# Adapting Database Object Models to Knowledge Representation Needs

**Stéphane Demphlous**
INRIA, BP 93, F-06902 Sophia Antipolis Cedex, France
E-mail: Stephane.Demphlous@sophia.inria.fr

**Abstract.**
The use of external object-oriented databases to offer persistence to knowledge bases often induces a problem of heterogeneity between the knowledge representation model and the persistent object model. Most often a translation engine has to be provided between the two systems and the knowledge structures are not mirrored in the database schema structures. We propose an object-oriented database management system, based on a object-relational couple and a complete metaobject approach, allowing to customize both its object model and the translation engine between its two constituting elements. So the object model of the database can become similar to the one of the knowledge representation system, and, with our specializable persistence metaobject protocol, the underlying relational structures can be adapted.

## 1 OBJECT-ORIENTED DATABASES AND KNOWLEDGE REPRESENTATION NEEDS

One of the first aim of our research team is to provide a link between artificial intelligence applications and the database world. Initially our research topic was to provide persistence to the SMECI expert system [4]. This task has been completed with the DRIVER system [11]. However DRIVER does not limit itself to the persistence of SMECI. It appears clearly that, as the knowledge may evolve, the knowledge representation may have to evolve too. So DRIVER provides a straightforward way to use different object modeling to represent identical data. To summarize the DRIVER concept, one can say that it is an object layer on a relational database. So, final data are saved in a simple and standardized way, when knowledge representation applications construct appropriate object modeling. The interest of such an approach has been shown in [10] and [12].

However DRIVER adaptability has some restrictions. As a matter of fact, DRIVER provides ways to link most of object modelings to persistent data, but the object model stays the same. We differ object modeling and object model. The first is the way information is structured, that is, roughly speaking, the classes used and the relations between them. The second addresses the internal structure itself of objects and classes. We center our study on the second one.

DRIVER can be objectively seen as an object-oriented database. So we have to follow the evolution of the databases domain. Now the *Object Database Management Group* proposes a standard, the ODMG one [1], that may federate all object-oriented databases. This standard includes an object model. So we have to fix whether we only have to set DRIVER ODMG-compliant, which is, by the way, currently under process. From a database viewpoint it may appear as sufficient. However, from a knowledge representation viewpoint, centering the DRIVER system on an unique model appears as a mistake.

As a matter of fact, it seems obvious that the ODMG object model cannot fulfill the needs of every knowledge representation application. SMECI itself , which historically has been the origin of the works of our team, uses a different object model than the ODMG one. One can find many examples where it may be hard to express fundamental notions in the ODMG model: how to express frames, how to express the point of view paradigm in the ODMG model? Using an ODMG database, the developer would have to implement from scratch a translation engine between the object model used in his knowledge representation application and the database object model. This problem has been faced in [6]. A knowledge representation system has been coupled with an object-oriented database, or, when needed, with a relational database. Moreover, a way to open the persistent knowledge representation model is to offer persistence to a *generic frame protocol* [7]. However the knowledge representation models that can be inserted are only the one whose concepts strictly fit in the features provided by this protocol.

We think that one of the best way to provide extensibility of object models is to provide a metaclass paradigm, as shown in [8]. In this work, the CLOS language is taken as an example, and different ways to introduce a new object model are shown, creating new metaclasses and specializing on them the instantiation metaobject protocol. Some object-oriented databases, like ADAM [3] and PCLOS [14][15][16][17] provide a metaclass paradigm.

However these systems are more persistent languages owning the metaclass concept than object-oriented databases. They often consider themselves as the owner of the underlying database, so data can be hardly reused outside the language or the system. We want a same group of data to be seen differently according the object model used, that is, different meta-instances representing same data, but following different representations. We want data to be stored in an elegant way, defined by the developer, in the underlying relational database, and we want them to be accessed by different translation engines that may coexist upon them. We think that the "elegance" of the underlying relational schema is an important goal. We do not want knowledge structures to be saved in the underlying database in a kind of "bytes string".

