ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

B Novikov, University of St Petersburg, Russia and J W Schmidt, University of Hamburg, Germany. (Eds)

Advances in Databases and Information Systems, Moscow 1996

Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS '96). Moscow, 10-13 September 1996

Databases Evolution: A Metaobject Approach

Stéphane Demphlous

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Databases Evolution: A Metaobject Approach

Stéphane Demphlous

INRIA/CERMICS

B.P.93 F-06902 Sophia Antipolis Cedex, France

E-mail : Stephane.Demphlous@sophia.inria.fr

Abstract

We show how the use of a metaobject layer is useful to allow an easier evolution of an object-oriented database integrating existing relational systems. A correct use of metaobjects allow to hide the persistence contingencies when linking an application with an external application. Moreover a metaobject layer allows the evolution of the database object model. We show in these lines how a underlying relational database can be used for final storage of data in a system allowing metamodeling abilities. We have designed a correspondence engine, driven by rules, linking standard object and relational structures. We want the designer to be able to alter these rules according to the evolution of the object model, in order to keep coherent relational schemes. So the correspondence engine has been designed as a specialisable protocol discriminating on the metaobjects of the object layer.

1 Object-Oriented Databases and Legacy Systems

For some years, in database community, a growing interest has focused on object-oriented databases. Most of the current object oriented database management systems (OODBMS) are now uniform, that is, all levels, including the storage level, deal with objects. However, as it is shown in [10], the introduction of such information systems into companies induces many problems. On the one hand, companies have often made big investments in relational databases, and they do not want to abandon them; on the other hand, even if a company decides to transfer its data from a relational database to an object one, such a transfer may be costly. First, databases may be out of order during the transfer. Then, it is not obvious that all the applications using the concerned data are themselves object-oriented. Hence, either the system administrator would have to upgrade these applications toward the object paradigm, or two equivalent databases would have to be kept. So, many companies prefer an incremental move toward object-oriented databases, keeping untouched relational data for older applications.

The DRIVER system [11] solves this problem. It offers an object layer on relational databases. It has been shown in [9] that many object-oriented notions can be expressed with relational databases. With DRIVER, every object-oriented application can see a relational schema in its own object modeling, and can store its new data in the relational database, provided, of course, that the objet modeling is compatible with the underlying relational schema. So, relational databases can still be used and be relevant in new object-oriented applications. DRIVER offers an evolutionary way to introduce objects in a legacy relational system, rather than a revolutionary one.

Moreover, one of the DRIVER's goal is to provide an important modularity. Unlike some other object-relational systems like *Persistence* [5], DRIVER does not consider itself as the master of the underlying relational database, imposing a systematic relational schema on it. The developer is free to choose both his object modeling and his relational schema, provided, of course, that they are compatible, that is, they must follow the generic correspondence rules presented in [9]. The developer just has to provide to the DRIVER system a correspondence schema linking, roughly speaking, classes and relations. DRIVER deals with three conceptual layers: the ODMG object layer, the NF^2 layer and the relational one. DRIVER introduces an intermediate level between the object and the relational one, whose interest is more widely described in [13]. Briefly, The NF^2 layer is a Non-First Normal Form (NF^2) description layer that includes all the

properties formalized in [14]. It allows the developer to define nested relations and orthogonal constructors. It has been shown in [12] that *list* fields, *set* fields or *structured* fields can be automatically defined with joined tables. The NF^2 level frees the database administrator from this systematic tasks.

We consider the object-relational couple as the first step of databases evolution. Now, we will mainly ponder on object model evolutions. However we do not forget that the object-relational couple stays the frame of our works and that relational databases, driven by DRIVER, are the only storage tools that we have chosen for our persistence needs.

2 New Needs in Object-Oriented Databases

In this section, we see some needs that appear in the object oriented database domain. Most of them are common to all object oriented databases. But, in a second part, we see how the use of legacy data can emphasize some of them.

2.1 Orthogonality needs

Object-oriented databases are rarely considered as a whole, only acceded by query languages, but often as a part of a software engineering project. The object concept gives a common modeling structure to databases and programming languages. So the developer may desire to directly manipulate objects stored in a database inside his application. One problem is the differences of handling between transient and persistent objects. The developer definitely wants to define and access uniformly all types of data. However, frequently it is not the case, and it induces some problems when it influences the object modeling.

It is shown in [1], that two important points have to be considered in object-oriented databases: orthogonality with the object modeling and orthogonality with the instantiation process. We will refine this notion, speaking of orthogonality with the way the developer modelizes the universe, and orthogonality with the way he, or the user, accesses persistent data.

