O_2Views
User Manual

VERSION 2 - JANUARY 1995

The software described in this document is freely available.

Warning: This is a prototype. There is absolutely no warranty of any kind and you should use it at your own risk.

Contents

List of Figures

1 Introduction

O_2Views is a tool allowing the definition and implementation of views for the object-oriented database system O_2 [BDK92]. Once defined, a view can be used for querying and updating the database. The tool has been implemented with the O_2C programming language on top of the O_2 system.

This manual focuses on the use of the O_2Views prototype and do not go into detail on the view mechanism and its motivations, nor on the implementation of the prototype itself. For more details, the reader is referred to [AB91, SDA94, dS94].

In addition, the reader is supposed to know the O_2 data model as well as the O_2 system [Tec94]. Knowledge of another OODBMS must be sufficient to follow the manual but in order to be able to use the prototype some basic knowledge of the O_2 system is required.

The software presented in this manual is a preliminary prototype freely available and we provide absolutely no warranty of any kind, the user being warned to use it at his or her own risk. In particular, performance has not been an issue in the development of the current prototype, a more efficient and extended version being under development.

Before presenting the language and the prototype implementing it, we give an overview of the view mechanism. O_2 related concepts are taken for granted.

A view in O_2Views is defined as a virtual schema derived from another (real or virtual) schema, called the root schema of the view. Views can therefore be composed and nested to any degree. A virtual base corresponds to the image of a root base (a base instantiated from the root schema) through the view. Figure 1 illustrates the relationship between virtual entities and their corresponding roots.

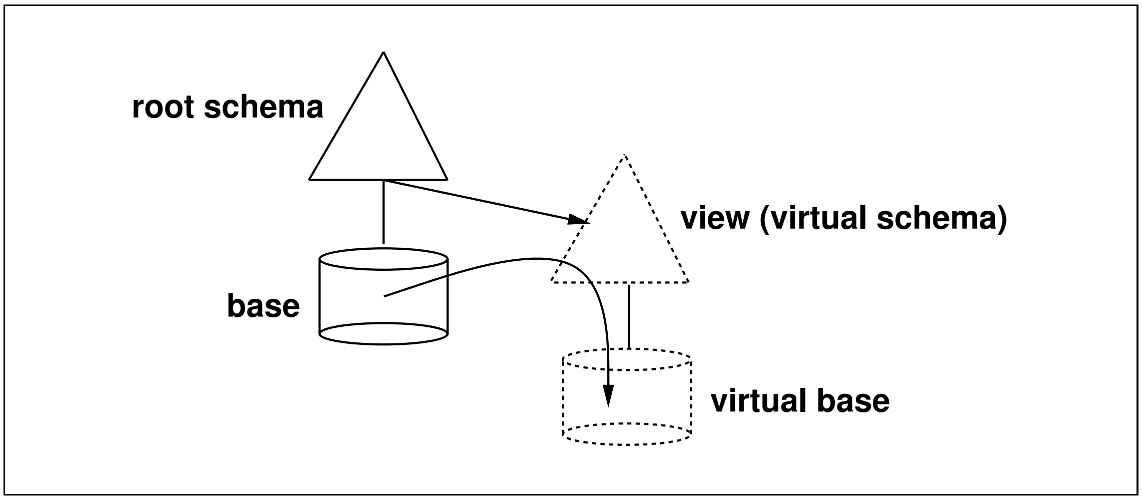

Figure 1: Relationship among schemas and bases

A virtual schema is defined using an extension of the O_2C language, the O_2Views language. This language provides a set of primitives for defining virtual schemas and its syntax and semantics will be informally described in Section 2.

Once defined, a virtual schema definition must be compiled in order to be used. The O_2Views compiler loads the view definition into an O_2 system and stores it as an object in a special base (a view repository). Once loaded in an O_2 system, O_2Views can generate the implementation of a view, which is a standard O_2 schema with some system-supplied classes, methods and functions implementing the view definition. Such schema can then be used as any other O_2 schema and, in particular, different applications can be defined on it. The process of defining, loading and generating the implementation of a view is depicted in Section 3 and its use is explained in Section 4. Section 5 gives a complete example and Section 6 describes how to install O_2Views and how views should be maintained in an O_2 system.

If you have any question, comment or suggestion on the prototype, they are very welcome and can be sent to Cassio.Souza@inria.fr. Bugs, if any, can be reported to the same address.

2 The O_2Views Language

The O_2Views language is an extension of O_2C for defining views. In what follows, we present the syntax of the commands of the language and explain the semantics informally. These commands extend the set of commands provided by the O_2C language for schema development, but they can be used only in the definition of virtual schemas. The O_2Views commands must be compiled by the O_2Views compiler and cannot be directly submitted to the standard O_2C compiler. The O_2Views compiler generates standard O_2C code implementing a virtual schema definition, which can then be loaded into an O_2 system.

As the language extension is conservative, the same general features of the standard O_2 commands syntax apply (case-sensitiveness, semicolon as command separator, comments between /* and */).

The same conventions used in the O_2 User Manual for syntax notation and naming will be used here. We assume the reader knows these conventions.

2.1 The Concrete Syntax of the O_2Views Language

The concrete syntax will be given using the EBNF notation. A vertical bar (|) represents a choice, braces ({ and }) represent zero or more repetitions and brackets ([and]) represent an optional construct. Terminals are quoted. Non-terminals are given in italics.

virtual_schema ::= "`virtual schema`" *schema_name* "`from`" *schema_name* "`;`"
    [*command_list*]
    ["`end virtual schema`"]

command_list ::= *command* "`;`"{*command* "`;`" }

command ::= *import_command*
    | *hiding_command*
    | *virtual_class_command*
    | *imaginary_class_command*
    | *virtual_attribute_command*
    | *virtual_name_command*
    | *type_command*
    | *renaming_command*

import_command ::= "`import schema`" *schema_name* "`class`" *class_name*
    | "`import base`" *base_name* "`real name`" *name*
    | "`import base`" *base_name* "`virtual name`" *name* "`:`" *type_expression*

virtual_class_command ::= "`virtual class`" *class_name* [*signature*]
    "`from`" *class_name*
    ["`inherit`" *class_name_list*]
    ["`virtual attributes`" *virtual_attribute_list*]