It has been shown in [9] that most of the object-oriented concepts can have mirrored relational structures expressing their semantic. We think that it can be useful to let knowledge representation structures be explicitly mirrored in the relational schema. Moreover, we want to capitalize translation experience, so we think that a persistence metaobject protocol between the knowledge representation system and the database is the best answer. We have discussed this architecture in [2].

# 2 INTRODUCING A NEW PERSISTENT OBJECT MODEL

## 2.1 Managing Persistence

We offer a three-layered architecture. The lower layer is a relational database. An intermediate level uses a *Non-First Normal Form* ($NF^2$) paradigm. This level allows the user to define embedded tables as attributes. So the mapping from object slots to relational attributes is simplier when the slots are of *list* type or *structure* type [13]. The upper layer is the CLOS dialect *Power Classes* [5]. We have chosen a CLOS dialect since many works have shown the ability of this language to integrate new object models. Moreover, CLOS structure appears often appropriate to make evolve an object approach toward a knowledge representation one. An example can be found in [18].

Between the upper and the medium layer we have defined a *persistence metaobject protocol* that we present now.

We have pondered on the various mechanisms that succeed one another during persistence, that is, either when host relational structures corresponding to object types are created, or when final data are saved. We have brought to the fore "strategic" events inside these mechanisms. Modifying these events leads to systematically define new coherent correspondence schemas. This event sequence is the *persistence metaobject protocol* (PMOP). We have implemented it in *Power Classes.* Its ability to be specialized on new metaclasses allows the developer to exercise an influence on these "strategic places".

We show in figure 1 a skeleton of our PMOP. It is a sequence of multi-method calling one another and specializing at least on the persistence-devoted metaclass we provide. The indent between the lines shows the calling relations between methods.

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

Figure 1: Persistence metaobject protocol skeleton

So we provide *Power Classes* and the *persistence metaobject protocol.* This package includes a persistence-devoted metaclass `<persistent-class>` inheriting from the standard metaclass. The developer has to use it to define persistent structures. The use of this metaclass is the only constraint added in the application modeling phase. The developer can send whenever he wants a persistence message to any object. Then, using the standard PMOP, underlying $NF^2$ and relational schemas corresponding to addressed object types are defined; the metadata of these object types are saved; and, finally, data are stored. The retrieve and the storage of data are incremental. Object defaults are managed, and the user is the master of the persistence activity: an object is completely created in memory only when the user wants it to; otherwise, the upper layer converses with the database.

## 2.2 Customizing the Object Model

Now, when the developer wants to define a new persistent object model more suitable to his knowledge representation needs, he has to follow two steps. First, he has to define a new metaclass `<MC>` and to specialize the *Power Classes'* metaobject protocol on it. It is the usual way to introduce or modify an object model, and this way is not altered by the persistence requirements. Then, when the object model is consistent with the application needs, the second step to follow is to define a new metaclass `<persistentMC>` inheriting from our persistence-devoted class `<persistent-class>` and from the metaclass introduced by the developer. Then, the developer just has to declare his object types as instances of `<persistentMC>`: they will persist, of course only when the final user wants them to.

However the relational schema will be the one computed by our translation protocol. Since the object model is supposed to have substantially changed, the computed corresponding relational schema would not be very adequate. Moreover it may reflect implementation contingencies. If the developer wants more coherent corresponding relational schemas, he has to specialize the PMOP on the new metaclass.

Let us take a simple example. Let us imagine that the developer introduces an object model similar to the standard one except the field structures. He may want to add facets to the slots. Facets add informations on slots. Now, when a persistence message is sent to a class, corresponding $NF^2$ and relational structures are created and metadata about these structures are saved. With the standard objet model, these metadata are saved in $NF^2$ and relational tables whose skeleton is briefly shown in the figure 2. A table of slots, whose keys are the class and slot identificators, keeps various informations about slots, like its type.