In many object-oriented databases, the application developer has to directly manage persistence. For example, to give persistence to C++ objects in an ODMG-compliant database, the developer has to insert in the inheritance graph a class devoted to persistence. So, persistence is not orthogonal with the object modeling activity. When the developer describes his universe, he has to sort the obviously persistent data and the obviously transient data. Now, the frontier between these two data types does not always clearly appear. Sometimes, the use of an application states precisely and makes evolve the limits between persistent and transient data. So offering later persistence to a class which has not been originally described as persistent implies that the graph of classes of the application will be modified. A straightforward solution is to set the class devoted to persistence as root of the complete inheritance tree. So persistence potentiality is given to every class of the application. This seems to be a good answer. However a point has to be considered: in many object-oriented databases, for each class described as potentially persistent an equivalent structure is created in the database. Hence, according to the subsequent actual application use, there may be useless classes stored in the database.

We think that persistence contingencies must be hidden as much as possible to the developer when he modelizes an application. Moreover, we think that there must exist no difference between the ways transient and persistent data are acceded. As a matter of fact, a persistent object creation often appears slightly different than a transient object one. Even if the ODMG standard proposes some answers to reduce this problem, there are still some points where the access to persistent data differs from the transient data accesses. For example, in every C++ binding for an ODMG-compliant database, the way handles on persistent objects are declared is different than for transient data. These problems create many constraints in software engineering. As a matter of fact, during the development phase, a developer has to know if he deals with persistent structures or transient ones.

A simple way to consider persistence during software engineering is to use a fully persistent language where every object is persistent, whatever its purpose may be. Then the persistence is fully orthogonal to the language. The object-oriented database O_2 provides a way to realize this kind of orthogonality with its language O_2C [16]. If persistence contingencies actually disappear when defining an application, this solution is not always satisfactory. As a matter of fact, it may be a waste of storage space to apply such a policy, and sometimes it may be a non-sense, for example, when saving typically transient data that never have to be retrieved and that must be frequently recomputed.

2.2 Object Model Needs

We cannot speak about object-oriented databases without pondering on the most suitable object model. In fact, as we have shown in [2], a perfect persistent object model does not exist. The Object Database Management Group has proposed a model, which can be classified as a *class language* model, according to the classification presented in [15]. However, this model cannot be considered as the best and the universal object model. For example, in artificial intelligence or in knowledge representation, non-*class languages* object models are very useful. Let us cite, among others, *frames* object models, or models using the *point of view* paradigm. Whatever their behaviors are, they are often very different among themselves in respect of their object structures.

The adequation of the object model to the application needs is a really important point since if it is not the case, the developer has to use two different object models: one for the application, another one for the database. Hence, one of the goal of the object oriented databases, which was to unify the application and information systems models, would be lost. Moreover, the actual coexistence of two different object models may induce a heavy cost in software engineering. As a matter of fact, the developer would have to implement from scratch a translation engine between both models.

We think that a very important goal of an evolutive object-oriented database is to provide an object model that can be easily adapted to the one of the querying application.

2.3 Needs Induced by Coexistence of Object and Relational Models

We have seen in the previous parts some new needs that appear in the object-oriented database domain. However, as explained in section 1, we focus on the evolution of relational databases. Since the goal of an object-relational database is to be fully an object-oriented database, each proposal presented in the previous parts stays relevant. However we have to show how some distinctive features of the object-relational couple emphasize some proposals, particularly on object model evolution purposes.

Most uniform object-oriented databases do not allow to modify their object model because they do not want to add a mapping engine upon their system. Now we have seen that there is always a mapping engine between the object and relational worlds in an object-relational database. This mapping is an additional argument to provide an adaptable object model, as proposed in section 2.2. Whatever the chosen object model is, there is an “impedance mismatch”, that is, a great difference in model theory, between the object layer and the relational one. So, inserting a new object model does not add another translation engine: it only changes the already existing one. So there is no reason to keep the object model fixed. It is quite natural to provide the developer some ways to modify the translation engine, and, consequently, the object model.

3 Metaobject Protocols: An Answer

We think that a good way to elegantly solve these problems is to use a “metaobject level”.

3.1 Orthogonality

As seen in section 2.1, an interesting way to manage persistence is to let final users dynamically set the persistence frontier. The system may be able, during an application session, to save in the database only the classes required by the user. Hence, whenever the user wants a new class to be persistent, the system has to be able to dynamically inspect the structure of this class to consequently modify the database schema. Of course, such an activity is possible only if classes are inspectable at runtime, hence are themselves objects, instances of meta-classes.

