Deductive Object-Oriented Database Systems: A Survey

Pedro R. Falcone Sampaio and Norman W. Paton
Department of Computer Science, University of Manchester
Oxford Road, Manchester, M13 9PL, United Kingdom
Fax: +44 161 275 6236 Phone: +44 161 275 6124
E-mail: [sampaiop,norm]@cs.man.ac.uk

Abstract. Deductive object-oriented databases (DOODs) seek to combine the complementary benefits of the deductive and the object-oriented paradigms in the context of databases. Research into DOODs has now been taking place for almost ten years, and a significant number of designs and implementations have been developed. This paper categorises proposals for DOODs, based on the language design strategy pursued, and compares the resulting systems in terms of the support provided for specific deductive and object-oriented features. It is shown how comprehensive proposals have emerged from significantly different design strategies, and it is argued that research on DOODs is now quite mature, in that consensus is emerging on the capabilities that it is appropriate for DOODs to support.

1 Introduction

The observation that object-oriented and deductive database systems generally have complementary strengths and weaknesses gave rise to interest in the integration of the two paradigms in the late 80s. Initial proposals involved the extension of deductive languages with limited facilities for modelling complex values [AG88], and the view was sometimes expressed that DOODs could not be developed without detracting substantially from the benefits of at least one of the component paradigms [Ull91]. However, although there has never been a flood of proposals for DOODs, there has been a steady flow of new ideas, techniques and systems throughout the 90s, so that there are now more than ten working prototypes and one product [FLV96]. It seems suitable, therefore, to review the work carried out to date, to identify the approaches that have been taken, the achievements that have been made, and to assess the quality of the current state-of-the-art.

This survey provides an overview of DOOD systems and an evaluation of language proposals. The survey is organised according to the language design strategy adopted, which highlights how new proposals for DOODs build upon earlier work on deductive systems, programming languages, or logic theory. The systems that result from the different approaches are then compared according to two orthogonal language criteria: *declarative support:* language features that support the declarative programming style; and *object-orientation support:*

language features that support the object-oriented programming style. The following proposals are reviewed in this work: Chimera, ConceptBase, Coral++, ESQL2, FLORID, Gulog, Logidata+, Logres, Noodle, Orlog, Peplom^d, Quixote, ROCK & ROLL, ROL and Validity. The choice of systems was based on the availability to the authors of an implemented prototype and/or suitable papers describing the language. There is evidence that at least a partial prototype exists for every language in the survey except Gulog, and versions of ConceptBase, FLORID, ROCK & ROLL, ROL and Quixote are available in the Internet.

The paper is organised as follows. Section 2 presents an overview of the different design strategies for DOODs from the deductive language perspective. In section 3 the proposals derived using the strategies of section 2 are compared in terms of the functionalities they support. Section 4 presents some conclusions. Throughout the paper, some familiarity with deductive database concepts, as described in [CGT90], is assumed.

2 Design Strategies

In this section, DOODs are classified according to the language design strategy exploited during their development. There is more variety in the approaches taken to the development of DOODs than was the case with deductive relational databases (DRDBs), as in the latter case an agreed data model gave rise to the development of Datalog as a widely accepted starting point for the development of practical DRDBs.

The following strategies, which are illustrated in figure 1, have been adopted in the design of DOOD systems:

Language Extension: an existing deductive language model is extended with object-oriented features. This approach is familiar from work on DRDBs such as LDL [NT89] or CORAL [RSS92], in which the syntax and semantics of Datalog were extended incrementally with negation, set terms, built in predicates, etc. In the language extension approach to DOODs, Datalog is generally an ancestor of the DOOD language, which is extended to support identity, inheritance, etc. Note that this strategy does not imply the existence of an earlier deductive database implementation or syntactic conformity with existing languages.

Language Integration: a deductive language is integrated with an imperative programming language in the context of an object model or type system. In this strategy, the resulting system supports a range of standard object-oriented mechanisms for structuring both data and programs, while allowing different and complementary programming paradigms to be used for different tasks, or for different parts of the same task. The idea of integrating deductive and imperative language constructions for different parts of a task was pioneered in the Glue-Nail DRDB [DM93], and is now adapted for object-oriented databases. The success of this strategy depends on the seamlessness of the integration of the deductive language and the imperative language.

