ConceptBase - A Deductive Object Base Manager*

Matthias Jarke, Stefan Eherer, Rainer Gallersdörfer
Manfred A. Jeusfeld, Martin Staudt

Informatik V, RWTH Aachen, Ahornstr. 55, D-52056 Aachen, Germany
{jarke,eherer,gallersd,jeusfeld,staudt}@informatik.rwth-aachen.de

Abstract

Deductive object bases attempt to combine the advantages of deductive relational databases with those of object-oriented models. We review modeling and optimization issues encountered during the development of ConceptBase, a prototype deductive object base supporting the Telos data model. We also report on a number of application experiences in the field of meta data management.

*This work was supported in part by the Commission of the European Community under ESPRIT Basic Research Action 6810 (Compulog 2) and by the Ministry of Science and Research of Nordrhein-Westfalen.

1 Introduction

Proposals for next-generation databases tend to stem from three traditions [BMS84]:

- *Databases:* This stream is best typified by object-oriented and extended relational databases (e.g., NF2). The main extension is the addition of complex domains, shared sub-objects, and procedures to the database. Examples include Postgres, Damokles, Cactis, Iris, DASDBS, and many others.
- *Programming languages:* The main goal is to provide persistence and sharing to one or more programming languages, ideally orthogonal to the type systems of these languages. Examples include the O_2 system for imperative programming languages, and typed versions of Prolog such as LOGIN or PROTOS-L.
- *Knowledge representation:* Some of these systems aim at providing database service to AI applications, others come from the tradition of deductive databases or semantic data models and aim at formal as well as practical support for general database systems and applications.

A sample of the former two groups of systems is described in [TKDE90], whereas several prototypes of the latter are reported in [SIGA91]. Although a certain confluence can be observed, these systems do not only differ in their background theories but also in their intended application domains.

The first group is typically intended to support non-standard applications such as the handling of complex engineering objects. The second group mostly aims at an easier programming environment for standard applications: additionally, they emphasize the goal of making applications written in different languages interoperable.

The third group pursues, as one of its goals, the support for AI applications such as natural language understanding, expert systems, and recursive search tasks. However, this may remain a fairly limited portion of the software market. The reason why we became interested in this group of languages is therefore a slightly different one: the important role we expect them to play in meta data management. Such applications range from the uniform access to heterogeneous data sources, to the coordination of design processes, to the integration of heterogeneous information services in networked production chains (CIM), whole enterprises, or even transnational networks. In these applications which are crucial for the huge integration tasks to be tackled in the 1990s, the possibility not only to *execute* systems but also to *reason* formally about their structure and capabilities, can be considered a competitive advantage over the other approaches. The ConceptBase prototype, described in this paper, has provided some validity to this claim by extensive usage experiments in almost all of the above-mentioned areas.

The language supported by ConceptBase, Telos ([STAN86],[MBJK90]), has been one of the earliest attempts to integrate deductive and object-oriented database features, Telos takes a very conservative approach to this problem, with simplicity as the main goal. This emphasis on simplicity has paid off both in user acceptance and in ease of implementation.

In contrast to other object-oriented databases, with a minor transformational trick, the O-Telos [JEUS92] variant of Telos used in ConceptBase can be shown to be equivalent in expressiveness to Datalog with stratified negation and perfect model semantics. Thus, our design goal of being able to reason about a system's structure and capabilities is addressed. Moreover, all existing and future analysis and implementation techniques for deductive databases can be reused for Telos implementations. For example, the currently distributed version of ConceptBase supports a variant of the Supplementary Magic Set method for recursive query evaluation, and an extension of the deductive integrity checking method proposed by [BDM88].

For large systems, deductive capabilities are not enough but must be augmented with semantic structuring as offered in the object-oriented approach. In contrast to standard deductive databases, any O-Telos database includes a number of structural axioms that can be expressed as facts, rules, or integrity constraints of the corresponding Datalog model. These axioms define the naming conventions as well as the abstraction principles (classification, generalization, aggregation) of object-oriented databases and also allow the specification of certain methods through deductive rules and constraints. An important design goal was that all this was achieved without leaving the Datalog framework [JJ91]; recently, other researchers have also recognized the importance of this point [ALU93]. ConceptBase exploits the structural axioms in three ways:

- At the user interface level, it offers a choice of structured frame and graph syntax in addition to the usual logical language. Unusual features include the hypertext-like switching between these notations, the treatment of attributes as full-fledged objects, and the availability of an infinite classification hierarchy which is particularly suitable for repository applications.

- At the transformational level, the axioms lead to certain semantic controls and pre-optimizations that can be applied to rules, constraints, and repetitive queries when these are entered, and improve correctness and efficiency of rule evaluation.

- At the storage level, they lead to a special-purpose object store with a fully inverted representation of objects and a special object algebra on which set-oriented bottom-up processing of Magic Set transformed rules is realized.

In section 2, we describe the syntax and semantics of O-Telos, including the semantic optimizations. The ConceptBase system itself has a client-server architecture where the server supports queries (ASK) and updates (TELL) of O-Telos object bases and clients offer user interface facilities and application-specific processing. The implementation of the server is described in section 3, the client-server interaction in section 4.

Versions of ConceptBase have been distributed for research experiments since early 1988. The current distribution version, ConceptBase V3.2, has been installed at more than one hundred sites worldwide and is seriously used by about a dozen research projects in Europe and North America [1]. Section 5 summarizes some of the application experiences. Section 6 concludes with a brief comparison to related work and the description of ongoing ConceptBase extensions.

2 The O-Telos Data Model

There are two equivalent definitions of a deductive object base, depending on the parent discipline you wish to emphasize:

- **logic perspective:** A deductive object base is a deductive database (i.e., a logical theory) which consistently includes a pre-defined set of so-called structural axioms.
- **object perspective:** A deductive object base is an object-oriented database in which deductive rules and integrity constraints are the only means to specify methods.

In this section, we describe the O-Telos deductive object base model mostly from the logic perspective. A deductive object base is a triple $DOB = (OB, R, IC)$ where OB is the extensional object base, R and IC contain deduction rules and integrity constraints. As usual we require that (OB, R, IC) is consistent, i.e. $OB \cup R \models IC$. The object-oriented dictions like object identity are encoded as predefined deductive rules and integrity constraints. Thereby, O-Telos formally remains in the framework of deductive databases and inherits the well-known fixpoint semantics from there.

The following subsections define the O-Telos data model in three steps. *First*, the semantics of the O-Telos object structure is defined by specifying the extensional database structure and the predifined axioms. The semantic structure allows three equivalent surface syntax varieties which the ConceptBase user interface combines in a hypertext-like style: the underlying logical language, a graph syntax which interprets instances of the base relations as nodes or arcs, and a frame syntax which groups arcs around nodes or other arcs in a certain way. From a structural viewpoint, O-Telos is a very powerful semantic modeling language whose distinguishing features include attributes as full-fledged objects and a potentially infinite hierarchy of classification (meta classes).

Next, the integration of deduction rules, integrity constraints, and query classes in the same framework is described; query classes, another novel feature of O-Telos, are a form of parameterized view definition which seems to capture particularly well the integration of deductive and object-oriented aspects.

[1] A scaled-down version is available via anonymous ftp from ftp.informatik.rwth-aachen.de, directory pub/CB.

With these two sections, we have a complete picture of the O-Telos syntax and semantics. However, this "external" object base semantics is not very suitable for semantic analysis of the database with respect to features such as correct typing, stratification, or precision of query processing and integrity checking algorithms. We convert user-defined deductive rules, integrity constraints and query classes into an "internal" format which partitions the single original base relation in a large number of very small base relations. Constants are assigned uniquely to objects identifiers, variables are bound to classes, and predicates are assigned to attributes resp. classes. An assignment failure is reported as structural error (according to O-Telos axioms). This partitioning then leads to a number of optimizations on rules, queries, and constraints which can be applied at compile time and independently of the actual evaluation techniques used later on. In the process of converting the external to the internal DOB, constants are largely replaced by internal OIDs which is later exploited in efficient storage access (cf. section 3).

The language and implementation techniques are illustrated with the following simple scenario:

> *A company has employees, some of them being managers. Employees have a name and a salary. They are assigned to departments which are headed by managers. The boss of an employee can be derived from his department and its respective manager. No employee is allowed to earn more money than his boss.*

In the formalisms below, we employ small italic letters for variables, capital italic letters for predicate names, and computer font for example constants.

2.1 The O-Telos Object Structure

Let ID and LAB be sets of identifiers, and labels resp. An **extensional O-Telos object base** is a finite subset

$$OB \subseteq \{P(o,x,l,y) \mid o,x,y \in ID, l \in LAB\}.$$

The elements of OB are called **objects** with identifier o, source and destination components x and y and name or label l. Object identifiers are system-generated; to represent them, we shall adopt the convention that the identifier of an object with name `i` is written as `#i`

An extensional object base can be visualized as a structured semantic net. Objects of the form $P(o,o,l,o)$ (called *individuals*) are represented as nodes with name l. Instantiations of the form $P(o,x,in,c)$ (= "x is in class c") and specializations of the form $P(o,c,isa,d)$ (= "c is subclass of d") are represented as dotted resp. shaded directed links. All other objects $P(o,x,l,y)$ are called attributes and are drawn as l labeled directed links between x and y.

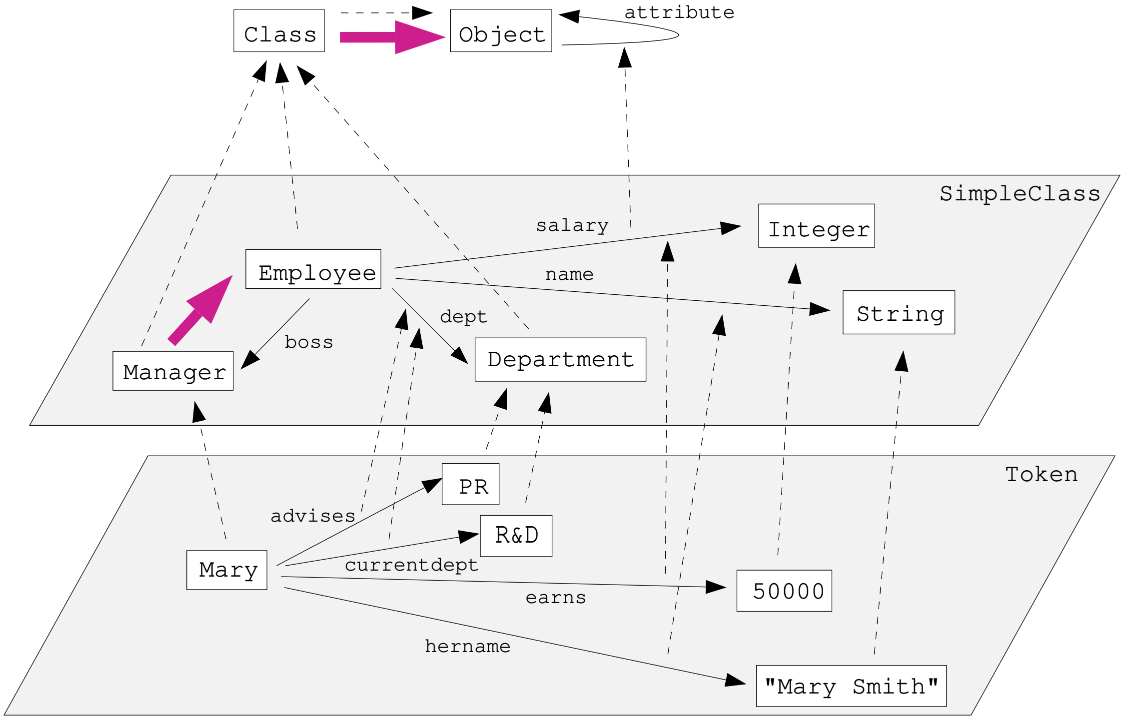

Figure 1: Semantic net view of an example object base

Fig. 1 shows an example graph. *OB* contains an object named `Mary` who is instance of the object `Manager` and works in two departments `PR` and `R&D`. In addition, the object base contains `Mary`'s full name and her salary. Obviously *OB* has two layers: a so-called *Token* layer which comprises all concrete objects which are direct representatives of real world objects and a *SimpleClass* layer which corresponds to the schema in traditional databases. There is no restriction on the number of layers; the schema can be instance of a set of metaclasses, these can be instances of metametaclasses, and so on. As can be seen from fig. 1 , the top layer of an object base is predefined by the objects `Object` and `Class` . The former contains all objects in *OB* as instances and the latter all objects which have instances themselves.