["hide attributes" *attribute_name_list*]
["rename attributes" *renaming_list*]
[*population_clause*]
[*method_clause*]
"end"

imaginary_class_command ::= "imaginary class" *class_name* [*signature*]
["key attributes" *key_attribute_list*]
["virtual attributes" *virtual_attribute_list*]
["hide attributes" *attribute_name_list*]
"with extension" *extension_name* "=" *query_expression* ";"
[*method_clause*]
"end"

virtual_attribute_command ::= "virtual attribute" *attribute_name* ":" *type_expression*
"in class" *class_name* "=" *query_expression* ";"

hiding_command ::= "hide" *attribute_name_list* "in class" *class_name*

virtual_name_command ::= "virtual name" *virtual_name* [*signature*] ":" *type_expression*
"=" *query_expression* ";"

type_command ::= "type" *type_name* ":" *type_expression*

renaming_command ::= "rename attribute" *attribute_name* "as" *attribute_name*
"in class" *class_name*

population_clause ::= "with extension" *extension_name* "=" *query_expression* ";"
| "if" *boolean_expression* ";"

method_clause ::= "method" *method_decl_list*

method_decl_list ::= *method_decl* { "," *method_decl* }

method_decl ::= ["public"] *method_name signature* ":" *type_expression*

parameter_list ::= *parameter* { "," *parameter* }

parameter ::= *parameter_name* ":" *type_expression*

signature ::= "(" *parameter_list* ")"

class_name_list ::= *class_name* { "," *class_name* }

virtual_attribute_list ::= *virtual_attribute* { *virtual_attribute* }

virtual_attribute ::= *attribute_name* ":" *type_expression* "=" *query_expression* ";"

attribute_name_list ::= *attribute_name* { "," *attribute_name* }

renaming_list ::= *renaming* { "," *renaming* }

renaming ::= *attribute_name* "as" *attribute_name*

key_attribute_list ::= *key_attribute* { "," *key_attribute* }

key_attribute ::= *attribute_name* ":" *type_expression*

The terminals *schema_name*, *class_name*, *attribute_name*, *virtual_name*, *extension_name*, *type_name* and *base_name* follow the naming conventions of O_2. The terminal *query_expression* represents a regular O_2SQL expression and the terminal *type_expression* represents a regular O_2 type expression.

2.2 The O_2Views Commands

Virtual Schemas

- “`virtual schema`” *schema_name* “`from`” *schema_name* “`;`”

This command allows the creation of a virtual schema and must be the first command in a view definition. The name of the virtual schema must not conflit with the name of any existing real or virtual schema or base. The schema name declared after the reserved word `from` is the name of the *root schema* of the virtual schema being defined and must be an existing real or virtual schema. This rule, which we will call “*first define then use*” applies to all commands in the list of commands of a view definition, and the order of definitions is therefore relevant.

Export and Import

- “`import schema`” *schema_name* “`class`” *class_name*

- “`import base`” *base_name* “`real name`” *name*

The semantics of the two import commands above is as defined for O_2 and allows the import of classes from schemas and names from bases respectively. Imports are usually given in the beginning of a virtual schema definition since imported definitions are supposed to be used by other definitions in the view. They can nevertheless be given at any point, provided the “*first define then use*” rule is respected.

Virtual Classes

- “`virtual class`” *class_name* [*signature*]
  “`from`” *class_name*
  [“`inherit`” *class_name_list*]
  [“`virtual attributes`” *virtual_attribute_list*]
  [“`hide attributes`” *attribute_name_list*]
  [“`rename attributes`” *renaming_list*]
  [“`with extension`” *extension_name* “`=`” *query_expression* | “`if`” *boolean_expression*] “`;`”
  “`end`”

Creates a virtual class definition under the virtual schema. The name of the virtual class must not conflict with the name of another class in the virtual schema or the name of an imported class or name. The `from` clause gives the *root class* of the virtual class.

The name through which the extension of the virtual class is accessed in the view is given in the `with extension` clause. If no extension name is given, a default name is provided which follow the following extension name derivation rule: the prefix “`The_`” is appended at the beginning of the class name and an “`s`” is appended at the end.

There are now two ways of defining the population of a virtual class, through a query expression (clause `with extension`) or through a boolean expression (clause `if`).

The query expression (called the *defining query*) defines the population of the virtual class and must return a unique set of instances of the root class (or its subclasses, if any). Type checking is not performed by the O_2Views compiler at this level and only at query evaluation the system will be able to trap ill typed queries.

The boolean expression is an O2C boolean expression (like in an O2C `if` statement) and gives the condition that an instance of the root class must satisfy in order to be able to become an instance of the virtual class, i.e. it gives the characteristic function of the virtual class. If a class is defined through a characteristic function, its extension cannot be retrieved and its objects are converted on demand by passing instances of the root class as parameters to the corresponding conversion function.

If neither the `with extension` nor the `if` clause are used, a default boolean expression `true` is assumed.

If a class signature is provided, the class is generic. In that case, the extension is accessed through a function named *extension_name* which takes the class parameters as input in order to return the set of corresponding objects. Formal parameters can be used in the defining query or boolean expression but then the character "`:`" must be appended to the beginning of each reference to a class parameter.

Methods (behavior) and attributes (structure) are "inherited" through delegation from the root class. In other words, in the current implementation, each instance of a virtual class holds a pointer to an instance of its root class (which can be real or virtual). Methods and attributes which are not defined or redefined in the virtual class are thus automatically delegated to the root object. This can be viewed as a special case of instance inheritance (delegation) which is provided automatically by the compiler so as to provide the same functionality of class inheritance.

In order to loosen the constraints imposed by the "*first define then use*" rule, the definition of a virtual class can be given incrementally, as in standard O_2, so that classes can be mutually dependent (e.g. two virtual classes can depend one on the other through the type declaration of their virtual attributes). The only mandatory clause is the root class one, the others being optional or being able to be given in separate (except for the population and the inheritance clauses, which are optional but can be given only in the virtual class definition command).

A virtual class can inherit form other virtual classes only . The root class of a virtual superclass of a given virtual class C must itself be a superclass of the root class of C. In other words, the subclassing relationship defined in the root class hierarchy must be preserved in the corresponding virtual class hierarchy. Inheritance in the view is aimed at allowing polymorphism to be preserved in the view, as well as incremental "virtualization" of classes. Properties from the superclass(es) of the root class of a virtual class are "inherited" through the root class rather then its virtual superclass(es), except if the property is redefined in a virtual superclass, in which case the virtual redefinition is taken.

Imaginary Classes

- "`imaginary class`" *class_name* [*signature*]
  ["`key attributes`" *key_attribute_list*]
  ["`virtual attributes`" *virtual_attribute_list*]
  ["`hide attributes`" *attribute_name_list*]
  "`with extension`" *extension_name* "`=`" *query_expression* "`;`" "`end`"

Creates an imaginary class definition under the virtual schema. The name of the imaginary class must not conflict with the name of another class in the virtual schema or the name of an imported class or name. The query expression must return a unique set of tuple values whose type is that defined by the key attributes. Type checking is not performed at this level, as for virtual classes. The `with extension` clause works as for virtual classes, that including the rule for deriving default extension names.