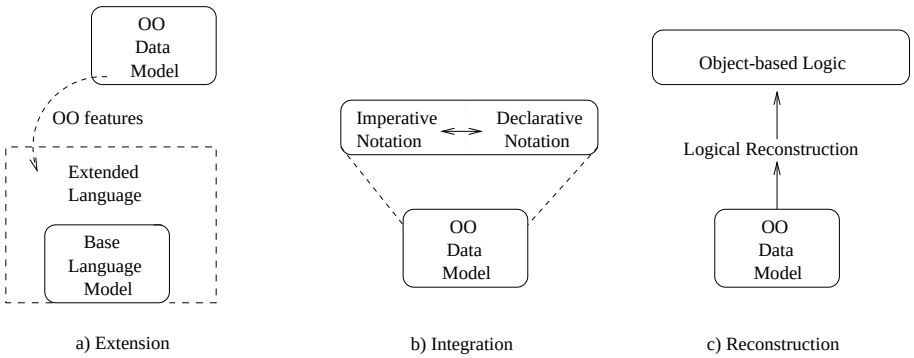

Fig. 1. Language design strategies for DOODs

Language Reconstruction: an object model is reconstructed following the rationale of Reiter [Rei84], creating a new logic language that includes object-oriented features. In this strategy, the goal is to develop an object logic that captures the essentials of the object-oriented paradigm and that can also be used as a deductive programming language in DOODs. This is a revolutionary approach in the sense that much of the associated implementation technology may have to be conceived from scratch. The driving force behind the language reconstruction strategy is an argument that language extensions fail to combine object-orientation and logic successfully [KLW95], by losing declarativeness through the introduction of extra-logical features, or by failing to capture all aspects of the object-oriented model. The major drawbacks of reconstruction are the difficulty of providing efficient implementations of all the features of the object logic and the lack of agreement on the target model to be formalised.

Within the strategies described, different approaches can be adopted, depending on the choice of the base language to be extended, the the nature of interaction between the integrated languages, or the object model reconstructed by the logic. The main approaches, examples of which are given in table 1, are:

1. Extend a Datalog-based language with object-oriented features.
2. Extend a Prolog-based language with object-oriented features.
3. Extend a SQL-based language with deduction and object-orientation.
4. Integrate a declarative rule language with an object-oriented programming language, allowing one of the languages to borrow programming constructs from the other (unilateral integration).
5. Integrate a declarative rule language with an object-oriented programming language, allowing each of the integrated languages to borrow programming constructs from the other (bilateral integration).

6. Reconstruct a data model from the database community.
7. Reconstruct a data model from the knowledge representation community.

Strategy	Approach	Proposals
Extension	1	ConceptBase[JGJS94] Logres[CCF+93] Logidata+[NST93] ROL[Liu96b]
	2	Quixote[YTM93]
	3	ESQL2[GV92b]
Integration	4	Chimera[CM94] Peplom^d[DR94] Coral++[SRSS93] Validity[FGVLV95]
	5	Noodle[MR93] ROCK & ROLL[BPF+94]
Reconstruction	6	Gulog[Dob96] Orlog[JL92]
	7	FLORID[FHKS96]

Table 1. Strategies and approaches to DOOD language design

The different strategies and approaches reflect general disagreement on how to combine object-orientation and deduction. An interesting issue is the expectations towards the delineation of an *Oblog*: the equivalent of Datalog for the object-oriented world [FPWB92]. Three lines of thinking seem to emerge from the literature: that *Oblog* is Datalog extended with object-oriented features, that *Oblog* is a reconstructed language, or that *Oblog* will never exist.

2.1 Language Extension Strategy

Language extension proposals aim to enhance the modelling power and programming style of declarative languages by supporting object-oriented features within the data models associated with such languages. The critical issues in this strategy relate to the choice of declarative language model that underlies the extension (logic programming, relational programming) and to the technique used to map features like objects, classes, object sharing, inheritance, methods and encapsulation into the declarative language.

Datalog and Prolog Extensions Extension-based DOODs have been derived incrementally from Datalog or Prolog by adding structural and behavioural features from the object-oriented model. Theoretical languages [CG88, AK89, BM92, ACM93] were proposed to provide a foundation to the strategy, with IQL [AK89] being the first theoretical language extension proposal to support the structural complexity of the main object model features (classes, identity, complex objects and inheritance). IQL had a major impact on many later DOOD languages.

There is also a substantial amount of work on concepts and techniques for extending logic programming with object-oriented features – for a comprehensive overview see [McC92, Mos94]. A brief summary of how the main object-oriented concepts can be mapped into the logic programming world is shown in Table 2.

Logic Programming	**Object-Orientation**
complex terms	complex values
object theories or predicate extensions	classes
OIDs in predicate extensions labels in object theories	object identity
ground instances of predicate extensions labelled object theory	objects
OID-based references label-based references	object sharing
unification through generalisations	inheritance

Table 2. Support for object-orientation within logic programming

Two alternatives to mapping structural object-oriented concepts into logic programming have been identified [CT93]: *predicate extensions* define an object as an identity-based ground instance of an extended predicate and *logical theories* define an object as a set of facts and rules defining properties of an object.

Predicate extensions

In implementations based on predicate extensions, identity is represented in the predicate by a special argument that stands for the object identifier (OID), providing an identity-based data model on top of a value-based system implementation. The creation and manipulation of OIDs is under the control of the system, which provides special methods for object creation (e.g. a *new* operator or OID invention). The management of *is-a* relationships can be done by adding clauses to each program, that enforce the *is-a* relationships defined over the corresponding schema. The following rules show the definition of system generated clauses added for managing the relationship C_1 *is-a* C_2 in Logres and Logidata+, where facts for the satisfaction of containment constraints are generated, propagating OIDs along *is-a* relationships, assuring that instances of the subclass C_1 are included in the extension of the superclass C_2.