A third syntactic representation of deductive object bases results in a frame-based notation which only relies on object labels rather than object identifiers. Around the label l of an object o we group the labels of all other objects which have o as source component. The frame notation of the objects `Manager`, `Employee` and `Mary` is e.g.

```
Employee in Class with
  attribute
    dept:Department;
    boss:Manager;
    salary:Integer;
    name:String
end

Manager in Class isA Employee end
```

```
Mary in Manager with
  dept
    currentdept:R&D;
    advises:PR
  salary
    earns:50000
  name
    hername:"Mary Smith"
end
```

For each attribute the frame description includes the labels of its attribute classes, e.g. `dept` is assigned to `currentdept`.

The object model O-Telos has a couple of object-oriented dictions present in most object-oriented data models and each addressing a certain objective. In our approach, these objectives are simply encoded as predefined deductive rules and integrity constraints, referred as the axioms of O-Telos. The following list presents the four interacting dictions and some of the axioms realizing their semantics (see appendix for a complete list).

Object identity and referencing. Each object must have a unique identification, and each referenced object must exist.

External naming of objects. Each object can be referenced by a unique sequence of labels. For an individual object, this reference is just its label. For other objects, e.g. attribute, the sequence is constructed from the labels of its source and destination components. This justifies the usage of only external labels in the frame syntax and guarantees a unique mapping of frames to the other syntax forms.

Definition of abstraction principles as derived relations. Instantiation, generalization, and attribute relationships between two objects can also be derived. This goal is realized by defining the following deductive ruler for the three predicates In (for instantiation relationships), Isa (generalization)and A[2] (attribution).

$$\forall o,x,y\ P(o,x,in,y) \Rightarrow In(x,y)$$

$$\forall o,c,d\ P(o,c,isa,d) \Rightarrow Isa(c,d)$$

$$\forall o,x,l,y,p,c,m,d\ P(o,x,l,y) \wedge P(p,c,m,d) \wedge In(o,p) \Rightarrow A'(x,l,m,y)$$

$$\forall x,l,m,y\ A'(x,l,m,y) \Rightarrow A(x,m,y)$$

For our example we can e.g. deduce that *In(#Mary,#Manager)*, *A(#Mary,dept,#PR)*, *A(#Mary,dept,#R&D)*, *A(#Mary, name, "Mary Smith")*, and *A(#Mary,salary,50000)* hold in the object base. One should note that these rules are only the predefined ones. Any O-Telos object base may define further rules with the same conclusion predicate (see next subsection).

Definition of abstraction semantics. The attributes of an object must be correctly typed according to the attribute definitions of its classes. This goal is formally represented as an integrity constraint:

$$\forall o,x,l,y,p\ P(o,x,l,y) \wedge In(o,p) \Rightarrow \exists c,m,d\ P(p,c,m,d) \wedge In(x,c) \wedge In(y,d)$$

In combination with an axiom that defines the semantics of generalization (isA) links, namely, instances of an object are also instances of its superclasses,

$$\forall o,x,c,d\ In(x,d) \wedge P(o,d,isa,c) \Rightarrow In(x,c)$$

[2] We will show in a later section why we need the predicate A' which is less general than A.

the typing condition allows attribute inheritance of subclasses from superclasses[3]. In the example, `Mary` is also an instance of the object `Employee`.

The above properties make up the "core" of the object-orientation in O-Telos. Additional properties like non-circularity of part-of relationships can easily be defined in the same way. Since each property is defined within the framework of deductive databases (Datalog with negation) the unique model is guaranteed by the fixpoint semantics. Of course, as for all deductive databases containing integrity constraints, the database designer should make sure that there exists at least one extensional object base satisfying the integrity constraints.

2.2 Deduction Rules, Constraints, and Query Classes

In addition to the O-Telos axioms the sets R and IC of a deductive object base (OB, R, IC) may contain application-specific deduction rules and integrity constraints which are specified in a many-sorted first order language interpreted under closed-world assumption. Quantified variables range over classes and are interpreted as instantiation relationships. The admissible literals of the language are label-based abbreviations of the predicates In, Isa and A.

We illustrate this for our standard example by extending the definition of `Employee` with a rule and a constraint. The (recursive) rule deduces for a given employee the managers who are his bosses, i.e. are head of a department he works for or are themselves bosses of one of his department chairs. Based on this, one may impose an integrity constraint on `Employee` that all its instances are not allowed to earn more money than their bosses.

```
Employee in Class with
  attribute
    ...
  rule
    bossrule:$ forall t/Manager
                ( exists d/Department (this dept d) and (d head t)
               or exists m/Manager (this boss m) and (m boss t) )
                ==> ( this boss t) $
  constraint
    salaryIC:$ forall m/Manager x,y/Integer
                (this boss m) and (this salary x) and (m salary y) ==> x <= y $
end
```

`this` refers to the instances of `Employee` and e.g. `(d head t)` corresponds to $A(d, head, t)$. The constants `Manager`, `Department` etc. are replaced by object identifiers according to axiom 2 to 4 for external naming of objects. Using our convention, the identifier for

[3]Since rules and constraints are seen as attribute classes in the object perspective of O-Telos, method inheritance is also guaranteed by this axiom.

`Manager` is *#Manager*. By this, we get the following logical representation of the boss rule:

$$\begin{aligned}\forall e,t\quad & In(e,\#Employee) \wedge In(t,\#Manager) \wedge \\ & \exists d\; In(d,\#Department) \wedge A(e,dept,d) \wedge A(d,head,t) \vee \\ & \exists m\; In(m,\#Manager) \wedge A(e,boss,m) \wedge A(m,boss,t) \\ & \Rightarrow A(e,boss,t)\end{aligned}$$

Queries are treated in a very similar way to rules and constraints. However, to reflect the deductive and object-oriented style of the language, a special language construct is introduced which generalizes the notion of view in relational databases in several ways: the *query class*. The ConceptBase query language CBQL [STAU90] represents queries as classes whose instances are the answer objects to the query. Semantically, the system-internal object `QueryClass` may have as instances so-called query classes which contain necessary and sufficient membership conditions for their instances (answers). These conditions can be used to check whether a given object is an instance of a query class or not. On the other hand, they can be used to compute the set of answer objects.

Query classes have superclasses to which they are connected by an isa-link. These superclasses restrict the set of possible answers to their common instances. Two different kinds of query class attributes can be distinguished. *Retrieved attributes* are already defined for one of the superclasses of the query class. An explicit specification of such an attribute in a query class description means that answer instances are given back with values for this attribute, as in the projection operation of relational algebra. In addition, a necessary condition for the instantiation by an admissible value is included. The attributes of superclasses can also be refined, i.e. the target class is substituted by a subclass which results in an additional value restriction. The query class

```
QueryClass TopFemaleManager isA Manager,FemalePerson with
  attribute
    salary:HighSalary
end
```

where `HighSalary` is defined as a subclass of `Integer` (with a range specified e.g. by a deduction rule) has those female managers as instances who draw a high salary. In addition this salary is included in the answer description.

Computed attributes have values derived in the query evaluation process. Neither the extensional object base contains this relationship between the answer instance and the attribute value, nor is it derivable by deduction rules. For prescribing how to deduce these new relationships and for expressing arbitrary other constraints (comparable with relational selection) for the answer instances by analogy to deduction rules and integrity constraints many-sorted first order logic expressions are admissible as building elements for query classes. The example query class

```
QueryClass IndEmp isA Employee with
  attribute, parameter
    lowersal : Employee
  constraint
    lowersal_noboss:$ exists s,t/Integer (this salary s) and (lowersal salary t)
                      and (s > t) and not exists m/Manager (this boss m) $
end
```

retrieves all employees from the object base who don't have a boss and deduces those other employees who have a lower salary. As above `this` refers to the answer instances of `IndEmp`. The computed attribute `lowersal` is identified with the variable of the same name within the formula. Note that, for answering this query, we have to evaluate the recursive deduction rule concluding the `boss` attribute.

In order to avoid the frequent reformulation of similar more specialized queries, attributes of query classes can be declared as instances of an attribute class *parameter*. Substitution of a concrete value for such an attribute or specialization of its target class by a subclass leads to a subclass of the original query which implies a subset relationship of the answer sets. Since the `lowersal` attribute is declared as parameter the expressions

```
IndEmp(Bill/lowersal)
IndEmp(lowersal:UnionMember)
```

denote two derived query classes which restrict the answer instances of `IndEmp` to those employees without boss who earn more money than `Bill` or resp. some other employee of a special subclass `UnionMember` of `Employee`.

In deductive databases queries are represented as intensional relations derived by a set of rules. Analogously, the definition of a query class `Q` induces a so called *query literal* $Q(x, x_1, ..., x_n)$ whose arity depends on the number of attributes and parameters of `Q`. The first argument x of Q stands for the answer object identifiers. Query classes are transformed to rules concluding their corresponding query literals ([SNJ93]).

Our example query class `IndEmp` is transformed to the following rule:

$$\begin{aligned}\forall e, c \quad & In(e, \#Employee) \wedge In(c, \#Employee) \wedge \\ & \exists s, t\; In(s, Integer) \wedge In(t, Integer) \wedge A(e, salary, s) \wedge A(c, salary, t) \wedge \\ & (s > t) \wedge \neg \exists m\; In(m, \#Manager) \wedge A(e, boss, m) \\ & \Rightarrow IndEmp(e, c)\end{aligned}$$

2.3 Internal Deductive Object Bases

Let us summarize how far we got up to now. We have shown that an O-Telos deductive object base is equivalent to a relational deductive database which consists of a single base relation, P, a number of additional pre-defined axioms (facts, rules, and constraints), and a set of user-defined rules, constraints, and query definitions.

However, looking at this structure from the viewpoint of deductive database theory as well as from the viewpoint of the intent of object-oriented modeling, we can identify a number of shortcomings. First, if negation is present in rules it becomes close to impossible to have a stratified database if there is only a single relation. Second, the given structure makes it very hard to type-check logical formulas, i.e., to find out whether their literals correspond to existing object classes; moreover, the notation leaves the possibly complicated mapping from external names to OIDs open.

To overcome these problem, we introduce two axioms for every (class) object $P(p, c, m, d) \in OB$. The first one defines a dedicated predicate $In.p(o)$ which is true if o is an instance of p. The second one is true if y is the destination of an attribute instantiated from attribute class p:

$$\begin{array}{l} \forall\ o\ In(o,p) \Rightarrow In.p(o) \\ \forall\ o,x,l,y\ P(o,x,l,y) \wedge In(o,p) \Rightarrow A.p(x,y) \end{array}$$

As a consequence, we get a multitude of predicates for class membership of objects and attributes of objects. A formula generated from Telos frames is rewritten with these new predicates according the following rules:

- A predicate $In(x, c)$ with constant c is rewritten to $In.c(x)$.
- A predicate $A(x, m, y)$ is rewritten to $A.p(x, y)$ where $P(p, c, m, d) \in OB$ is the unique attribute guaranteed by axiom 17 for the class c of x (which is known since variables in frame formulas are bound to classes).

In our example, the concerned class of $A(d, head, t)$ e.g. is the attribute class `#head` of `#Department` and the concerned class of $In(t, \#Manager)$ is `#Manager`. As a result we can rewrite the `bossrule` by replacing the literals A and In by specializations derived from their concerned classes.

$$\begin{array}{ll} \forall e,t & In.\#Employee(e) \wedge In.\#Manager(t) \wedge \\ & \exists d\ In.\#Department(d) \wedge A.\#dept(e,d) \wedge A.\#head(d,t) \vee \\ & \exists m\ In.\#Manager(m) \wedge A.\#boss(e,m) \wedge A.\#boss(m,t) \\ & \Rightarrow A.\#boss(e,t) \end{array}$$

The same can be done for the query class defined in the previous section:

$$\begin{array}{ll} \forall e,c & In.\#Employee(e) \wedge In.\#Employee(c) \wedge \\ & \exists s,t\ In.\#Integer(s) \wedge In.\#Integer(t) \wedge A.\#salary(e,s) \wedge A.\#salary(c,t) \wedge \\ & (s > t) \wedge \neg \exists m\ In.\#Manager(m) \wedge A.\#boss(e,m) \\ & \Rightarrow IndEmp(e,c) \end{array}$$

By using axiom 14 and generalizing it to deductive rules it can be shown that a predicate $A.p(x,y)$ implies both $In.c(x)$ and $In.d(y)$. This typing lemma [JEUS92] further simplifies formulas since it allows the elimination of many class membership predicates. The two formulas above shrink to:

$$\begin{array}{ll}\forall\ e,t & \exists\ d\ A.\#dept(e,d) \wedge A.\#head(d,t) \vee \\ & \exists\ m\ A.\#boss(e,m) \wedge A.\#boss(m,t) \\ & \Rightarrow A.\#boss(e,t)\end{array}$$

$$\begin{array}{ll}\forall\ e,c & \exists\ s,t\ A.\#salary(e,s) \wedge A.\#salary(c,t) \wedge \\ & (s > t) \wedge \neg\exists\ m\ A.\#boss(e,m) \\ & \Rightarrow IndEmp(e,c)\end{array}$$

There are two cases where the rewriting fails. The first one are **type errors** in predicates which should lead to a rejection of the insertion of these predicates. For example, a predicate $A(d, dept, e)$ where d is bound to `#Department` cannot be assigned to a concerned class since `#Department` has no attribute labeled `head`.