Now, there may be various facets linked to one slot. To offer persistence to these facets, the way the metadata are organized must be changed. To keep coherent underlying relational structures, the user would probably need structures like the ones shown in the figure 3. We can see in the relational layer that metadata dealing about basic structures of slots stay in a main table while metadata of facets are stored in an auxiliary relational table. This table is joined to the main one considering the class and the slot identificators as a foreign key. This new schema is useful since it can be reused with an object model not owning the facet concept. We see that the evolution of the object model structure has no destructive consequence on the underlying relational data.

To systematically provide these corresponding schemas, the developer has to specialize only the part of the PMOP generating persistent metadata describing slots. The rest of the PMOP, dealing, for example, with classes, inheritance, reference or aggregation relations, stays unchanged. Hence the only method to be specialized are the one shown in bold in the figure 4. The first one manages the generation of $NF^2$ structures corresponding to slots; the second one links relational structures to $NF^2$ attributes; the last one deals with slot storage, which may have side effect on metadata since facets default may be dynamically changed.

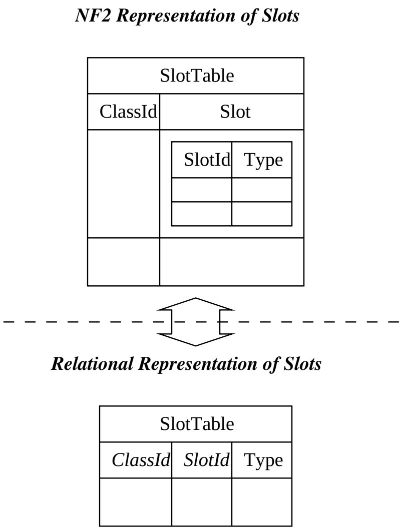


Figure 2: Standard storage structures

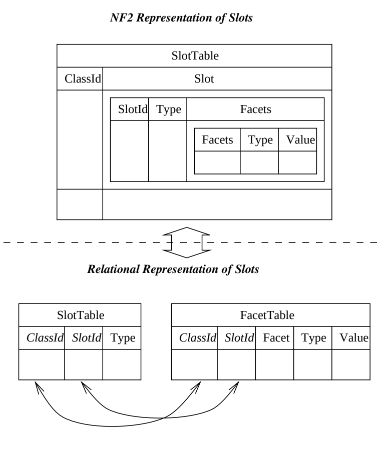


Figure 3: Altered storage structures

```
_pst-SaveObj (<persistent-class> <object>)
  _pst-SaveClass (<persistent-class>)
    _pst-ClassToNF2 (<persistent-class>)
      _pst-LinkedClasses (<persistent-class>)
        _pst-SuperClasses (<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-SaveSlot (<persistentMC> <NF2Table>
                 <slot> <object>)
  _pst-DefineRequest (<persistent-class> <object>)
```

Figure 4: Persistence metaobject protocol skeleton

The PMOP offers a "open translation engine". When the developer customizes the object model, he can customize the translation engine with a weak cost. As a matter of fact, he can reuse the most part of the engine, customizing only the essential parts. Whatever the new object model used for knowledge representation may be, the rewriting cost is lighter than using a "black box" ODMG database.

## CONCLUSION

Offering persistence to a knowledge representation application often leads to a hard problem: the use of a common object model, since the object models used in knowledge representation are often different than the object databases one. Most often the only application has to move toward the underlying database. We think that the database must move as far as possible toward the application. The PMOP we have described is a way to fill the gap between them. The application object model becomes the database's one. Our system gives a way to derive a knowledge representation model from a persistent object model. Our object-oriented database is based on an object-relational couple. With the PMOP, when a knowledge representation model is introduced, the evolution of semantic can be mirrored on the underlying database. Hence knowledge can be more easily accessed and shared from a lower layer than the knowledge representation system itself. Finally we will conclude highlighting two key points. First, introducing a new persistent object model may appear as a hard activity. However it is an uniform activity. As a matter of fact, the developer has to specialize two orthogonal metaobject protocols: the first one to create his object model, the second one to adapt its persistence ability. So there is only one concept to consider: the specializable metaobject protocol one. Moreover we have seen that all the persistence activity is protocolized. Hence, whatever the knowledge representation application is, the amount of code to write in order to offer persistence to it, is minimal. Since the developer can reuse parts of our translation engine, the rewrited code is alway less or equal than the one done for a complete translation engine.