$C_2(OID : x_0, A_1 : x_1, \ldots, A_k : x_k) \leftarrow C_1(OID : x_0, A1 : x_1, \ldots, A_k : x_k, \ldots, A_{k+h} : x_{k+h})$
$TYPE(C_2) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$ $TYPE(C_1) = (A_1 : \tau_1, \ldots, A_k : \tau_k, \ldots, A_{k+h} : \tau_k)$
and $x_i \in$ Attribute Names, $\tau_i \in$ Attribute Types.

Logical theories

Within the implementation based on logical theories, a set of facts and rules defines each object or class. Class hierarchies are represented by specialisation rules and OIDs can be supported by a user-defined label attached to the theory defining an object or by a built-in predicate *new* that generates the OIDs. The example in figure 2 specified in the syntax proposed in [BM92], shows two classes (stop, train_arrival) defined as theories, with declarative database updates expressed as a side-effect of the deductive process by applying the operators minus and plus to denote deletion and insertion, respectively.

```
stop:(NM,CD)
      { name(NM).
        code(CD).
        CN:change_name(N,NewN) <- -name(N), +name(NewN).
      }
train_arrival:(TC,T,PS,OC)
      {
        train_code(TC).
        time_at_stop(T).
        place_of_stop(PS) <- stop:code(PS).
        origin_code(OC) <- stop:code(OC).
      }

?- new(stop,euston,lon_e,Oid).
?- new(stop,manchester_picadilly,mcr_p,Oid).
?- new(train_arrival,t42,16:55,mcr_p,lon_e,Oid).
```

Fig. 2. Object theories

In the example, goals are used to create objects using the built-in *new* predicate. When the predicate receives as parameters the class name and the values of the properties of the object to be created, the predicate succeeds and an OID is generated for the object.

Example systems

ConceptBase is a multi-user DOOD adopting a client-server architecture that provides a CORBA compliant interface. ConceptBase supports a wide range of deductive and object-oriented features that are accompanied by graphical tools for browsing and manipulating the system. ConceptBase's language O-TELOS can be seen as an amalgamation of a declarative rule language and an object-oriented data model, where the object structure is mapped as facts, rules and integrity constraints within Datalog with negation.

Logres and Logidata+ provide a deductive language to query an object-relational data model. Logres introduces the notion of *application* as an abstraction unit for grouping rule definitions and goals relating to a specific task.

Transitions between database states in Logres are controlled by the execution of *applications* that can update the database as a side effect of the deductive process.

Quixote is a persistent constraint logic programming language with top-down evaluation extended with identity-based objects represented as labels in object theories. Methods define properties of objects, and inheritance is supported through subsumption constraints. Quixote supports a concept of module that provides encapsulation and a modular programming style, and also provides support to updates as a side effect of the query process.

ROL is a typed extension to Datalog that supports the widest range of deductive and object-oriented features within the surveyed prototypes (see tables 3 and 4). ROL provides a mechanism for representing both partial and complete information on sets and avoids procedural extensions.

The example in figure 3 in Logres illustrates the language extension strategy. In the example, the database describes a graph representing train and flight connections between stops. The flights and trains connect stops that can be airports or train stations. Classes `STOP`, `TRAIN-STOP` and `FLIGHT-STOP` are declared in the database schema and rules are provided to recursively compute paths among cities and the total duration of the travel. A set of intermediate nodes is computed to avoid cyclic paths of infinite duration.

```
DOMAINS SECTION
   NAME = STRING; CODE = INTEGER; TIME = INTEGER;
CLASSES SECTION
   STOP = (NM:NAME, CD:CODE);
   TRAIN-STOP = (ST:STOP, ORIGIN:STOP, DEST:STOP, T:TIME);
   FLIGHT-STOP = (ST:STOP, ORIGIN:STOP, DEST:STOP, T:TIME);
   TRAIN-STOP isa STOP;
   FLIGHT-STOP isa STOP;
ASSOCIATIONS SECTION
   PATH = (ORIGIN:STOP, DEST:STOP, INT:{S}, T:TIME);

path(origin:X,dest:Y,int:I,t:Z) <- train-stop(st:S,origin:X,dest:Y,t:Z),I=0.
path(origin:X,dest:Y,int:I,t:Z) <- flight-stop(st:S,origin:X,dest:Y,t:Z),I=0.
path(origin:X,dest:Y,int:I,t:Z) <- path(origin:X,dest:X1,int:I1,t:Z1),
                                   train-stop(st:S,origin:X1,dest:Y,t:Z2),
                                   Z=Z1+Z2,append(X1,I1,I),
                                   not member(Y,I1).
path(origin:X,dest:Y,int:I,t:Z) <- path(origin:X,dest:X1,int:I1,t:Z1),
                                   flight-stop(st:S,origin:X1,dest:Y,t:Z2),
                                   Z=Z1+Z2,append(X1,I1,I),
                                   not member(Y,I1).
```

Fig. 3. Example rules in Logres

Language extension proposals based on predicate extensions solve the problems of infinite set generation summarised in [Ull91] by supporting value-based as well as identity-based entities. In the Logres example, `path` is modelled as an association (unnormalised relation), and thus individual paths are not represented as objects with distinct identities.