The second case are **meta formulas** which contain predicates like $In(x,c)$ where c is a variable. Since c is bound to a class such formulas make statements about objects two (or more) instantiation levels below the class of c. A subset of these formulas can be partially evaluated whenever a new class is entered into the database [JJ91]. An example is the *necessary* or *required* category defined in many semantic data models. It can be defined as an integrity constraint of a meta class `#Necessary`:

$$\begin{array}{ll}\forall\ p,c,m,d,x & In.\#Necessary(p) \wedge In(x,c) \wedge P(p,c,m,d) \Rightarrow \\ & \qquad \exists\ y\ In(y,d) \wedge A(x,m,y)\end{array}$$

By partially evaluating the predicates $In.\#Necessary(p)$ one gets a version of this constraint for each instance of `#Necessary`. Let us assume that the `#head` attribute of class `#Department` is such an instance. Together with the typing lemma optimization, we obtain:

$$\forall\ x\ In.\#Department(x) \Rightarrow \exists\ y\ A.\#head(x,y)$$

The examples show that the "internal" representation of formulas has several advantages that distinguish O-Telos from relational-style deductive databases:

- The predicates have a finer granularity since single attributes opposed to whole tuples are supported. For example, updates on attributes of an object which don't occur in a formula won't trigger integrity checking.
- The assignment of predicates to classes makes it possible to store the simplified form of integrity constraints and deductive rules as attributes of the concerned class. The simplified forms are only evaluated for the instances of this class.

- The multitude of predicates allow to use a "normal" stratification test for negation in deductive rules, rather than the inefficient test for local stratification.
- The predicate $A.p(x, y)$ saves the access to $P(p, c, m, d)$ which was needed in the original form $A(x, m, y)$.
- The typing of variables induces a typing of predicates which can be used for elimination of class membership predicates and thus unnecessary view maintenance operations.
- The meta classes of O-Telos find their natural counterpart in the meta formulas. Often, partial evaluation of these yields efficient representations not offered by other systems.

3 The ConceptBase Server

The ConceptBase implementation is organized in a client-server architecture. The server supports the generic operations TELL (updating an O-Telos database) and ASK (querying an O-Telos database) through transformation and storage services. Transformation servcies are implemented in BIM-Prolog, storage services in C++. Clients, implemented in C or Prolog, provide user interfaces for the various syntax variations of O-Telos, but users can also add their own clients.

The implementation of the server reflects the observations made about O-Telos. On the one hand, it reuses only slightly adapted variations of well-known query and integrity processing algorithms for transformation. On the other hand, it maps these to an object algebra tailored to the O-Telos object structure, taking into account its great flexibility with respect to schema evolution – classes are normal objects, attributes can have attributes, etc. Therefore, a fully decomposed storage structure with sophisticated single-column and join indexing is used as a target instead of the usual frame-oriented structures of OODB.

3.1 Overview

In the development of the ConceptBase server, it was recognized that the tasks of query and rule evaluation, integrity enforcement, updates and view maintenance are strongly related. This is reflected in the server architecture by a small number of functional components which are reused for different purposes.

The modules involved in updating and querying the knowledge base are presented in fig. 2 along with their dependencies due to data and control flow. The architecture is separated horizontally in two layers. The upper layer deals with transforming representation forms noted in section 2.1 and the compilation of declarative expressions described in

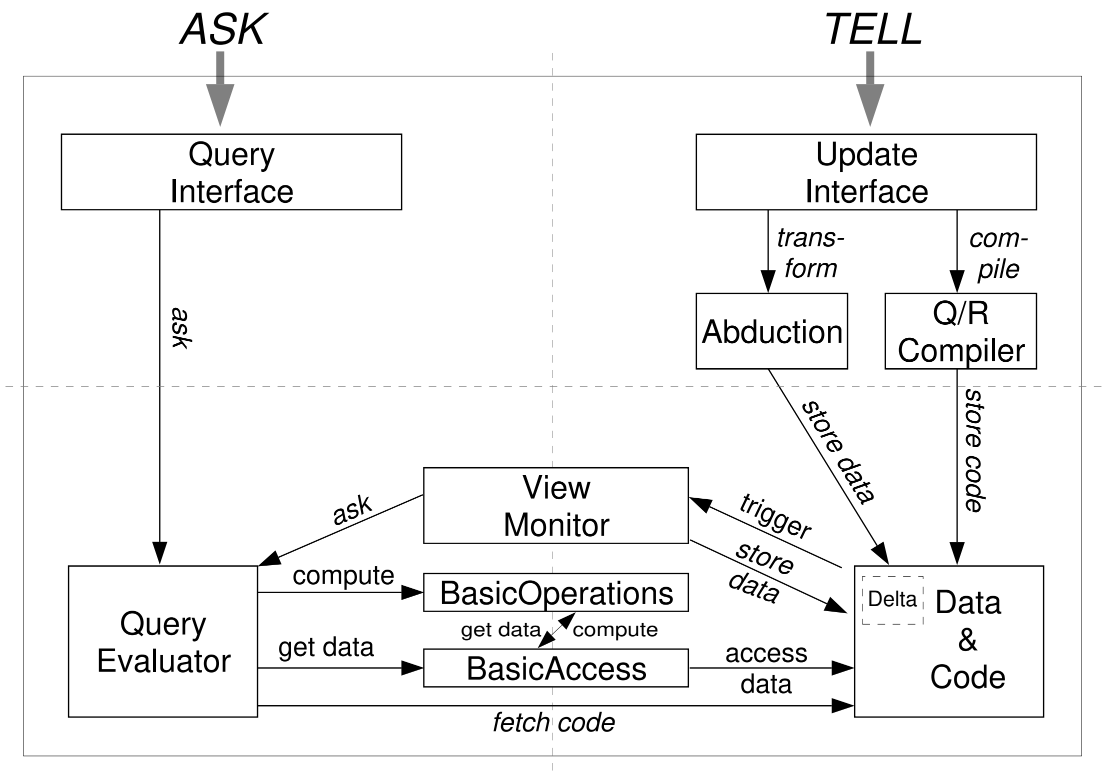

Figure 2: The Architecture of the ConceptBase Server

section 3.2.3. The lower layer works directly on the object base when computing query results. The architecture is vertically divided by the main operations $TELL$ and ASK.

Let us step through fig. 2 clockwise starting from the module `UpdateInterface`. This module takes object descriptions from the client programs and passes the information as fully instantiated predicates A, In and IsA to the module `Abduction` which generates the base data predicates P representing the objects. A declarative description, e.g. a query or a rule, is passed to the module `Q/R Compiler` which generates algebraic expressions necessary for the computation of the query result. This compilation phase involves rewriting steps such as the supplementary magic set method. Both, the object and the algebra expressions, are placed into a storage management system represented by the box at the bottom right in fig. 2. As new informations are collected in a temporary storage, they trigger the module `ViewMonitor`. The `ViewMonitor` propagates all changes in an *event/ condition/action* manner; it consults the module `QueryEvaluator` for evaluating the conditions which are realized by ASK statements.

The `QueryEvaluator` first fetches the corresponding algebra expressions directly from the storage system. Then it applies a semi-naive fixpoint procedure involving the modules `BasicOperations` and `BasicAccess`. The module `BasicOperations` evaluates algebra operations like *union*, *intersection* or *transitiveclosure* by accessing the storage system according to its set implementation. Correspondingly, the module `BasicAccess` realizes the access to the base data dependent on the index representation. These two modules guarantee the independence of the algebra evaluation and the object storage representation. The evaluation of stored queries is not only initiated by the `ViewMonitor` but also directly from client programs through the `QueryInterface`. The following subsections describe the modules of fig. 2 in more detail.

3.2 TELLing Objects and Rules

3.2.1 Abductive Update Processing

Since all the surface notations are derived notations in O-Telos, all updates have in principle to be treated as view updates. Therefore, it has been natural to add more sophisticated view update facilities (e.g., updates on rule conclusions) to the system, along the lines of abductive reasoning proposed by [KM90].In other words, ConceptBase takes a declarative rather than an imperative approach to updates. Below, we describe the simple case; the general view update case in presented in [SNJ93].

Updates are pushed from the client program into the ConceptBase server in Telos frame notation using the operation `TELL`. The update process transforms this representation into the base predicates P and places them in the storage system.

The transformation is divided in two main steps provided by the modules:

1. `UpdateInterface`: The Telos frame notation is transformed into a literal form as described in section 2.1. Object names are replaced by object identifiers using the restrictions of O-Telos axioms A2-A4 (see Appendix).

2. `Abduction`: The literal representation is compiled into base predicates P.

For example, the client program updates information about the manager `Mary` using the operation

```
TELL (   Mary in Manager with
            dept
               currentdept:R&D;
               advises:PR
            salary
               earns:50000
         end   )
```

The `UpdateInterface` analyses this Telos frame and generates one In literal for each instantiation description, one Isa for each specialization description and an A' literal for each attribute in the frame:

$$\begin{gathered} In(\#Mary, \#Manager), \\ A'(\#Mary, currentdept, dept, \#PR), \\ A'(\#Mary, advises, dept, \#R\&D), \\ A'(\#Mary, earns, salary, 50000) \end{gathered}$$

The literal A' is an extended version of the literal A which retains additionally the user-defined attribute label.

The literals are now forwarded to the module `Abduction` to generate base predicates P. The main problem is to determine the optimal realization of the update if a set of plausible explanations arises. A detailed description of the *abduction* algorithm is presented in [SNJ93]. This algorithm does not only accept the literals In, Isa and A but also query literal terms. The elimination of different explanations is reached by defining a priority order [FUV83], e.g. preferring the elimination of attribute relations to elimination of instantiation relationships.

In our example, the base data can be easily generated because there are no rules involved:

$$\begin{gathered} P(\#Mary, \#Mary, Mary, \#Mary), \\ P(\#in1, \#Mary, in, \#Manager), \\ P(\#a1, \#Mary, advises, \#PR), \\ P(\#ai1, \#a1, in, \#dept), \\ P(\#a2, \#Mary, currentdept, \#R\&D), \\ P(\#ai2, \#a2, in, \#dept), \\ P(\#a4, \#Mary, earns, 50000), \\ P(\#ai4, \#a4, in, \#salary) \end{gathered}$$

The literals A' are transformed into two predicates P, one to describe the attribute itself and another one to attach this attribute to its category.

