ODL-DESIGNER. The second phase is devoted to top-down and bottom-up refinements in order to achieve the appropriate modelling of a target application domain. This phase is supported by the ADD function of ODL-DESIGNER.

The CREATE/MODIFY function is activated in the first phase of the schema design. A skeleton schema is created and class and value-type descriptions are added and modified many times. Figure 4 shows the behaviour of ODL-DESIGNER when the CREATE/MODIFY function is activated. Ellipses denote ODL-DESIGNER sub-modules, thick arrows sub-module links, and dashed arrows the data flow. The user interface allows schema creation by typing descriptions at the terminal or by giving the name of a text file containing a schema. The input schema is then analyzed from a syntactic and semantic point of view. First, syntactic and semantic checks are performed, implementing the well-formedness conditions for value-types and the inheritance relation, as spelled out in Section 2.3. If an error is detected, the systems supports the user interactively in producing a well-formed ODL schema. When the input schema is well-formed, the canonical transformation algorithm is applied, followed by the application of the incoherence algorithm specified in Section 3. The input schema and the canonical schema are made persistent and the list of incoherent classes and types (if any) is displayed. At the presence of incoherent classes or value-types the designer can modify descriptions, activating **F1** again.

After a successful run of the **F1** module, which resulted in a canonical schema, the module **F2** is activated. This module applies the subsumption

