Using Prolog to Provide Access to Metadata in an Object-Oriented Database

Suzanne M. Embury[1,2]

Zhuoan Jiao[1]

Peter M. D. Gray[1]

Departments of [1]Computing Science
and [2]Molecular and Cell Biology
University of Aberdeen
King's College
Aberdeen, Scotland

ABSTRACT

P/FDM is an object-oriented database implemented in Prolog that is intended to provide a platform for the development of data intensive applications (e.g. scientific databases). It is being used to store information about protein structures. A Prolog application has been developed that uses this large database to assist biochemists in homology modelling of proteins. Because of the large amounts of data involved, it is essential that database access be efficient. This is particularly true of metadata, which must be accessed several times to retrieve even a single value from the database. Unfortunately, this causes a conflict with user applications, which also need to access metadata. Now uniformity of access replaces efficiency as the main consideration. This paper examines this conflict of requirements and a solution is proposed. Finally, the suitability of Prolog for the implementation of such a solution is discussed.

1. Introduction

Object-oriented databases have been proposed as suitable platforms for the development of large, data-intensive applications [Zdonik 90]. This class of applications, which includes design and scientific systems, typically requires fast access to very large amounts of data, an expressive data model for the representation of complex domains and the ability to extend that representation during or after development with a minimal effect on existing code. Databases are also used when applications need to operate on more data than will fit conveniently into virtual memory, or when several users or modules of an application need to operate on a central body of shared data [Fox 86].

To a database developer, a database management system (DBMS) is also an application that consists of several modules all accessing common data. However, that data is not the *raw data* that other kinds of application use; a database is a general-purpose tool and it is not written with any specific domain in mind. When building a system on top of a DBMS, the structure of the domain data must be described using the building blocks offered by the database's data model. It is this description of data (also called a *schema* or *data dictionary* or *metadata*) that is shared between the components of a DBMS.

How is this shared data to be managed? Since schemata are defined by individual users for the storage of particular sets of data, they must be stored with that data somehow. The generally accepted solution is to store metadata in some specially designed format and to read it into memory at the beginning of each session. This can be very efficient but it means that metadata is accessed differently from raw data. In other words, we have lost *uniformity* of access and have thus reduced the usefulness of the metadata.

In what follows we use P/FDM, an object-oriented database implemented in Prolog, to illustrate the ramifications of this conflict of requirements and then propose a solution. The ease with which the solution has been incorporated into the existing P/FDM system owes much to the choice of Prolog as the implementation language. In the final section of this paper we discuss the features of Prolog that make it particularly suitable for the implementation of such an solution.

2. An Overview of the P/FDM Architecture

The Functional Data Model (FDM) was proposed in [Shipman 81] along with the data definition and manipulation language Daplex. P/FDM is a natural extension of the FDM, implemented in Prolog and C, which incorporates the concepts of the object-oriented paradigm [Gray 92]. It provides three building blocks with which the application domain may be described: entity classes, functions and actions. Entity classes represent real-world objects (e.g. people, proteins) and can be organised into inheritance hierarchies. Unlike many other object-oriented databases, P/FDM requires that a key (possibly compound) be specified for each entity class. The advantages of this are spelt out in [Paton 88].

A function which maps an entity class to an atomic type represents a property of that entity class (e.g. a name, a molecular weight). A function which maps an entity class to another entity class represents a relationship between those classes (e.g. a protein consists of one or more chains). In P/FDM a type may be an entity class, an atomic type or a set of either of these. Consequently, functions may be either single- or set-valued. A function may be defined by an explicit specification of its extension, in which case the range and domain values are stored in the database, or by giving an intensional definition, in which case the code to compute the result is stored in the database.

The third building block is the action [Kemp 91]. An action is also a piece of code which may be stored in the database; however, unlike a function, it does not return a result. Rather, actions define some operation to be carried out; possibly updating the database or displaying some data. An example action taken from the protein modelling application is *new_modeller(Modeller)* which sets up *Modeller* as a valid worker on a particular version of a protein model.

The P/FDM system consists of several system utilities and a set of database access primitives (see Figure 1). These primitives represent the operations that may be performed against the database. They are defined in Prolog but call out to C routines to do the actual file handling. They fall into three categories: those primitives that operate on raw data - adding, deleting and retrieving data; those primitives that operate on metadata - creating and deleting new types; and those dealing with access to database modules. They are listed in Appendix I. A database module is the largest grain of storage provided by P/FDM and is also the unit of locking [Jiao 90]. It represents a conceptual grouping of data. For example, for storing protein structure information we have one module storing standard biochemical data (such as Van der Waal's radii), another storing ‘‘high level’’ data about proteins as a whole and about their chains, and a third storing ‘‘low level’’ details about actual atom positions. Objects are defined in one particular module but, subject to certain constraints, they may reference objects in other modules. To open or create a module is to gain access to the objects which it contains (either for shared reading or exclusive writing). To close a module is to yield up access permission and to commit any changes that have been made to metadata during the session (changes to raw data in the database occur immediately).

A P/FDM database can be queried and updated from Prolog using the primitives for manipulating raw data. Here is an example program that prints the value of the *name* function for all instances of the entity class *person*:

```
query :- getentity(person, Instance),
         getfnval(name, [Instance], NameOfInstance),
         write(NameOfInstance), nl,
         fail
         ;
         true.
```

