An Axiomatic Model for Deductive Object-Oriented Databases

Zsolt Tivadar Kardkovács, Gábor Mihály Surányi

Department of Telecommunications and Media Informatics,
Budapest University of Technology and Economics,
Magyar tudósok körútja 2., H–1117 Budapest, Hungary
{kardkovacs, suranyi}@db.bme.hu

Abstract

Databases with both object-oriented and deductive features (DOOD) seem to be the most promising database technology. There are successful approaches to designing such systems. However, the resulting systems, in our opinion, still lack several important properties of full-featured DOOD's. Realizing that some practical problems cannot be tackled with the former implementations, we reconsidered the mathematical foundations of the existing models. We propose a new model for DOOD's that incorporates the whole range of deductive capabilities including constraints. We sketch out a proof that shows our model complies with Object Management Group's Unified Modeling Language Specification (OMG's UML).

1 Introduction

Databases, object-oriented paradigm and logic programming are three independently developed areas in computer science. Both the logical and the object-oriented approaches to database design attracted considerable interest in the previous decades. Logic forms the basis for efficient knowledge-based systems, while object-orientated database languages are popular for their straightforward interface to applications. A number of attempts to combine the two approaches have been reported in the literature. Their goal was to create more expressive database models than the object-orientated ones can be.

Why is this important? We mention here only two reasons. First of all, describing the surrounding world is not limited to the identification of the objects and their properties, but it must employ logical elements to express relations between objects as well. For example the structure of a watch and a mill is greatly identical and we want represent it somehow. Or a Lamborghini, a diamond ring or da Vinci's Mona Lisa are all precious, etc.

Second, there are several problems, for instance in decision support, which can often not be solved efficiently by means of monoparadigmic methods. Consider, e.g. your

private doctor, who makes decisions in order to cure people. Obviously, the effect of a therapy varies from person to person because the reactions depend on the anamnesis. In general, the effect of a treatment is most satisfactory if similarities to former cases are not only recognized, but also exploited. To identify similarities, one would most naturally choose a deductive language. To represent diseases with their evolutions, e.g., object-orientation seems to be the most powerful paradigm.

Our work is aimed to advance the integration of database theory, object-oriented paradigm and logic programming by developing the most flexible and expressive database design technology. In the next section we point out some defects of previous DOOD models. Then an axiomatic model is laid down as the foundation of the next generation of DOOD systems.

2 Problems of Existing Models

Though the meaning of the term DOOD highly varies between models and implementations, in the mathematical basis they have some common elements taken from David Maier's work [13]. Its main goal was to find a sound, complete and computable logic corresponding to the object-oriented paradigm. As it is seen in the papers [10, 11, 6, 7, 4, 1, 16], there are well-defined structures which fulfill object-orientated criteria but it is also observable these models do not go beyond the capabilities of the object-oriented paradigm: several logical and dynamic features are lost or there is trouble with reasoning (cf. [2, 12]). Our work focuses on incorporating the solutions for the problems outlined next into the existing results.

Mandatory vs. optional conditions. In the case of a Prolog clause, e.g.

$$\texttt{proper(X)} : -\texttt{odd(X)}. ,$$

there is practically no clue of the intended usage. In other words, it is not clear if X *must be* odd or we want to *check whether* X is odd. None of the DOOD models[10, 11, 6, 7, 3, 1, 16] is capable of specifying object invariance constraints. Only via integration of further languages could these logics be partially reconciled with constraint modeling[6, 1]. Nevertheless, it is unclear how all the mentioned models could be extended in practical ways.

Contradiction in inheritance. Take a look at the problem called Nixon's diamond[7, 9]. Nixon is a quaker and also a republican. Quakers are pacifists while republicans are hawks, thus exactly one of the properties can be inherited by Nixon. If all objects are represented by a single mathematical structure, one of the features must be blocked non-deterministically in order to retain soundness [10, 7, 16]. Another option is the integration either of an object-oriented programming language (OOPL)[6] or of active elements[5]. In the latter case, it is determined in runtime which property is eventually inherited[1, 3]. Independently of the actual realization, it always has to be decided to which class the child logically and uniquely belongs. The subclass should have the properties of both superclasses but it is not able to do so. Some information is therefore lost during reasoning.