Moreover, it has been shown in [20] how the use of a meta-level, especially the CLOS’ meta-classes and metaobject protocol, help offering orthogonality. With a persistence meta-class and a corresponding refined instantiation metaobject protocol, it is possible to automatically introduce a persistence-devoted class as root of a graph of classes, avoiding most of the possible conflicts. Moreover, refining in an adequate way the metaobject protocol, one can, transparently, reroute slot accesses toward the database when needed. For example, some CLOS-like languages, like *Power Classes* [4], have protocolized the slot accessors. So, they offer means to realize a very refined management of data

access, and hide the persistence distinctive features during a slot access. So, the need for the system to interrogate the database in order to fill an object slot, can be hidden.

With these merged approaches, the developer can design his application without taking into account the persistence contingencies. When he wants to introduce persistence, he just has to change the meta-class of his classes, and to choose a persistence-devoted meta-class. We can notice that it corroborates the idea presented in the part 2.1, that is, a persistence-devoted root class for all classes. However there are some differences. First, the developer does not have to manage the inheritance of the persistence-devoted class, since the introduction is managed by the specialized instantiation metaobject protocol. Then, due to the dynamic aspect, there is no link between presence of the persistence devoted class inside a graph of class and classes creation in the database. Persistent structures are created only when final users decide to.

3.2 Object Model Evolution

A meta-level appears to be one of the best ways to easily define or specialize an object model. Some ways to introduce a new object model defining new meta-classes and specializing their related metaobject protocol are shown in [6]. Some object-oriented databases have adopted such an object layer. We can cite the ADAM object-oriented database [3], with its meta-class paradigm. Works on ADAM show the improvement in data expressivity that a correct use of meta-classes can introduce. The VODAK object-oriented database [8] also offers a well structured meta-layer allowing to widely customize the native object model in order to comply with new needs in expressivity. However these databases do not meet our requires, since they are “proprietary” databases: data are not saved in any standard format, neither the relational, nor the ODMG one.

We want to provide an object-oriented database whose object model may be evolutive, thanks to a metaobject layer. But we also want the storage tool to be a relational database.

3.3 Metaobject protocols and RDBMS

Works have been performed in order to integrate legacy relational databases in the OODBMS VODAK [7]. These works permit to reengineer existing relational databases. A meta-class linking the object and relational worlds is provided. This meta-class gives access handles to the relational database and offers to classes some generic methods allowing to get a value from a relational attribute through a path between tables. So the developer, when he defines a class, can use these methods to specify slot accessors that retrieve corresponding values from the underlying relational database. However we have, in our current works, a different point of view on object model evolution in an object-relational database. The intend of the relational link with VODAK is, as far as the author knows, to reengineer relational databases inside an object-oriented database using meta-classes, not to directly provide a metaobject layer on top of a relational database. Our purpose in the current part of our works, is to systematically link a special type of relational structures with a given part of an object model. Of course, this approach implies some limitations in the relational schema. As a matter of fact, the schema will have to fit in special frames induced by the correspondences chosen for the different parts of the object model. But we will see in the next parts that it allows us to completely hide to the developer all the communications with the database.

Finally PCLOS [17, 18, 19] has interested us since it is a persistent CLOS on a underlying relational database. It takes the same “up to down” approach regarding object model correspondences. PCLOS provides an automatic way to get good relational representations for every standard CLOS class. Now, when a developer specializes the CLOS’ metaobject protocol to create another object model, he often desires a different relational representation than the standard one in order to mirror the language semantic on the relational support.

So, we have designed a persistence metaobject protocol between an object layer using a CLOS dialect and a relational database.

4 A Persistence Metaobject Protocol

Now we present our choices, and how Extended DRIVER may be used. We used the *Power Classes* language as interface language.

4.1 Standard Use

When the provided standard object model appears to be sufficient for an application, the developer just has to use the persistence-devoted meta-class `<persistent-class>` we provide, as meta-class of all his classes. The use of this meta-class is the only constraint added on the application modeling phase. We think it is a light constraint even if it implies some syntactic “noise”. As a matter of fact the absence of the meta-class specification line in standard classes definition is only a syntactic easiness, since every class should precise its meta-class. We just want the developer to use a subclass of the standard meta-class `<standard class>`, instead of `<standard-class>` itself.