SQL extensions ESQL2 is a language extension proposal that extends the relational model of SQL2 with deductive and object-oriented capabilities. Classes are implemented as relations extended with a system controlled attribute that stands for the OID. A rich set of generic abstract data types are supported, which can be specialised, providing a tool for building complex types based on sets, lists, bags and vector constructs. Updates are provided as an extension of the SQL2 syntax and deduction is implemented as an extension of the view mechanism of SQL2.

2.2 Language Integration Strategy

The rationale behind language integration is expressed by the following argument stated in [UZ90]:

> A declarative formulation cannot compete with the cogency and optimality of textbook algorithms for specific problems. These situations call for a mixed mode, and for the harmonious cooperation between the two modes at the language and system levels.

DOODs with integrated languages are designed to support the synergy of the different components: the logic language can be used to support those parts of an application that are suitable for declarative rule-based expression, and the imperative language can be used to support updates, input/output, and the expression of algorithms normally written in a procedural manner. Both languages operate in the context of an OO data model or type system.

A significant strength of language integration is the conventionality of its individual components. Most of the integration can be carried out without altering the defining principles of the component paradigms or introducing complex new concepts in order to achieve integration. Originality exhibited in the enactment of this strategy is found in the methodology adopted to achieve integration, rather than in the individual components that have been integrated.

The critical issues in this strategy are related to the flexibility of the integration approach (unilateral or bilateral), the choice of the source languages, and the selection of object model or type system underlying the integration.

Example systems

Chimera integrates sub-languages supporting active, imperative and declarative constructs in the context of the Chimera object model. Together with Validity and unlike other language integration proposals, it provides support for

declarative integrity constraints, and with ROCK & ROLL [DPWF96], is unusual among DOODs in supporting Event-Condition-Action rules. Chimera is classified as an unilateral language integration approach, as there are limitations on the bilateral embedding of the deductive and imperative sub-languages.

Coral++ integrates the Coral rule language with the C++ type system. Arbitrary C++ objects are allowed in database facts and C++ expressions are allowed in rules (unilateral integration). An advantage of Coral++ in relation to other integrated proposals is the access provided to the popular and powerful language C++, whereas other DOODs generally exploit their own data models and languages. However, Coral++ manifests impedance mismatch problems.

Noodle provides a declarative query language for the SWORD object-oriented database system [MR92]. Noodle is integrated with the O++ language of the Ode DBMS [AG89] following a bilateral approach to integration. Noodle defines a powerful declarative query language supporting higher order features, OID-based and value-based elements in collections, and collections as first-class objects. It is the only language integration proposal to support schema browsing.

Peplom^d integrates a deductive notation with the Peplom database programming language, which is superficially similar to C. The deductive language is a significantly extended Datalog.

In ROCK & ROLL the starting point for the development of the subsequently integrated languages is an object-oriented semantic data model. The deductive language ROLL and the imperative language ROCK are then derived from this model, and integrated with minimal impedance mismatches. Type inference in ROLL prevents runtime type errors occurring across the language interface, and later work has added active [DPWF96] and spatial [FDPW97] facilities.

Validity is the first DOOD to become a product. The system integrates declarative and imperative constructs in the DEL language. Validity is classified as an unilateral language integration approach as there are limitations on the bilateral embedding of the deductive and imperative sub-languages (methods can only be implemented by the imperative component).

To illustrate language integration ideas, the example in figure 4 shows the definition of a type *stop* in ROCK & ROLL, combining imperative object-oriented and declarative styles of programming— the type definition indicates that stop is an association of other adjacent stops (defining stops that are directly linked to a stop, denoted by the set brackets symbol) and that it has properties *name*, *code* and three public methods *print*, *adjacent* and *reachable*. The *print* method displays the properties of a stop and is implemented in the imperative language ROCK. The *adjacent* method indicates if a stop is directly linked to another and *reachable* indicates if a stop is reachable from another stop. Both *adjacent* and *reachable* are implemented in the ROLL language. All methods are statically type checked, can be inherited or overridden, and can be invoked from the same program or interactively. Methods starting with the prefix *get* are system generated and are used to retrieve properties and to access members of collections.

```
type stop:
  properties:code:integer,
             name:string;
             public: {stop};
  interface: ROCK: public: print();
             ROLL: public: adjacent(stop):bool, reachable(stop):bool;
end-type;

class stop
      public:
         print()
         begin
          write "name=", get_name()@self, ",code=",get_code()@self, nl;
         end
         adjacent(stop)
         begin
          adjacent(OtherStop)@ThisStop
                           :- OtherStop == get_member@ThisStop;
         end
         reachable(stop)
         begin
           reachable(OtherStop)@ThisStop
                           :- adjacent(IntermediateStop)@ThisStop,
                              reachable(OtherStop)@IntermediateStop;
         end
end-class
```

Fig. 4. Imperative and declarative methods in ROCK & ROLL

2.3 Language Reconstruction Strategy

The language reconstruction strategy is based on the formal description of data model concepts in a mathematical framework with the aim of formalising the notions underlying implemented systems. In the database field, logic foundations are useful for defining a declarative semantics for the data model, for underpinning query optimisation, for providing concise specifications of implemented systems, and for adding deductive capabilities.

The critical decision in this strategy relates to the nature of the object model to be reconstructed. The expressiveness, tractability and programming style of the logic will be affected by this decision.