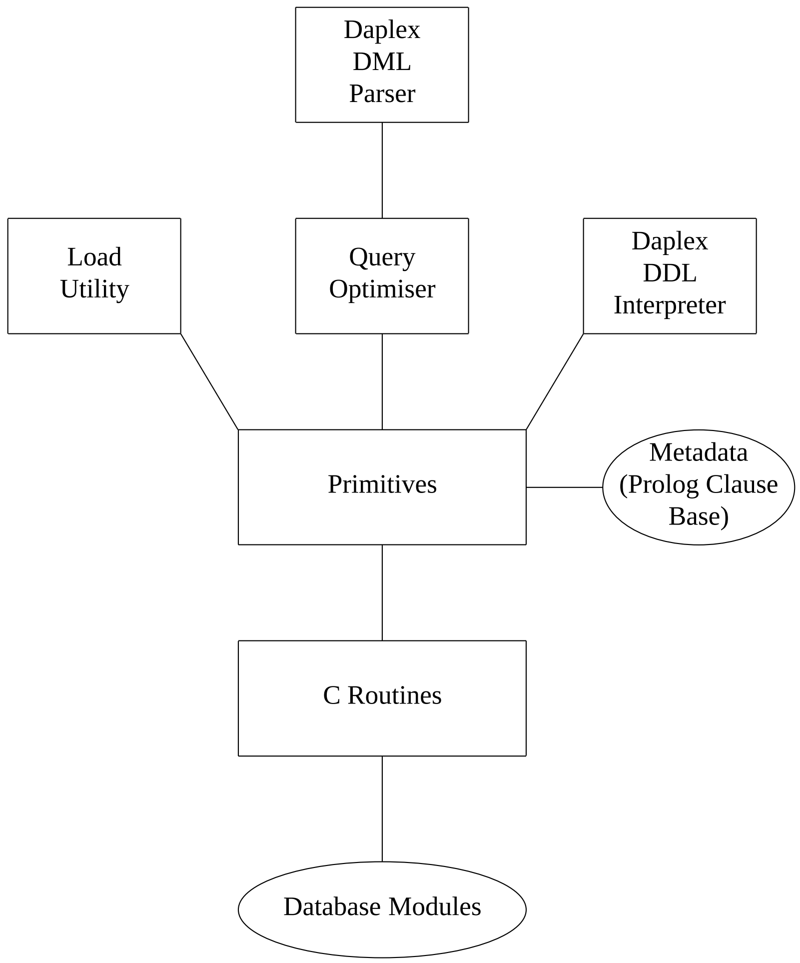

Figure 1 - The Architecture of P/FDM

Notice that entities (and results of set-valued functions) are enumerated by backtracking - hence the failure-driven loop. The following program illustrates an update to the database, the purpose of which is to create an instance of the class *graduand* for every final year student who has passed his or her examinations.

```
update :- getentity(student, Student),
          getfnval(year, [Student], 4),
          getfnval(exam_mark, [Student], ExamMark),
          ExamMark >= 40,
          getfnval(matriculation_number, [Student], MatricNumber),
          newentity(graduand, [MatricNumber], _),
          fail
          ;
          true.
```

Note here that either the function *exam_mark(student)* might be a stored function or else the student's final mark may be calculated *on the fly* from a set of marks achieved on individual courses. In both cases the same primitive is used in exactly the same way; the user does not need to know whether the function is stored or derived before retrieving its result. This is a consequence of the principle of *encapsulation* - an important feature of object-oriented databases [Atkinson 89].

Queries may be expressed even more easily using the language Daplex. The Daplex equivalents of the two programs given above are:

```
for each p in person
print(name(p));

for each s in student such that year(s) = 4 and exam_mark(s) >= 40
create a new graduand with key = (matriculation_number(s));
```

The P/FDM system compiles such queries into Prolog programs with embedded calls to the data model primitives (or, at least, those primitives that operate on raw data). A parser converts the Daplex into an intermediate form which is then optimised before being converted into a Prolog program. Schemata can be expressed using the data definition constructs provided by Daplex. These constructs are interpreted and result in the invocation of one or more of the primitives for metadata update.

3. Demands on Metadata in P/FDM

Although, to a user or application developer, metadata appears to be ‘‘ordinary’’ data like their raw domain data, a database developer cannot take such a naive view. We have already pointed out that metadata cannot be stored in the ordinary database structures. This is partly due to the problem of circularity - we would then need more metadata in order to be able to access the metadata. It is also due to the need for efficient access which, as we shall see, is an important factor in the design of metadata storage structures. Moreover, metadata has different access requirements from raw data. Even a very large database will have relatively few entity classes compared with the number of instances of those classes. Since fast access is required and the volume is low, it seems sensible to copy each module's metadata into the Prolog clause base, as it is opened.

Compared with raw data, metadata is relatively static so update time is not important. Nor are issues of concurrent update of much concern. Integrity of metadata, however, is vital. The database management system depends upon the metadata in order to be able to access raw data. The odd error in raw data need not be so calamitous but even the smallest corruption or inconsistency in metadata can render large portions of or even whole database modules inaccessible.

These, then, are the basic demands on metadata in P/FDM

- the schema for a database module is user-definable; therefore the metadata for that module must be stored with its raw data
- the schema must be extensible; therefore it must be possible to update a module's metadata (although this need not be a particularly efficient process)
- the integrity of metadata must be preserved.

In addition to these, the differing patterns of usage of system and user applications impose other requirements: namely, efficient retrieval and uniformity of access respectively. These are discussed in more detail below but for now it is worth noting an important corollary of uniformity. If metadata can be retrieved using the same constructs as raw data, then any general-purpose application program (i.e. one which makes no assumptions about the schema on which it is to operate) can also automatically operate on metadata. A graphical query interface, for example, designed to be used by end-users, can also be used by programmers as a development tool. Thus, in providing a uniform interface to both raw data and metadata, we are extending the usefulness of existing applications without having to make any changes to them whatsoever.

3.1. System Demands on Metadata

A database is not merely a large collection of raw data; rather, the data must be organised so that it can fulfil the different needs of its users. This is just like a bank, which is not merely a place to keep money. Apart from money, a bank also has at least a manager, some staff, and account books. The DBMS in a database system functions just like the manager of a bank. In order to manipulate the raw data (the ‘‘money’’), the DBMS depends on its knowledge of the metadata (the ‘‘account books’’).

Metadata exists, principally, to facilitate access to raw data. A request to retrieve raw data causes several components of P/FDM's DBMS to access metadata. The Daplex parser checks the syntactic and semantic correctness of queries against the metadata of the accessible modules. Queries are optimised and then compiled into Prolog programs which contain calls to the data manipulation primitives. These primitives access metadata at run-time to decide where to find the raw data and which method definitions to use if method name overloading occurs.

Since each operation on raw data involves several metadata accesses, the efficiency of these accesses has a direct bearing on the efficiency of raw data access. As pointed out in [Maier 86, p.25]

> ‘‘... a mechanism to convey relation schemes, database constraints or types of attributes ... is essential for database design, secondary storage management, optimization, and efficient evaluation. ... Database systems get much of their speed and power advantage over general knowledge representation systems from the declaration of schemes. If a user makes promises in advance about the form of the data, the database system can promise performance advantages.’’

So, efficient access to metadata is not an end in itself for a DBMS but is, rather, an important prerequisite for achieving efficient access to raw data.

While metadata can be used to improve the speed of raw data manipulation at the level of the database access primitives, it can also be used to optimise the Prolog queries produced by the Daplex parser. The need for query optimisation of this kind is well known, and descriptions of relevant research can be found in much of the literature [e.g. Selinger 79, Warren 81].

A query optimiser makes use of such metadata as the *cardinalities* of entity classes, the *fan-out factors* (cardinalities) of set-valued functions, the existence of function inverses, etc. An entity class with a low cardinality is obviously less expensive to enumerate than a higher cardinality class. The higher a fan-out factor of a function is, the more result instances that function will generate, leading to a further expansion of entity instances that must be accessed. If a function has an inverse, the object access path can be reordered so that it starts from a different entity class and results in fewer retrievals. A more detailed description of how the query optimiser uses this information to perform optimisation can be found in [Jiao 91]. Here, we wish only to illustrate the use a query optimiser makes of metadata and thus further stress the need for efficient metadata access. It is of limited use if an optimiser takes longer to optimise a query than is gained in that query's final execution time. More efficient access to metadata means that, within the same amount of time, more query evaluation strategies can be produced and thus a better one can be chosen from amongst them.