3.2.2 Efficient Access to Objects by Indexed Storage

The O-Telos literals suggest a special index structure for the storage system. [GALL90]. It is designed especially to support the `QueryEvaluator` and generalizes to inverted indices [CARD75] and join indices [VALD87]. The goals for the design of the indexes are the following:

1. Fast access to objects via their identifiers
2. Fast access to instances/classes of an object
3. Fast access to specializations/generalizations of an object
4. Fast access to the attributes and attribute values
5. Support for computation of algebra operations

As shown in section 2.1 the extensional object base OB can be seen as a set of tuples $P(o, x, l, y)$. To fulfill the first requirement we associate the identifier o of type `TOID`

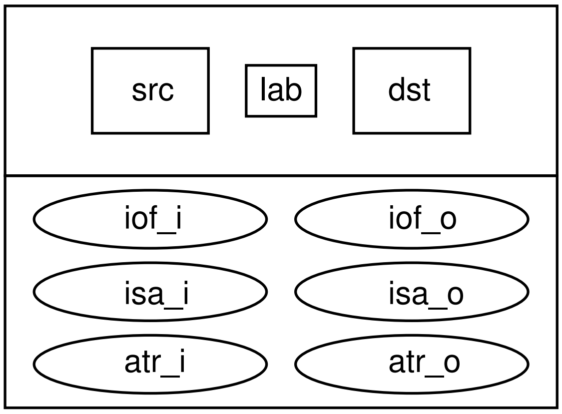

Figure 3: Graphically visualization for a storage record of type TOBJ

with a reference to a small piece of the storage of type `TOBJ` containing the components x, y, l and further indexing informations. The indexes are defined by sets of `TOID`s called type `TOIDSET` and can be implemented as classes in C++ [STRO92], as shown in figure 3. The components `src`, `lab` and `dst` correspond to the components x, y and l in the predicate $P(o, s, l, d)$, while the record components `iof_i(o)`, `iof_o(o)`, `isa_i(o)`, `isa_o(o)`, `atr_i(o)` and `atr_o(o)` are the indexes according to the object base structure. These indexes are defined to fulfill the requirements 2, 3 and 4. Their semantics can be described by:

$$iof\_i(o) = \{o_i \mid P(o_i, s, in, o) \in OB\}$$

$$iof\_o(o) = \{o_i \mid P(o_i, o, in, d) \in OB\}$$

$$isa\_i(o) = \{o_i \mid P(o_i, s, isa, o) \in OB\}$$

$$isa\_o(o) = \{o_i \mid P(o_i, o, isa, d) \in OB\}$$

$$atr\_i(o) = \{o_i \mid P(o_i, s, l, o) \in OB \wedge l \notin \{in, isa\}\ \}$$

$$atr\_o(o) = \{o_i \mid P(o_i, o, l, d) \in OB \wedge l \notin \{in, isa\}\ \}$$

In the version of the storage system described here, we make the design decision that everything is an object, so instantiation links are objects, too. This means that the sets `iof_i` resp. `iof_o` do *not* contain the identifiers of the objects which are the instances, but the identifiers of the instantiation links itself.

The sets contain only objects that are *stored* in the object base which means the instantiation relations explicitly known. Inherited instantiation information is accessible by combinations of access operations.

Summarizing, this data structure has two major advantages. The O-Telos frame syntax is easily reconstructed at the storage level, and an inverted index structure records the

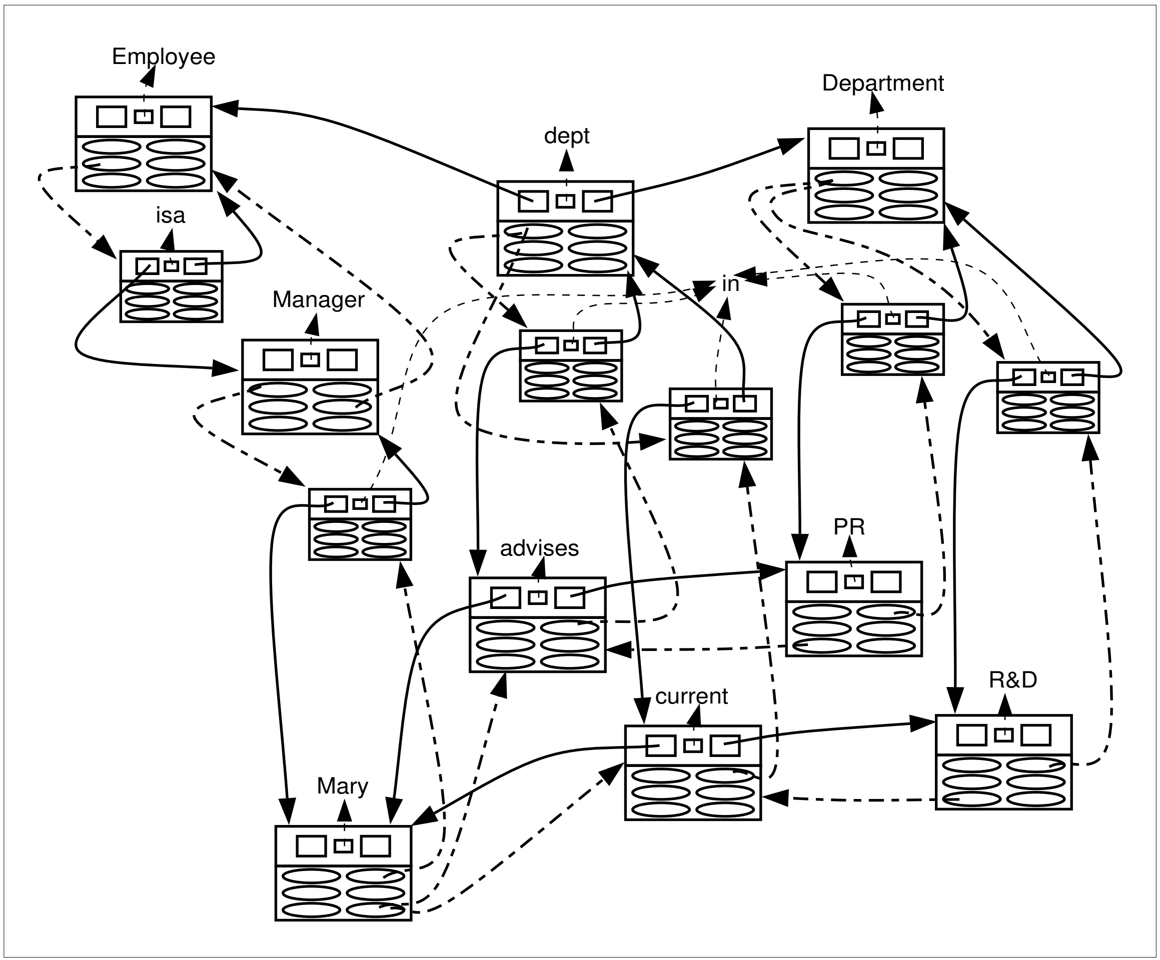

Figure 4: Indexed storing of objects in the semantic net

appearance of any object o as the source or destination object of a relation to an other object p. Portions of the storage structure for our example is shown in fig. 4.

Finally, requirement 5 is fulfilled both by the indexing of the object structures and the implementation of the sets. These sets are realized as B-trees [BBDG90] which makes the processing of basic set operations like *union* or *intersection* very fast. The evaluation of set operations like `union`, `difference` and `intersection` is encapsulated in the module `BasicOperations`. While `BasicAccess` directly retrieves data from the structure shown in fig. 3, `BasicOperations` also considers information derived according to the basic O-Telos axioms:

`instances`: delivers all instances of an object, including those deducible through subclass inheritance (see axiom 13 in appendix).

`classes`: delivers all classes of an object. As for `instances` the inheritance through superclasses is considered, too.

```
void apply_access( void *accessOperation(); TOIDSET& towhat )
{  TOID    x;
   TOIDSET result, help;

   forall( x, towhat )             // this is a macro for scanning a set
   {  help.clear();                // reinitialize the buffer help
      accessOperation( x, help ); // get stored data from the object structure
      result |= help; }            // add the buffer help to collect the result
   towhat.clear();                 // empty the input because
   towhat |= result;  }            // the result is given back in the parameter

void closure_objects( void *op(); TOIDSET delta; TOIDSET& result )
{  while( ! delta.empty() )        // while we have information that
   {  result |= delta;             // add the delta from the last step
      apply_access( op, delta );  // compute the desired operation on delta
      delta -= result;             // we only want the new information,
   } }                             // so remove what already exists

void instances( TOID c; TOIDSET& result )
{  TOIDSET start;
   start.add( c );                 // add c to initialize the first delta
                                   // compute all specialized classes
   closure_objects( stored_subclasses, start, result );
                                   // compute all instances of the classes
   apply_access( stored_instances, result );  }
```

Figure 5: Operations to evaluate basic equations

`specializations`: delivers all subclasses of a given class considering inheritance (closed under axioms 10-12).

`generalizations`: delivers all superclasses of a given class considering inheritance.

`attributes`: delivers all attributes of an object. Note that this is different to `attributeValues`.

`attributeValues`: delivers all attribute values of an object, namely the destination components of the attributes.

`attributeValueOf`: delivers all objects o having the given object as an attribute Value.

This encapsulation ensures the independence from the other modules so that the storage system can be seen as an abstract data type; indeed, we have experimented with several different storage managers.

As an example of such an implementation, fig. 5 shows some pieces of C++ code for the computation of `instances`. To compute all instances of an input object, first all specialized classes have to be computed by the routine `closure_objects`; the union of all instances of these classes forms the result.

The use of functions as parameters makes them more general leading to higher-order programming [SOKO91]. For instance, the routine `closure_objects` takes a function describing the "closure-direction" for the semantic net navigation and as a second argument a set of starting objects. Then it computes the closure according to the direction operation using a semi-naive method [ULLM88, AGJA90]. In our example, the functions to be applied are `stored_subclasses` and `stored_instances` which are simply built on `ISA_I` and `sour` resp. `IOF_I` and `sour`. The routine `apply_access` applies the access function provided as the first parameter to all elements of the set given as the second parameter. The result is placed in the set to which the access function was applied. This simple routine shows how the basic operations can be realized easily and *fast*, provided fast implementation of the basic set operations [LEA91].

3.2.3 From Rules to Algebraic Expressions

In section 2.1 we showed how query classes can be transformed into deductive rules concluding specific query literals. The module `Q/R-Compiler` implements these transformation steps finally leading to sets of algebraic equations.

We continue our employee example by demonstrating the different stages with the recursive `bossrule` and the query class `IndEmp`. The semantically optimized rules described in section 2.3 are compiled to plain Datalog$^\neg$ utilizing the algorithm proposed for general logic programs in [LT84].

$$\begin{aligned}
A.\#boss(e,t) &\ :-\ A.\#dept(e,d), A.\#head(d,t) \\
A.\#boss(e,t) &\ :-\ A.\#boss(e,m), A.\#boss(m,t) \\
IndEmp(e,c) &\ :-\ A.\#salary(e,s), A.\#salary(c,t), s > t, \neg IE_1(e) \\
IE_1(e) &\ :-\ A.\#boss(e,m)
\end{aligned}$$

The last Datalog$^\neg$ rule is an auxiliary rule concluding a new literal IE_1. It results from resolving the negated existential quantification. Among the standard deductive optimization techniques, we chose the supplementary magic-set rewriting method ([BR87]) because of its closeness to the idea of parameterized query classes.

An application of a bottom-up fixpoint procedure for evaluation purposes to magic rule sets guarantees to make only intensional information explicit which is relevant for a given query. The magic-set rewriting always takes place with respect to different instantiation patterns of queries denoted by b for bound and f for free arguments. The so-called adorned form of the query literal, e.g. $IndEmp$, with a concrete second argument is $IndEmp^{fb}(e,c)$. A rewriting of the four rules above concerning this query and a subsequent partial predicate elimination following [ULLM89] results in the rule set shown in fig. 6.

$$
\begin{array}{rcl}
A.\#boss^{bf}(e,t) & :- & q\_A.\#boss^{bf}(e), sup_{1,1}(e,d), A.\#head^{bf}(d,t). \\
sup_{1,1}(e,d) & :- & q\_A.\#boss^{bf}(e), A.\#dept^{bf}(e,d). \\
q\_A.\#dept^{bf}(e) & :- & q\_A.\#boss^{bf}(e). \\
q\_A.\#head^{bf}(d) & :- & sup_{1,1}(e,d).
\end{array}
$$

$$
\begin{array}{rcl}
A.\#boss^{bf}(e,t) & :- & q\_A.\#boss^{bf}(e), sup_{2,1}(e,m), A.\#boss^{bf}(m,t). \\
sup_{2,1}(e,m) & :- & q\_A.\#boss^{bf}(e), A.\#boss^{bf}(e,m) \\
q\_A.\#boss^{bf}(m) & :- & sup_{2,1}(e,m).
\end{array}
$$

$$
\begin{array}{rcl}
IndEmp^{fb}(e,c) & :- & q\_IndEmp^{fb}(c), sup_{3,2}(c,e), \neg IE_1(e). \\
sup_{3,1}(c,e,s) & :- & q\_IndEmp^{fb}(c), A.\#salary^{ff}(e,s). \\
sup_{3,2}(c,e) & :- & sup_{3,1}(c,e,s), A.\#salary^{bf}(c,t), s > t. \\
q\_A.\#salary^{ff}() & :- & q\_IndEmp^{fb}(c). \\
q\_A.\#salary^{bf}(c) & :- & sup_{3,1}(c,e,s). \\
q\_IE_1^{b}(e) & :- & sup_{3,2}(c,e).
\end{array}
$$

$$
\begin{array}{rcl}
IE_1^{b}(t) & :- & q\_IE_1^{b}(t), A.\#boss^{bf}(t,m). \\
q\_A.\#boss^{bf}(t) & :- & q\_IE_1^{b}(t).
\end{array}
$$

Figure 6: Magic-set rules generated for the example.