Example systems

FLORID is a partial implementation of the features and syntax defined by F-logic [KLW95]. FLORID supports a powerful logic, departing from a frame-based knowledge representation scheme, adopting a style of programming that mixes intensional and extensional definitions and supporting higher order syntactic constructs. The notion of schema is formed by the definition of class signa-

tures and class hierarchies, and no distinction is made between data and schema declarations.

Gulog tries to provide a more database-oriented formal foundation than F-logic, by reconstructing a more conventional object-oriented data model and supporting only first-order syntactic constructs. Gulog distinguishes between schema and instance level constructs.

Orlog departs from an object-relational data model, and supports the notions of schema and integrity constraints. Schema definitions are separated from data declarations and higher order syntactic constructs are supported.

Another difference between reconstruction proposals stems from the way that ambiguously defined object-oriented features like multiple inheritance and overriding are dealt with: conflicts due to multiple inheritance in FLORID are solved by a non-deterministic choice of the inherited property (one of the definitions is inherited). Gulog prohibits programs with ambiguities due to multiple inheritance and Orlog does not address the issue.

Overriding is the redefinition of methods and attributes in subtypes. The combination of overriding and inheritance can cause non-monotonic behaviour. This can be noticed in the following example in FLORID:

```
person[believes_in*->god].
employee::person.
john:employee.
?- john[believes_in->X].
X/god
```

In the example, every `person` believes in `god`, `employee` is a subclass of `person` and `john` is an `employee`. With non-monotonic inheritance, if the fact `john[believes_in]-> ghosts]` is added to the database, the answer to the previous query is `X/ghosts`. Monotonic inheritance is inheritance without overriding. In the example, after the new clause is added with monotonic inheritance, both values for `X` could be derived. It is interesting to note that in this example, overriding occurs at the instance level.

Overriding can be static or dynamic. With static overriding, overriding definitions replace the definition in the parent regardless of the validity of the body of the overriding definition. In dynamic overriding, a definition is overridden only if the body of the overriding clause is true. FLORID and Gulog adopt dynamic overriding while Orlog adopts static overriding.

The distinction between objects and classes also differentiates the proposals. FLORID gives full object status to all elements in the language, while Orlog and Gulog distinguish between the two concepts. Giving full object status to all elements in the language allows FLORID to perform schema browsing operations (query schema elements like classes and attributes) and also to attach properties (e.g. attributes) to schema elements.

Figure 5 shows a program in FLORID that illustrates the mix of data declarations and schema definitions, and gives an overall idea of the F-logic programming style:

```
/* class signatures ------------------------------------------*/
/* 1 */ stop[name=>string].
/* 2 */ stop[departure@(train)=>>time;arrival@(train)=>>time].
/* 3 */ stop[adjacent=>>stop;reachable=>>stop].
/* 4 */ train[destiny=>stop;origin=>stop].
/* subclass relationship -------------------------------------*/
/* 5 */ cargo_train::train.
/* 6 */ passenger_train::train.
/* data declarations -----------------------------------------*/
/* 7 */ london_euston:string.
/* 8 */ stockport:string.
/* 9 */ mcr_piccadilly:string.
/*10 */ mcr_victoria:string.
/*11 */ lon_e:stop[name->london_euston].
/*12 */ stc_1:stop[name->stockport;adjacent->>{lon_e}].
/*13 */ mcr_v:stop[name->mcr_victoria].
/*14 */ mcr_p:stop[name->mcr_piccadilly;adjacent->>{mcr_v,stc_1}].
/* Rules -----------------------------------------------------*/
/*15 */ X[reachable->>{Y}] :- X:stop[adjacent->>Y].
/*16 */ X[reachable->>{Y}] :- X:stop,X[adjacent->>Z],Z[reachable->>Y].
```

Fig. 5. Train program in FLORID

Lines 1-4 define signatures for single-valued methods(name,destiny,origin) and multi-valued methods (adjacent,departure, arrival, reachable) applicable to instances of the classes *stop* and *train* respectively. Lines 5 and 6 define the subclass relationship between classes (*cargo_train, passenger_train*) and *train*. Lines 7-14 declare instances and their properties, and lines 15 and 16 define deductive rules.

3 Comparison Criteria

In this section, general criteria are defined and used to compare the different proposals. Two main groups of criteria are used: *declarative support* and *object-orientation support*.

3.1 Declarative Support

This subsection outlines a range of declarative language features that can be used to distinguish between DOOD proposals, whatever strategy has been adopted in the design of the DOOD. It can be taken that all proposals support standard deductive features like recursive rules and negation.

Declarative Database Updates: the capability to update the database as a side-effect of the deductive process using the rule language, instead of separating query and update functions as is traditionally done in database

languages. This feature is implemented in DRDBs like LDL [NT89], and is supported by extending the set of predicates that can occur in a rule, allowing predicates of the form $\alpha p(t_1, t_2, \ldots, t_n)$. The intuitive meaning of $+p(t_1, t_2, \ldots, t_n)$ is to insert the tuple or object $p(t_1, t_2, \ldots, t_n)$ into the corresponding relation or class p and the intuitive meaning of $-p(t_1, t_2, \ldots, t_n)$ is to delete the tuple or object $p(t_1, t_2, \ldots, t_n)$ from the corresponding relation or class p.