3.2. Application Demands on Metadata

The prime function of a database application is to access and manipulate the raw data in the database. This, of course, involves indirect access to metadata via the database access primitives. However, here we are more concerned with *direct* access to metadata, whether via Daplex or Prolog. If we wish to help users and programmers to make more direct use of metadata we must ensure that access to it is at least as convenient as access to raw data. This can be most easily achieved if we provide *the same interface* to both raw data and metadata; in other words, a *uniform* interface.

There are two types of database application: schema-dependent and schema-independent. A schema-independent application is intended to be a general-purpose tool that can be used with any database module. On-screen browsers and graphical query interfaces are good examples. Such an application has no built-in knowledge of the schemata on which it is to operate. What it does know is how to extract such information from metadata at *run-time*. An on-screen browser, for example, might need to know the names of all top-level entity classes or all the attributes of a particular entity type. A graphical query interface might want to know all the relationships in which an entity type participates or the names of the functions that make up its key. A uniform interface to metadata at this level would allow schema information to be retrieved in the following way (the metadata schema used here is given in Appendix II):

Produce a list of all top-level entities (called TopLevelEnts), i.e. all those for whom the function *supertype* is undefined

```
findall(TopLevelEnt, (
        getentity(entmeta, TopLevelEnt),
        \+ getfnval(supertype, [TopLevelEnt], _)
), TopLevelEnts)
```

Produce a list (called KeyFns) of the functions making up the key of the entity type EntType

```
findall(KeyFn, (
        getfnval(key_component, [EntType], KeyFn)
), KeyFns)
```

Schema-independent application programs often need to be able to re-engineer a description of a schema from its metadata (e.g. recreate the description using the data definition constructs of Daplex). A uniform interface to metadata facilitates this and provides a form of communication protocol for schema definitions.

We have already pointed out that uniformity will allow schema-independent applications, built to operate on raw data, to operate on metadata too. Most usefully, this principle extends to the system utilities such as the Daplex parser and optimiser. Since the Daplex parser converts Daplex queries into Prolog programs including calls to the database access primitives, it is only necessary to achieve uniformity at the Prolog level and uniformity at the Daplex level will automatically follow.

A schema-dependent application is written with a specific schema or set of schemata in mind. The details of the schema (entity class names, function names, for example) are embedded into the code of the application by the programmer. The application program itself does not need to make direct queries to metadata at run-time as all the necessary information is supplied at compile-time.

Although schema-dependent applications make no direct use of metadata, the people creating, maintaining and using such applications do. Users require help when formulating queries (what is the name of the relationship function that maps a protein model onto its versions? What are the types of the arguments to the search_database_for_fragments action?) and when gaining access to data (what shared modules do I have open? Is module ‘‘protein_data’’ open for reading or writing?) Application developers need similar assistance when extending schemata or defining a module that is to be used in conjunction with a pre-existing one (does an entity class called ‘‘version’’ already exist? Does this module contain any cross-module functions?) This is particularly useful where several developers are working on a single application and need to coordinate their work. It is not our aim to provide sophisticated help facilities such as those described in [Gray 88] but to ensure that access to metadata is at least as convenient as access to raw data.

If a uniform interface to metadata is provided, queries such as those given above can be expressed neatly using the Daplex query language:

What are the types of the arguments to the search_database_for_fragments action?

```
for the a in actmeta such that aname(a) = "search_database_for_fragments"
print(oname(act_args(a)));
```

What shared modules do I have open?

```
for each m in modmeta such that mstatus(m) = "shared"
print(mname(m));
```

4. The Implementation of Metadata in P/FDM

Having illustrated the need for both efficient and uniform access to database metadata, we will now describe how this has been achieved in P/FDM. Our original intention was to imitate the strategy used by EFDM [Atkinson 84], an implementation of the FDM in PS-Algol. Here, a set of ordinary Daplex functions are automatically populated as a result of the data definition constructs **declare**, **define** and **drop**. In this way, metadata are stored twice: once in the internal format and once in the standard database format.

While the difficulties of maintaining two copies of data are well known, in P/FDM the problem is compounded by the use of modules. When using EFDM, one interacts with a single database. In a P/FDM environment, on the other hand, several modules of varying types may be in use simultaneously, and extra modules may be opened or closed at any time during a session. Thus, the metadata are subjected to regular bulk insertions and deletions as module schema are loaded and unloaded. Maintaining a database copy of metadata under such circumstances is not trivial and it was thought better to adopt a different approach.

Basically, the structures for storing metadata (the internal descriptors) have been designed for efficiency and the system components access these directly. Uniformity has been provided via a *view* onto these structures. This view is implemented at the level of the database primitives, so that any program operating above this level will be able to make use of it. This scheme has the additional advantage that it insulates application programs from changes to the internal metadata format.

We will first describe some of the issues involved in the design of the internal descriptors and then show how the uniform view has been implemented.

4.1. The ‘‘Efficient’’ Implementation

A request to retrieve raw data causes several components of P/FDM’s DBMS to access metadata. The efficiency of access to metadata has a direct bearing on overall system performance. For this reason, the need for *fast access* is an important factor in the design of metadata storage structures. The nature of metadata determines that they are generally much lower in volume, compared with the raw data they describe. Therefore, to improve access speed, it is necessary to store metadata in a specially designed format, rather than in the same format as raw data. Based on these considerations, we chose to represent metadata in P/FDM as Prolog terms, making use of the first-argument-index facility of Quintus Prolog.

A database in P/FDM can be partitioned into several modules, which are defined using the three building blocks described in Section 2. Therefore, metadata are needed to describe the arrangement of these building blocks within each module.

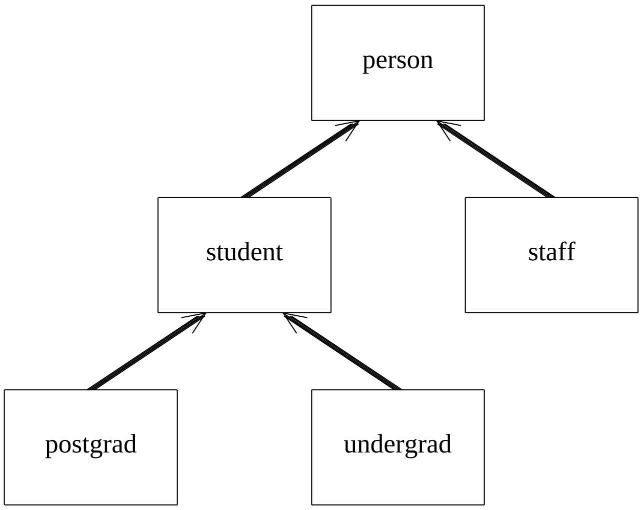

Figure 2 - An Example Inheritance Hierarchy in P/FDM

An **entity class** represents a group of real-world objects which share the same characteristics. P/FDM supports the concepts of *object identity* and *class hierarchies*. An object identifier is a unique, system-generated attribute of an entity instance that is used as an internal identifier. Each entity class also has an associated *key* which is composed of entity properties, and whose values act as external identifiers of that class's instances. A class hierarchy captures the generalisation relationship between a class and its direct and indirect subclasses. In P/FDM, an entity class can have one superclass and several subclasses (see Figure 2), and these classes must reside in the same module. Therefore, metadata about entity classes should describe:

(i) The name of the entity class.