The next step translates the generated magic-set rules into equations of the COBRA object algebra [THOE92]. COBRA is based on the relational algebra and adopts concepts from other object-oriented algebras (as e.g. [SZ90],[VD91],[VW91]), tailored to the storage structure for objects. One advantage of this approach is a clear separation of algebraic equations generated from user-defined rules and queries, and so-called basic equations allowing the access to the stored extensional data. Note that only the latter directly use operations from the `BasicAccess` module introduced in section 3.2.2. Thus, alternative storage can easily be integrated.

Compiling our example rules into COBRA equations (compare fig. 7) we get subset relationships[4] for the A-, magic (prefix q) and query literals and equations for the unique defined supplementary relations ($sup_{i,j}$).

COBRA allows functions and predicates defined in a λ-calculus like way within these operations occurring as an extension of simple component selection functions and equality

[4]The evaluation process performs the union of all subset expressions with the same left hand side [CGT90]. There could be other rules concluding e.g. solutions for the `boss` attribute of an employee, especially algebraic expressions generated for other instantiation patterns.

$$
\begin{array}{rcll}
A.\#boss & \supseteq & \prod_{1,3}((q\_A.\#boss^{bf} \ltimes_{1=1} sup_{1,1}) \ltimes_{2=1} A.\#head) & (1)\\
sup_{1,1} & = & q\_A.\#boss^{bf} \ltimes_{1=1} A.\#dept & (2)\\
q\_A.\#dept^{bf} & \supseteq & q\_A.\#boss^{bf} & (3)\\
q\_A.\#head^{bf} & \supseteq & \prod_{2} sup_{1,1} & (4)\\
\\
A.\#boss & \supseteq & \prod_{1,3}((q\_A.\#boss^{bf} \ltimes_{1=1} sup_{2,1}) \ltimes_{2=1} A.\#boss) & (5)\\
sup_{2,1} & = & q\_A.\#boss^{bf} \ltimes_{1=1} A.\#boss & (6)\\
q\_A.\#boss^{bf} & \supseteq & \prod_{2} sup_{2,1} & (7)\\
\\
IndEmp & \supseteq & (q\_IndEmp^{fb} \ltimes_{1=1} sup_{3,2}) \setminus_{2=1} IE_1 & (8)\\
sup_{3,1} & = & q\_IndEmp^{fb} \times A.\#salary & (9)\\
sup_{3,2} & = & \prod_{1,2}(sup_{3,1} \ltimes_{2=1 \wedge 3>2} A.\#salary) & (10)\\
q\_A.\#salary^{ff} & \supseteq & \prod_{0} q\_IndEmp^{fb} & (11)\\
q\_IE_1^{fb} & \supseteq & \prod_{2} sup_{3,2} & (12)\\
\\
IE_1 & = & \prod_{1}(q\_IE_1^{b} \ltimes_{1=1} A.\#boss) & (13)\\
q\_A.\#boss^{bf} & \supseteq & q\_IE_1^{b} & (14)
\end{array}
$$

Figure 7: COBRA equations compiled from the example rules.

tests, similar to [SZ90]. For example, in the definition of the $\prod$-operator

$$\prod_{\lambda v.<f_1(v),...,f_n(v)>}(V) = \{< f_1(v), ..., f_n(v) > | v \in V\})$$

where V is a set of objects or tuples of objects and the f_i are functions, the $\prod$ is a general apply operator for functions. In figure 7 we use numbers for tuple component selection functions and leave out variables as it is usually done in relational algebra[5]. In the next section we show with a simple example the evaluation of the generated equations by computing the relevant part of the fixpoint of all equations with respect to a given query.

[THOE92] contains the description of a C++ data structure for COBRA-equations which sets up a network between so called *virtual classes* corresponding to the relations occurring within the equations. These classes are filled with objects resp. tuples of objects during the evaluation process and share the data structures (esp. for sets) of the extensional object base.

For directly accessing the extensionally stored data, the `QueryEvaluator` employs base equations in the same way as the equations above. The expressions (3), (4), (7), (11),

[5]Thus $\prod_1$ means a projection on the first tuple component whereas $\prod_0$ is just an emptiness test.

(12) and (14) in fig. 7 insert new tuples into the magic relations on their left side in order to trigger other rules. In equation (3), a query asking for the bosses of an employee `John` (therefore $\#John$ is a member of the relation $q\_A.\#boss$) leads to copying $\#John$ into $q\_A.\#dept^{bf}$ because for the next step (e.g. the join operation in (2)) all departments d of $\#John$ are needed: either computed from further equations which have the magic relation $q\_A.\#dept^{bf}$ in their body or retrieved from the extensional object base. The goal is to have all derivable $(\#John, d)$-tuples in $A.\#dept$. The base equation

$$A.\#dept \supseteq \prod_{\lambda x.\{x\} \times dest_s(instances(\#dept) \cap attributes(x))} q\_A.\#dept^{bf}$$

retrieves all stored $(John, d)$-tuples by applying an efficient intersection operation between the instances of the attribute class $\#dept$ and all attributes of objects in $q\_A.\#dept^{bf}$.

Due to the overhead it is obviously not desirable to actually have base equations of the above kind for any attribute class in the object base. The system therefore provides parameterized equations (not explained here). There are additional equations for the *In*, *Isa* predicates, and for the possible parameter instantiations of the magic relations.

The algebra equations are stored in a dedicated code space within the storage subsystem and serve as input of the module `QueryEvaluator`. This module implements a semi-naive fixpoint procedure which is applied to all stored equations that are relevant for evaluating a given query. Thus we can distinguish three stages of a query evaluation process:

- An external ASK operation is interpreted by the module `QueryInterface` (compare fig. 2). After performing all necessary compilation steps the `QueryEvaluator` is initiated.
- From the internal network of algebraic equations the `QueryEvaluator` selects the relevant set of equations and initializes the magic relations matching with the query to be evaluated by inserting the constants given within the query description.
- The fixpoint procedure is started with respect to a previous partitioning of the found equations into stratas in order to cope with negation.

The evaluation process results in a set of fully instantiated query literals corresponding to the query. The `QueryInterface` provides suited answer formats from this internal representation to satisfy the requirements of the external clients.

A query evaluation applied to the running example is listed in the appendix. It combines the recursive boss rule with the parameterized query `IndEmp`.

3.3 View Maintenance in ConceptBase

Views are answers to queries which have to be kept up-to-date with the database[6]. The problem of view maintenance is trivially computable: a simple method is to evaluate each query belonging to a view after any update. However, this would imply unnecessary computations since updates are not likely to affect a change on all views concurrently. A better solution is to evaluate only the difference implied by the update.

In ConceptBase, we have realized a view maintenance method that generalizes the simplification method developed for integrity checking proposed in [BDM88]. For the purpose of this paper, we restrict ourselves to the case of deduction rules without negation but the system also handles negation.

Consider the deductive rule:

$$\forall\ X\ A_1(X_1) \wedge ... \wedge A_k(X_k) \Rightarrow Q(X)$$

The upper case arguments $X, X_1, ...$ denote lists of variables or constants, resp. An **update** to the extension of a predicate is a literal $A(C)$ or $\neg A(C)$ where the C is a sequence of constants. Positive literals denote insert updates and negative deletion updates. If A is a base predicate then we call the update **base update**.

If $Q(D)$ is derivable in the new database (i.e., the database after performing a base update) but not in the old database, then $Q(D)$ is called a **derived insert update** on Q. If it is derivable in the old but not in the new database then it is called a derived delete update, denoted as $\neg Q(D)$. The problem can now be reformulated as the efficient computation of derived updates from base updates.

Now, suppose a base update has led to a derived insert update $Q(D)$. The substitution $[D/X]$ in the deductive rule implies substitutions $[C_i/X_i]$ in the arguments of the condition literals $A_i(X_i)$. Since $Q(D)$ is a new solution there must be an index j such that $A(C_j)$ is an (insert) update. This observation implies that $Q(D)$ is in the extension of the **simplified rule** (without loss of generality $j = 1$):

$$\forall\ X'\ A_2(X'_2) \wedge ... \wedge A_k(X'_k) \Rightarrow Q_1(X'')$$

where $X'_2, ..., X'_k, X''$ are obtained by applying the substitution $[C_1/X_1]$ to the arguments of the literals.

In other words: the extension of Q_1 (in the new database!) delivers a superset of the insert updates on Q. If more than one $A_j(C_j)$ is updated then we have to compute all extensions of the corresponding Q_j and build their union.

The case of delete updates $\neg A_j(C_j)$ is similar to the above. Instead of the new database the simplified rule is evaluated against the old database. Once again, this method delivers a superset of the actual updates on Q.

[6] A special case are integrity constraints represented as deductive rules with the 0-ary *Inconsistent* as conclusion predicate. Then, maintaining integrity means to make sure that the view on *Inconsistent* is empty through the lifespan of the database.

View maintenance is implemented in the `ViewMonitor` module (see fig. 2). During a transaction the updates to the object base are located at the `Delta` part of the object store. The inserted (deleted) objects trigger the query evaluation of the simplified rules shown above. The obtained solutions are potential updates to the views. In case of insertion the view monitor checks whether the new fact $Q(D)$ is already in the view, otherwise it is inserted. In case of deletion, the view monitor first verifies that $Q(D)$ is not derivable from the new object base and then removes it from the view.

Due to the strong connection between classes and predicates (section 2.3) it is sufficient to check only instantiations to classes for triggering (proof in [JEUS92]). The extinction of unnecessary literals is especially beneficial for view maintenance since it also avoids unnecessary trigger evaluations due to updates on these literals.

3.4 Performance Issues

ConceptBase has been used in several configurations of algorithms and storage structures. Moreover, its typical applications are characterized by very many classes (hundreds to thousands) but not that many instances in each class. For these reasons, giving a table with typical performance benchmarks is not very meaningful. Nevertheless, a few qualitative observations should be made here.

First, the transformation to internal object base form and the consequent semantic optimizations have had a very substantial impact on performance: the internal form reduces directly the search space, integrity checks are completely dropped for many updates and made much more precise in others, and true extensibility of the language through meta formulas is only made realistic through the partial evaluation techniques discussed in section 2.3. In one medium-size application in software configuration management, a representative set of query classes and integrity checks improved by factors of 20-50. Of course, we have to admit that some of these improvements just remove performance problems we have created ourselves through the introduction of class as objects; but we argue that the introduction of type checking is very useful for large databases, and that it is important to see that this can be done statically, at formula update time.

A similar improvement over naive relation-like storage structures is achieved by the storage structure introduced in section 3.2.2. However, here the improvement is a complexity jump in terms database size. Essentially, certain important operations, such as the construction of a frame or the computation of a simple join, obtain the complexity of their actual result rather than of the cross-product of their operands. This corresponds to experiences with a similar data structure for a relational database in [PT92].