Then, at runtime, the user can send whenever he wants, a persistence message to any object. If the considered object is a final instance, that is, not a class, the system retrieves the object’s class and, if no correspondent persistent structure has been yet computed, sends the same persistence message to it. So the considered class is inspected and corresponding NF^2 and relational structures are automatically generated. This generation is done using the standard translation engine. This translation engine from object models to NF^2 and relational representations is designed as a protocol specializing on the object model’s meta-classes. Since it specializes on meta-classes and only deals with persistence, we call it Persistence MetaObject Protocol (PMOP). The PMOP is graded in numerous multi-methods, always specializing at least on the standard persistence meta-class we provide. Each method addresses one elementary part of an object model, or one elementary activity of the translation engine. A short skeleton of the PMOP is shown in figure 1. Each line symbolizes a method call and the indents show the calling relations between two methods.

Figure 1: Persistence metaobject protocol skeleton

We can see that there are some distinctive parts in this protocol. It first deals with the definition of (1) the NF^2 representations associated with object types; (2) the relational structures to be linked to the NF^2 one; finally it deals with (3) the actual creations of persistence structures and (4) retrieve or storage call. The key point to consider is that this protocol gives the insurance that systematic shapes of NF^2 and relational structures will be generated as correspondence of a given part of the object model. Let us take a simple example that we will follow along this paper. We do not explain in details the standard correspondences of classes and the way the inheritance is managed, but, in a didactic purpose, we intend to focus on the NF^2 and relational correspondences of slots. Very briefly, a class is mirrored by a table, and the inheritance is managed adding tables owning the new properties, thanks to the concept of inheritance between NF^2 tables presented in [13]. In *Power Classes*, slots in classes cannot be described as structured: their types are atomic, list or reference. The PMOP maps atomic slots to NF^2 and relational attributes, list slots to list NF^2 attributes and to a relational table joined to the main one, and references slots to NF^2 and relational foreign keys. Due to the dynamic properties of the language and to the systematism of these correspondences, the PMOP can, in a transparent way, modify the standard slot accessors so that they interrogate the database when an object is not fully built in memory. We see that the persistence is fully transparent for the developer, and orthogonal to the object model.

4.2 Persistent Object Models Developers

Now, when a developer wants to make evolve the standard persistent object model in order to comply to a new standard, or to interoperate with a special application, he has to follow two steps: first, he has to specialize the *Power Classes*' metaobject protocol; then, he has to specialize our persistence metaobject protocol.

The way the developer introduces in our system a new object model is the classical one in CLOS dialects. It is in no way altered by some persistence need. The developer defines a new meta-class <MC> and refines the instantiation metaobject protocol on it. Then, since *Power Classes* manages meta-classes multiple inheritance, the developer declares a new meta-class <persistent-MC> that inherits from the persistence devoted meta-class that we provide, and from the newly created meta-class <MC>.

If the developer does not want the underlying NF^2 and relational structures to be in special shapes, the new object model can yet be considered as persistent. Instead of declaring classes as instances of the meta-class <MC>, the developer just has to declare them as instances of the persistent meta-class <persistent-MC>. So all the runtime features described in the previous section stay relevant. However, as presented in section 3.3, the generated database structures may not be satisfactory. Let us consider again the example of slots correspondences. It seems obvious that one of the first evolutions in the *Power Classes*' object model that a meta-level developer would like to perform, would be to add structured slots, that are widely used in current object models. So the developer will define a new persistent meta-class <persistent-MC> implementing these new features.

we can imagine a class <car>, instance of the meta-class <persistent-MC>, whose slot engine, expressing the power of the car's engine, owns two attributes: the tax linked with the engine, and the engine's cubic capacity. The *Power Classes* corresponding code is shown in figure 2.

```
(defclass <car> ()
  ((.name accessor: name)
   (.engine accessor: engine
    attributes: ((.tax accessor: engine-tax)
                 (.cm3 accessor: engine-cm3)))
  )
  constructor: make-car
  meta-class: <persistent-MC>)
```

Figure 2: Class <car>

However, the values added to a slot, like the tax or the cubic capacity must be stored somewhere in memory. This problem has to be solved by the developer when he specializes the *Power Classes*' object model to introduce his new object model. There is many ways to do it and almost no hypothesis can be done on the way that will be chosen. One can, for example, store them in the actual slot value and redefine the slot accessors to access only the needed information, hiding the additional one. An example of an instance of this class is shown on the top part of the figure 3. When adequately redefined, the accessor engine will render only the value 84 while the pseudo-accessor engine-tax will find 7 in the value of the engine slot.

