INFORMATICA, 1998, Vol. 9, No. 1, 107–118
© 1998 *Institute of Mathematics and Informatics, Vilnius*

O!-LOLA – Extending the Deductive Database System LOLA by Object-Oriented Logic Programming

Günther SPECHT
Technische Universität München, Department of Computer Science
Orleansstr. 34, D-81667 München, Germany
e-mail: specht@informatik.tu-muenchen.de

Received: January 1998

Abstract. This paper presents the declarative extension of the deductive database system LOLA to the object-oriented deductive database system O!-LOLA. The model used for O!-LOLA is "objects as theories", extended by state evolution. O!-LOLA combines logic programming and OO programming in two different ways: First, methods are implemented as logic programs. These methods can be inherited, encapsulated and overloaded. Second, logic programs can be defined over classes, meta-classes, instances, attributes and values. Dynamic updates of attributes of objects and dynamic instantiations of classes are supported.

O!-LOLA is implemented as a preprocessor. O!-LOLA programs are transformed into LOLA rules and facts, which are evaluated set-oriented and bottom-up, using fixpoint semantics. Some object-oriented features concerning dynamic aspects are handled via built-in predicates in LOLA.

We describe the applied theory, the system and the preprocessor, including an example of how methods are translated and we discuss dynamic updates of objects in O!-LOLA.

The benefits of our system in contrast to others are: a single integrated language, clear semantics and a set-oriented evaluation. O!-LOLA uses fixpoint semantics (not any procedural semantics like other systems) and still evaluates set-oriented (and not in a mixed manner like other systems). Thus, we can fully use all optimization techniques developed for deductive databases and gain a very efficient system.

Key words: object-oriented logic programming, objects as theories, dynamic updates.

1. Introduction

Within the last years the two main areas of research in extending database programming languages have been deductive database languages and object-oriented database languages. Logic (or deductive) databases augment the relational model by Herbrand terms instead of flat attributes and arbitrary recursive views (or rules), whereas object oriented databases enhance the relational model by complex objects, classes, abstract data types, inheritance, methods, and encapsulation. Approaches to combine these two paradigms are heavily discussed today.

Besides various language proposals (e.g., Abiteboul, 1989; Kifer and Wu, 1989; Cacace *et al.*, 1990; Bertino and Montesi, 1992; Atzeni, 1993; Barja *et al.*, 1994) several

systems have been developed as prototypes or are on the way to combine these two extensions, each including a different aspect or combination idea. Some extend deductive databases with C++ as object definition language like CORAL++ (Srivastava *et al.*, 1993). Others add rule systems to object-oriented databases, such as Peplom^d (Dechamboux and Roncancio, 1994) or Noodle (Mumick and Ross, 1993). Still others were developed from scratch like Rock&Roll (Barja *et al.*, 1994) or Logidata++ (Atzeni, 1993).

A drawback of most of the systems is, that the user has to deal with two completely different programming languages within one system, a declarative rule-based language for the retrieval and a procedural language (mostly C++) for the definition of objects. Our system O!-LOLA is an integrated extension of the deductive programming language LOLA. O!-LOLA includes classes, instances, attributes, methods, (multiple) inheritance, encapsulation etc. and rules, all in one language. The integration of object-orientation and rule based systems comes into effect at least at two different points:

- Methods can be specified declaratively as logic programs. If they should cause side effects, such as updates of attributes or creation and deletion of new instantiations, built-in predicates can be called. These logic programs, implementing methods, are encapsulated and can be inherited and overridden.
- In addition it is possible to define logical rules over meta-classes, classes, instances, attributes and properties of objects.

The underlaying model of O!-LOLA is "objects as logical theories" extended by state evolution as McCabe (McCabe, 1992) defined it for the Prolog based system L&O and Bertino/Montesi (Bertino and Montesi, 1992) specified it as a programming language for databases.

Using a preprocessor, O!-LOLA is completely transformed into the deductive query language LOLA, which is enriched by some additional built-in predicates to the dynamic aspects of O!-LOLA. Thus, our approach is a bit related to some Prolog based OO-systems like L&O (McCabe, 1992) or OL(P) (Fromherz, 1993), which are also implemented as preprocessor, but are non-persistent and do not work set-oriented.

This preprocessor technique has several advantages: The extension can be done without any changes in the LOLA kernel. The deductive functionality of LOLA is fully available in O!-LOLA. The semantics can be defined as an extended fixpoint semantics. Thus, it is still declarative. The huge amount of optimization techniques developed for deductive database systems can be reused to gain a highly efficient deductive and object-oriented database system.