To each tuple value returned by the query a new object identifier is associated, so that the tuple values are "objectified". No two objects are created for the same tuple value. This is already guaranteed by the fact that the query must return a unique set of tuples, but further attempts to extend the extension of an imaginary class will not create redundant objects, i.e. if an imaginary object already exists for a given tuple, it will be returned, otherwise a new object is created for that tuple.

An imaginary class can also be generic. In this case, the genericity is implemented in the same way as for virtual classes, as described above.

The definition of an imaginary class can also be given incrementally, for the same reason as for virtual class definitions. Virtual and imaginary classes can be mutually dependent (though the root class clause of the virtual class or through type declarations of their respective virtual and key attributes). View definitions can thus be composed to any extent (intra-view and inter-views), provided the "*first define then use*" rule is observed.

Virtual Attributes

- "`virtual attribute`" *attribute_name* "`:`" *type_expression*
  "`in class`" *class_name* "`=`" *query_expression* "`;`"

Adds a virtual attribute to the virtual or imaginary class given in the clause `in class`. The name of the virtual attribute must not conflict with the name of another virtual attribute defined in the same class, nor with the name of an attribute (real or virtual) defined in the root class of the class where the virtual attribute is defined, in case it is virtual, nor with the name of a key attribute, if this class is imaginary.

The type of the virtual attribute must be given and the query expression giving the value of the virtual attribute (its defining query), must be of the declared type, but, again, type checking is performed only at query evaluation time, i.e. when the view is activated and virtual and imaginary objects are accessed.

Hiding of Attributes

- "`hide`" *attribute_name_list* "`in class`" *class_name*

Hides the attributes listed in the attribute list in the class given in the `in class` clause.

Virtual Names

- "`virtual name`" *virtual_name* [*signature*]"`:`" *type_expression*
  "`=`" *query_expression* "`;`"

Defines a virtual name in the virtual schema. A virtual name is a named value or object defined through a query. Its value is initialized when the virtual base is activated. As classes in the view, a virtual name can also be generic, if it is defined with a signature. In this case, its contents is accessed through a function named after it and taking as input its parameters.

Types

- "`type`" *type_name* "`:`" *type_expression*

Defines a named type in the virtual schema. This command allows the definition of auxiliary types in the virtual schema as in an ordinary O_2 schema. The reason why it is considered as a command of the O_2Views language rather than being inherited from the standard O_2 command language is that in the current implementation, the type closure of a view[1] is not inferred by the system and must be provided along with the view definition.

Renamings

- "`rename attribute`" *attribute_name* "`as`" *attribute_name*
  "`in class`" *class_name*

This command is valid only for virtual classes and is used for renaming attributes "inherited" from the root class of a virtual class. Conventional renaming can be achieved through the renaming command of O_2.

What's New?

We summarize the new features of the O_2Views language with respect to its previous version:

- parameterized virtual and imaginary classes;
- parameterized virtual names;
- inheritance in the virtual schema between virtual classes;
- virtual classes defined through a characteristic function (a predicate) rather than a query;

Some small syntactic modifications were performed, like adding the keyword `end` as a terminator for virtual and imaginary classes, virtual names and virtual attributes definitions and declaring virtual attributes through a separate clause rather than through a class signature (the signature being used to declare class parameters).

3 Defining and Implementing a View

Figure 2 gives an overview of the different stages from view definition to its use. Gray ovals correspond to the modules implemented by O_2Views. This section describes how to define a view through a definition file using the O_2Views language or through the O_2Views graphical interface. Then the generation of a view implementation, the output of the compiler called the view implementation schema, will be explained. The next section explains how the view implementation schema can be used.

The O_2Views compiler can be used in two modes: the interactive and the batch mode. A virtual schema definition, in turn, can be loaded into an O_2 system from a *view definition file* or through the O_2Views graphical interface. We will describe how this tasks can be accomplished in this section.

Before compiling a view definition, the user must load it into an O_2 system. View definitions are loaded in a base instantiated from schema `o2views`. The user can load it from a unix shell or through the graphical interface of the application `View_Management` defined in schema `o2views`, which was designed for maintaining the view repository. Before describing the two loading modes, let us see how a view definition file is composed.

[1] The type closure of a view is the set of type definitions necessary for the compilation of a view. It includes, among others, the root classes of virtual classes.

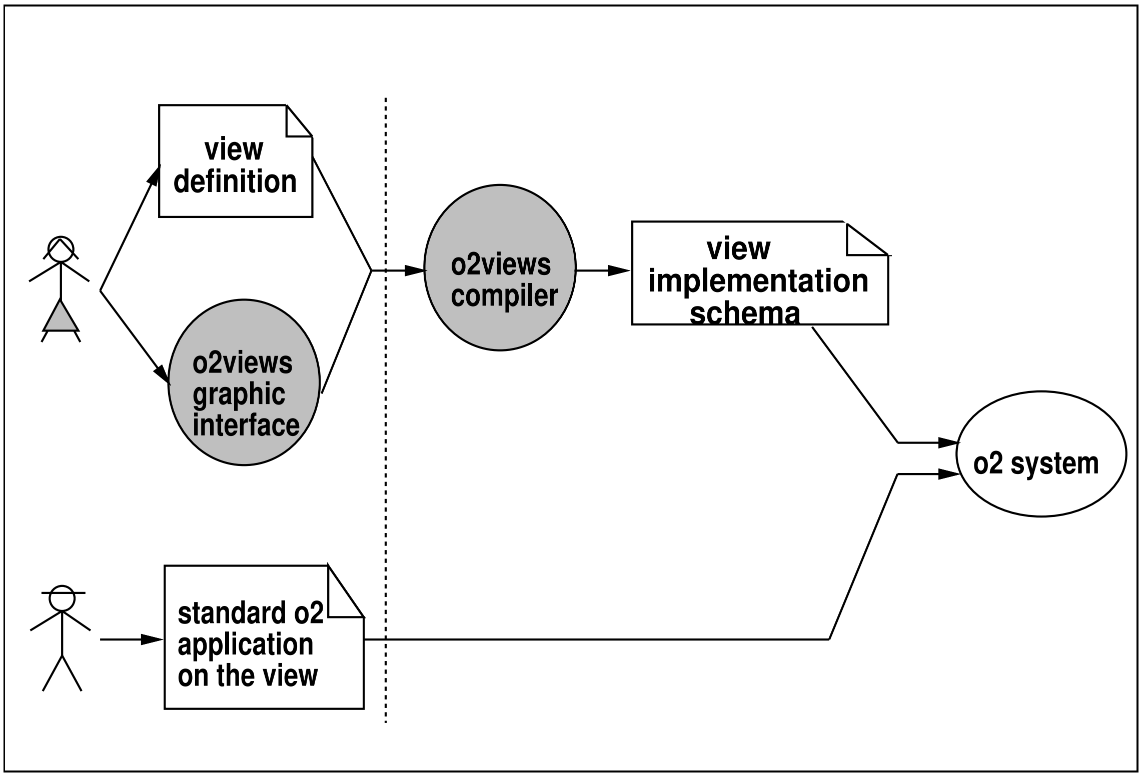

Figure 2: Diffenrent stages from view definition to its use

3.1 The View Definition File

A *view definition file* is a unix file where a view definition is recorded. The string stored on this file is passed to the O_2Views compiler which generates the associated *view implementation file*. The implementation file contains standard O_2C code which can be loaded in a newly created schema and which implements the view defined in the definition file.