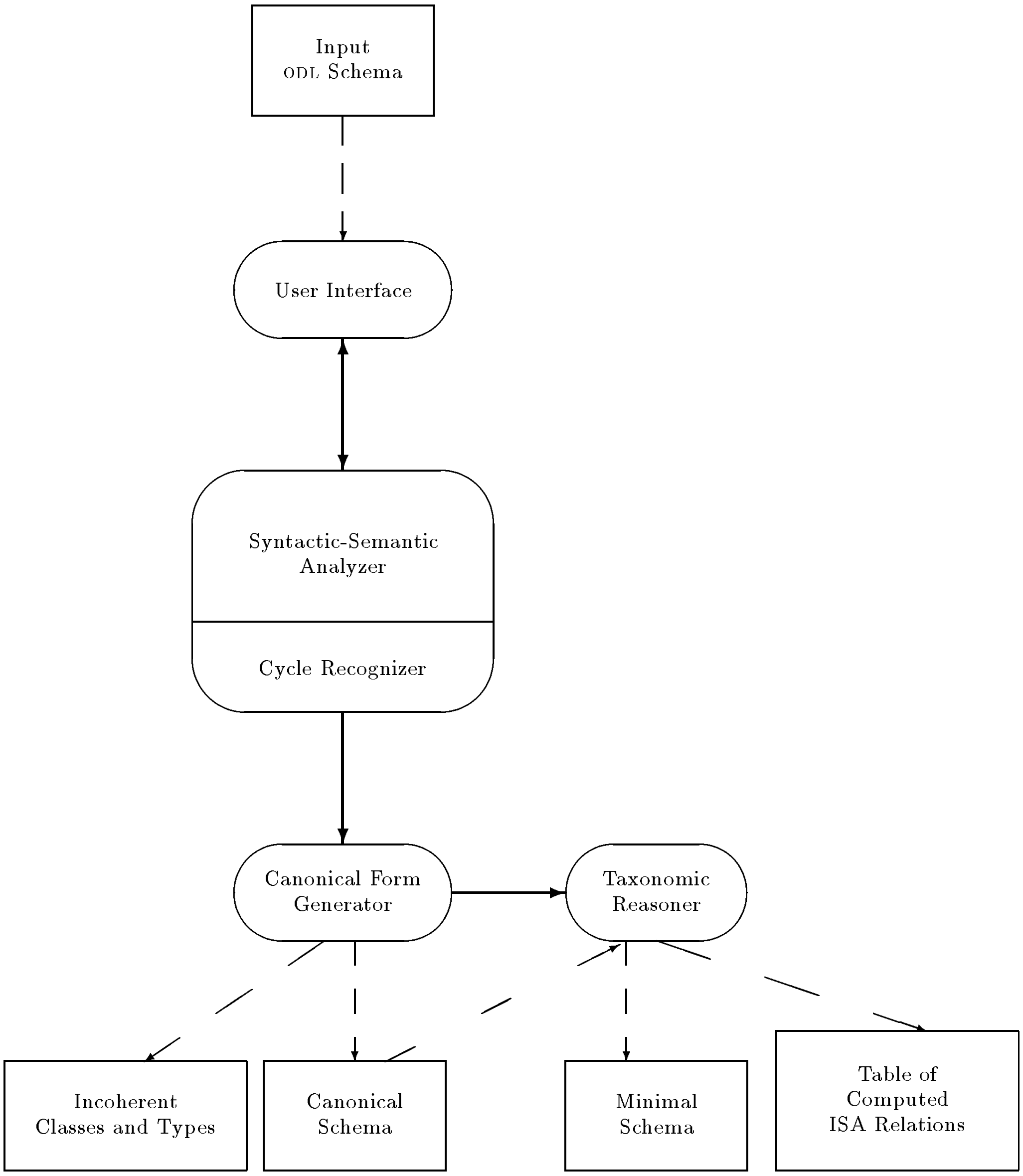

Figure 4: ODL-DESIGNER: CREATE Function

algorithm specified in Section 3 to the canonical schema in order to compute the table of *isa* relations. Finally, the minimal schema is derived by computing the MSGs for all classes and value-types. The table of computed *isa* relations is made persistent and is displayed.