O!-LOLA offers a fully object-oriented functionality including classes, instances, methods, inheritance, encapsulation, and overriding. But some of these items cannot be implemented as easily as in the Prolog systems mentioned above, since the target program has to fulfill the well-known restrictions of deductive databases. These are: 1) range restriction[1], 2) safe negation, 3) local stratification and 4) some restrictions with regard

[1] Of course, Magic-Set Transformation can propagate bindings from the query to rules and facts, and only the transformed program has to be range restricted. But the restriction still holds in unresolvable cases. It can be completely omitted only if the target system is based on a unificational relational algebra.

to updates of facts within subgoals. The last item includes some still remaining problems and is an ongoing research activity among the deductive database community (for more details refer Section 5).

Since deductive databases use a fixpoint semantics, the semantics of O!-LOLA can be defined by an extended fixpoint semantics as well. Consequently multiple inheritance (even of methods) is done by a set-oriented evaluation of all inherited methods or values. This behaviour can be controlled by explicitly defining the inheritance-path or by overriding.

This paper is organized as follows: Section 2 defines objects as logical theories. Section 3 shows an example of an O!-LOLA program. Section 4 explains the preprocessor, the OO-kernel and the target system of the translation. The transformation of a method is shown in detail. Section 5 discusses the problem of dynamic updates. In Section 6 further related works are mentioned, although most of them appear already interleaved in the text. Finally the conclusion sums up the main results.

2. Objects as Theories

2.1. *Definitions*

In most cases there are two different techniques of incorporating objects and OO-techniques into logic programming. One can be characterized as "objects as terms", in which objects, often represented by their OID, appear at term positions in rules. Another, rather different approach is the "object as theory" approach (McCabe, 1992; Bertino and Montesi, 1992), in which an object is characterized by a set of clauses defining the properties, i.e., attributes and methods, of an object. Together they are the *theory* of this object. The structure of such a theory can be represented as

object_name : { $axiom_1$.
...
$axiom_n$.
}

An *axiom* can either be a rule, notated `head :- body`, or a fact or an attribute assertion like `attribute := value`, where the value may be a term structure or an evaluable (arithmetic) expression.

Rules and facts (since facts are just rules without a body) represent the methods of the object. Theoretically there is no inherent need to distinguish between attributes and facts, because it does not make any difference whether to write `attr := x` or `attr(x)`. The semantics introduced in O!-LOLA is the following: attributes defined as facts may have more than one value, whereas attributes defined by attribute assertion have at most one value.[2]

A *theory* $T = \{A_1, \ldots, A_n\}$ is a set of axioms, characterizing one or more objects. An object o satisfies a theory T if it satisfies all axioms, i.e., if $A_1(o) \wedge \ldots \wedge A_n(o) = true$.

[2] As an example think of a person having more than one address at a time, but exactly one birthday.

From this point of view, an object is the sum of all axioms known by it. Objects may be classes or instances. Each object defines its own theory and deductive object programming is the simultaneous work with different theories in the system (McCabe, 1992).

The call of a method (or an attribute) of an object is called message. Thus a message is a query, sent to a theory. Since different theories may include different implementations of the same method, the object name (= theory name) to which the message should be sent is put in front of the message. Example: `:- john:wife(X), john:weddingday(D).`

Of course, different messages on different theories (and meta-theories) can be combined for deducing new results. Even the class name to which a method should be sent needs not to be known statically at programming time. Example: `seabird(X) :- classes:isa(X,bird), X:can(swim).` Here `classes` is a meta-class containing the method `isa`, which computes the isa-hierarchy. The class `X` to which the method `can` with parameter `swim` is sent, is not known before runtime.

2.2. *Dynamic Aspects*

Objects, modeled by theories, are *static*. Dealing with dynamic aspects, they can just show snapshots. But updates are important for OO systems. We can distinguish updates in

- changes of the internal state of an object: Since the state of an object is represented in its attributes, this implies updates of the attributes.
- changes of the behaviour of an object: This implies updates of the methods, i.e., the rules, corresponding to an object.

While the first can be done dynamically in O!-LOLA (it is just an update of a specialized base relation), the latter implies a recompilation of this theory. Traditionally, all database languages strictly separate data manipulation language (DML) and data retrieval language (DRL). Deductive database languages are DRLs. Although some have been extended by update statements, such as LDL and RDL, there are still a lot of restrictions on updates.