Some rules must be observed when composing a view definition file:

- A view definition file is composed of two parts. The first part contains the view definition using the O_2Views language whose commands are described in the previous section. The second part is not mandatory and contains standard O_2C code that complements the view definition. The reserved word `end virtual schema` followed by ";" delimits the two parts and it can be ommited in case the second part is empty.
- The first command of the first part of the definition file must be a virtual schema creation command, which is followed by a possibly empty list of view primitive commands (virtual class, imaginary class, virtual attribute, virtual name, hiding, renaming and type commands) up to the end of file or to the view definition terminator `end virtual schema`.
- The second part of the file contains code for methods, applications, programs and auxiliary functions which are to be associated to the virtual schema implementation. This code won't be compiled by the O_2Views compiler, as it is the case for the first part of the definition file, but will be pasted into the implementation file to be further submitted to the O_2C compiler.

An example of a complete view definition file is given in Section 5.4.

3.2 Loading a View into O_2Views

Once a view definition file is composed, the user must load it in order to submit the view definition to the compiler, who will then generate the implementation of the view.

Interactive Mode

In the interactive mode, the user must first activate the base where the view definition is to be stored (a base instantiated from schema `o2views`) and then run the application `View_Management`. Figure 3 shows the main window of this application. Three views have been loaded in the currently active base, namely `special_promotion`, `management` and `vendors`. These three views do not necessarily relate to each other and may have been defined from different schemas.

Views can be stored in many different bases instantiated from schema `o2views` and the set of views can be thus partitioned according to a particular criteria. For instance, views defined from different schemas can be grouped into different bases (partition based on the root schema) or views can be grouped according to the groups of users which will be granted access to them (partition based on access rights). All views defined in a given O_2 system can also be ultimately stored in a single base and it is up to the view designer to decide on which base his or her view definition will be stored.

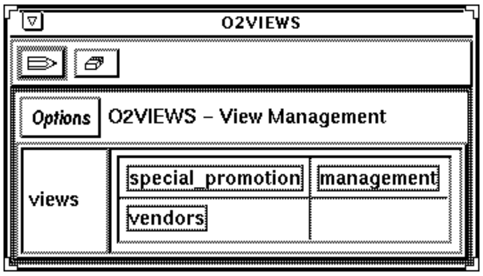

Figure 3: Main window of application `View_Management`

Figure 4 shows the `Options` menu.

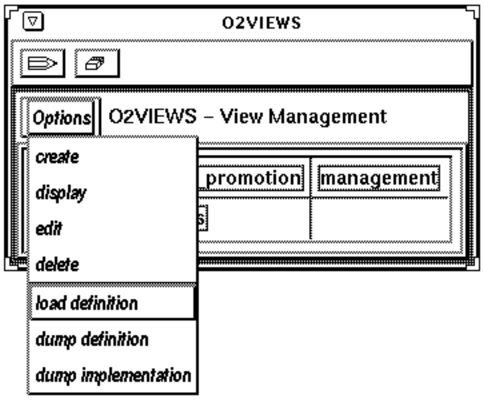

Figure 4: The `Options` menu of the application `View_Management`

The option selected in Figure 4, namely `load definition`, allows the user to load the definition of a view from a unix file. The system asks for the name of the file. The string stored on the file will be parsed and syntactical errors, if any, are reported on the shell from which O_2 was launched (or to a redirectioned output, if some has been defined when launching O_2). If a view with the same name as that defined in the file has been previously loaded on the same base, the system asks for confirmation before overwriting the existing view definition. If no error is reported, the view is successfuly loaded into the system.

Another way of loading and editing a view in the interactive mode is by means of the graphical interface of O_2Views, which will be described later on.

Batch Mode

A special program is furnished in the O_2Views package which allows automatic loading and implementation of views into an O_2 system. The program is `o2views_batch` and must be called as follows:

```
o2views definition-file implementation-file
```

This program makes the connection with an o2server through the O2API interface and allows the loading of a view whose definition is stored in file `definition-file`.The program requires schema `o2views` and associated default base `o2views_base` (as these are built through the installation procedure of the O_2Views package) to be available in the O2 system `$O2SYSTEM`. Variable `$O2HOME` must also be defined. If no syntactic errors are found in the view definition, a view implementation file named `implementation-file` is dumped, which can be subsequently loaded into O2.

3.3 The O_2Views Graphical Interface

A view can be loaded into an O_2 system through the graphical interface. From the menu in the main window of the view management application, illustrated in Figure 3, the user can issue the creation of a new view (option `create`) and then edit it (option `edit`). View edition can also be performed on views which have been loaded from a definition file. Also, a view definition can be dumped (option `dump definition`) so as to keep a definition file consistent with the view definition stored into the system. The definition file can be used as a backup of the view.

Figure 5 shows the view edition window, from which virtual and imaginary classes can be created and edited. Figure 6 and 7 show the windows used for editing virtual and imaginary classes definitions respectively. Finally, Figure 8 shows the window used for editing a virtual attribute definition together with the associated window used for editing the defining query of the virtual attribute. The snapshots were taken from the view example described in Section 5.

The advantage of using the graphical interface is that the user is released from knowing the syntax of the view definition language. It is also useful when small changes have to be performed to a view definition, so that the user does not have to reload the entire view definition. In addition, as mentioned before, a view definition file can be generated for every view definition stored in the system, so that after updates to a view definition through the graphical interface, the user can keep an associated definition file up to date as a backup or for further reloading.