Dynamic classification. Often only partial information is available about an entity. Let us consider, e.g., the problem of hunt for presents (which is being more and more acute now with the approach of the Christmas season). In these cases we need to evaluate a generalized query that matches dynamically any object regardless of its position in the inheritance hierarchy. Hence, although two (or more) instances are not related to each other in the hierarchy, they could fit into the same type. A type system should manage this dynamism. The notion of types is not supported in DOOD's and is even not elaborated unlike the concept of classes[10]. In this paper we try to clarify, amongst others, what type might mean in a DOOD.

The following section demonstrates that our concept formally corresponds to the most common object-oriented specification, to UML[14]. It is proved in [10] that the model theory of Maier (improved by Kifer) is a sound and complete OOPL. Thus we also point out indirectly that the expressiveness of the axiomatic model is comparable to Maier's idea and additionally, it elegantly solves the aforementioned problems.

3 The Axiomatic Model

3.1 Syntax

In this section we present our object-oriented data model formally and show that the notions of the object-oriented paradigm have their equivalent in it. The latter we accomplish informally as we compare our concepts to the definitions of the Core subpackage of the Foundation package of UML[14], which are themselves informal.

The axiomatic object model is a logic and therefore has a language, $\mathcal{L}$, that consists of:

- an infinite set of variables,
- four disjoint sets of identifiers (namely $\mathcal{C}$, $\mathcal{T}$, $\mathcal{O}$, $\mathcal{I}$, $\mathcal{F}$ and $\mathcal{X}$ for classes, types, objects, type-instances, object fields and auxiliaries respectively — all of them will be introduced later),
- a set of predicate symbols, $\mathcal{P}$,
- auxiliary symbols, such as $($, $)$, @, $[$, $]$,
- the usual logical connectives (i.e. $\wedge, \vee, \neg, \leftarrow$),
- the usual quantifiers (i.e. $\exists, \forall$).

The elements of the identifier sets play the role of function symbols in the model. Elements of $\mathcal{C}$, $\mathcal{T}$, $\mathcal{O}$, $\mathcal{I}$ and $\mathcal{X}$ are constants. Since this is an object-oriented data model for databases, every individual in the corresponding structure has a constant in $\mathcal{L}$.

Atomic and complex formulae can be constructed the usual way,only the @, [,] signs need further explanation.

@ corresponds to the field retrieval/message sending operator. In fact, it is a syntactic sugar: $i@f$ stands for $f(i)$ and $i@f(a_1, \ldots)$ for $f(i, a_1, \ldots)$, where i is a class, type, object or type-instance identifier, a_j's are arguments, i.e. terms. Thus the arity of every element of $\mathcal{F}$ is at least one.

Comma-separated terms between [and] denote a special term: a set. The set's elements are the enumerated terms. Sets can be used as "normal" terms, i.e. their occurrences in functions, predicates are allowed. Using sets in basic (i.e. not user-defined) predicates has the meaning of conjunctions of atomic formulae with non-set terms in all combinations. User-defined predicates and field identifiers (operations) can handle set-valued arguments in the way they wish[1]; sending a message to a set is interpreted as sending the message to each element of the set, and the results are collected in a set.

Now we are ready to define the object-oriented notions in our model.

3.2 The Concept of Classes

A *class* **C** is a triple $\langle \mathbf{c}, \mathbf{S}, \mathbf{M} \rangle$ where $\mathbf{c} \in \mathcal{C}$ (the class identifier), **S** is a set of axioms (true closed statements), **M** is the implementation of operations (methods) describing the class. The elements of **S** can be divided into three groups: *attribute, operation* and *constraint declarations*.

Attribute declarations have the form of $attr(f)$ where $attr \in \mathcal{P}$ and $f \in \mathcal{F}$ meaning the class has an attribute named f. Operations provided by a class described via $oper(f, arity)$ predicates where $oper \in \mathcal{P}$, $f \in \mathcal{F}$ (the name) and $arity$ is a natural number.