O!-LOLA can handle dynamic changes of the internal state of an object and dynamic creations and deletions of instances. Thus, our model is "*objects as theories with dynamic state evolution*". Updates of methods in classes or creation of new classes need a (possibly incremental) recompilation. But from a theoretical point of view this means restarting with a new set of theories (for more details refer to Section 5).

3. An Example of an O!-LOLA Program

Program 1 shows a first simple O!-LOLA program for managing applications in a company. Classes for persons, applicants, and employees are defined.

O!-LOLA supports encapsulation of attributes and methods, which can be declared private or public. Attributes and methods can be overloaded in subclasses at two levels: by

```
class person {
      public  attribute address.
      public  method knows, health.
      private method illnesses.

      health(ok) :- $not ($self:illnesses(_)).


class applicant {
      public method qualified, applicable.

      qualified(programmer) :- $self:knows('C++'),
                               $self:knows('O!-LOLA').
      qualified(salesmen) :- $self:knows(marketing).

      applicable(Pos) :- $self:health(ok), $self:qualified(Pos).


applicant isa person.


class employee {
      private attribute salary.
      public  method  head_of_department, superior.

      superior(X) :- $self:head_of_department(X).
      superior(Z) :- $self:superior(Y), Y:head_of_department(Z).

employee isa person.


john instance_of applicant.
     john:address := residence(state('Germany'),city('Munich'),
                               street('Mainstreet', 12)).
     john:knows('C++').
     john:knows('O!-LOLA').


invitable(Pos, X, Addr) :- classes:isa(X,applicant),
                           X:applicable(Pos),
                           X:get_attribute(address, Addr).
```

Fig. 1. Program 1: O!-LOLA example for applications.

signature declaration and by code implementation. In our example, `illness` is private (encapsulated) and only the state of health is visible for the company which wants to hire somebody.

Methods are defined as rules that can be inherited. But rules over classes are also available, like `invitable`. The query `:- invitable(programmer, X, A).` is answered by the set of invitable programmers.

Additional to the user-defined methods each class has predefined ones, such as `<class>:get_attribute(<attribute>, <value>)`, `set_attribute`, `has_attribute`, `has_method`, etc.

A predefined meta-class called `classes` provides often needed public methods, such as `classes:isa(<class1>,<class2>)` (which can be used for traversing the class hierarchy), `create_new_instance(<class> <instance_name>)` and `destroy_instance(<instance_name>)` (which are dynamic), `instance_of (<class>, <instance>)`, etc.

The definition of the method "superior of an employee" is worth looking at. Since superior hierarchy is a recursive problem, this method has to be defined recursively, which can be easily done using the deductive rule system. This method extends common features. Although class Y, the receiver of the next `head_of_department` call, cannot be bound statically, it is efficiently computable. Internally this query is evaluated using seminaive iteration optionally combined with deductive database optimizations such as pushing selections, Magic-Set Transformation, etc.

An example for dynamic changes in the system may be "hire John":

```
:- classes:move_instance(john, applicant, employee),
   john:set_attribute(salary, 6000).
```

If additionally John knows Prolog his salary will rise to 7000:

```
:- john:knows('Prolog'),
   john:set_attribute(salary, 7000).
```

4. Translating O!-LOLA into LOLA

A preprocessor translates O!-LOLA programs into LOLA programs. The target system consists of two parts, the kernel and the application dependent LOLA code:

Kernel: The kernel includes all functions that are independent from the O!-LOLA source program. It consists of two parts: On one hand, there is an application independent OO-kernel implemented in LOLA. It controls object hierarchies, inheritances, encapsulations, overriding etc. On the other hand, a small number of built-in predicates for the dynamic parts of O!-LOLA exists. Essentially these predicates implement access methods for internally used efficient data structures for classes, class schemes, attributes, instances etc. These functions are implemented in Lisp, LOLA's host language.

Application dependent LOLA code: These rules are produced by the O!-LOLA compiler from the O!-LOLA source code. Even each method (already defined as a logic program) is expanded by additional rules checking accessability within inheritance, encapsulation and overriding of the method. Most of these generated rules call kernel-predicates.

The rule system of the kernel is precompiled and preoptimized, so that it becomes highly efficient.