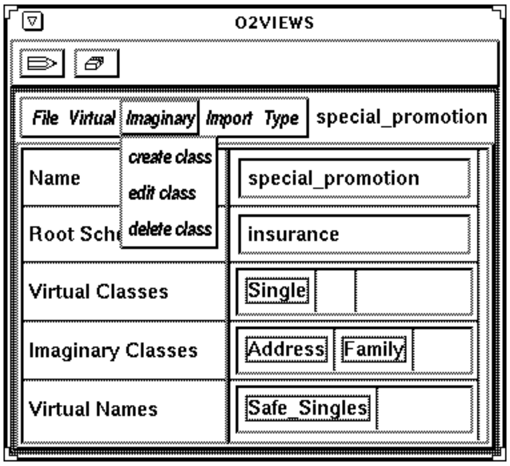

Figure 5: Window for editing a view

3.4 Generating the Implementation of a View

The implementation of a view corresponds to the O_2 schema implementing a view definition. Such schema can be generated automatically through the program `o2views` by passing the optional `schema-name` argument or be created by hand and then have a view implementation file loaded into it.

The generation of a view implementation file is achieved through the graphical interface of the application `View_Management`, by choosing the option `dump implementation` from the menu of the main window of the application (Figure 3) or by choosing the same option from the `File` menu of the view edition window (Figure 5). In any case, the system will ask for the name of the file into which the code of implementation schema is saved. This is pure O_2C code and must be further loaded into a newly created (empty) schema, i.e. the view implementation schema.

The second part of the corresponding definition file is appended to the end of the implementation file when this one is generated, but the user can decide to keep it in a separate file and load it into the view implementation schema after the view implementation file has been loaded.

Different applications can be defined on a view and they do not have to be necessarily put in the second part of the corresponding view definition file. The view implementation schema is a standard O_2 schema to/from which auxiliary functions, types, methods and applications with associated programs and variables can be attached/suppressed at any time. But the user must be careful so as to be able to (re)generate the implementation schema from scratch, so (s)he is advised to keep all additional definitions (functions, types, applications and methods) in a separate file or on the second part of the definition file. That's why changes to a view definition should not be performed in the implementation file, which is not supposed to be edited, nor in the resulting schema, which can be extended with new auxiliary definitions (conservative changes) but should not have its definitions changed except through the corresponding definition file. Once updated, a view definition file must be recompiled (reloaded into the system) and a new implementation schema must be generated. The schema implementing the old version of the view definition should be deleted before generating the new implementation schema.

The name of the view implementation schema is not necessarily the same as the name of the virtual schema defined in the corresponding view definition file. But the user

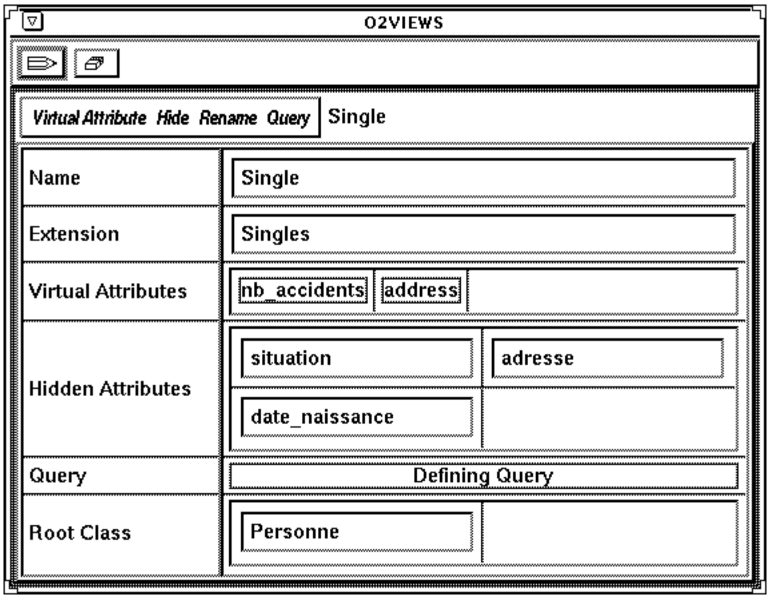

Figure 6: Window for editing of a virtual class definition

is advised to name the implementation schema after the name of the virtual schema, specially if (s)he intends to define other views from such view, in which case definitions are likely to be imported from the virtual schema through its original name.

4 Using a View

Once a view has been compiled and the implementation schema properly generated, the view can be used for querying and updating the data.

4.1 Activating a View

In order to be able to access data imported from other base(s), a new base has to be created from the view implementation schema. This base does not contain data of its own and it serves only as an entry point for accessing data from other base(s)[2]. This base implements a virtual base as it is defined in [SDA94]. There is no point in creating more than one base from the same implementation schema, since they would be all identical (i.e. isomorphic up to object identifiers)[3]. The contents of a virtual base are initialized and updated through system supplied functions, which are described in Section 4.2.4.

[2] Actually, such a base keeps some internal persistent data, for the sake of performance, which should not be visible to the end user of the view, but in the current implementation they are not hidden.

[3] An extension of the view mechanism for modelling roles is being considered through which new data can be stored in the virtual base, along with the view. A different kind of class in the view, a *role class*, could be defined for that end, allowing the user to model the evolving nature of some kinds of data, such as those representing people, who are likely to change of role during their lifetime as well as to play different roles simultaneously. In that case, it would make sense to instantiate different bases from the same implementation schema since they could store different role instantiations.

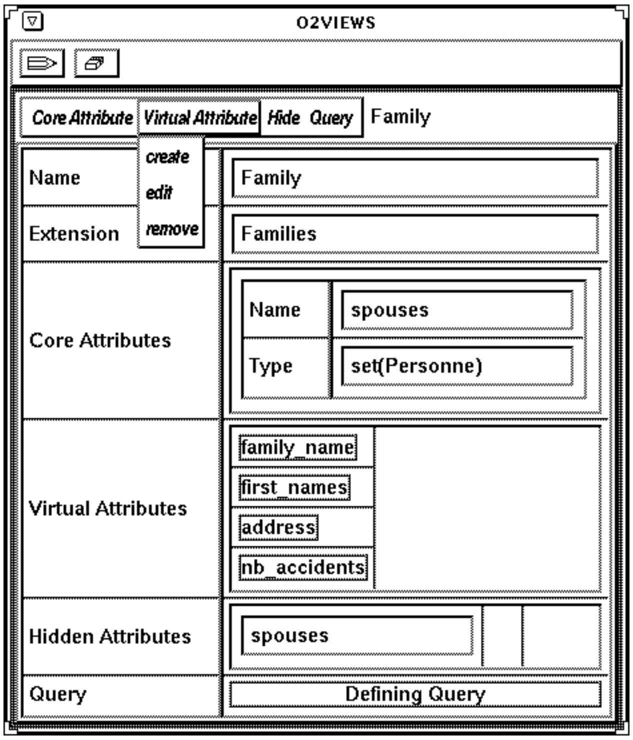

Figure 7: Window for editing of an imaginary class definition

4.2 The View Implementation Schema

The view implementation schema is supplied with classes, associated methods, types, names and auxiliary functions which together implement the corresponding virtual schema. These will be described in the sequel.