(ii) The cardinality of the entity class, i.e. how many entity instances it has.

(iii) The formation of object identifiers for entity instances.

(iv) How the *key* of the entity class is formed, and its type (i.e. integer, string, etc).

(v) The configuration of entity class hierarchies.

(vi) The module in which the entity class resides.

Entity class metadata are represented by Prolog terms with functor *edesc* (**e**ntity **desc**riptor) of arity seven:

edesc(ClassName, Is-A, KeyType, KeyDesc, NumInst, LastInstID, ModuleName)

The first argument of the term is the name of the entity class. The uniqueness of an entity class name within a module makes it an ideal first argument of *edesc*. The *Is-A* argument is the name of its immediate superclass. The *KeyDesc* describes the entity properties that are used to construct the key of the entity class. Examples of an *edesc/7* to store the metadata about entity classes ‘‘person’’ and ‘‘student’’ are:

edesc(person, entity, string, [forename, surname], 100, 101, db1)

edesc(student, person, string, [forename, surname], 40, 101, db1)

The root of a hierarchy is an ‘‘artificial’’ entity class called **entity**. The key of a person is composed of his or her name and surname. Object identifiers of entity instances are generated by concatenation of the *ClassName* with *LastInstID* + 1, e.g. person(41). The cardinality of an entity class is kept in *NumInst*, which is used by the query optimiser, as explained in Section 3.1.

Functions in P/FDM can be used to model properties of or relationships between entity classes. Some functions may be used in constructing the key of an entity class, which means they should be treated with care in function value update operations. The extension of a function is either stored in a database, or derived at runtime. P/FDM supports function overloading and overriding. *Overloading* allows functions defined on different entity classes to have the same name. *Overriding* means that a subclass can redefine (i.e. specialise) functions defined on its superclass. These useful and powerful concepts require the metadata to provide the following information about each function:

(i) The name of the function.

(ii) The entity classes on which the function is defined (i.e. its argument types).

(iii) Whether function values are stored in a database or derived at runtime.

(iv) Whether the function is used in constructing the keys of some entity classes.

(v) The module in which the function is defined.

(vi) The type of the function, i.e. whether it is a single-valued or multi-valued function.

(vii) The result type of the function. (This is to help type-checking at run-time)

(viii) The function identifier. (This is to facilitate function binding.)

Function metadata are represented by Prolog terms with functor *fdesc* (**f**unction **desc**riptor) of arity eight. The first argument of the term is the name of the function:

fdesc(FName, ArgType(s), ResultType, FType, Status, Inverse, FId, ModName)

For example, the fdesc/8’s for the functions ‘‘surname’’ and ‘‘age’’ defined on the class ‘‘person’’ are:

fdesc(surname, [person], string, single, key-function, no-inverse, 2, db1)

fdesc(age, [person], integer, single, optional, no-inverse, 3, db1)

Although function name overloading is allowed in P/FDM, duplication of function names is still less likely to occur than duplication of the other attributes of function metadata. Therefore, we chose the function name as the first argument of the *fdesc* structures. The value of *FType* is either ‘‘single’’ or ‘‘multiple’’. The *Status* argument indicates whether a function is used in constructing a key (*key-function*), or whether it is a derived function (*method*) or just an ordinary function (*optional*). If a function has an inverse, the value of *Inverse* will be ‘‘has-inverse’’.

For query optimisation purposes, the *fan-out factor* (cardinality) of each set-valued function (i.e. the number of instances the result set contains) is also recorded by the P/FDM system. This metadata is stored in a Prolog term:

function_cardinality(FName, ArgType(s), Cardinality)

An **action** in P/FDM is a piece of code which does not return a specific value as result, but which performs some operation. The system requirements on the action metadata are:

(i) The name of the action.

(ii) The entity classes to which the action will be applied (i.e. its argument types).

(iii) The module in which the action is defined.

(iv) The action identifier. (To ease the binding of an action name to its definition.)

The term structure representing metadata about actions is *adesc* (**a**ction **desc**riptor) of arity four:

adesc(ActionName, ArgType(s), ActionId, ModuleName)

A **module** is the largest grain of storage in P/FDM. There are three types of module: *shared*, *private*, and *temporary*. Shared and private modules are persistent, i.e. they are stored on disk. A temporary module resides in memory and the data it contains last for only one session. A module must know how to generate unique identifiers for functions and actions defined in the module. Thus metadata about modules include:

(i) The module name.

(ii) The module type (shared, private, or temporary).

(iii) The generators for new function and action identifiers.

Metadata about modules are represented by Prolog terms with functor *mdesc* (**m**odule **desc**riptor) of arity four:

mdesc(ModuleName, ModuleType, NextFunctionId, NextActionId)

where the *NextFunctionId* and the *NextActionId* are used to generate identifiers whenever new functions or actions are defined.

Metadata are stored with the database modules in which they are defined, and copied into the Prolog clause base when modules are opened for a session. To ensure the validity of metadata, metadata storage structures are known only to the various ‘‘trusted’’ components of P/FDM’s DBMS, such as the parser, query optimiser, and primitives. These special structures should be shielded from users to prevent them accessing and updating metadata directly. In [Moffat 86] this protection was achieved by using the Prolog ‘‘module’’ facility, i.e. metadata structures were kept in system modules rather than user modules, so that a user could not operate on metadata directly unless he or she knew the names of system modules. Since we do not distinguish system modules and user modules in P/FDM, we provide an interface which allows users some limited access to metadata while preserving its integrity. This interface is described below.

4.2. The ‘‘Uniform’’ Implementation

In providing a uniform interface to P/FDM metadata it is only necessary to implement it at the level of Prolog and the database access primitives, as this will automatically confer uniformity of access at the Daplex level. Such an interface must provide a view onto the internal descriptors so that, above the level of the primitives, metadata appears to be stored in an ordinary temporary module. Each user will have their own private version of this special module; it is stored in memory and therefore cannot be shared. The implementation consists of two parts:

- the definition of a schema for the virtual metadata module using the standard data definition language for raw data, and
- the redefinition of the database access primitives so that they access the internal descriptors if the object on which they are operating has been defined in the special metadata module.

A diagrammatic representation of the metadata schema currently in use is given in Figure 3, while the full version, expressed in the Daplex data definition language, is given in Appendix II. As can be seen from the diagram, this schema is also a description of the P/FDM data model, showing the relationships between objects, functions, actions and modules. Each application schema can be thought of as an instantiation of this ‘‘data model’’ schema.