Constraint declarations provide new information about the fields defined by the previously introduced declarations. These constraints may use built-in predicates, such as $typeof(f, t)$ ($typeof \in \mathcal{P}$, $f \in \mathcal{F}$, $t \in \mathcal{T}$), which asserts that the value of f is compatible with t.[2] Or, alternatively, the constraints can be expressed via user-defined predicates (which are members of $\mathcal{P}$, too). These predicates, as obvious, always have a "body", an intended meaning usually expressed by an implication formulae that is included in the constraint declarations.

Of course the components **S** and **M** are not independent. There are several well-formedness rules that must hold otherwise a class is considered ill-formed. They include e.g. field-name uniqueness, each operation must have a realization (method).

$\mathcal{L}$ has a special binary predicate, $=$. Its existence makes possible to define the sameness of two classes, $\mathbf{C_1}$ and $\mathbf{C_2}$, ensuring no classes with same identifiers but dif-

[1] However it requires the existence of special set-functions.

[2] We will discuss type-compatibility later in details.

ferent axiom-sets/methods can exist:[3] $\mathbf{C_1} = \mathbf{C_2}$, i.e. are the same if and only if the same holds for their identifiers: $\mathbf{c_1} = \mathbf{c_2}$. The case will be similar for objects, types and type-instances. Thus in the rest of this section identifiers will be used instead of their owner, which one we mean will always be unambiguous from the context.

Returning to our reference, UML, a class is "a description of [...] attributes, operations, methods, relationships, and semantics."[14] Relationships, i.e. generalizations and associations, will be discussed later in this paper. But, we have just shown, all properties of the other elements are expressible in our model, however, some of the concrete symbols etc. was suppressed for simplicity. Examples are field-scopes and visibility. Their transformation into our model is an easy task. Note that the axiomatic model is capable of asserting pretty complex facts about the objects via its rather free constraint declaration mechanism.

3.3 Objects

The second notion in our model is the *object (instance)*. For its definition we need a notation: let $\mathbf{t} = \langle \ldots, \mathbf{c}, \ldots \rangle$ be a tuple, then $\mathbf{t^{(c)}}$ denotes the component of $\mathbf{t}$ which is denoted by $\mathbf{c}$.

Every object is a triple $\langle \mathbf{o}, \mathbf{c}, \mathbf{M} \rangle$ where $\mathbf{o} \in \mathcal{O}$ (object identifier) and $\mathbf{M}$ is a model of $\mathbf{c^{(S)}}$ for the primary class $\mathbf{c}$: $\mathbf{M} \models \mathbf{c^{(S)}}$. $\mathbf{M}$ has to be minimal in the sense that it assigns individuals to unary function symbols introduced by $\mathbf{c}$ only. In fact the value assigned to a particular attribute is one of the followings:

- NULL$\in\mathcal{X}$ (the well-known special symbol for unknown values),
- $[\,]\in\mathcal{X}$ (the empty set),
- a set of object identifiers.[4]

The properties of NULL and $[\,]$, although they are fairly intuitive, require further discussion, and are not in the scope of this paper.

The concept of object we have just introduced is directly compatible with the one of UML: "an object is an instance of a class [...] which has a state that stores the effects of the operations".[14]

3.4 The Type System

UML makes distinction between (implementation) classes and *types*. The former's equivalent in our model is already known to the reader while the latter's is not. Now we fill this gap.

[3] Were classes in the object model a single partially defined function, $\mathcal{C} \longrightarrow 2^{F(\mathcal{L})}$ where $F(\mathcal{L})$ is the set of all closed formulae of $\mathcal{L}$, that would be trivial.

[4] A set containing only one element represents the element itself.

In fact, the notion of types in the axiomatic model is more sophisticated than UML's. A type is not only "used to specify a domain of objects together with operations applicable to the objects without defining the physical implementation of those objects"[14] but may implement features similar to delegation, dynamic "classification" and templates (via so-called *type-instances*).

There fundamental ideas of our types are as follows.