On the whole, the kernel consists of about 40 precompiled LOLA rules and 25 built-in predicates. Kernel predicates are prefixed with #. As an example the very simple rules for the isa-hierarchy look like

```
#isa(X, Y) :-  #instance_of(X, Y).
#isa(X, X) :-  #class(X).
#isa(X, Y) :-  #subclass(X, Y).
#isa(X, Y) :-  #isa(X, Z), #subclass(Z, Y).
```

More complex are kernel rules for methods and the actual access to them. Each method declaration is transformed into a fact of the form
`#has_method(<method_name>, <class_name>, #public | #private).`

The following (simplified) rules determine whether (or not) a called method is accessable and which one is taken if it has been overwritten. The semantics of the parameters of `#is_public` and of `#access_method` is defined as follows: the first term is the method name, the second term is the class for which this method is called, and the third term is a result parameter, returning the actual valid owner class of the specified method for this caller.

```
#is_public(Method, Class, Class) :-
      #has_method(Method, Class, #public).

#is_public(Method, Class, Owner) :-
      $not(#has_method(Method, Class, _)),
      #subclass(Superclass, Class),          % 1 Step in isa
      #is_public(Method, Superclass, Owner).

#access_method(Method, Owner, Owner) :-
                              #is_private(Method, Owner).
#access_method(Method, Caller, Owner) :-
                               #is_public(Method, Caller, Owner).
```

Now let us look at the translation of a user-defined method: The method `superior` in class `employee`

```
superior(X) :- $self:head_of_department(X).
superior(Z) :- $self:superior(Y), Y:head_of_department(Z).
```

is transformed into the following three rules

```
superior(Caller, X) :-
        #access_method(superior, Caller, Owner),
        superior_trafo(Owner, Caller, X).
```

The first subgoal #access_method tests whether a method "superior" is defined for the caller, and which class the owner of this method is. This subgoal controls encapsulation and overriding. Thus the second subgoal is always called instantiated with Owner and Caller. X is the obtained parameter from the original method. And superior_trafo is coded as:

```
superior_trafo(employee, Caller, X) :-
        head_of_department(employee, Caller, X).

superior_trafo(employee, Caller, Z) :-
        superior_trafo(employee, Caller, Y),
        head_of_department(employee, Y, Z).
```

A query like `:- john:superior(X).` is finally translated into:

```
:- superior(john, X).
```

Now, we got a brief idea of how the declarative part of the kernel looks like and how the preprocessor works. Of course, some aspects are a bit more complicated and we simplified them for didactic reasons. One of the more complicated cases is the integration and interaction of the dynamic parts of O!-LOLA.

5. Dynamic Updates in Objects

The procedural part of OO-systems is necessary for processing changes of the internal states of objects at runtime. Deductive databases, as all databases, distinguish strongly between update- and retrieval queries. This implies that, dealing with Horn logic, updates of facts or rules are not allowed in subqueries at runtime.

Bottom-up evaluation and several optimization techniques benefit from the advantage that there is neither a fixed rule order nor a fixed subgoal order. But subgoal-reordering (i.e. join-reordering) could cause a later evaluation of delete or insert predicates with unpredictable results. Another example is the Magic-Set Transformation which causes bound predicates in the SIP to be evaluated more than once. Again this might not result in the intended meaning. Thus, this problem is essentially coupled with the applied optimization and evaluation technique.

A lot of papers have discussed this topic within the past several years. Many different solutions have been proposed. Coral++ (Srivastava *et al.*, 1993) does not allow rules to create new objects and instances. Rock&Roll (Barja *et al.*, 1994) prohibits non local updates of objects within rules. LDL++ (Zaniolo *et al.*, 1993) only allows inserts and deletes of facts, introducing a procedural semantics of those rules. Even the non persistent OO-Prolog systems include restrictions on updates: OL(P) (Fromherz, 1993) forbids all dynamic updates in the definition of classes and L&O (McCabe, 1992) distinguishes between static and updatable program parts. Kramer *et al.* (1992) discuss updates of objects in rule based language in more detail.

We soften the restriction of forbidden updates in the DRL, so that so-called safe updates are allowed in subgoals and queries. Only definitions and updates of attributes and instances are allowed evolutionary. Updating methods or defining new classes need a recompilation of this theory and thus a restart in the deduction process.

It is important that the fixpoint that is reached is always the same. Single updates, i.e., updates corresponding to DDL or single DML statements of SQL, are safe if they occur

in a positive stratum and if all necessary variable bindings can be bound at runtime. Both are checked at compile-time by rule inspection.