On entering the P/FDM system each user finds that a temporary module called *metadata* is already open. As modules are opened and closed, internal descriptors will be created and deleted, and the contents of the metadata module will apparently change to reflect this. In other words, the special module contains

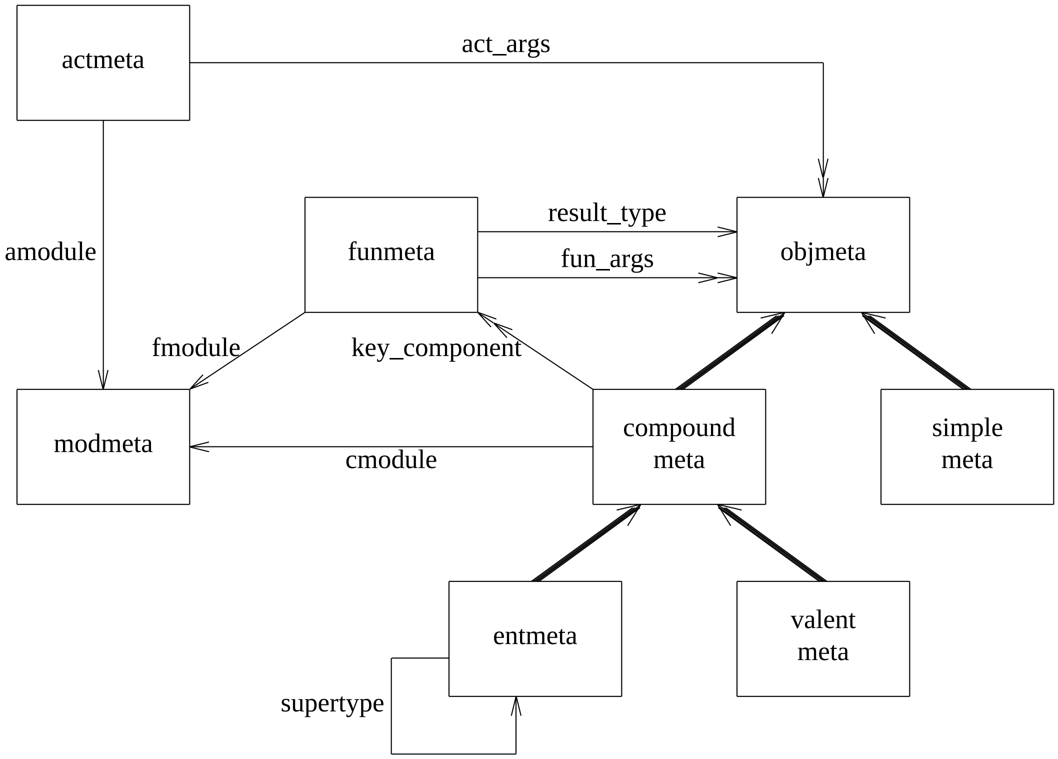

Figure 3 - Diagrammatic Version of the Metadata Schema
(Thick arrows indicate sub-supertype relationships)

metadata for all modules currently open. The advantage of this is that we do not need to provide a mechanism to manage concurrent updates to metadata.

In the current version of P/FDM the only way to create a new entity class from Daplex is to use the data definition part of the language. When parsed, this will result in the invocation of one of the special primitives provided to update metadata (see Appendix I). With the addition of a metadata schema, the same updates can be expressed using the ordinary update constructs, so that

create a new entmeta with key = (person)

is equivalent to

declare person ->> entity

This is fine for those updates that have direct equivalents in the data definition language as the required database primitives will already exist. However, there is no primitive to perform the following update:

let supertype(the e1 in entmeta such that oname(e1) = "student")
= the e2 in entmeta such that oname(e2) = "university_member";

The system as it stands has no provision for schema evolution other than purely incremental changes; schema elements may be added or deleted but existing ones may not be modified. Since only a subset of the possible updates to metadata would be feasible it was thought better disallow all updates expressed in the data manipulation language. The simplest way to do this is to ensure that the special metadata module can never be opened in *write* mode. In fact, this is doubly useful as it also ensures that the metadata-update primitives can never be used on the metadata module itself. Thus, in constraining the module's status to be *read* we are preventing both direct updates to metadata and indirect updates to the metadata metadata!

The business of the uniform interface, of course, is retrieval; retrieval of entity instances and function values. If the metadata module were an ordinary Daplex module and we wished to discover the name of the superclass of the class called ‘‘student’’ we would use the following Prolog query (result in SuperClassName):

```
% Find the instance of entmeta whose name is ‘‘student’’
getentity(entmeta, [student], ClassInst),
% Find the instance of entmeta which is the supertype of ClassInst
getfnval(supertype, [ClassInst], SuperClassInst),
% Find the name of the superclass
getfnval(oname, [SuperClassInst], SuperClassName)
```

The DBMS, with direct access to the internal descriptors, would express the equivalent query as:

```
edesc(student, SuperClassName, _, _, _, _, _)
```

This is a very graphic illustration of how uniformity clashes with efficiency! Why is the uniform version so verbose compared with the efficient version? Part of the problem is that the two parts of the system use different identifying schemes. Entity instances each have their own unique identifier, of the form

<class_name>(<identifying_integer>)

(e.g. person(243), protein(18)) whereas the internal descriptors are identified by the names of the objects that they represent (e.g. person, protein). We need to simulate entity instances in the metadata module and so we must invent some kind of indentifier for them. We need something with a unique value, not confusable with ordinary object identifiers and which complements the identification scheme already in use among the internal descriptors. We use a compromise solution that borrows something from both styles of identifier. We retain the class name as the functor, except that now we are using the name of a metadata class (e.g. objmeta, entmeta). For the argument we use the identifying values of the internal descriptors (i.e. the key of each metadata entity class). This gives identifiers of the form

<metadata_class_name>(meta_id(<metadata_instance_key>))

So, the entity class called ‘‘person’’, for example, would have

entmeta(meta_id(person))

as its identifier, because *entmeta*s are keyed on the function *oname* - the name of the entity class. Functions are keyed on their name and first argument type so the function *courses(undergrad)* would be identified by

funmeta(meta_id(courses, undergrad))

This illustrates the purpose of the *meta_id* term. It allows us to bundle together several values into a single argument for the instance identifier. Notice that this identification scheme is compatible with the existing scheme used within the internal descriptors. While it has all the disadvantages attendant on any attribute-based identification scheme it does have one significant advantage over an integer-based method. Namely that the existing ordinary data routines, expecting integer arguments, should fail automatically if given metadata instance identifiers to work with, and vice versa for the metadata predicates.

Now that we have an identifying scheme, how are we to simulate the presence of metadata instances? Each internal descriptor will represent one instance of some metadata class. For example, the set of *fdesc* descriptors represents all the instances of the metadata class *funmeta*. Similarly with the *adesc*s and the *mdesc*s which represent the instances of the *actmeta* and *modmeta* classes respectively. The classes of the *objmeta* hierarchy (see Figure 3) require a little more care. The set of *edesc* internal descriptors actually represents the union of the sets of *entmeta* instances and *valentmeta* instances. *Edesc*s representing value entities have ‘‘value’’ as their KeyType (see Section 4.1) while those representing full entities have either ‘‘string’’ or ‘‘integer’’. We have introduced a new set of internal descriptors - *sdesc/1* - to represent the instances of the set of simple types. Their function is merely to record the presence of the type, and we have no more information to record about them than their name. In the current system there are three such types:

sdesc(string)
sdesc(integer)
sdesc(float)

and since there is no facility to create simple types dynamically these three terms are asserted when the system is bootstrapped. The remaining two classes in the *objmeta* hierarchy are *compoundmeta* and *objmeta*. There are no internal descriptors corresponding to these classes and they require special treatment, which shall be described shortly.