1. As opposed to UML, objects do not belong to types indirectly, through their primary class, but they do directly, on their own, satisfying the types' axioms.
2. It was realized a long ago that using a strictly typed language helps to develop bug-free applications. Thus objects are usually manipulated as instances of types.[5] We treat typed objects (type-instances) standalone entities which may have their own attributes and functions.

The formalism is the following: a type is a tuple $\langle \mathbf{t}, \mathbf{CS}, \mathbf{TS}, \mathbf{M} \rangle$, where

- $\mathbf{t} \in \mathcal{T}$ (type identifier),
- **CS** is a set of axioms describing fields provided by the object itself,
- **TS** defines fields of type-instances and constraints on any field as the **S** component of a class,
- **M** is the implementation of operations (methods) defined by **TS**.

The specification of the fields belonging to the objects and not to type-instances (e.g. **CS**) can be performed in a number of different ways, from the user's view. One may say, for example, $t_1 \in \mathcal{T}$ is a union of $c_1 \in \mathcal{C}$ and $c_2 \in \mathcal{C}$ or a union of $t_2 \in \mathcal{T}$ and $t_3 \in \mathcal{T}$, or, like an interface, all instances of $t_4 \in \mathcal{T}$ must have an operation `foo`. In any case

$$\neg\exists\varphi \quad \mathbf{CS} \vdash \varphi \ \wedge \ \mathbf{TS} \vdash \varphi$$

must hold, i.e. fields of a type-instance must be different from its "disguised" object. It is just a formal restriction because such name collisions can be avoided via renaming the field(s) considered if necessary.

Because the $typeof$ predicate requires types, not classes, for each class there must exist exactly one type containing the same declarations as the corresponding class. By means of separating types and classes it is easily possible to assign NULL's and $[\,]$'s to any member field: the semantics of the predicate symbol for the integer type may be, e.g., "it is an element of the integer class or NULL or $[\,]$".

[5] Furthermore, classes and types often have one to one correspondence.

To make the discussion of type-instances simpler let us introduce a few notations:

$$\mathbf{S} \stackrel{\text{def}}{=} \mathbf{CS} \cup \mathbf{TS}$$

$$\mathbf{T^{(S)}} \stackrel{\text{def}}{=} \mathbf{T^{(CS)}} \cup \mathbf{T^{(TS)}}$$

for any type **T**. Finally, $\prod_{\mathbf{S}} \mathbf{M}$ is a model, where **S** is an axiom-set, **M** is a model, such that it contains only truth-values of formulae in which there are no function symbols in addition to the ones used in **S**. It can be interpreted as a projection, what explains the usage of the symbol $\prod$.

A type-instance is represented by a tuple $\langle \mathbf{i}, \mathbf{t}, \mathbf{o}, \mathbf{M} \rangle$ where $\mathbf{i} \in \mathcal{I}$ is the type-instance identifier, $\mathbf{t} \in \mathcal{T}$ is its type, $\mathbf{o} \in \mathcal{O}$ is the disguised object, and **M** is a model such that:

$$\prod\nolimits_{\mathbf{t^{(CS)}}} \mathbf{o^{(M)}} \cup \mathbf{M} \models \mathbf{t^{(S)}}.$$

As at the definition of objects **M** must be minimal in a sense: $\prod_{\mathbf{t^{(TS)}}} \mathbf{M} = \mathbf{M}$. Type-instances may replace objects in any expression, the meaning is the same as if the instance's **o** component were there.

Because types are independent of the class hierarchy, type-compatibility of our data model does not need the concept of generalization:

$$\mathbf{o} \circ \mathbf{t} \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \exists \mathbf{M} \quad \prod\nolimits_{\mathbf{t^{(CS)}}} \mathbf{o^{(M)}} \cup \mathbf{M} \models \mathbf{t^{(S)}},$$

that is an object **o**, is compatible with a type **t**, if and only if an assignment of instance-fields exists such that the assignments together satisfy all of the type's axioms.