4.2.1 Virtual Classes

For each virtual class in the virtual schema definition, a standard O_2 class is created in the view implementation schema. This class has the name of the virtual class and the structure and behavior of its instances, called virtual objects, is defined as follows.

The class implementing a virtual class encapsulates a pointer to an instance of the root class of the corresponding virtual class. Such instance is pointed by the private attribute `root_object`.

Every public and read attribute of the root class is propagated to it as a read attribute. Private attributes are propagated as private attributes. In addition, it has extra attributes corresponding to the virtual attributes defined in the virtual class. Let us consider an example. Given the class `Person` defined in Figure 9, we define the virtual class `Young_Person` as shown in Figure 10, along with some virtual attributes and hiding commands.

The structure of the class implementing this virtual class is shown in Figure 11.

The implementation shown in Figure 11 highlights many of the features of the current implementation. Hidden attributes are not part of the structure of the class implementing the virtual class. If they are to be used in the definition of a virtual attribute or in the code of a method, they can be however accessed through the private attribute `root_object`, as

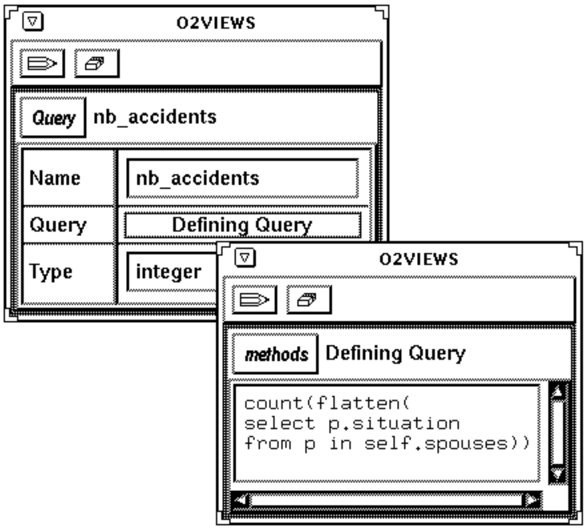

Figure 8: Window for editing of a virtual attribute and associated defining query

```
class Person type
tuple (read family_name: string,
       public first_name: string,
       private date_of_birth:Date,
       public addresse: Addresse)
end;
```

Figure 9: Class `Person`

in the definition of virtual attribute `name` given in Figure 10.

The attribute `root_object` is a bridge towards the root of an object seen in the view. It is not part of the public interface of such an object but is available for methods and query definition. The ability to use the semantics of the real base is crucial when defining a view of it.

In the current implementation, `root_object` stands for the object from which the virtual object is an image through the view. This object can be, in turn, a virtual object, since the root class of a virtual class can be itself virtual (view composition). A private method, namely `real_object`, is defined in the class implementing the virtual class to access the real object of a virtual class hierarchy in a single method call but this method is implemented through recursive calls up to the real root class in the virtual class hierarchy , so it represents no gain in performance but just a user convenience (a syntactic shorthand). Here, the term hierarchy should not be confused with inheritance hierarchy. The virtual class hierarchy results from the dependence relation (a partial order, i.e. reflexive, antisymmetric and transitive) defined between a virtual class and its root class. In the current implementation, there is no standard inheritance relationship between classes implementing virtual and imaginary classes defined in a virtual schema.

```
virtual class Young_Person from Person
with extension Young_People =
      select distinct p
      from p in The_People
      where p->my_age < 21;
end;

virtual attribute age:integer in class Young_Person =
      self->my_age;

virtual attribute name:string in class Young_Person =
      self->root_object->first_name + " " +
      self->root_object->family_name;

hide attribute family_name, first_name in class Young_Person;
```

Figure 10: Virtual class `Young_Person`

```
class Young_Person type
tuple (private root_object:Person,
       private date_of_birth:Date,
       read addresse: Addresse,
       read age:integer,
       read name:string)
end;
```

Figure 11: Implementation of virtual class `Young_Person`

4.2.2 Imaginary Classes

Similarly to virtual classes, for each imaginary class in the virtual schema definition, a standard O_2 class is created in the view implementation schema. This class is named after the imaginary class and the structure and behavior of its instances, called imaginary objects, is defined as follows.

The structure of the class implementing an imaginary class is that defined by the tuple of key attributes, which can be extended with virtual attributes. Hidden key attributes are defined as private, differently from hidden attributes in a virtual object, which are not part of its structure and can be accessed only through its root object. An imaginary object has no root object, and pointers to its key attributes are part of its structure, even if they are hidden, as they determine its identifier[4].

Let us consider an example. Given the class `Person` defined in Figure 9, we define the imaginary class `Family` as shown in Figure 12, along with some virtual attributes and hiding commands. We assume that people with the same family name are married, and a family is simply a couple of spouses.

The structure of the class implementing this imaginary class is shown in Figure 13.

[4] Of course this is just an implementation decision since hidden key attributes could have been kept in an auxiliary structure such as a table instead of becoming part of the imaginary object structure itself.

```
imaginary class Family
   key attributes spouses: set(Personne)
   with extension Families =
   select distinct tuple(spouses: set(h,w))
      from h in The_People,
           w in The_People
      where h.family_name = w.family_name
            and h <> w;
end;

virtual attribute family_name : string in class Family =
   element(select distinct p.family_name
                  from p in self.spouses);

virtual attribute first_names : set(string) in class Family =
   select p.first_name
          from p in self.spouses;

hide attribute spouses in class Family;
```

Figure 12: Imaginary class `Family`

```
class Family type
tuple (private spouses:set(Person),
       read family_name:string,
       read first_names:set(string))
end;
```

Figure 13: Implementation of imaginary class `Family`

4.2.3 Virtual Names

An example of a virtual name definition is given in Figure 14. It illustrates the composition of view definition primitives since it uses a virtual class extension.

```
virtual name Very_Young_People: set(Young_Person) =
   select p from p in Young_People where p->age < 15;
```

Figure 14: Virtual name `Very_Young_People`

The implementation of such a name is given in Figure 15. The value of virtual names in a virtual base is initialized when the view is initialized and can be further updated through apropriate update functions, as it will be explained in Section 4.2.4.