Lastly, as in other deductive databases, our experiences with the magic set method have been mixed. In the applications it was designed for, it proves very useful but a naive implementation introduces substantial overhead; the actual context of algebraic equations to be considered for evaluation must be carefully restricted. As a practical short-term solution, the ConceptBase user has the option to turn off the magic set

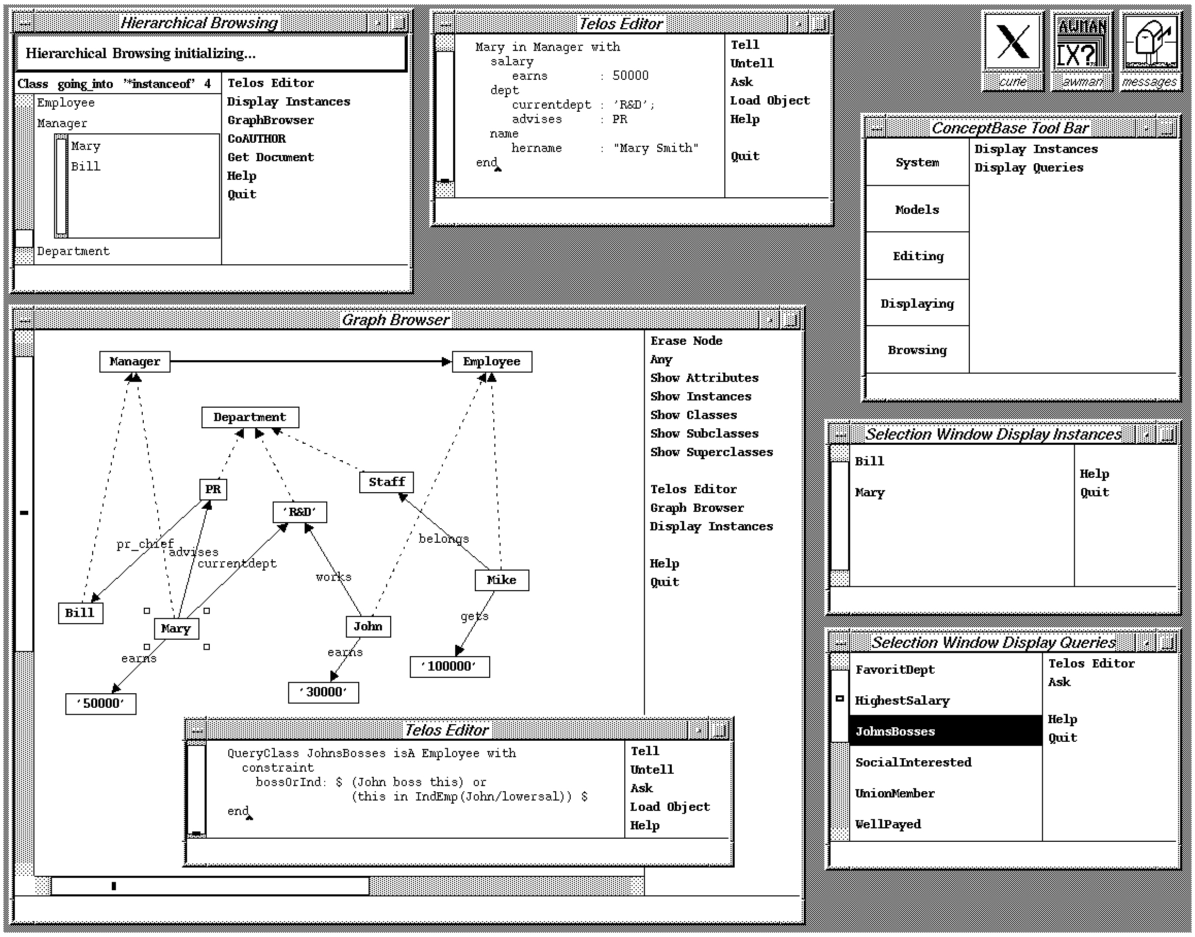

Figure 8: Screendump of a sample session with the ConceptBase System

transformation. A solution, which considers special cases of recursion as well as a more refined integration between query processing and view/integrity maintenance, is under development.

4 The ConceptBase Usage Environment

The ConceptBase Usage Environment is a collection of *tools* working in a graphical environment, namely the *X Window System* and the *Andrew Toolkit* [BORE90]. These tools are invoked by the *ConceptBase Toolbar*:

Telos Editor: The Telos Editor deals with objects in the Telos frame notation. You can insert or remove the information shown in textual form using the buttons TELL or UNTELL. If the object in the editor is a query object, you can evaluate it by pressing an ASK button.

Display Instances: Lists all instances of a given object in a single-column table.

Display Queries: This tool is a specialization of *Display Instances*; it shows all instances of the object `QueryClass` and offers the possibility to evaluate a selected query with the `ASK` button.

Graph Browser: This tool shows the contents of the knowledge base in its semantic net representation, and allows menu-based operations on both nodes *and* links since links are first-class objects in O-Telos. For example, you can show all attributes of a link. The graphical layout of the displayed objects is configurable as you can associate object classes with graphical types for the nodes or links that represent their instances (e.g., *oval, rectangle, gray, blue, dotted*) .

Hierarchical Browser: The Hierarchical Browser displays hierarchical relations between objects relative to a given set of start objects and given link types. The specified link label is followed not only by direct connected links but also in a transitive manner. A depth parameter specifies how far the links should be followed in the semantic net. The found objects are presented in a tabular form with the possibility to further unfold the table dynamically.

Query: The query interface is used to specify the parameters for the evaluation of a stored query class. The computed result is displayed in an answer window containing all frames which are instances of the given query class and therefore are the answers to the query.

Fig. 8 shows a screendump of a session using the example from the previous chapters. At the top you can see a `Hierarchical Browser` starting from the object `Class` displaying all instances for four levels. The table is unfolded at the object `Manager`. The `ConceptBase Toolbar` is placed at the top right under some icons. On the right, there are two windows from `Display Instances`, one containing the instances of `Manager`, the other some available queries. The main part of the screen is consumed by a `Graph Browser` that shows a part of the semantic net. The specialization link between `Manager` and `Employee` is drawn as a bold arrow, instantiation links are dotted. The textual representation of the object `Mary` and the query used in the previous section are shown in two `Telos Editor`s.

In addition to these basic interface tools of the ConceptBase system, users can define special tools for applications. Figure 9 shows the server as part of a more complex environment connected by a communication channel. The server offers the two operations TELL & ASK to the environment. The numbers behind the tool names indicate that more than one instance of a tool may participate in the environment, for instance, there are two processes of class `relDBMS` in the environment. The communication protocol uses standard interprocess communication (IPC). The reason for selecting IPC are its large distribution (almost any UNIX implementation offers IPC) and its ability to hide

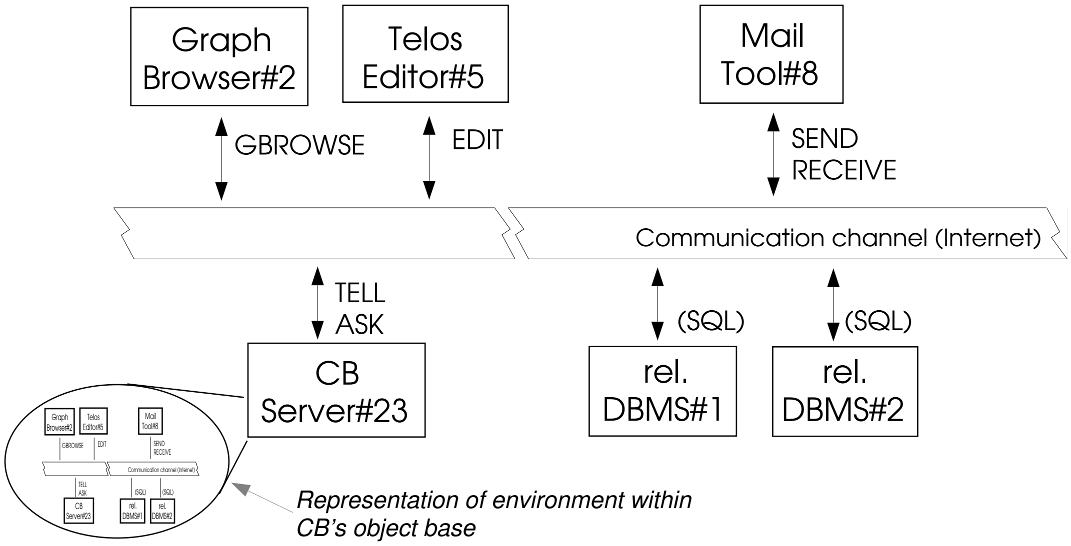

Figure 9: Client-Server architecture of ConceptBase

details on where the communicating processes are located on the network. Thus, wide-area (up to continent-spanning) distribution of ConceptBase is easily available and has been used in several applications.

While interprocess communication solves the principal problem of exchanging data between distributed processes and calling remote procedures, it does not offer *semantic* control. ConceptBase therefore contains three builtin classes `Tool`, `Operation`, and `Object`. An environment is abstractly modeled by enumerating the allowed tools, their operations, and the object types processed by the operations. At runtime, the configuration of the environment is an instance of the abstract model represented as part of ConceptBase's object base. As soon as a tool registers itself as an instance of its tool class in ConceptBase, its operation become accessible to the rest of the environment.

Representing the environment as part of the object base makes it subject to the deductive abilities of ConceptBase. For example, access rights can be encoded as deductive rules [SJ92]. Operations applicable to an object can be deduced from the object's class, and then be passed to a tool supporting that operation. The most significant advantage, however, is that updates to the environment are mirrored as updates on their abstract representation in the object base. The following query class computes applicable operations for any object:

```
QueryClass ApplicableOperation isA EnvOperation with
  parameter
     obj: Object
  attribute
     tool: EnvTool
  constraint
     whatOp: $  exists c/Class (obj in c) and (this input c) and
               (this supported_by tool) $
end
```

The class `EnvOperation` is an instance of class `Operation` and subsumes all operations allowed in the environment. Given a selected object `obj` the formula expresses that a solution `this`, i.e. an applicable operation, is computed by retrieving all operations which specify the class `c` of `obj` as their input. The tools supporting the operation are attached as attributes of the solutions.

5 Applications

As mentioned before, many users inside and outside our research group have made experiments with the language. In these applications, the abstraction mechanisms offered by O-Telos for structuring information turned out to be equally important as the deductive database capabilities. Quite often, O-Telos models are being used to structure complex information such that the system serves as a means of analyzing systems at the requirements or meta data level. In such applications, non-computer scientist users found the system superior to traditional simple modeling tools such as SADT diagrams but also easier to understand than very complex object-oriented modeling formalisms. In the following, we briefly summarize experiences in two major application domains, software information management and hypertext authoring. The first one has been DAIDA [JARK93] where ConceptBase served as a global database covering the development process from requirements models to database application programs.

Software information systems are an example of applications where heterogeneous objects and tools have to be integrated with a particular task in mind. More specifically, a software repository is a system for managing evolving (software) objects where dependencies between objects (like "A is compiled to B") have to be maintained. A language for software repositories has to combine abstractional, assertional, and dynamic classifying features. The first requirement stems from the fact that software objects (specifications, documentation, designs, implementations) typically are dissimilar in their instances and that the language should allow for easily describing common properties. The assertional feature is required for integrity constraints, e.g. each imported property of a software module must be exported by another module.

```
QueryClass ReleasedProgram isA Program with
   parameter
      producer: Agent
   constraint
      is_released: (this in_state Released) and
                   (this owner producer)
end
```

The above example (taken from [RJM92]) shows a typical query class used for dynamic classification of objects. ConceptBase allows to store such queries as permanently just like any other class. Note that membership to this class is derived and that updates to the software repository may change the membership of certain objects. The parameter in the query class is used to specialize the query. For example,

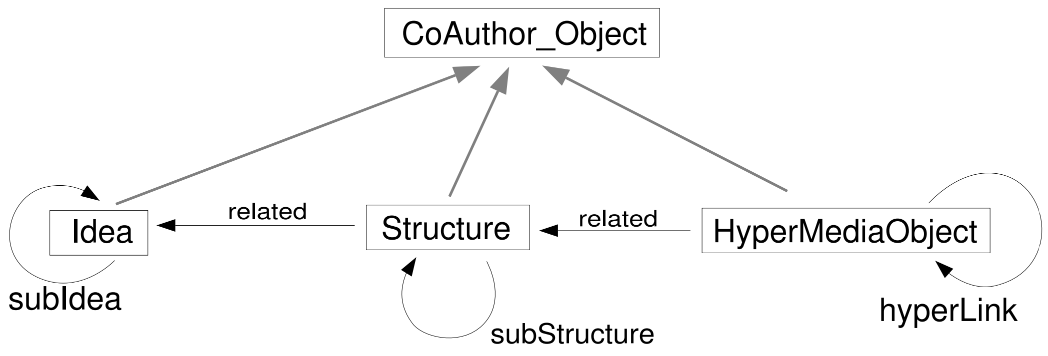

Figure 10: Conceptual model of authoring

`ReleasedProgram[John/producer]` denotes all released programs owned by an agent with name `John`. A software repository offering these features has been implemented with ConceptBase within the Canadian network of excellence program. Other applications within this domain include similarity-based component retrieval in software reuse [CHE93], group support, prototyping, and traceability for requirements engineering [RD92], [RL93].