Schema Browsing: the capability to express queries on metadata contained in the schema without explicit reference to the database catalog (e.g 'Find all classes or subclasses of the *Vehicle* class that have an attribute engine_capacity'). This feature is supported in deductive languages by allowing variables to range over schema elements.

Higher Order Syntax: the capability to express rules that are not syntactically first-order logic (e.g. to quantify over predicates).

Integrity Constraints: the capability to directly support logical rules that define conditions that cannot be violated by the data, in addition to defining intensional predicates.

Formal Semantics: this criterion relates to the formal definition of the semantics of the deductive language. The semantics can be defined directly (D) or by an indirect approach of translating to a language that has a well-defined semantics (I).

The declarative support provided by the DOOD proposals from table 1 is summarised in table 3.

3.2 Object-Orientation Support

This subsection outlines a range of object-oriented features that can be used to distinguish between DOOD proposals, whatever strategy has been adopted in the design of the DOOD. It can be taken that all proposals support standard object-oriented features such as some notion of identity, some form of inheritance (in fact, all the systems reviewed here support multiple inheritance), and some form of overriding.

Object-Relational: the capability to support relations in addition to classes as elements in the data model.

Modularity: the capability to organise a complex program in terms of abstraction units. Units can be abstract data types implemented as classes that define the structure and behaviour of objects, or modules that group rules relating to a specific subpart of a task.

Encapsulation: the capability to control access to software components (rules, data, methods) defined in the abstraction units.

DOOD	Criteria				
	Declarative Database Updates	Schema Browsing	Higher Order Syntax	Integrity Constraints	Formal Semantics
Chimera	No	No	No	Yes	No
ConceptBase	No	Yes	No	Yes	[JGJS94] (I)
Coral++	No	No	No	No	No
ESQL2	No	No	No	Yes	[GV92a] (I)
FLORID	No	Yes	Yes	No	[KLW95] (D)
Gulog	No	No	No	No	[DT95] (D)
Logidata+	No	No	No	Yes	[ACMT93] (D)
Logres	Yes	No	No	Yes	[CCCR+90] (D)
Noodle	No	Yes	Yes	No	No
Orlog	No	No	Yes	Yes	[JL92] (D)
Peplom[d]	No	No	No	No	No
Quixote	Yes	No	No	No	No
ROCK & ROLL	No	No	No	No	[FBPHW97] (D)
ROL	No	Yes	Yes	Yes	[Liu96a] (D)
Validity	No	No	No	Yes	No

Table 3. Declarative support criteria

Collections as 1st Class Citizens: the capability of attaching properties to collections (e.g., set attributes or class attributes) and querying the properties within the rule language.

Method Implementation: the capability to implement methods declaratively and/or procedurally. The implementation can be done using an integrated imperative language (language integration), or through functions or adorned predicates in the rule language.

The support for object-orientation provided by the DOOD proposals from table 1 is summarised in table 3.

3.3 Summary of Comparison

The comparison provided by this section has to be interpreted with some care – it is not necessarily the case that systems that have more features than others in tables 4 and 3 are the best. In many cases, it depends upon the emphasis taken by the developers. For example, declarative database updates are considered

DOOD	Criteria				
	Object-Relational	Modularity	Encapsulation	Collections as 1st Class	Method Implement.
Chimera	No	Class	Yes	Yes	Dec + Imp
ConceptBase	No	Class	Yes	Yes	Dec
Coral++	Yes	Class + Module	Yes	No	Imp
ESQL2	Yes	Class	Yes	No	Imp
FLORID	No	Class	No	Yes	Dec
Gulog	Yes	Class	No	No	Dec
Logidata+	Yes	Class	No	No	Dec
Logres	Yes	Class + Module	Yes	No	Dec
Noodle	Yes	Class	Yes	Yes	Dec + Imp
Orlog	Yes	Class	Yes	Yes	Dec
Peplom[d]	No	Class + Module	Yes	No	Dec + Imp
Quixote	No	Module	Yes	No	Dec
ROCK & ROLL	No	Class + Module	Yes	No	Dec + Imp
ROL	Yes	Class	No	Yes	Dec
Validity	No	Class	No	No	Imp

Table 4. Object-orientation support criteria

beneficial by many as they allow updates as well as querying to take place within the logic language. However, supporting updates often complicates the semantics of the logic language, and proponents of the language integration approach claim that procedural issues such as updates and I/O should be addressed outside the logic language.

4 Conclusions and Future Work

In spite of the perceived difficulties involved in combining the deductive and object-oriented paradigms for use in database systems, considerable progress has been made in this area during the last ten years. It was at first felt by many that the paradigms were incompatible, or that combining them would lead to unacceptable compromises, but in fact a number of proposals have been made that support both comprehensive deductive inference and rich object-oriented modelling facilities.