4.2.4 System-Supplied Functions

System supplied functions include view initialization and update functions. For each virtual and imaginary class, three special functions are provided in the implementation

```
name Very_Young_People: set(Young_Person);
```

Figure 15: Implementation of virtual name `Very_Young_People`

schema, namely the update extension, the conversion and the characteristic function.

Update Extension Function The update extension function allows the user to update the class extension, i.e. the named unique set of all instances of a virtual or imaginary class. The signatures of two such functions, for the virtual and imaginary classes defined in Figure 10 and 12 are shown in Figure 16.

```
function Update_Young_Person:set(Young_Person);

function Update_Family:set(Family);
```

Figure 16: Extension functions for classes `Young_Person` and `Family`

These functions are generated automatically, as the conversion and characteristic functions. The rule for creating the name of an update extension function is the following: the prefix "`Update_`" is appended at the beginning of the class name.

If the class is generic, the update extension function has the same signature as that of its corresponding class.

Conversion Function For a given virtual class, a conversion function allows the retrieval of a virtual object corresponding to a given root object. The signature of the convertion function associated to the virtual class given in Figure 10 is shown in Figure 17.

```
function As_Young_Person(obj:Person):Young_Person;
```

Figure 17: Conversion function for virtual class `Young_Person`

The parameter passed to the conversion function must be of the type of the root class of the virtual class. It can be an instance of the root class or of any of its subclasses.

It must be noted that the current prototype is built on the top of O_2 and that at this level class extensions are not available. Therefore, the defining queries of virtual classes are based on named sets such as the defining query of virtual class `Young_Person` which uses the named set `The_People`. It is up to the view designer to make sure that such a set holds not only instances of the root class `Person` but also intances of its subclasses, in which case the ability to convert instances of the subclass of the root class would cause no problem of consistence between the update extension function result and the objects actually belonging to a virtual class at a given moment as a result of the use of the conversion function. Put another way, conversion functions must be used in accordance to the way the extension of the virtual class is defined, i.e. the object returned by the conversion function must belong to the set returned by the corresponding update extension function.

For a given imaginary class, a conversion function allows the retrieval of an imaginary object corresponding to a tuple of key attributes. As pointed out before, the identifier of an imaginary object is a function of its key attributes values. The signature of the convertion function associated to the imaginary class given in Figure 12 is shown in Figure 18.

```
function As_Family(spouses:set(Person)):Family;
```

Figure 18: Conversion function for imaginary class `Family`

The rule for generating the name of a conversion function is the following: the prefix "`As_`" is appended at the beginning of the class name.

The conversion function also takes the same parameters as those defined in its corresponding class (if any), in addition to the object to be converted (in case of a generic virtual class) or to the key attributes (in case of a generic imaginary class). Such class parameters must be passed first, in the same order as they are defined in the class.

Characteristic Function For a given virtual class, the characteristic function allows the test of membership of an instance of the root class to the virtual class. As discussed in [SDA94], an instance of a virtual class actually corresponds to an instance of its root class with which it shares its identifier. In the current implementation, a new object implements the virtual object and this new object keeps a pointer to the root object. Testing the membership through the characteristic function in the current implementation corresponds to asking whether the given instance of the root class matches the conditions implicitly stated in the defining query of the virtual class to be a member of such a class.

The signature of the characteristic function associated to the virtual class given in Figure 10 is given in Figure 19 along with its specification. In the current prototype, the implementation of the characteristic function is very simplistic and amounts to test whether the object passed as a parameter belongs to the set returned by the evaluation of the defining query. Here, conflicts may arise if instances of a subclass of the root class is passed and the defining query (from which the update extension function is derived) is based on a named set which holds only instances of the root class but not of its subclasses. In a future version, some rewriting based on the defining query must allow the derivation of more efficient characteristic functions by avoiding the evaluation of the defining query. In that sense, the implementation of the characteristic function of `Young_Person` would then resemble its specification, as it is shown in Figure 19.

```
function Is_Young_Person(obj:Person):boolean;

function Is_Young_Person(obj:Person):boolean {
      return(obj->my_age < 21);}
```

Figure 19: Characteristic function for virtual class `Young_Person`

For a given imaginary class, the characteristic function allows to test if a tuple of key attributes corresponds to an instance of the imaginary class.

The signature of the characteristic function associated to the imaginary class given in Figure 12 is given in Figure 20, along with its specification. Analogously to virtual classes, in the current prototype, the implementation of the characteristic function associated to an imaginary class amounts to test whether the tuple passed as a parameter belongs to the set returned by the evaluation of the defining query.

```
function Is_Family(spouses:set(Person)):boolean;
```

Figure 20: Characteristic function for imaginary class `Family`

The rule for generating the name of a characteristic function is the following: the prefix "`Is_`" is appended at the beginning of the class name.

If the corresponding class of a characteristic function is generic, its signature is extended in the same way as for convertion functions.

View Initialization Function Another system-supplied function is the view initialization function, whose rule formation name is the following: the prefix `Init_` is appended at the beginning of the virtual schema name. This function must be called once a virtual base is activated. It updates all class extensions (by calling the update extension functions for virtual and imaginary classes) and all virtual names in the view. It takes no parameter.

Virtual Name Update Function To each virtual name defined in the view, an update function is provided which allows its value to be updated through the evaluation of the corresponding defining query. Such functions are called from the view initialization function.

Internal Functions Other auxiliary functions are generated in the implementation schema but they are not supposed to be used by the user of a view but only internally by the implementation of the view mechanism and will not be described here. In a future version they must be hidden from the schema interface and kept by the O_2 schema manager in an internal library private to the system. Also, some auxiliary named sets are created in the implementation schema but they should not be available to the user, who should ignore them in the current implementation.

4.2.5 System-Supplied Methods

Some system supplied methods are present in the interface of virtual and imaginary objects.

For virtual objects, these methods are the following:

- `public is_consistent:boolean`

  This methods uses the corresponding characteristic function to test whether the virtual object happened to become inconsistent as a result of an update to its root object. An instance of `Young_Person`, as it is defined in Figure 10, can have its attribute `date_of_birth` updated and ceases to be an instance of the virtual class[5].

- `public has_root_object(obj:Root_Class):boolean`

  `Root_Class` stands for the actual name of the root class of the corresponding virtual class. This allows the user to test if a virtual object corresponds to a given instance of the root class.

[5] Actually, in this example, an instance of `Person` can cease to be an instance of `Young_Person` even if there is no update to the date of birth, as time goes by...

- `private real_object:Topmost_Root_Class`

  This method allows the retrieval of the real object corresponding to a virtual object, as discussed above. If the root class of a virtual class is real (not virtual), then the object returned by this method will coincide with the one stored in the `root_object` private attribute. If the root class is virtual, the same method is recursively applied to the root object up to the top of the virtual class hierarchy.