3.5 Generalization

One of the key concepts of the object-oriented paradigm is *generalization*. It is "a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information."[14]

In our model generalization is not of that great significance because the cases, in which it is needed to handle objects with several identical properties in a uniform way, can be modeled more elegantly by means of the flexible type system described previously. But UML mentions taxonomic relationships, too, and they have to be retained. The user may thus specify $\mathbf{c_1} : \mathbf{c_2}$ ($\mathbf{c_1}, \mathbf{c_2} \in \mathcal{C}$, $: \in \mathcal{P}$) *suggesting* $\mathbf{c_2}$ is a generalization of $\mathbf{c_1}$.

In fact, $\mathbf{c_2}$ is a generalization of $\mathbf{c_1}$ ($\mathbf{c_1} \prec \mathbf{c_2}$) if and only if

$$\mathbf{c_1^{(S)}} \vdash \mathbf{c_2^{(S)}} \wedge \mathbf{c_1} : \mathbf{c_2}.$$

This definition conforms to UML:

- If $\mathbf{c_2}$ is a generalization of $\mathbf{c_1}$, the latter has all fields of the former with their properties and this fact is reflected by the axiom-sets (see the definition of the concept class). And finally, the generalization is asserted by the user, $\mathbf{c_1}\colon\mathbf{c_2}$, otherwise taxonomically $\mathbf{c_1}$ is not an offspring of $\mathbf{c_2}$, they are only somewhat similar to each other.
- If the user asserts $\mathbf{c_1}\colon\mathbf{c_2}$ and $\mathbf{c_1^{(S)}} \vdash \mathbf{c_2^{(S)}}$ holds, all fields and properties of $\mathbf{c_2}$ are possessed by $\mathbf{c_1}$ (because of the definition of $\vdash$), i.e. $\mathbf{c_2}$ is a generalization of $\mathbf{c_1}$.

Our interpretation of generalization implies two obvious things:

1. From the view of the implementation of the axiomatic model the simplest (and most evident) way to realize generalization (inheritance) is to include $\mathbf{c_2^{(S)}}$ in $\mathbf{c_1^{(S)}}$ if $\mathbf{c_1}\colon\mathbf{c_2}$.
2. If one is interested in structural, behavioral inclusion (but not in the original taxonomic facts) only entailment is to be considered.

The previously defined subclass relationship ($\prec$), as usual, is a partial order on classes, i.e. it is reflexive, transitive and antisymmetric. Consequently, we can speak about class hierarchy. Furthermore, ”an instance of the more specific element may be used where the more general element is allowed”[14], i.e.

$$\mathbf{o^{(M)}} \models \mathbf{c_1^{(S)}} \wedge \mathbf{c_1} \prec \mathbf{c_2} \quad \Longrightarrow \quad \mathbf{o^{(M)}} \models \mathbf{c_2^{(S)}}.$$

The proofs of the propositions are pretty straightforward taking into consideration the properties of $\vdash$ and :. (The latter, because of its semantics, is a partial order, too.)

In an object-oriented model, certain classes are said to be parents (children) of other classes. What is the equivalent of those relationships in the axiomatic model? As the user supplies ancestors/offsprings with :, $\prec$ does not specify parents/children directly. It is true that if

$$\mathbf{c_1} \prec \mathbf{c_3} \quad \wedge \quad \neg\exists \mathbf{c_2} \ \left(\mathbf{c_1} \prec \mathbf{c_2} \ \wedge \ \mathbf{c_2} \prec \mathbf{c_3}\right),$$

then $\mathbf{c_1}$ is a child of $\mathbf{c_3} \in \mathcal{C}$, but not conversely: in the user’s taxonomic model a class may be parent and grandparent of another class at the same time. That information is lost in this way. Alternatively, : might *suggest* parent–child relationship and then $\prec$ would be its transitive closure (not forgetting the entailment that is obligatory).

Generalization can be defined among types, too. But, since their usage differs from the one of classes, it is usually unnecessary.