If an updated value is referred within the same subquery, the subgoal order has to be fixed. Thus the occurrence of update predicates omits all subgoal reordering optimizations in O!-LOLA. Magic-Set Transformation, using a strict left to right SIP, can still be applied in several cases, since it fixes the subgoal order too. But multiple evaluation of SIP predicates implies that update predicates in SIP predicates have to be idempotent. Since LOLA works set-oriented and since sets include duplicate elimination, there are idempotent update operations, such as creating new attributes, deleting attributes, non-incremental value assertion on attributes, etc. But this detail does not need to be known by the end-users, it is controlled by the preprocessor and the kernel.

Summarizing, state evolution of objects and dynamic class instantiation are available in O!-LOLA. They are called by special built-in predicates with side effects. The system can optimize queries including updates only in a restricted way. Updates of methods need a recompilation.

6. Related Works

In addition to the already presented related systems and frameworks, we compare O!-LOLA to these systems now in a more summarizing form.

For "a not very much annotated bibliography on integrating Object-Oriented and Logic Programming" see (Alexiev, 1993).

Some object-oriented systems, like Peplom^d (Dechamboux and Roncancio, 1994) and Noodle (Mumick and Ross, 1993), were developed by extending an existing object-oriented system by deductive mechanisms, while others, like ROCK&ROLL (Barja *et al.*, 1994), were developed from scratch. O!-LOLA and CORAL++ (Srivastava *et al.*, 1993) are extensions of an existing deductive database system with object-oriented features.

But CORAL++ links CORAL to C++, using C++ as object definition language, and provides object access only by special built-ins, using an "object as term" like approach. O!-LOLA is an integrated extension of the deductive programming language LOLA, supporting an "object as theory" model.

In this aspect O!-LOLA is related to L&O (McCabe, 1992), which also uses the "object as theory" model. But O!-LOLA uses the deductive mechanisms of LOLA and evaluates, similar to CORAL++ and Noodle, bottom-up. This is a major difference to all PROLOG based object-oriented systems, like PROLOG++ (Moss, 1994), L&O and OL(P) (Fromherz, 1993), which evaluate top-down and "one tuple at a time".

Like Peplom^d and Coral++, O!-LOLA can hold persistent data.

Similar to the Prolog-based systems L&O and OL(P), O!-LOLA is implemented as a preprocessor, too. We adapted this proposal to deductive database based systems in order to gain their advantages for object-oriented logic programming.

7. Conclusion

We have presented O!-LOLA, which is an object-oriented deductive database system implemented as a preprocessor on top of LOLA. O!-LOLA uses an "objects as theories with state evolution" model. This includes the dynamic creation and deletion of instances. We have presented the architecture of O!-LOLA and have shown in detail how the preprocessor transforms the OO source language into the logic target language. While inheritance, encapsulation and overriding can be specified purely declaratively, dynamic updates of objects are done by side effects of built-in predicates. Our combination of object-orientation and deduction allows logical rules as methods and logic programs over classes, meta-classes and their properties on the top level. We have given a summary of the most important issues of the theory and the dynamical aspects of the language.

Since LOLA evaluates bottom-up and 'set at a time', O!-LOLA is evaluated in that way too. O!-LOLA benefits from the rich set of optimization techniques developed for deductive databases and included in LOLA. Since LOLA provides an integrated access to relational databases, O!-LOLA can use it, whereas the external relational database can be seen as one big external object with all granted relation names as methods.

O!-LOLA was developed at the Technische Universität of Munich. Further research will include performance tuning and optimizations, extending the dynamic behavior, user interfaces and building applications.

References

Alexiev, V. (1993). *A (not very much) annotated bibliography on integrating object-oriented and logic programming*. ftp://menaik.cs.ualberta.ca/pub/oolog.

Abiteboul, S. (1989). Towards a deductive object-oriented database language. In *Proc. of the 1st Int. Conf. on Deductive and Object-Oriented Database Systems (DOOD)*, Kyoto, Japan. pp. 419–438.

Atzeni, P. (Ed) (1993). LOGIDATA+: deductive databases with complex objects. In *Lecture Notes in Computer Science*, Vol. 701. Springer.

Barja, M., N.W. Paton, A.A. Fernandes, M.H. Williams, A. Dinn (1994). An effective deductive object-oriented database through language integration. In *Proc. of the 20th Int. Conference on Very Large Data Bases (VLDB)*, Santiago, Chile. pp. 463ff.