However, with the standard PMOP, the implementation contingencies will be directly mirrored in the relational schema. For the standard PMOP the two slots of the class engine are only standard slots. So the corresponding generated NF^2 and relational structures would be as diagrammed on the center and the bottom of the figure 3, that is, an unique NF^2 or relational table owning two attributes corresponding to the two slots. Hence, slot values will be straightforward saved in the corresponding attributes.

Such computed underlying NF^2 and relational schemas would probably not be very convenient. Their structures do not take into account the evolve of the object model: they are similar to the ones computed for the standard object model. This is not satisfactory because the values stored in a relational attribute may lose any semantic. Let us imagine that the developer has added this new property to the *Power Classes*' object model in order to make evolve an existing application modeling cars. Now, if, due to any reason, a previous version of the application has to be used, it will find in the relational attribute engine, values that depend on a subsequent implementation that it cannot understand. So, this kind of evolution may induce many compatibility problems. One way to avoid such conflicts is to systematically break the information present in attributed slots in several tables, as diagrammed in figure 4.

Figure 3: Standard storage structures

Figure 4: Altered storage structures

The point to consider is that each attributed slot in a class would own, as an NF^2 correspondence, an embedded table containing the slot value itself, and the slot attribute values as separated NF^2 attributes. The desired underlying relational schema would include a main table owning all the primitive values of the slots, and, for each attributed slot, an auxiliary table owning the values of attributes. Of course slots attributes could be themselves attributed. Then NF^2 attributes representing them would have to be in their turn embedded tables, and additional relational tables would have to be defined. We limit our study on an unique level of attributes for didactic purposes.

So, we see that we have to let the developer freely modify the parts of the translation engine dealing with slot correspondences. Such a customization can be done specializing our persistence metaobject protocol. In our system, with an object model defined by a meta-class specializing the standard meta-class, the developer just has to specialize on it the parts of the PMOP where the translation of the new object model differs from the older one. In our example, only parts of the PMOP dealing with slot translation have to be adapted.

In figure 5 we have pointed by an arrow the only methods that will have to be specialized on the new meta-class `<persistent-MC>`.

```

_pst-SaveObj (<persistent-class> <object>)
_pst-SaveClass (<persistent-class>)
_pst-ClassToNF2 (<persistent-class>)
_pst-LinkedClasses (<persistent-class>)
_pst-Inheritance (<persistent-class>)
➔ _pst-SlotToNF2 (<persistentMC> <slot>)
_pst-NF2ToRelational (<persistent-class>
                     <NF2Table>)
➔ _pst-NF2AttributeToR (<persistentMC>
                       <NF2Attribute>)
_pst-CreateRelational (<persistent-class>
                      <RTable>)
_pst-CreateNF2 (<persistent-class> <NF2Table>)
_pst-DefineRequest (<persistent-class> <object>)
➔ _pst-SaveSlot (<persistentMC> <NF2Table>
                <slot> <object>)
```

Figure 5: Modified PMOP

These methods deal with: first, the mapping of object slots to NF^2 attributes; then the mapping of NF^2 attributes to relational ones; finally the type of the NF^2 -queries – near to OQL – that will be computed at runtime for saving slots. The first must now link an embedded NF^2 table to each slot, the second a joined relational table to each embedded NF^2 table, and the last sets how the system must, at runtime, break the information contained in slot values, separating direct slot values and values of attributes, in order to save them. Of course, the developer does not have to completely rewrite these parts of the protocol. Since the new persistent meta-class inherits from the persistence-devoted meta-class we provide, the methods specialized on the last one can be freely reused.

4.3 Main interest of the approach

This approach presents two main interests. First, there is only one concept to consider: the metaobject protocol one. To introduce a new persistent object model, there are two metaobject protocols to specialize. The first one to create the object model, the second one to set it persistent. Then, the second interest is that experience is capitalized. When a need of different object model evolution appears in a classic object-oriented database, developers have to write a complete transaction engine “from scratch”. With our approach, experience is yet capitalized since a translation engine has been created and protocolized and only concerned elements have to be changed.