The O-Telos object base structures developed in these projects seem to be of interest beyond the software engineering domain. Computer-integrated *quality management* is the theme of a project that tackles the problem of communication gaps between isolated quality assurance tools in industrial engineering. The project started with the definition of a shared ontology which contains definitions for items like `Product`, `Team`, `ProductionProcess` etc. and their relationship to the quality assurance tools. The definitions are written using O-Telos classes and stored in ConceptBase. Currently, the definitions contributed by the project partners are informally discussed and voted upon via the CoAUTHOR mechanism described below.

CoAUTHOR is an extension of ConceptBase in which nodes can be multimedia objects, thus generating a knowledge-based hypertext environment [EJ91]. Additionally, CoAUTHOR includes an O-Telos meta model which defines a real-time environment for the development of hypertext documents by multiple authors. CoAUTHOR's document model distinguishes between *ideas* (subsuming the issues to be covered by the document), *structures* (enriched tables of contents), and hypermedia objects (constituting the final document, see also fig. 10).

CoAUTHOR uses the client/server architecture of ConceptBase to support multiple authors developing a document. The descriptions at any of the three levels can be annotated by the authors via so-called voting links (`Pro` for accepting, `Counter` for denying, and `Alternative` for proposing a variant). The deductive capabilities of ConceptBase are used to classify the created objects on their voting. The following query defines ideas which have both pro and counter votes.

```
QueryClass Conflicting isA CoAUTHOR_Object with
   constraint
     conflicting: $ exists p/Pro c/Counter
                    (p target this) and (c target this) $
end
```

6 Conclusions

ConceptBase is a deductive object-oriented database which is particularly close to the deductive database approach. Without leaving the Datalog (with negation) framework, it makes object-oriented abstraction mechanisms available to the user, thus providing significant help for data structuring as compared with relational deductive databases. By its semantic closeness to the relational approach, it differs from the many recent attempts for a logical reconstruction of object-oriented database which, because they tend to work with more complicated logics (e.g., F-Logic [KL89], COL [AG91]), typically have to invent new implementation techniques rather than building on existing ones. This simplicity may also have been one of the factors determining the relatively large practical success of the system. The query language supported is similar to those offered by other object-oriented databases [CD91] but, like in deductive databases, more directly integrated with the rest of the data model; this has led to some useful ideas with many applications, such as the parameterization of query classes [SNJ93].

The O-Telos data model also naturally supports a two-layered implementation technique for deductive object bases, discussed in [JJR89], which is currently widely used in making active databases more declarative (STARBURST is a typical example [CW90]). The surface layer of the language works with global assertions (rules, constraints, query classes). These are then compiled to object-centered triggers which attach evaluations of specialized forms to exactly those classes of objects where insertions or deletions necessitate view maintenance. In ConceptBase, this approach is formally supported through the notion of an internal deductive object base in which the triggers are embedded. Experiments by ourselves and others [BM91], [RB90] show that the declarative form is much more concise and less error-prone than directly implementing the triggers as methods of object-oriented databases such as O_2.

The overall system architecture has by now reached a "sandwich" structure where the deductive, logic-centered part provides the "butter" between two pieces (the clients and the storage structure) which are essentially object-oriented in nature. This seems to exploit well the strength of the deductive approach in program transformation. We plan to strengthen further this approach by two measures. First, by using the logic layer as a tool of generating direct interoperability between client and storage structure, without passing data through the logic layer. This is expected to increase performance for complex object databases considerably, without making the logic much more complicated. Second, by extending the idea of query classes to a full integration of our approach with so-called concept logics in the style of KL-ONE [BS85] which allows for an automatic classification of certain objects and thus for a further reduction of the search space in many applications [BJNS93].

Acknowledgments. The development of ConceptBase over the past seven years would have been impossible without the help of our students. Eva Krüger designed the deductive integrity component. Thomas Wenig extended ConceptBase with an interval-based time calculus not described in this paper. Michael Gocek, Christian Rank and others implemented the first window-based user interface. Alain Rouge contributed his graphical browser. Hans W. Nissen organized the version and configuration management based on the model developed by Thomas Rose. Andre Klemann implemented a dependency browser on top of Alain's tool and extended ConceptBase by an interface to relational databases. Markus Baumeister re-implemented the user interface for the X11 library. The current version was implemented with the help of Masoud Asady, Michael Gebhardt, Farshad Lashgari, Thomas List, Christoph Radig, Kai von Thadden, and Claudia Welter. We have to thank our customers who were never satisfied with the expressiveness and performance of the system and thus stimulated us to continue thinking of better alternatives.

7 Appendix

7.1 Complete List of O-Telos Axioms

- Axiom 1: Object identifiers are unique.
  $\forall\ o, x_1, l_1, y_1, x_2, l_2, y_2\ P(o, x_1, l_1, y_1) \wedge P(o, x_2, l_2, y_2) \Rightarrow$
  $(x_1 = x_2) \wedge (l_1 = l_2) \wedge (y_1 = y_2)$
- Axiom 2: The name of individual objects is unique.
  $\forall\ o_1, o_2, l\ P(o_1, o_1, l, o_1) \wedge P(o_2, o_2, l, o_2) \Rightarrow (o_1 = o_2)$
- Axiom 3: Names of attributes are unique in conjunction with the source object.
  $\forall\ o_1, x, l, y_1, o_2, y_2\ P(o_1, x, l, y_1) \wedge P(o_2, x, l, y_2) \Rightarrow (o_1 = o_2) \vee (l = in) \vee (l = isa)$
- Axiom 4: The name of instantiation and specialization objects (*in, isa*) is unique in conjunction with source and destination objects.
  $\forall\ o_1, x, l, y, o_2\ P(o_1, x, l, y) \wedge P(o_2, x, l, y) \wedge ((l = in) \vee (l = isa)) \Rightarrow (o_1 = o_2)$
- Axioms 5,6,7,8: Solutions for the predicates *In*, *Isa*, and A are derived from the object base.
  $\forall\ o, x, c\ P(o, x, in, c) \Rightarrow In(x, c)$
  $\forall\ o, c, d\ P(o, c, isa, d) \Rightarrow Isa(c, d)$
  $\forall\ o, x, l, y, p, c, m, d\ P(o, x, l, y) \wedge P(p, c, m, d) \wedge In(o, p) \Rightarrow A'(x, l, m, y)$
  $\forall\ x, l, m, y\ A'(x, l, m, y) \Rightarrow A(x, m, y)$
- Axiom 9: An object x may not neglect an attribute definition in one of its classes.
  $\forall\ x, y, p, c, m, d\ In(x, c) \wedge A(x, m, y) \wedge P(p, c, m, d) \Rightarrow$
  $\exists\ o, l\ P(o, x, l, y) \wedge In(o, p)$
- Axioms 10,11,12: The *isa* relation is a partial order on the object identifiers.
  $\forall\ c\ In(c, \#\mathrm{Obj}) \Rightarrow Isa(c, c)$
  $\forall\ c, d, e\ Isa(c, d) \wedge Isa(d, e) \Rightarrow Isa(c, e)$
  $\forall\ c, d\ Isa(c, d) \wedge Isa(d, c) \Rightarrow (c = d)$
- Axiom 13: Class membership of objects is inherited upwardly to the superclasses.
  $\forall\ p, x, c, d\ In(x, d) \wedge P(p, d, isa, c) \Rightarrow In(x, c)$
- Axiom 14: Attributes are "typed" by their attribute classes.
  $\forall\ o, x, l, y, p\ P(o, x, l, y) \wedge In(o, p) \Rightarrow \exists\ c, m, d\ P(p, c, m, d) \wedge In(x, c) \wedge In(y, d)$
- Axiom 15: Subclasses which define attributes with the same name as attributes of their attributes must refine these attributes.
  $\forall\ c, d, a_1, a_2, m, e, f$
  $Isa(d, c) \wedge P(a_1, c, m, e) \wedge P(a_2, d, m, f) \Rightarrow Isa(f, e) \wedge Isa(a_2, a_1)$
- Axiom 16: If an attribute is a refinement (subclass) of another attribute then it must also refine the source and destination components.
  $\forall\ c, d, a_1, a_2, m_1, m_2, e, f$
  $Isa(a_2, a_1) \wedge P(a_1, c, m_1, e) \wedge P(a_2, d, m_2, f) \Rightarrow Isa(d, c) \wedge Isa(f, e)$

- Axiom 17: For any object there is always a unique "smallest" attribute class with a given label m.
  $\forall\ x, m, y, c, d, a_1, a_2, e, f\ (In(x,c) \wedge In(x,d) \wedge P(a_1, c, m, e) \wedge P(a_2, d, m, f)$
  $\Rightarrow \exists\ g, a_3, h\ In(x,g) \wedge P(a_3, g, m, h) \wedge Isa(g,c) \wedge Isa(g,d))$
- Axioms 18-22: Membership to the builtin classes is determined by the object's format.
  $\forall\ o, x, l, y\ (P(o,x,l,y) \Leftrightarrow In(o, \#\text{Obj}))$
  $\forall\ o, l\ (P(o,o,l,o) \Leftrightarrow In(o, \#\text{Indiv}))$
  $\forall\ o, x, c\ (P(o,x,in,c) \Leftrightarrow In(o, \#\text{Inst}))$
  $\forall\ o, c, d\ (P(o,c,isa,d) \Leftrightarrow In(o, \#\text{Spec}))$
  $\forall\ o, x, l, y\ (P(o,x,l,y) \wedge (o \neq x) \wedge (o \neq y) \wedge (l \neq in) \wedge (l \neq isa) \Leftrightarrow In(o, \#\text{Attr}))$
- Axiom 23: Any object falls into one of the four categories.
  $\forall\ o\ In(o, \#\text{Obj}) \Rightarrow In(o, \#\text{Indiv}) \vee In(o, \#\text{Inst}) \vee In(o, \#\text{Spec}) \vee In(o, \#\text{Attr})$
- Axioms 24-28: There are five builtin classes.
  $P(\#Obj, \#Obj, \text{Object}, \#Obj)$
  $P(\#Indiv, \#Indiv, \text{Individual}, \#Indiv)$
  $P(\#Attr, \#Obj, \text{attribute}, \#Obj)$
  $P(\#Inst, \#Obj, \text{InstanceOf}, \#Obj)$
  $P(\#Spec, \#Obj, \text{IsA}, \#Obj)$
- Axiom 29: Objects must be known before they are referenced. The operator $\preceq$ is a (predefined) total order on the set of identifiers.
  $\forall\ o, x, l, y\ P(o,x,l,y) \Rightarrow (x \preceq o) \wedge (y \preceq o)$
- Axioms 30*: For any object $P(p, c, m, d)$ in the extensional object base we have two formulas for "rewriting" the In and A predicates.
  $\forall\ o\ In(o,p) \Rightarrow In.p(o)$
  $\forall\ o, x, l, y\ P(o,x,l,y) \wedge In(o,p) \Rightarrow A.p(x,y)$

The objects `Class`, `QueryClass`, `Tool` etc. mentioned in the paper are not part of the O-Telos axioms but predefined in ConceptBase. ConceptBase implements some of the axioms as rules (e.g., axioms 13, 30*), others as integrity constraints.

7.2 Example of a Query Evaluation

First, we complete our example object base with some additional objects and then demonstrate the evaluation process for a query class that is based both on the `bossrule` and on the query class `IndEmp`. In addition to the instances `PR` and `R&D` of `Department` we assume a department `Staff`, declare `Mary` as head of `R&D`, a new manager `Bill` as head of `PR` and introduce two other new employees `John` and `Mike` (see fig. 8).

```
Mike in Employee with
  dept
    belongs:Staff
  salary
    earns:100000
end
```

```
John in Employee with
  dept
    works:R&D
  salary
    sal:30000
end
```

```
QueryClass JohnsBosses isA Employee with
      constraint
        bossOrInd: $ (John boss this) or
                     (this in IndEmp(John/lowersal)) $
end
```

The query interface receives a message to execute the operation `ASK(JohnsBosses)`, i.e. to retrieve all existing employees from the object base who are bosses of `John` or are just independent employees but have a higher salary than `John`.