3.6 Differential Inheritance

An object-oriented model is ill-formed if it is inconsistent.[14] So is the axiomatic model. But sometimes it would be nice if we could retain the taxonomic relationship although a property of the subclass contradicts one of those of the superclass. Consider a database of a hypermarket with two classes: `wooden_spoon` and `wooden_spoon_made_of_plastic`. Of course, the former's material is wood, the latter's plastic. To model it, one needs to block the inheritance of the constraint on the field named `material`, or, as we call it, specify *differential inheritance.*

In our model differential inheritance, denoted by $\mathbf{c_1}:_d \mathbf{c_2}$ where d means differential, seems to be very simple, only the axioms considered must not exist in the offspring class, $\mathbf{c_1}$. But then $\mathbf{c_1}$ is no longer a subclass of its ancestor, $\mathbf{c_2}$, since $\mathbf{c_1^{(S)}} \not\vdash \mathbf{c_2^{(S)}}$. How can we make use of the taxonomic relationship then?

The purpose of a taxonomic relationship, when differential inheritance is in effect, is to be able to answer queries such as "for which entities in the group $\mathbf{c_2}$ is f true" or "what is the value of the field(s) of the entities in the group $\mathbf{c_2}$ for which f is true" where f is a condition and $\mathbf{c_2}$ has at least one differential offspring, $\mathbf{c_1}$. If inheritance blocking does not affect the relationships among the fields involved in the query, one may say, $\mathbf{c_1}$ is a subclass of $\mathbf{c_2}$ considering the query q. Returning to our spoon example if we are not interested in the material of the wooden spoon and no condition refers to it in a query, `wooden_spoon_made_of_plastic` *is a* `wooden_spoon` (for the time of the query).

Formally, let $\prod_{[q]} \mathbf{c^{(S)}}$ ($\mathbf{c} \in \mathcal{C}$) denote the maximum subset of $\mathbf{c^{(S)}}$ such that it contains no function symbols which stand for fields in $\mathbf{c}$ other than those mentioned in the query, q. In other words, $\prod_{[q]} \mathbf{c^{(S)}}$ is the transitive closure of the function symbols from q inside $\mathbf{c^{(S)}}$. $\prod_{[q]} \mathbf{c^{(S)}}$ can also be interpreted as a projection of $\mathbf{c^{(S)}}$ respect to the function symbols of q. Then $\mathbf{c_1}$ is a subclass of $\mathbf{c_2}$ considering the query q, denoted $\mathbf{c_1} \prec_q \mathbf{c_2}$, if and only if

$$\prod\nolimits_{[q]} \mathbf{c_1^{(S)}} \vdash \prod\nolimits_{[q]} \mathbf{c_2^{(S)}} \qquad \wedge \qquad \mathbf{c_1}:_d \mathbf{c_2}.$$

3.7 Overloading, Overriding

The careful reader might have wondered how the axiomatic model introduced in the previous subsections supports *overloading* (operations with the same name but with different signatures). The answer is easy, somewhat trivial after knowing tons of object-oriented models and implementations. Operations with different number of arguments match different function symbols and so do the ones that have the same number of arguments but their type differs. Type constraints for operations, as for attributes, must be placed into the constraint field of the axiom-set.

Overriding means replacing the implementation of an operation. In our model there are two modeling parts that may represent methods. One of them is the **M** compo-

nent of classes, which is not used for reasoning. Therefore its elements can be freely superseded in subclasses.

User-defined predicates may require functions, too. The body of those functions has to reside in the **S** component of the classes because they are used for deduction. They are not overridable, however. This is not a restriction since if more constraints are to be added, they can be, but if a constraint (or its definition) is replaced, the subclass relationship may no longer hold (and we cannot speak about overriding). In any case one can specify differential inheritance or add the new constraints, definitions as appropriate.

3.8 Extending the Model

Association is ”the semantic relationship between two or more classifiers that specifies connections among their instances.”[14] Attributes of the axiomatic model can be viewed as incomplete connections, i.e. navigation is supported only in one direction. By means of the constraint field of the classes one can specify that ”backward attributes” must exist ensuring the existence of real *binary* connections. Non-binary associations always require a class for the connections in the model, then they can be split into several binary associations between the participants and the association class. The method is similar if a binary association has properties as well.