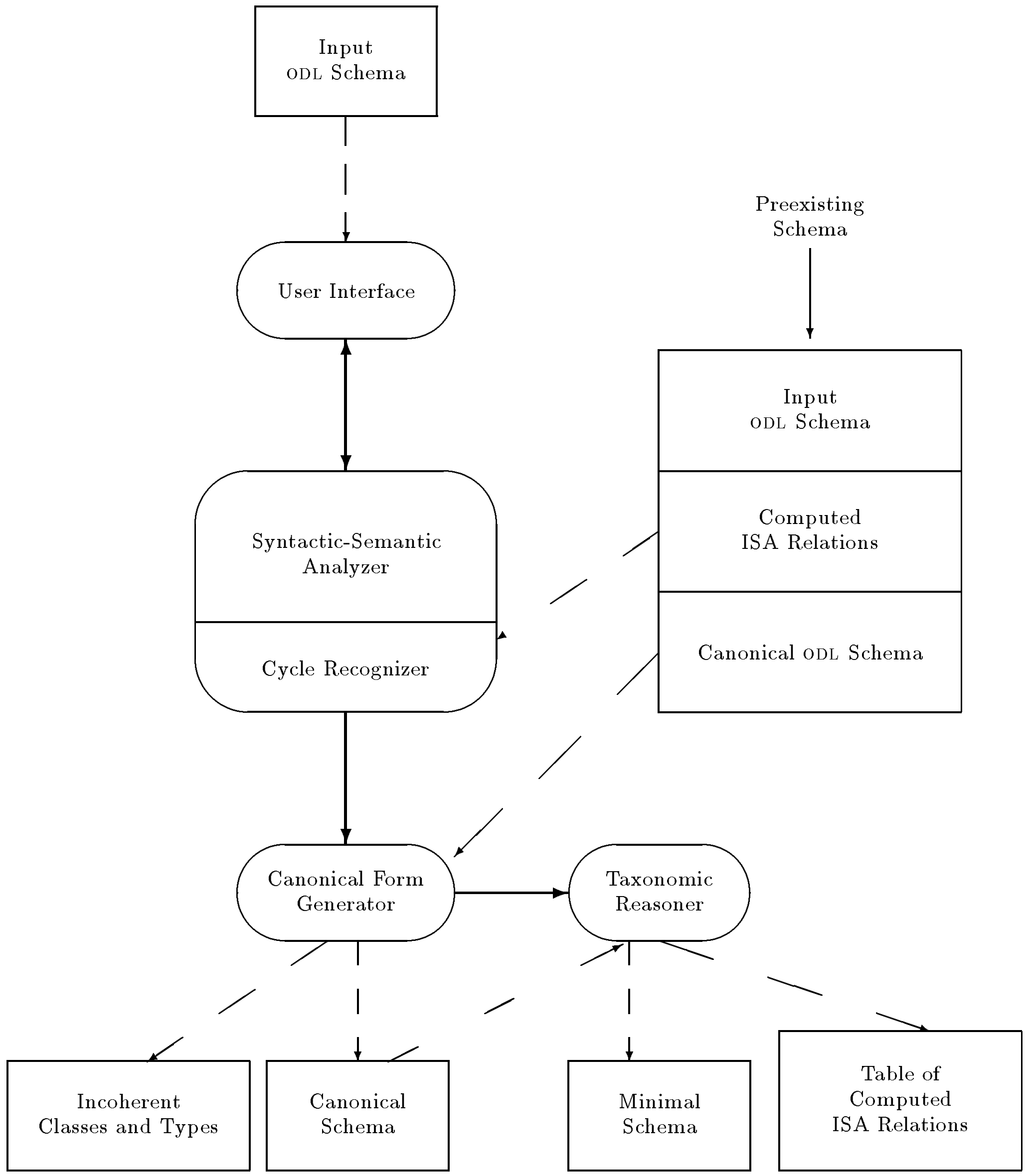

Figure 5: ODL-DESIGNER: ADD Function

In the second design phase, the database schema is quite stable and schema modifications have the flavor of a little at a time refinements. The ADD function of Figure 5 is useful for this second phase. The main difference, with respect to the CREATE/MODIFY functions, is that the ADD function exploits the results of the "compilation" of a pre-existing database schema, avoiding syntactic and semantic analysis algorithm, since it compares new descriptions, which are potentially incorrect, with a corpus of well-formed definitions. If the user tries to modify descriptions of the pre-existing schema during an ADD activation, the system refuses the operation, pointing out that it is necessary to use the CREATE/MODIFY functions.

Let us give an example of a sample session. When ODL-DESIGNER is activated, the options: *create-db, modify-db, add-db* are available. Suppose we want to create the *company* domain schema. In response to the system prompt we type

```
create-db(company).
```

and insert the schema description by Prolog assertions as follows:

```
s([type,Level,=,1,-,10]).
s([type,MdmLevel,=,2,-,7]).
s([type,LowLevel,=,1]).
...
s([class,Person,=,/,\,'[',name,:,String,']']).
...
s([virt-class,Clerk,=,Employee,&,
      /,\,'[',works-in,:,Department,']']).
...
s([end]).
```

Typing "`s([end]).`" triggers the execution of the module **F1**, which detects no errors or incoherencies, and the *company* ODL schema and its canonical form are made persistent. After that the user is asked whether he wants to compute the subsumption relation or to modify the schema. If we select the first alternative, module **F2** is activated, the table of *isa* relations depicted in Figure 1 and the minimal ODL schema depicted in Figure 2 are made persistent and displayed on the terminal.

Suppose we select the second alternative because we forgot to insert a `Typist` description. After invoking the ADD function by

```
add(company).
```

we type the following assertions:

```
s([virt-class,Typist,=,Secretary,&,
     /,\, '[',Level,:,LowLevel,']']).
s([end]).
```

The ODL-DESIGNER detects that the class `Typist` incoherent, as `LowLevel` and `MdmLevel` are disjoint, and suggests to correct its description. Suppose we want to modify the definition of `Person` and type the command

```
modify(company).
```

followed by

```
s([class,Person,=,Clerk,&,/,\,'[',name,:,String,']']).
s([end]).
```

In this case, ODL-DESIGNER detects that there is an `isa` cycle (`Clerk isa Person` and `Person isa Clerk`) and, again, suggests to correct this definition.

The current version of ODL-DESIGNER has been implemented in Quintus Prolog on a SUN SPARC station. The code amounts to 4130 code lines (158 predicates) and has been tested on a number of different sample database schemata.

6 Conclusions

We specified a formalism, called ODL, for complex objects following recent approaches in the areas of object-oriented and complex object data models. Instead of viewing inheritance as a syntactic transformation (as in O_2 [21]), inheritance is expressed by conjunctions of types. Additionally, we introduce the notion of virtual classes, which is similar to database views, and allow the definition of cyclic references in class definitions.

We specified algorithms to check schemata for well-formedness on the syntactic level (value-type well-formedness and inheritance relation well-formedness) and the semantic level (coherence of the schema), and addressed the problem of automatically computing the taxonomy of classes and value-types by specifying a subsumption algorithm that detects all implied specialization relationships.

Using recent results from knowledge representation research, we showed that coherence detection and subsumption computation in ODL are computationally intractable. However, we argued that in all situations appearing in practice worst cases are rare and that our algorithms are feasible in practice because schemata are usually “almost canonical.”

Based on these theoretical results, a schema acquisition and validation tool, the ODL-DESIGNER, has been implemented, which supports the above hypothesis of practical feasibility. Furthermore, this tool demonstrates the relevance of basing schema acquisition and validation on the ODL framework.

Let us briefly compare our work with some recent works close to our approach. The idea of introducing virtual classes and subsumption in a database schema has been considered in a number of papers [8, 11, 12]. Our work extends the framework of all the above quoted papers since we allow *cyclic* virtual classes specification. In [1] virtual classes are introduced in an object-oriented database schema, with the aim of introducing a sophisticated view mechanism. Our formalism does not enable all the facilities proposed in this paper, but includes possibilities for populating cyclic virtual classes not considered in this paper. Relating our work to research in knowledge representation, it turns out that ODL-DESIGNER viewed as a terminological knowledge representation system is one of the few systems that are able to deal correctly with cyclic classes [19].

References

[1] S. Abiteboul and A. Bonner.
Objects and views.
In *Proceedings of the 1991 ACM SIGMOD International Conference on Mangement of Data*, pages 238–247, 1991.