To retrieve an instance of a metadata class, then, we must search for an appropriate internal descriptor (following the rules just described) and, from this, generate the corresponding identifier. Suppose, for example, that we wish to enumerate instances of the *modmeta* class. First, we look for an *mdesc* term

mdesc(ModuleName, _, _, _)

and then use the module name to generate the identifier

modmeta(meta_id(ModuleName))

This process is basically the same for all metadata classes: find an internal descriptor that matches the current requirements and extract enough data from it to form the identifier. The *get_meta_instance/2* predicate given in Appendix III implements this behaviour.

However, there is one further complication. P/FDM, unlike many semantic data models, supports *multiple subclass membership*. This allows a natural representation of, for example, staff members who are also students, and swimmers who also play golf. To provide this facility each conceptual instance is represented by a set of instances, arranged into an *instance hierarchy*. So, given the inheritance hierarchy in Figure 2, an undergraduate student might be represented by instances with identifiers

person(45)
student(45)
undergrad(45)

and a staff member who is also a postgraduate student by

person(67)
staff(67)
student(67)
postgrad(67)

Notice that the integer identifier remains constant within each instance hierarchy. In other words, it identifies the conceptual instance while the functors indicate the classes of which that instance is a member.

For retrieval purposes the metadata module must appear to be an ordinary module and therefore must appear to contain fully-populated instance hierarchies. The class ‘‘protein’’, for example, would be represented by three instances with identifiers

objmeta(meta_id(protein))
compoundmeta(meta_id(protein))
entmeta(meta_id(protein))

However, there is only one internal descriptor corresponding to the entity class ‘‘protein’’ - its *edesc*. The remaining instance identifiers must be generated from this. The algorithm is very simple. To enumerate all instances of a metadata class

(i) retrieve all instances for which appropriate internal descriptors exist, and

(ii) retrieve all instances of all subclasses and *convert their identifiers to ones appropriate to the class being enumerated*.

This conversion of identifiers is important. The identifiers of subclass instances will have the name of that subclass as their functor and we require identifiers with the name of the class being queried as a functor. We cannot return entmeta(meta_id(protein)) as the identifier of an instance of *compoundmeta*. It must first be converted to compoundmeta(meta_id(protein)).

So much for the retrieval of metadata instances; what about function values? Instance retrieval is concerned only with the presence of internal descriptors. In retrieving the results of metadata functions we are extracting *values* from particular internal descriptors. Each metadata function takes an instance of some metadata class as its argument. This will identify the particular internal descriptor from which the result will be taken. *mstatus/1* (i.e. module status) is typical and its definition is shown in Figure 4 as an example. The argument gives the name of a module, which is used to retrieve an internal descriptor. The second argument of the internal descriptor contains the module’s status and this is returned as the result of the function.

```
% mstatus(+ListOfArgumentTypes, +ListOfFormalParameters, ?Result)
mstatus([modmeta], [modmeta(meta_id(MName))], MStatus) :-
    mdesc(MName, MStatus, _, _).
```

Figure 4 - Prolog Definition of the Metadata Function mname/1

Inverse functions are easy to define thanks to the declarative nature of the definitions. The body of the clause remains unchanged but the argument specification is exchanged with the result specification in the head. As an example, consider the definitions of the function *cmodule/1* and its inverse *cmodule_inv* given in Figure 5.

```
cmodule([compoundmeta], [compoundmeta(meta_id(CName))],
                                    modmeta(meta_id(MName))) :-
    edesc(CName, _, _, _, _, _, MName).

cmodule_inv([modmeta], [modmeta(meta_id(MName))],
                                    compoundmeta(meta_id(CName))) :-
    edesc(CName, _, _, _, _, _, MName).
```

Figure 5 - Prolog Definitions of the Function cmodule/1 and its Inverse cmodule_inv/1

5. The Role of Prolog in the Implementation

The implementation of the uniform view onto metadata described here, required a small amount of new code to be written (approximately 500 lines of Prolog code) and, perhaps most significantly, only a few changes to the existing system. *No changes have been made to either the Daplex parser, the query optimiser or the code generator*. This was due, in part, to the narrow and well-defined interface between application programs and the database system. While it was this feature of the P/FDM architecture that kept changes to existing code low, the brevity of the additional code owes much to the choice of Prolog as the implementation language.

What are the features of Prolog that have helped to keep these routines concise? Firstly we have the simple but expressive type system. Prolog atoms can be used in much the same way as elements of an enumerated type in a standard procedural language but without the attendant syntactical baggage. Of course, we lose the security of compile-time checks but we gain in generality of use and the ease with which new atoms may be introduced at run-time. The fixed nature of enumerated types in most procedural languages reduces their usefulness in practice. Dynamic type checking has another advantage: namely that it is very easy to extend a procedure to take a different type of value for an *existing* argument. For example we have been able to extend the database primitives to deal with metadata instance identifiers as well as ordinary instance identifiers without having to introduce complicated union or variant type declarations. Many procedural languages do not even allow such types because they cannot be statically type-checked. Even though system developers may want the flexibility of dynamic type-checking, database users need more security. Thus, though our query language sits on top of weakly-typed Prolog, the Daplex language itself is strongly typed and can give the extra protection required for user's queries.

Secondly, Prolog provides several powerful operators for constructing and taking apart term structures and pieces of Prolog code. This feature has been used often in the implementation of the uniform view in, for example, the construction of instance identifiers for metadata instances (see *getentity/3* in Appendix III). Since Prolog code is stored as an ordinary term structure we can build and execute Prolog programs at run-time. This gives us the power of a callable compiler but without the difficulties of producing type declarations for the code we are to generate.

Thirdly, we have found that Prolog queries are simplified if the database predicates enumerate instances and function values one at a time by backtracking, rather than producing (possibly large) lists of results. This allows a kind of lazy evaluation over the large sets resulting from many database queries. Queries are not obscured with the details of list processing, and recursive definitions are often avoided in favour of failure-driven loops.

6. Conclusion

We have described the factors affecting the design of storage structures for database metadata, and have shown how the needs of the DBMS and the needs of its users conflict. The DBMS requires efficient access to metadata in order to satisfy the user's demand for efficient access to raw data. Database users require a uniform access to both raw data and metadata in order to simplify their interaction with the DBMS. Here, we propose a compromise solution that provides a uniform interface to metadata via a view onto more efficient data structures. We have been able to implement this solution due to the following three properties of P/FDM:

(i) the data model is flexible enough to be able to describe itself

(ii) access to the database proper is limited to a small set of well-defined primitives which hide the implementation of the storage system

(iii) the concept of a *module of definition* for objects provides a convenient handle by which to decide which version of a primitive is required.