Although different strategies have been adopted to the development of DOOD systems, as outlined in section 2, much of the challenge for researchers has in-

volved reconciling a tension between: (1) the development of powerful but complex and inefficient languages; and (2) the development of efficient but semantically inexpressive languages. It might have been anticipated that research would move incrementally from less expressive systems towards more expressive systems, but this is not how things have worked out in practice. Early proposals fell at opposite ends of this spectrum. F-logic, which was first proposed in 1989, can be seen with hindsight as rather too expressive to yield practical implementations, whereas some of the other early proposals, such as COL, can be seen as not expressive enough. Various research projects have proceeded by following the reconstruction strategy of F-logic, but with rather less expressive models and languages, and others have sought to extend systems such as COL with additional modelling and programming features. This has led to something of a consensus among the more recent systems, such as ROL and Orlog, as to what facilities it is practical to support.

While DOOD researchers working on language extensions or language reconstruction were focusing on the expressiveness of logic languages for objects, language integration researchers were playing a different game altogether. The power of the logic language is not the principal concern in a system with integrated languages, as the effectiveness of the system derives from the capabilities provided by all its components and from the ease with which they can be used together. This has led some systems, such as ROCK & ROLL, to provide a logic language that is much less powerful (complex) than would be considered appropriate in a stand-along language, on the grounds that if a task cannot be carried out by the logic language, it can always be addressed by the closely associated imperative facilities. Thus, while three strategies have been identified for the development of DOOD systems, there are really only two competing camps – one based on single language systems and the other based on multiple (two) language systems.

How does the future look for DOOD systems? The basic technology for developing DOOD systems is now reasonably well understood, although there are not yet many systems that have proved their worth on large scale applications.

Some applications that have been reported in the literature relate to metadata management [JS93] (ConceptBase), GIS [PAHW96] (ROCK & ROLL), natural language processing [TTYY93] (Quixote) and data mining [FGVLV95] (Validity).

The current risk is probably that DOOD systems will go the way of their deductive relational predecessors, by providing facilities that may be desirable in packages that are not attractive to consumers. The near total commercial failure of DRDBs can be ascribed, at least in part, to the fact that such systems were poorly integrated with existing database systems or programming languages. A number of proposals for DOODs successfully combine the complementary features of their component paradigms, but if these benefits are to be widely exploited they will probably have to be made more readily available to existing users of object-relational or object-oriented database systems.

Acknowledgements: The first author is sponsored by Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Brazil) – Grant 200372/96-3.

References

[ACM93] P. Atzeni, L. Cabibbo, and G. Mecca. Isalog(¬): a deductive language with negation for complex-object databases with hierarchies. In *Proc. of the 3rd Intl. Conference on Deductive and Object-Oriented Databases*, 1993.

[ACMT93] P. Atzeni, L. Cabbibo, G. Mecca, and L. Tanca. The logidata+ language and semantics. In *LOGIDATA+: Deductive Databases with Complex Objects*, number 701 in LNCS. Springer-Verlag, 1993.

[AG88] 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, International Conference on Extending Database Technology*, LNCS 303, pages 271–293, Venice, Italy, March 1988. Springer-Verlag.

[AG89] R. Agrawal and N. Gehani. Ode (object database and environment):the language and the data model. In *Proc. of ACM Sigmod Intl. Conference on Management of Data*, 1989.

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proc. of The ACM SIGMOD Intl. Conference on Management of Data*, 1989.

[BM92] E. Bertino and M. Montesi. Towards a logical object-oriented programming language for databases. In *Proc. Intl. Conference on Extending Database Technology EDBT*, number 580 in LNCS, pages 168–183, 1992.

[BPF+94] M. L. Barja, N. W. Paton, A. A. Fernandes, M. Howard Williams, and Andrew Dinn. An effective deductive object-oriented database through language integration. In *Proc. of the 20th VLDB Conference*, pages 463–474, 1994.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, 1990.

[CCF+93] F. Cacace, S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. An overview of the logres system. In I. S. Mumick, editor, *Proc. of the Workshop on Combining Declarative and Object-Oriented Databases*, 1993.

[CG88] Q. Chen and G. Gardarin. An implementation model for reasoning with complex objects. In *ACM-SIGMOD International Conference on Management of Data*, 1988.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

[CM94] S. Ceri and R. Manthey. Chimera: A model and language for active dood systems. In *Proc. of the East/West Database Workshop*, pages 3–16, 1994.

[CT93] S. Ceri and L. Tanca. Bridging objects with logical rules: Towards object-oriented deductive databases. In *Logidata+: Deductive Databases with Complex Objects (LNCS 701).* Springer-Verlag, 1993.

[DM93] M. A. Derr and S. Morishita. Design and implementation of the glue-nail database system. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, 1993.

[Dob96] G. Dobbie. *Foundations of Deductive Object-Oriented Database Systems.* PhD thesis, Victoria University Of Wellington, Comp. Science Dept., 1996.

[DPWF96] A. Dinn, N.W. Paton, M.H. Williams, and A.A.A. Fernandes. An Active Rule Language for ROCK & ROLL. In R. Morrison and J. Kennedy, editors, *Proc. 14th British National Conference on Databases*, pages 36–55. Springer-Verlag, 1996.

[DR94] P. Dechamboux and C. Roncancio. Integrating deductive capabilities into an object-oriented database programming language. In *Proc. of 10 Journées Bases de Données Avancées*, 1994.