The logic language, $\mathcal{L}$, defined in Section 3.1 uses sets to represent multi-valued results of the functions. Sometimes not only the elements but their order is relevant: such structures are called *lists*. To manage lists, of course, new functions need to be added to $\mathcal{L}$. Many other extensions are realizable in this way, too.

4 Conclusion

Real DOOD’s would provide much benefit, however, current DOOD models exhibit deficiencies in logical modeling. We thus introduced a novel axiomatic approach to DOOD’s.

In the previous section we showed that this approach leads to an at least as expressive language as the UML. Furthermore, our model supports both mandatory and conditional elements using distinctive notions. We succeeded in reducing the information loss as much as possible in the presence of contradiction in inheritance. We also enriched the model with a type system. If a query is generalized, i.e. it specifies a type instead of an object class, the evaluation returns all instances that match the type definition irrespective of the class they belong to. Hence, our type definition supports these queries, too.

In our future endeavor we exhaustively verify the axiomatic model via the implementation of a complete system. Currently the class and the instance managers are implemented and our experiences are about to be submitted. The solid basis of the type system is already specified by [15]. We believe that the next generation of

DOOD's will, according to the original goal, fit all the needs of classical data modeling along with those of artificial intelligence.

References

[1] Carsí J. A., Janós J. H., and Ramos I. Implementation of an Interpreter of Transaction-Frame Logic. In Spanish. DSIC-II/35/97, Valencia University of Technology, October 1997.

[2] Serge Abiteboul and Stéphane Grumbach. COL: A Logic-Based Language for Complex Objects. In Joachim W. Schmidt, Stefano Ceri, and Michele Missikoff, editors, *Advances in Database Technology - EDBT'88, Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 14-18, 1988*, volume 303 of *Lecture Notes in Computer Science*, pages 271–293. Springer, 1988.

[3] S. Ceri and R. Manthey. Chimera: a model and language for active DOOD Systems, 1994.

[4] Stefano Ceri and Rainer Manthey. Consolidated Specification of Chimera (CM and CL). Technical report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy, 29 November 1993.

[5] Andrew Dinn, M. Howard Williams, and Norman W. Paton. ROCK & ROLL: A Deductive Object-Oriented Database with Active and Spatial Extensions. In Gray and Larson [8], page 582.

[6] Alvaro Adolfo Antunes Fernandes. *An Axiomatic Approach to Deductive Object-Oriented Databases*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, Scotland, UK, September 1995.

[7] Jürgen Frohn, Rainer Himmeröder, Paul-Thomas Kandzia, Georg Lausen, and Christian Schlepphorst. FLORID: A Prototype for F-Logic. In Gray and Larson [8], page 583.

[8] Alex Gray and Per-Åke Larson, editors. *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.* IEEE Computer Society, 1997.

[9] Paul-Th. Kandzia. Nonmonotonic Reasoning in Florid. In *4th International Workshop on Logic Programming and Nonmonotonic Reasoning, Dagstuhl Castle*, LNAI 1265, pages 399–409. Institut für Informatik, Albert–Ludwigs–Universität, Freiburg, Germany, 1997.

[10] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.

[11] M. Liu. Incorporating Methods and Encapsulation into Deductive Object-Oriented Database Languages, 1998.

[12] Mengchi Liu. Deductive database languages: problems and solutions. *ACM Computing Surveys*, 31(1):27–62, March 1999.

[13] David Maier. Logic for Objects. In *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington, USA, August 1986.

[14] **O**bject **M**anagement **G**roup. **U***nified* **M***odeling* **L***anguage Specification Version 1.3*, June 1999.

[15] Gábor Mihály Surányi. The $\lambda\&\complement$-calculus and its basic properties. Technical report, Budapest University of Technology and Economics, June 2004. Available on-line: `http://www.db.bme.hu/~surprof/calculus.pdf`.

[16] Guizhen Yang and Michael Kifer. FLORA: Implementing an Efficient DOOD System Using a Tabling Logic Engine. In *LNAI 1861*, pages 1078–1093. Department of Computer Science, SUNY at Stony Brook, NY, USA, 24–26 July 2000.