The last two properties work in tandem. The restricted interface means that application programs are independent of the details of the storage system. Modules allow several different storage systems to be accessed by a single database query.

Uniformity has increased the usefulness of database metadata by making it safer and easier to access. Moreover, since the same query interface is used by both users and application programs alike, uniformity allows existing schema-independent applications to operate on metadata without them having to be changed in any way. Perhaps most usefully, this extends to system applications like the Daplex parser and optimiser. Since Daplex queries are ‘‘compiled’’ into Prolog queries that operate above the level of the database access primitives, they will automatically invoke the metadata versions at run-time.

Acknowledgements

Zhuoan Jiao is supported by a British Council Technical Co-operation Training Award. Suzanne Embury is supported by a grant from the SERC Biotechnology Directorate. We would like to thank Graham Kemp and Oscar Diaz for their helpful comments.

Appendix I - Primitives Provided by P/FDM

(i) Primitives that operate on raw data

getentity(+EName, -InstOid):
given the name of an entity class, this primitive enumerates over the entity class and returns the object identifiers *InstOid* of the instances one by one on backtracking.

getentity(+EName, +Key, -InstOid):
given the name of an entity class and the key value of an instance of that class, this primitive directly accesses the instance using the key value and returns its object identifier.

getfnval(+FName, +Argument(s), -Value):
returns the value of the given function with the given argument(s), whether stored or derived. If the function is multi-valued, then it will return the values one by one on backtracking.

perform(+AName, +Argument(s)):
perform the action with the given argument(s).

newentity(+EName, +Key, -InstOid):
given the name of an entity class and the key value of a new instance, add the instance into the module in which the entity class is defined.

addfnval(+FName, +Argument(s), +Val):
include *Val* in the set of results of the function *FName* applied to *Arguments*. If *FName* is single-valued then *Val* becomes its only result.

updatefnval(+FName, +Argument(s), +OldVal, +NewVal):
modifies the value of function *FName* from *OldVal* to *NewVal*.

deletefnval(+FName, +Argument(s), +Val):
if the given function does not form part of a key of an entity class, delete *Val* from the set of its results. (Key function values may be removed only by deletion of the instance.)

(ii) Primitives that operate on metadata

new_entity_class(+EDesc):
given the descriptor of an entity class, this primitive creates the entity class in a module.

new_function(+FDesc):
given the descriptor of a function, this primitive generates a new function in a module.

new_action(+ADesc):
given the descriptor of an action, this primitive generates a new action in a module.

new_module(+MDesc):
given the description of a module, this primitive creates a new module for a database.

NB: Refer to Section 3.1 for information on *EDesc*, *FDesc*, *ADesc* and *MDesc*.

delete_entity_class(+ClassName):
removes an entity class with the name *ClassName* from its module of definition.

function_delete(+FName, +ArgType(s), +Res):
given the name of a function, its argument type(s), and result type, this primitive deletes that function from its module of definition.

action_delete(+AName, +ArgType(s)):
given the name of an action, its argument type(s), this primitive deletes that action from its module of definition.

(iii) Primitives that operate on modules

open_module(+ModuleName, +Mode):
opens a module in either 'read' or 'write' mode.

close_module(ModuleName):
is used to close a module. If there are any changes to the module, it will commit the changes.

close_modules:
is similar to the above *close_module* primitive, but closes all modules that remain open.

Appendix II - Daplex Definition of the Metadata Schema

```
declare modmeta ->> entity
declare mname(modmeta) -> string
declare mstatus(modmeta) -> string
key_of modmeta is mname


declare objmeta ->> entity
declare oname(objmeta) -> string
key_of objmeta is oname

declare simplemeta ->> objmeta

declare compoundmeta ->> objmeta
declare cmodule(compoundmeta) -> modmeta

declare entmeta ->> compoundmeta
declare supertype(entmeta) -> entmeta
declare num_inst(entmeta) -> integer

declare valentmeta ->> compoundmeta


declare funmeta ->> entity
declare fname(funmeta) -> string
declare first_fun_arg(funmeta) -> objmeta
declare fun_args(funmeta) ->> objmeta
declare card(funmeta) -> string
declare fstatus(funmeta) -> string
declare has_inverse(funmeta) -> string
declare result_type(funmeta) -> objmeta
declare fmodule(funmeta) -> modmeta
key_of funmeta is key_of(first_fun_arg), fname


declare actmeta ->> entity
declare aname(actmeta) -> string
declare first_act_arg(actmeta) -> objmeta
declare act_args(actmeta) ->> objmeta
declare amodule(actmeta) ->> modmeta
key_of actmeta is aname, key_of(first_act_arg)


declare key_component(entmeta) ->> funmeta;
```

and the auxiliary function definitions

```
define subtype(e in entmeta) ->> entmeta in metadata
        s in entmeta such that e in supertype(s);

define subtypes(e in entmeta) ->> entmeta in metadata
        (subtype(e) union subtypes(subtype(e)));
```

```
define supertypes(e in entmeta) ->> entmeta in metadata
        (supertype(e) union supertypes(supertype(e)));

define functions_on(o in objmeta) ->> funmeta in metadata
        f in funmeta such that o in fun_args(f);

define functions_on(e in entmeta) ->> funmeta in metadata
        (functions_on(supertype(e))
        union
        f in funmeta such that e in fun_args(f) as entmeta);

define functions_yielding(o in objmeta) ->> funmeta in metadata
        f in funmeta such that result_type(f) = o;

define num_f_args(f in funmeta) -> integer in metadata
        count(fun_args(f));

define actions_on(o in objmeta) ->> actmeta in metadata
        a in actmeta such that o in act_args(a);

define actions_on(e in entmeta) ->> actmeta in metadata
        (actions_on(supertype(e))
        union
        a in actmeta such that e in act_args(a) as entmeta);

define num_a_args(a in actmeta) -> integer in metadata
        count(act_args(a));
```

and finally an action with a Prolog definition

```
define print_key(entmeta)

print_key([entmeta], [Entmeta]) :-
    write('key_of '),
    getfnval(oname, [Entmeta], Oname),
    write(Oname),
    write(' is '),
    getfnval(key_component, [Entmeta], First_Component),
    print_first_component(First_Component),
    !, % Cut to ensure that only one component is treated as the first
    (
      getfnval(key_component, [Entmeta], Component),
      Component \== First_Component,
      print_component(Component),
      fail
    ;
      nl,
      true
    ).
```

```
print_first_component(Function) :-
        print_function(Function).

print_component(Function) :-
        write(', '),
        print_function(Function).

/* Print a key indirection (ie. the result of the key function is another entity class) */

print_function(Function) :-
        getfnval(result_type, [Function], Result),
        getfnval(oname, [Result], Result_type),
        getentity(entmeta, [Result_type], _),
        !,
        getfnval(fname, [Function], Fname),
        write('key_of('),
        write(Fname),
        write(')').

/* Print a straightforward key component */

print_function(Function) :-
        getfnval(fname, [Function], Fname),
        write(Fname).
```