Bertino, E., G. Guerrin and D. Montesi (1994). Deductive object databases. In *Proc. of the 8th European Conference on Object-Oriented Programming (ECOOP)*, Bologna, Italy LNCS 821. Springer. pp. 213–235.

Bertino, E. and D. Montesi (1992). Towards a logical - object oriented programming language for databases. In *Proc. of the Int. Conf. on Advances in Database Technology (EDBT)* 1992, LNCS 580. Springer. pp. 168–183.

Cacace, F., S. Ceri, S. Crespi-Reghizzi, L. Tanca and R. Zicari (1990) Integrating object-oriented data modeling with rule-based programming paradigm. In *Proc. ACM-SIGMOD Conference on Management of Data*, Atlanta City, New Jersey. pp. 225–236.

Dechamboux, P., and C. Roncancio (1994). $Peplom^d$: an object oriented database programming language extended with deductive capabilities. In *Proc. of the 5th Int. Conference on Database and Expert Systems Applications (DEXA)*, Athens, Greece 1994, LNCS 856. Springer. pp. 2–14.

Freitag, B., H. Schütz and G. Specht (1991). LOLA – A logic language for deductive databases and its Implementation. *Proc. of the 2nd Int. Symposium on Database Systems for Advanced Applications (DASFAA)*, Tokyo. pp. 216-225.

Fromherz, M. (1993). *OL(P) User and Reference Manual.* Available via anonymous ftp from parcftp@parc.xerox.com in the directory /pub/ol.

Kifer, M., and J. Wu (1989). A logic for object-oriented logic programming (Maiers O-logic revisited). In *Proc. of the 8th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania. pp. 379–393.

Kramer, M., G. Lausen, G. Saake (1992). Updates in a rule-based language for objects. In *Proc. of the 18th Conference on Very Large Data Bases (VLDB)*, Vancouver, Canada. pp. 251–262.

McCabe, F.G. (1992). *Logic and Objects*. Prentice Hall.

Moss, C. (1994). *Prolog++; The Power of Object-Oriented and Logic Programming*, Addison-Wesley.

Mumick, I.S., and K.A. Ross (1993). Noodle: a language for declarative querying in an object-oriented database. In *Proc. 3rd DOOD Conference Phoenix*, Arizona. pp. 360–378.

Srivastava, D., R. Ramakrishnan, P. Seshadri and S. Sudarshan (1993). Coral++: adding object-orientation to a logic database language. In *Proc. of the 19th Conference on Very Large Data Bases (VLDB)*, Dublin, Ireland. pp. 158–170.

Zaniolo, C., N. Arni and K. Ong (1993). Negation and aggregation in recursive rules: the LDL++ approach. In *Proc. of the 3rd Conf. on Deductive and Object-Oriented Database Systems (DOOD)*, Phoenix, Arizona. pp. 204–221.

G. Specht is a member of the research staff of computer science at the Technische Universität München, Germany. His research interests include deductive databases, object-oriented programming systems, multimedia databases and natural language parsing. Dr. Günther Specht received his Ph.D. in 1992. Since 1993 he leads the multimedia database project MultiMAP, now founded by the German Research Network (DFN Association). He teaches graduate courses on multimedia database systems, deductive and object-oriented database systems. He is author of several international articles and two German books.

O!-LOLA: deduktyvinės duomenų bazių sistemos LOLA praplėtimas objektiško loginio programavimo priemonėmis

Günther SPECHT

Straipsnyje aprašyta kaip deduktyvinė duomenų bazių sistema LOLA buvo praplėsta iki objektiškos deduktyvinės duomenų bazių sistemos O!-LOLA. Sistemoje O!-LOLA naudojamas modelis "objektai kaip teorijos", praplėstas būsenų vertinimu. Loginis ir objektinis programavimai sistemoje kombinuojami dviem būdais. Pirma, objektų metodai realizuojami kaip loginės programos. Antra, rašant logines programas galima naudoti klases, metaklases, objektus, atributus ir jų reikšmes. Leidžiama dinamiškai keisti atributų reikšmes ir kurti naujus objektus. Sistema realizuota kaip LOLA preprocesorius. Sistemos privalumas, lyginant ją su kitomis objektiškomis loginio programavimo sistemomis, yra kalbos integruotumas, aiški semantika ir aibėmis grindžiami būsenų vertinimai. Naudojama nejudančiojo taško semantika. Todėl galima naudotis visais deduktyvinių duomenų bazių optimizavimo metodais.