Conclusion

As a first step in database evolution we wanted to allow relational databases to cooperate with application using the object paradigm. The DRIVER system provides ways to meet this goal, coupling an object-oriented layer with a relational database. However we have seen that a persistent object model itself may have to evolve. One of the best ways to allow these evolutions is to provide a metaobject layer. Such a layer allows to set the persistence orthogonal to the

object modeling and to customize the standard object model in order to comply with new needs. Now, when the object model evolves, the mapping from object structures toward relational representations may have to be modified, in order to mirror the new semantic on the persistence structures. Thus we have protocolized the correspondence generation in a persistence metaobject protocol. This protocol opens widely the translation engine between the two worlds. So a meta-level developer will just have to adapt the only parts of the translation engine dealing with the modified parts of the object model. Making evolve a persistent object model is now an uniform activity that can avoid compatibility problems in schemas. In future works, we intend to introduce the ODMG object model. This introduction will probably help us to refine our protocol's structure.

References

- [1] E. Amiel, M.-J. Bellosta-Tourtier, P. Valduriez, and F. Viallet. Etude de la persistance dans les SGBDOO. Rapport de Recherche 1592, INRIA, Domaine de Voluceau, Rocquencourt, B.P.105, 78153 Le Chesnay Cedex, France, Feb. 1992.
- [2] S. Demphlous and F. Lebastard. Persistence of Multiple Object Models. In *OOPSLA'95 Workshop on Metamodeling in OO*, pages 32–37, Austin, Texas, Oct. 1995.
- [3] O. Diaz and N. W. Paton. Extending OODBMSs Using Metaclasses. *IEEE Software*, pages 28–39, May 1994.
- [4] ILOG. *Ilog Power Classes Reference Manual, version 1.3*. BP85, 2 av. Galliéni, 94253 Gentilly Cedex, France, 1994. 202 pages.
- [5] A. Keller, R. Jensen, and S. Agarwal. Enabling The Integration of Object Applications With Relational Databases. Technical report, Stanford University and Persistence Software, Inc., Dec. 1994.
- [6] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.
- [7] W. Klas, G. Fischer, and K. Aberer. Integrating Relational and Object-Oriented Database Systems using a Metaclass Concept. *Journal of Systems Integration*, 4:341–372, 1994.
- [8] W. Klas and M. Schrefl. Metaclasses and their application – Data Model Tailoring and Database Integration. In *Lecture Notes in Computer Science*, number 943. Springer-Verlag, 1995.
- [9] F. Lebastard. *DRIVER : Une couche objet persistante pour le raisonnement sur les bases de données relationnelles*. PhD thesis, INSA de Lyon/INRIA/CERMICS Sophia Antipolis, Mar. 1993. 380 pages.
- [10] F. Lebastard. An Object Layer on a Relational Database more Attractive than an Object Database? In *KRDB-95 – Reasoning about Structured Objects: Knowledge Representation Meets Databases*, Sept. 1995.
- [11] F. Lebastard. *DRIVER V1.56 : Reference Manuel*. CERMICS/INRIA, F-06902 Sophia Antipolis, France, Feb. 1995. 103 pages.
- [12] F. Lebastard. Some Generic Correspondences to Define Object Databases on Top of Relational Databases. In *OOPSLA'95 Workshop on Objects and Relational Databases*, Austin, Texas, Oct. 1995.
- [13] F. Lebastard. DRIVER: Toward a Three-Layered Architecture. Technical report, INRIA/CERMICS, F-06902 Sophia Antipolis, 1996. To be published.
- [14] M. Levene. The Nested Universal Relation Database Model. In *Lecture Notes in Computer Science*, volume 595. Springer-Verlag, 1992.
- [15] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Trombre. *Les Langages à Objets*. InterEditions, 1991.
- [16] O2 Technologies. *O2C Reference Manual*. 7, rue du Parc de Clagny, 78035 Versailles Cedex, France, Mar. 1995.

- [17] A. Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In *ECOOP/European Conference on Object-Oriented Programming - Lecture Notes in Computer Science*, pages 374–389. Springer Verlag, 1988.
- [18] A. Paepcke. PCLOS: A Critical Review. In N. Meyrowitz, editor, *OOPSLA '89 Conference Proceedings*, volume 24/10, pages 221–237. SIGPLAN Notices, Oct. 1989.
- [19] A. Paepcke. PCLOS: Stress Testing CLOS. In N. Meyrowitz, editor, *OOPSLA – ECOOP '90 Proceedings – Conference on Object-Oriented Programming, Systems, Languages, and Applications – European Conference on Object-Oriented Programming*, volume 25/10, pages 194–211. SIGPLAN Notices, Oct. 1990.
- [20] A. Paepcke, editor. *Object-Oriented Programming – The CLOS Perspective*, chapter 3, pages 65–101. The MIT Press – Massachusetts Institute of Technology, 1993.