Appendix III - Definitions of Metadata Primitives

Below are given the Prolog definitions of the metadata versions of *getentity/2* and *getentity/3*.

```
% getentity(+EntityClassName, ?InstanceIdentifier)

getentity(EName, Inst) :-
    nonvar(EName),
    edesc(EName,_,_,_,_,_,metadata),
    (
        get_meta_instance(EName, Inst)
    ;
        transitive_sub_super_relation(SubClass, EName),
        get_meta_instance(SubClass, SubClassInst),
        related_identifier(EName, SubClassInst, Inst)
    ).
```

```
% getentity(+EntityClassName, +KeyValues, ?InstanceIdentifier)

getentity(EName,Key,InstID) :-
   nonvar(EName),
   nonvar(Key),
   edesc(EName,_,_,_,_,_,metadata),
   InstKey =.. [meta_id | Key],
   InstID  =.. [EName | [InstKey]],
   getentity(EName, InstID).


% transitive_sub_super_relation(?SubClass, ?SuperClass)

transitive_sub_super_relation(SubClass,SuperClass):-
  edesc(SubClass,SuperClass,_,_,_,_,_).

transitive_sub_super_relation(SubClass,SuperClass):-
  edesc(SubClass,Super1,_,_,_,_,_),
  transitive_sub_super_relation(Super1,SuperClass).


% related_identifier(+Class, +OtherClassInst, ?ClassInst)
%
% ClassInst is an identifier for an instance of class Class that corresponds to the instance
% identifier OtherClassInst. For example, if student is a subclass of person
%
%          related_identifier(person, student(6), X)
%
% will cause X to become instantiated to person(6)

related_identifier(Class, OtherClassInst, ClassInst) :-
   nonvar(Class),
   nonvar(OtherClassInst),
   OtherClassInst =.. [_ | ID],
   ClassInst =.. [Class | ID].
```

```
% get_meta_instance(+EName, ?Inst)
%
% This predicate should return successive base instances (Inst) of the meta-entity EName,  and
% finally fail.  An instance is a 'base instance' of a class if there are no corresponding instances
% of any of the sub-classes of that class.  This is achieved by using narrow selection  conditions
% rather than an explicit test for lack of subclass instances (e.g. since no instances  of  objmeta
% are base instances the selection condition is infinitely strong).
% It can also be used to test that Inst is an existing instance of EName.
%
% The general algorithm for this predicate is
%
%       - use a template to fetch appropriate internal descriptors
%       - check conditions are met
%       - construct identifier
%
% N.B. for those classes with a module attribute it is necessary to explicitly disallow anything
% that has been defined in the special module 'metadata'.  We do not want to see the metadata
% metadata.

get_meta_instance(modmeta, modmeta(meta_id(ModName))) :-
        mdesc(ModName, _, _, _),
        ModName \== metadata.

get_meta_instance(objmeta, _) :-
        fail.

get_meta_instance(compoundmeta, _) :-
        fail.

get_meta_instance(simplemeta, simplemeta(meta_id(Name))) :-
        sdesc(Name).

get_meta_instance(entmeta, entmeta(meta_id(Name))) :-
        edesc(Name, _, KeyType, _, _, _, Module),
        Module \== metadata,
        KeyType \== value.

get_meta_instance(valentmeta, valentmeta(meta_id(Name))) :-
        edesc(Name, _, value, _, _, _, Module),
        Module \== metadata.

get_meta_instance(funmeta, funmeta(meta_id(Name, FirstArg))) :-
        fdesc(Name, [FirstArg | _], _, _, _, _, _, Module),
        Module \== metadata.

get_meta_instance(actmeta, actmeta(meta_id(Name, FirstArg))) :-
        adesc(Name, [FirstArg | _], _, Module),
        Module \== metadata.
```

References

Atkinson, M.P. and Kulkarni, K.G. (1984)
Experimenting with the Functional Data Model *in* Stocker, P.M., Gray, P.M.D. and Atkinson, M.P. (eds.) *Databases - Role and Structure*, Cambridge University Press, pp. 311-338.

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S. (1989)
The Object-Oriented Database System Manifesto. *Rapport Technique Altair 30-89*.

Fox, M.S. and McDermott, J. (1986)
The Role of Databases in Knowledge-Based Systems *in* Brodie, M.L. and Mylopoulos, J. (eds.) *On Knowledge Base Management Systems*, Springer-Verlag, pp. 407-430.

Gray, P.M.D., Storrs, G.E. and du Boulay, J.B.H. (1988)
Knowledge Representation for Database Metadata *in AI Review* (1988)**2**, pp. 3-29.

Gray, P.M.D., Kulkarni, K.G. and Paton, N.W. (1992)
Object-Oriented Databases: a Semantic Data Model Approach, *Prentice-Hall* (to appear).

Jiao, Z. (1990)
Modules and Temporary Data in P/FDM *Technical Report AUCS/TR9016*, Aberdeen University.

Jiao, Z. and Gray, P.M.D. (1991)
Optimisation Of Methods In A Navigational Query Language *in Proc. 2nd International Conference on Deductive and Object-Oriented Database Systems*, December 1991, Germany.

Kemp, G.J.L. (1991)
Protein Modelling: a Design Application of an Object-Oriented Database *in* Gero, J.S. (ed.) *Proc. 1st International Conference on Artificial Intelligence in Design*, Butterworth Heinemann, pp.387-406.

Maier D. (1986)
Databases in the Fifth Generation Project: Is Prolog a Database Language? *in* Ariav, G. and Clifford, J. (eds.) *New Directions for Database Systems*, Ablex Publishing Corporation, pp.18-34.

Moffat, D.S. and Gray, P.M.D. (1986)
Interface Prolog to a Persistent Data Store *in 3rd International Conference on Logic Programming*, London, 1986.

Paton, N.W. and Gray, P.M.D. (1988)
Identification of Database Objects by Key *in* Dittrich, K.R. (ed.) *Advances in Object-Oriented Database Systems: Proc. OODBS-II*, Springer-Verlag, pp.280-285.

Selinger, P.G. et al. (1979)
Access Path Selection in a Relational Database Management System *in* P.A. Bernstein (ed.) *Proc. ACM SIGMOD79 Conf*, Boston.

Shipman, D.W. (1981)
The Functional Data Model and the Data Language DAPLEX *in ACM Trans. on Database Systems* **6**(1), pp. 140-173.

Warren, D.H. (1981)
Efficient Processing of Interactive Relational Database Queries Expressed in Logic *in Proc. 7th VLDB*, 1981.

Zdonik, S. and Maier, D. (1990)
Fundamentals of Object-Oriented Databases *in* Zdonik, S. and Maier, D. (eds.) *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, pp. 1-32.