As in 3.2.3 this query class leads to several equations that are used for the evaluation. Effectively, the system has to evaluate two subqueries based on the example equations, namely $A.\#boss(John, x)$ and $IndEmp(x, John)$. Hence the relevant set of equations selected by the `QueryEvaluator` is exactly the set displayed in fig. 7 completed with the missing base equations. The magic relations triggering the forward chaining process for computing both subqueries are initialized by inserting $John$ into $q\_A.\#boss^{bf}$ and into q_IndEmp^{fb}. Due to the only negated occurrence of IE_1 in equation (8) we can compute the least fixpoint of all other equations and in a second iteration include equation (8). Thus here the computation of strata is very simple. Fig. 11 shows the individual steps in both iterations. For a relation r we denote the changes after each step with Δr and indicate the applied equation where (BE) stands for basic equations. The result from this computation, namely the relations

- $A.\#boss = \{(John, Mary), (Mary, Bill), (John, Bill)\}$ and
- $IndEmp = \{(Mike, John)\}$

have to be joined to get the answer instances of the query class `JohnsBosses`: `Mary`,`Bill` and `Mike`.

Iteration 1			
	step 1	$\Delta q\_A.\#boss^{bf} = \Delta q\_IndEmp^{fb} = \{John\}$	
	step 2	$\Delta q\_A.\#dept^{bf} = \{John\}$	(3)
		$\Delta q\_A.\#salary^{ff} = TRUE$ [a]	(11)
	step 3	$\Delta A.\#dept = \{(John, RD)\}$	(BE)
		$\Delta A.\#salary = \{(John, 30000), (Mary, 50000), (Mike, 100000)\}$	(BE)
	step 4	$\Delta sup_{1,1} = \{(John, R\&D)\}$	(2)
		$\Delta sup_{3,1} = \{(John, John, 30000), (John, Mary, 50000), (John, Mike, 100000)\}$	(9)
	step 5[b]	$\Delta q\_A.\#head^{bf} = \{R\&D\}$	(4)
		$\Delta sup_{3,2} = \{(John, Mary), (John, Mike)\}$	(10)
	step 6	$\Delta A.\#head = \{(R\&D, Mary)\}$	(BE)
		$\Delta q\_IE_1 = \{Mary, Mike\}$	(12)
	step 7	$\Delta A.\#boss = \{(John, Mary)\}$	(1)
		$\Delta q\_A.\#boss^{bf} = \{Mary, Mike\}$	(14)
	step 8	$\Delta q\_A.\#dept^{bf} = \{Mary, Mike\}$	(3)
		$\Delta sup_{2,1} = \{(John, Mary)\}$	(6)
	step 9	$\Delta A.\#dept = \{(Mary, R\&D), (Mary, PR), (Mike, Staff)\}$	(BE)
	step 10	$\Delta sup_{1,1} = \{(Mary, R\&D), (Mary, PR), (Mike, Staff)\}$	(2)
	step 11	$\Delta q\_A.\#head^{bf} = \{PR, Staff\}$	(4)
	step 12	$\Delta A.\#head = \{(PR, Bill)\}$	(BE)
	step 13	$\Delta A.\#boss = \{(Mary, Bill)$	(1)
	step 14	$\Delta A.\#boss = \{(John, Bill)$	(1)
		$\Delta sup_{1,1} = \{(Mary, Bill)\}$	(6)
		$\Delta IE_1 = \{Mary\}$	(13)
	step 15	$\Delta q\_A.\#boss^{bf} = \{Bill\}$	(7)
	step 16	$\Delta q\_A.\#dept^{bf} = \{Bill\}$	(3)
Iteration 2			
	step 1	$\Delta IndEmp = \{(Mike, John)\}$	(8)

[a] We write TRUE and FALSE to indicate (non)emptiness of 0-ary relations.
[b] Please note that we don't evaluate (8).

Figure 11: An example fixpoint computation

References

[AG91] Abiteboul S., Grumbach S. "A rule-based language with functions and sets." *ACM Transactions on Database Systems* 16, 1, March 1991, pp. 1–30.

[AGJA90] Agrawal R., Jagadish H.V., "Hybrid transitive closure algorithms", In *Proc. 16th International Conference on Very Large Data Bases*, Brisbane, Australia 1990.

[ALU93] Abiteboul S., Lausen G., Uphoff H., Waller E., "Methods and Rules", In *Proc. ACM SIGMOD*, Washington, DC 1990, pp. 32–41.

[BBDG90] Bellosta M.J., Bessede A., Darrieumerlou C., Gruber O., Pucheral Ph., Thevenin J.M., Steffen H., "GEODE – concepts and facilities", INRIA - Rocquencourt 1990.

[BDM88] Bry F., Decker H., Manthey R., "A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases", In *Proc. EDBT*, pp. 488–505.

[BJNS93] Buchheit M., Jeusfeld M., Nutt W., Staudt M., "Subsumption between Queries to Object-Oriented Databases." In *Proc. EDBT*, Cambridge, UK, March 1994.

Also available as *Aachener Informatik-Berichte* 93-9, RWTH Aachen, Germany, 1993.

[BM91] Bouzeghoub M., Métais E. "Semantic modeling of object oriented databases." *Proc. VLDB'91*, Barcelona, Spanien, pp. 3–14.

[BMS84] Brodie M.L., Mylopoulos J., Schmidt J.W. (ed.), *On Conceptual Modelling*, Springer-Verlag.

[BORE90] Borenstein N.S., *Multimedia applications development with the Andrew Toolkit*, Prentice-Hall, N.J., 1990.

[BR87] Beeri C., Ramakrishnan R., "On the power of magic", In *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems.*

[BS85] Brachman R.J., Schmolze J.G., "An overview of the KL-ONE knowledge representation system." In *Cognitive Science* 9, 2, April 1985, pp. 171–216.

[CARD75] Cardenas A. F., "Analysis and performance of inverted data base structures", In *CACM*, Vol. 18, No. 5, May 1987.

[CD91] Cluet S., Delobel C. "Towards a unification of rewrite based optimization techniques for object-oriented queries." In *Proc. VII'eme journées Bases de Données Advancées*, Lyon, France.

[CGT90] Ceri S., Gottlob G., Tanca L., *Logic programming and databases*, Springer-Verlag.

[CHE93] Chen P.S., "On inference rules of logic-based information retrieval systems", To appear in *Intl. J. Information Processing & Management*, 1993.

[CW90] Ceri S., Widom J. "Deriving production rules for constraint maintenance." In *Proc. 16th VLDB Conf.*, Brisbane, Australia, pp. 566–577.

[EJ91] Eherer S., Jarke M., "Knowledge-based support for hypertext co-authoring", In *Proc. 2nd Intl. Conf. Database and Expert Systems Applications (DEXA'91)*, Berlin, Germany, Aug. 21-23, 1991, pp. 465–470.

[FUV83] Fagin R., Ullman J.D., Vardi M.Y., "On the semantics of updates in databases", In *Proc. of Second ACM SIGACT-SIGMOD*, pp. 352–365.

[GALL90] Gallersdörfer R., *Realization of a deductive object base by abstract data types* (in German), Diploma thesis, Universität Passau, Germany 1990.

[JARK93] Jarke M. (ed.), *Database application engineering with DAIDA*, Springer-Verlag, 1993.

[JBR*93] Jarke M., Bubenko J., Rolland C., Sutcliffe A., Vassiliou Y., "Theories underlying requirements engineering – an overview of NATURE at Genesis.", In *Proc. 1st Int. IEEE Symposium on Requirements Engineering*, San Diego.

[JEUS92] Jeusfeld M., *Update control in deductive object bases* (in German), Infix-Verlag, St.Augustin, Germany.

[JJ91] Jeusfeld M., Jarke M., "From relational to object-oriented integrity simplification", In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases, LNCS 566*, Springer-Verlag, pp. 460–477.

[JJR89] Jarke M., Jeusfeld M., Rose T., "Software process modeling as a strategy for KBMS implementation." In *Proc. 1st Intl. Conf. Deductive and Object-Oriented Databases*, Kyoto, Japan, Dec. 1989.

[KL89] Kifer M., Lausen G., "F-Logic: a higher-order language for reasoning about objects, inheritance, and scheme." In *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Portland, Oregon, 1989, pp. 134–146.

[KM90] Kakas A,C., Mancarella P., "Database updates through abduction." In *Proc. VLDB*, Brisbane, Australia, 1990, pp. 650–661.

[KUEC91] Küchenhoff V., "On the efficient computation of the difference between consecutive database states", In *Proc. 2nd Intl. Conf. Deductive and Object-Oriented Databases*, München, Germany, Dec. 1991.

[LEA91] Lea D., *User's Guide to the GNU C++ Library*, 1991.

[LT84] Lloyd J.W., Topor R.W., "Making PROLOG more expressive", In *Journal of Logic Programming*, pp. 225-240, March 1984.

[MBJK90] Mylopoulos J., Borgida A., Jarke M., Koubarakis M., "Telos – a language for representing knowledge about information systems", In *ACM Trans. Information Systems* **8**(4), pp. 325–362.

[OLIV91] Olivé A., "Integrity constraints checking in deductive databases", In *Proc. VLDB'91*, Barcelona, Spain, pp. 513–524.

[PT92] Pucheral P., Thevenin J.-M. "Pipelined query processing in the DBGraph storage model." In *Proc. EDBT'92*, Vienna, Austria, March 1992, pp. 516–533.

[RB90] Rios-Zertuche D., Buchmannn A. *Execution models for active databases – a comparison.* Technical Report, GTE Laboratories, Waltham, MA, 1990.

[RD92] Ramesh B., Dhar V., "Process knowledge-based group support for requirements engineering", In *Jounal of Intelligent Information Systems* 1,1.

[RJM92] Rose T., Jarke M., Mylopoulos J., "Organizing software repositories – modeling requirements and implementation experiences", In *Proc. 16th Intl. Computer Software & Applications Conf.*, Chicago, IL, Sept. 23-25, 1992.

[RL93] Ramesh B., Luqi "Process knowledge based rapid prototyping for requirements engineering." In *Proc. IEEE Intl. Symposium on Requirements Engineering (RE'93)*, San Diego, CA, Jan. 1993, pp. 248-255.

[SIGA91] "Special Issue on Implemented Knowledge Representation and Reasoning Systems", *SIGART Bulletin* 2,3, June 1991.

[SJ92] Steinke G., Jarke M., "Support for security modeling in information systems design", In *Proc. IFIP 11.3 Working Conf. on Database Security*, Vancouver, Canada, August 19-22, 1992.

[SLT91] Scholl M., Laasch C., Tresch M. "Updatable views in object-oriented databases." In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases, LNCS 566*, Springer-Verlag, pp. 189–207.

[SOKO91] Sokolowski S., *Applicative high order programming*, Chapman and Hall Computing Series, 1991.

[SNJ93] Staudt M., Nissen H.W., Jeusfeld M.A., "Query by class, rule and concept." To appear in *Applied Intelligence, Special Issue on Knowledge Base Management.*

[STAN86] Stanley M.T., *CML – a knowledge representation language with application to requirements modelling*, M.S.thesis, University of Toronto, Ontario.

[STAU90] Staudt M., *Query representation and evaluation in deductive object bases* (in German), Diploma thesis, Universität Passau, Germany.

[STRO92] Stroustrup B., *The C++ programming language*, Second Edition, Addison-Wesley, 1992.

[SZ90] Shaw G.M., Zdonik S.B., "A query algebra for object-oriented databases", In *Proceedings of the 6th Int. Conf. on Data Engineering*, pp. 154-162.

[THOE92] Thönnissen H.J. *Design and implementation of an object algebra for a deductive object base system* (in German), Diploma thesis, RWTH Aachen, Germany.

[TKDE90] Special Issue on Database Prototype Systems, *IEEE Trans. on Knowledge and Data Engineering* 2, 1, March 1990.

[ULLM88] Ullman J.D., *Principles of Database and Knowledge Base Systems*, Vol. I., Computer Science Press 1988.

[ULLM89] Ullman J.D., *Principles of database and Knowledge Base Systems*, Vol. II., Computer Science Press 1989.

[VALD87] Valduriez P., "Join indices", In *ACM TODS*, Vol. 12, No. 2, June 1987.

[VD91] Vandenberg S.L., DeWitt D.J., "Algebraic support for complex objects with arrays, identity and inheritance", In *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, pp. 158-167, 1991.

[VW91] Vossen G., Witt, K.-U., *FASTFOOD: a formal algebra over sets and tuples for the FOOD object-oriented data model*, Report 91/03, University of Gießen.