[2] S. Abiteboul and R. Hull.
IFO: A formal semantic database model.

ACM Transactions on Database Systems, 12(3):525–565, Dec. 1987.

[3] A. Albano, L. Cardelli, and R. Orsini.
Galileo: a strongly typed, interactive conceptual language.
ACM Transactions on Database Systems, 2:230–260, 1985.

[4] P. Atzeni, editor.
LOGIDATA+: Deductive Databases with Complex Objects.
Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 1993.

[5] P. Atzeni and D. S. Parker.
Formal properties of net-based knowledge representation schemes.
Data & Knowledge Engineering, 3:137–147, 1988.

[6] F. Baader.
Terminological cycles in KL-ONE-based knowledge representation languages.
In *Proceedings of the 8th National Conference of the American Association for Artificial Intelligence*, pages 621–626, Boston, MA, Aug. 1990. MIT Press.

[7] J. P. Ballerini, S. Bergamaschi, and C. Sartori.
The ODL-DESIGNER prototype.
In P. Atzeni, editor, *LOGIDATA+: Deductive Databases with complex objects*. Springer-Verlag, 1993.

[8] H. W. Beck, S. K. Gala, and S. B. Navathe.
Classification as a query processing technique in the CANDIDE semantic data model.
In *Proceedings of the International Data Engineering Conference, IEEE*, pages 572–581, Los Angeles, CA, Feb. 1989.

[9] D. Beneventano and S. Bergamaschi.
Subsumption for complex object data models.
In *Proceedings of the International Conference on Database Theory*, Berlin, Germany, 1992. Springer-Verlag.

[10] S. Bergamaschi and J. P. Ballerini.
Automatic building and validation of complex object database schemata.
Technical Report 91, CIOC-CNR, Bologna, Italy, Dec. 1992.

[11] S. Bergamaschi and C. Sartori.
On taxonomic resoning in conceptual design.
ACM Transactions on Database Systems, 17(3):385–422, 1992.

[12] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick.
CLASSIC: a structural data model for objects.
In *Proceedings of the 1989 ACM SIGMOD International Conference on Mangement of Data*, pages 59–67, Portland, OR, 1989.

[13] R. J. Brachman and J. G. Schmolze.
An overview of the KL-ONE knowledge representation system.
Cognitive Science, 9(2):171–216, Apr. 1985.

[14] L. A. Cardelli.
Semantics of multiple inheritance.
In *Semantics of Data Types*, pages 51–67. Springer-Verlag, Berlin, Heidelberg, New York, 1984.

[15] P. P. Chen.
The entity-relationship model - towards a unified view of data.
ACM Transactions on Database Systems, 1(1):9–36, 1976.

[16] L. Delcambre and K. Davis.
Automatic validation of object-oriented database structures.
In *5th International Conference on Data Engineering*, pages 2–9, Los Angeles, CA, 1989.

[17] T. W. Finin and D. Silverman.
Interactive classification as a knowledge acquisition tool.
In L. Kerschberg, editor, *Expert Database Systems—Proceedings From the 1st International Workshop*, pages 79–90. Benjamin/Cummings, Menlo Park, CA, 1986.

[18] M. R. Garey and D. S. Johnson.
Computers and Intractability—A Guide to the Theory of NP-Completeness.
Freeman, San Francisco, CA, 1979.

[19] J. Heinsohn, D. Kudenko, B. Nebel, and H.-J. Profitlich.
An empirical analysis of terminological representation systems.
In *Proceedings of the 10th National Conference of the American Association for Artificial Intelligence*, pages 767–773, San Jose, CA, July 1992. MIT Press.

[20] C. Lécluse and P. Richard.
Modeling complex structures in object-oriented databases.
In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database-Systems*, pages 360–367, Philadelphia, PA, 1989.

[21] C. Lécluse and P. Richard.
The O_2 database programming language.
In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 411–422, Amsterdam, The Netherlands, 1989.

[22] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong.
A language facility for designing interactive database-intensive systems.
ACM Transactions on Database Systems, 5(2):185–207, 1980.

[23] B. Nebel.
Reasoning and Revision in Hybrid Representation Systems, volume 422 of *Lecture Notes in Artificial Intelligence*.
Springer-Verlag, Berlin, Heidelberg, New York, 1990.

[24] B. Nebel.
Terminological reasoning is inherently intractable.
Artificial Intelligence, 43:235–249, 1990.

[25] B. Nebel.
Terminological cycles: Semantics and computational properties.
In J. F. Sowa, editor, *Principles of Semantic Networks*, pages 331–362. Morgan Kaufmann, San Mateo, CA, 1991.