[DT95] G. Dobbie and R. Topor. On the declarative and procedural semantics of deductive object-oriented systems. *Journal of Intelligent Information Systems*, 4:193–219, 1995.

[FBPHW97] A. A. A. Fernandes, M. L. Barja, N. W. Paton, and M. Howard-Williams. The formalisation of rock & roll: A deductive object-oriented database system. *to appear in Information and Software Technology*, 1997.

[FDPW97] A.A.A. Fernandes, A. Dinn, N.W. Paton, and M.H. Williams. Extending a Deductive Object-Oriented Database System with Spatial Data Handling Facilities. 1997. submitted for publication.

[FGVLV95] O. Friesen, G. Gauthier-Villars, A. Lefebvre, and L. Vieille. Applications of deductive object-oriented databases using del. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, chapter 1, pages 1–22. Kluwer Academic Publishers, 1995.

[FHKS96] J. Frohn, R. Himmeroder, P. Kandzia, and C. Schlepphorst. *How to Write F-Logic Programs in FLORID: A Tutorial for the Database Language F-Logic.* Institut Fur Informatik, Universitat Freiburg, Germany, version 1.0 edition, September 1996.

[FLV96] O. Friesen, A. Lefebvre, and L. Vieille. VALIDITY: Applications of a DOOD System. In *Proc. EDBT*, pages 131–134. Springer-Verlag, 1996.

[FPWB92] A. A. Fernandes, N. W. Paton, M. H. Williams, and A. Bowles. Approaches to deductive object-oriented databases. *Information Systems*, 34(12):787–803, 1992.

[GV92a] G. Gardarin and P. Valduriez. Esql: An object-oriented sql with f-logic semantics. In *Proc. Intl. Conference on Data Engineering*, 1992.

[GV92b] G. Gardarin and P. Valduriez. Esql2 - extending sql2 to support object-oriented and deductive databases. Technical report, INRIA, 1992.

[JGJS94] M. Jarke, R. Gallerrsdorfer, M. Jeusfeld, and M. Staudt. Conceptbase - a deductive object base for meta data management. *Journal of Intelligent Information Systems*, 3:167–192, 1994.

[JL92] H. M. Jamil and L. V. S. Lakshmanan. Orlog: A logic for semantic object-oriented models. In *Proc. of the ACM Conference in Knowledge Management - CIKM*, 1992.

[JS93] M. Jarke and M. Staudt. An application perspective to deductive object bases. In *Workshop on Combining Declarative and Object-Oriented Databases*, 1993.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, May 1995.

[Liu96a] M. Liu. Rol: A deductive object base language. *Information Systems*, 21(5):431–457, 1996.

[Liu96b] M. Liu. The rol deductive object base language. In *Proc. of 7th Intl. Workshop on Databases and Expert Systems Applications (DEXA)*. IEEE-CS Press, 1996.

[McC92] F. G. McCabe. *Logic and Objects*. Prentice-Hall International, 1992.

[Mos94] C. Moss. *Prolog++ The Power of Object-Oriented and Logic Programming*. Addison-Wesley, 1994.

[MR92] I. S. Mumick and K. A. Ross. Sword: A declarative object-oriented database architecture. Technical report, AT&T Bell Labs., 1992.

[MR93] I. S. Mumick and K. A. Ross. Noodle: A language for declarative querying in an object-oriented database. In *Proc. of the Third Intl. Conference on Deductive and Object-Oriented Databases*, volume 760 of *LNCS*, pages 360–378. Springer-Verlag, 1993.

[NST93] U. Nanni, S. Salza, and M. Terranova. The logidata+ prototype system. In *Logidata+: Deductive Databases and Complex Objects*, number 701 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[NT89] S.A. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, Rockville, MD, 1989.

[PAHW96] N. Paton, A. Abdelmoty, and M. Howard-Williams. Programming spatial databases: A deductive object-oriented approach. In Taylor & Francis, editor, *Innovations in GIS 3*, 1996.

[Rei84] R. Reiter. Towards a logical reconstruction of relational database theory. In M. L. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*. Springer-Verlag, 1984.

[RSS92] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL-Control, Relations and Logic. In *Proc. of the 18th Intl. Conference on Very Large Databases*, pages 239–250. Morgan Kaufman, 1992.

[SRSS93] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proc. of the 19th VLDB Conference, Dublin, Ireland*, 1993.

[TTYY93] S. Tojo, H. Tsuda, H. Yasukawa, and K. Yokota. Quixote as a tool for natural language processing. Technical Report TM-1282, ICOT Research Center - Japan, 1993.

[Ull91] J. Ullman. A comparision between deductive and object-oriented database systems. In *Proceedings of the 2nd Intl. Conference on Deductive and Object-Oriented Databases*, 1991.

[UZ90] J. Ullman and C. Zaniolo. Deductive databases: Achievements and future directions. *ACM - SIGMOD Records*, 19(4):75–82, December 1990.

[YTM93] K. Yokota, H. Tsuda, and Y. Morita. Specific features of a deductive object-oriented database language quixote. Technical report, Institute for New Generation Computer Technology (ICOT), 1993.

This article was processed using the LaTeX macro package with LLNCS style