## References

[1] Tom Atwood, Joshua Dubl, Guy Ferran, Mary Loomis, and Drew Wade. *The Object Database Standard: ODMG-93.* Morgan Kauffman, 1994, 176 pages.

[2] Stéphane Demphlous and Franck Lebastard. Persistence of Multiple Object Models. *OOPSLA'95 Workshop on Metamodeling in OO*, pp 32-37 October 1995.

[3] Oscar Diaz and Norman W. Paton. Extending OODBMSs Using Metaclasses. *IEEE Software*, pp 28-39, May 1994.

[4] ILOG. *SMECI Version 1.65, Manuel de référence.* Gentilly (France), May 1990.

[5] ILOG. *Ilog Power Classes Reference Manual, version 1.3.* BP85, 2 av. Galliéni, 94253 Gentilly Cedex, France, 202 pages, 1994.

[6] Peter D. Karp and Suzanne M. Paley. Knowledge Representation in the Large. *IJCAI'95 Proceedings volume 1 – International Joint Conference on Artificial Intelligence*, pp 751-758, August 1995.

[7] Peter D. Karp, Karen L. Myers, and Tom Gruber. The Generic Frame Protocol. *IJCAI'95 Proceedings volume 1 – International Joint Conference on Artificial Intelligence*, pp 768-774, August 1995.

[8] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[9] Franck Lebastard. *DRIVER : Une couche objet persistante pour le raisonnement sur les bases de données relationnelles.* PhD thesis, INSA de Lyon/INRIA/CERMICS Sophia Antipolis, 380 pages, March 1993.

[10] Franck Lebastard. An Object Layer on a Relational Database more Attractive than an Object Database? *KRDB-95 – Reasoning about Structured Objects: Knowledge Representation Meets Databases*, September 1995.

[11] Franck Lebastard. *DRIVER V1.56 : Reference Manuel.* CERMICS/INRIA, F-06902 Sophia Antipolis, France, 103 pages, February 1995.

[12] Franck Lebastard. Some Generic Correspondences to Define Object Databases on Top of Relational Databases. *OOPSLA'95 Workshop on Objects and Relational Databases*, Austin, Texas, October 1995.

[13] Franck Lebastard. DRIVER: Toward a Three-Layered Architecture. Technical report, INRIA/CERMICS, F-06902 Sophia Antipolis, 1996. To be published.

[14] Andreas Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. *ECOOP/European Conference on Object-Oriented Programming - Lecture Notes in Computer Science*, pp 374-389, Springer Verlag, 1988.

[15] Andreas Paepcke. PCLOS: A Critical Review. In Norman Meyrowitz, editor, *OOPSLA'89 Conference Proceedings*, volume 24/10, pp 221-237, SIGPLAN Notices, October 1989.

[16] Andreas Paepcke. PCLOS: Stress Testing CLOS. In Norman Meyrowitz, editor, *OOPSLA – ECOOP '90 Proceedings – Conference on Object-Oriented Programming, Systems, Languages, and Applications – European Conference on Object-Oriented Programming*, volume 25/10, pp 194-211, SIGPLAN Notices, October 1990.

[17] Andreas Paepcke. *Object-Oriented Programming – The* CLOS *Perspective*, chapter 3, pp 65-101. The MIT Press – Massachusetts Institute of Technology, 1993.

[18] Christian Rathke. Object-oriented programming and Frame-based knowledge representation. *In Proceedings of the 1993 IEEE – International Conference on tools with Artificial Intelligence*, pp 95-98, Boston, Massachusetts, November 1993.