- `public update_attributes`

  This methods allows the update of virtual attributes, whose values may become obsolete due to side effects (update to the root object).

In addition to the general methods above, for each public attribute propagated from the root object, an update method is provided which allows the update of such attributes in the virtual object as well as the propagation of such updates to the root object. The rule for generating the names of such methods is the following: the prefix "`set_`" is appended at the beginning of the attribute name. The signature of such a method for the virtual class `Young_Person` defined in Figure 10 is given in Figure 21. Note that this is the only such method generated for this class since the other public attributes of the root class, namely `family_name` and `first_name` are hidden in the virtual class.

```
method public set_address(address:Address) in class Young_Person;
```

Figure 21: Update method for attributes `address` of class `Young_Person`

For imaginary objects, the system supplied methods are the following:

- `public is_consistent:boolean`

  This methods uses the corresponding characteristic function to test whether the imaginary object happened to become inconsistent as a side effect of an update to its key attributes.

- `public has_tuple_value(key_signature):boolean`

  `key_signature` stands for the tuple of key attributes. This method allows the user to test if an imaginary object has a given tuple of key attributes.

- `public update_attributes`

  This methods allows the update of virtual attributes, whose values may become obsolete due to side effects (update to the key attributes).

Other private methods are defined in the implementation of virtual and imaginary classes but they are used for internal purposes only and will not be described here.

4.3 Querying a View

A view can be queried as a usual O_2 schema. In order to issue queries on virtual or imaginary class extensions, the user must use the appropriate named extensions and update extension functions. Other system-supplied functions and methods can also be used in the body of queries.

We give below some examples of queries on the young people and families defined by classes `Young_Person` (Figure 10) and `Family` (Figure 12).

1. Retrieving every instance of `Young_Person`:

```
Young_People
```

2. Retrieve every instance of `Young_Person` aged of 15 years:

```
select p from p in Young_People
       where p->age = 15
```

3. Group the names of young people by age:

```
group p in Young_People
      by (age:p->age)
      with (name:select x->name from x in partition)
```

4. Retrieve every family whose one of the spouses is a young person:

```
select distinct f from f in Families
       where exists p in Young_People:
                   (f->family_name = p->root_object->family_name)
```

The query above can be reformulated as follows:

```
select distinct f from f in Families
       where exists p in f.spouses:(Is_Young_Person(p))
```

In the query above, the attribute `spouses` of class `Family` is private (it is a hidden key attribute) but private attributes can be accessed in the query interactive mode (the O_2SQL interpreter breaks encapsulation). This query is equivalent to the previous one but is more efficient since it avoids the computation of the `Young_Person` class extension. It also illustrates the use of a characteristic function.

More examples of queries on views are given in Section 5.

4.4 Updating through a View

Updates through a view in O_2Views can be performed in a restricted way. Although the availability of object identifiers in the data model eases the propagation of updates from a view to the real base, threfore simplifyng the view update problem as it is commonly referred to in the relational context, such updates cannot always be properly propagated for different reasons.

As in the relational context, sometimes it is not possible to find the original object(s) from which a view object has been constructed. This is the case for some imaginary objects which do not keep pointers to the objects from which they were constructed.

Virtual attributes in turn cannot be updated in general, since their value is a function of other objects or values, but an inverse function can, if such a function can be defined, can be implemented by the view designer as a method which implements the propagation of the update.

Some restrictions to update through the view in the current prototype have to do with the way objects and values imported from an external base can be updated in O_2. In that sense, objects must be created in the context of the base where they are supposed to be stored (through a method or function defined in the schema of that base, for instance). Atomic values and complex values (lists, sets and tuples) composed of atomic values can be freely updated.

As mentioned in Section 4.2.5, for each public attribute propagated from the root object of a virtual object, an update method is provided which allows the update of such attributes in the virtual object as well as the propagation of such updates to the root object. The rule for generating the names of such methods is the following: the prefix "`set_`" is appended at the beginning of the attribute name. An example is given in Section 4.2.5.

Updates through the view remain to be properly studied in our approach and we do not pursue this issue here.

4.5 Writing Applications on a View

Applications are attached to schemas in O_2. In order to write applications on a view, the user must attach such applications to the corresponding view implementation schema. Applications on a view are therefore written as for any other O_2 schema. As each time the view definition is updated and recompiled the view implementation schema is overwritten, the code of applications should be kept in a separate file or in the second part of the view definition file so that they are not last as a result of a view recompilation.

An example of an application is given in section 5.

4.6 Limitations of the Current Prototype

The current prototype is a preliminary work and presents some limitations with respect to the mechanism presented in [SDA94]. Some of these limitations have to do with the fact that the prototype has been entirely implemented on top of the O_2 system. In that sense, some features of the view mechanism could not be implemented since they require modifications at lower level of the O_2 system.

On the other hand, some limitations result from the fact that we did not intend to implement the whole functionality of the view mechanism in the first prototype, in order to simplify the task and be able to have a simplified running mechanism in a shorter time. Extensions to the prototype are already under development and must become soon available.

The main limitations are listed below:

- A great level of control is left to the programmer.

  The current prototype encourages a defensive programming style, whereby the programmer is required to perform tests (through the characteristic functions) before performing conversions (through the conversion functions) in order to guarantee the soundness of his or her transactions.

- The implementation schema is a standard O_2 schema and we can not prevent the user from defining persistent data on it.

  On one hand, the user should use the implementation schema as any other schema for defining applications on the view. On the other hand, some restrictions should be applied on the use of such schema so as to preserve the view semantics, but such restrictions are currently not verified by the system and it is up to the user to make good use of the implementation schema. Actually, the implementation schema is just an implementation concept related to the current prototype. Idealistically, virtual schema should be managed by the schema manager at a lower level of the stratified O_2 system implementation so that restrictions on its use could be enforced by the system itself.

- The type closure of a view is not inferred and must be provided by the user.

  This implies a number of import of definitions from which the view designer should be released from taking care of. The type closure of a view is the set of types necessary for the compilation of a view implementation schema as it is currently generated. In particular, all class types involved in the defining query must be imported to the view. Also, named types must be redefined in the view (named types cannot be imported in the O_2 system).

- Defining queries (of virtual and imaginary classes and virtual attributes) are not compiled.

  Queries are copied to the implementation schema and no syntactic nor type verification takes place before query evaluation time. We did not want to reimplement the O_2SQL compiler and intend to use the existing compiler in a future version so as to trap errors at compile time.

- Dependencies between view definitions are not checked by the system.

  If there are dependencies between view definitions, the initialization function must be called more than once so that virtual and imaginary objects can be properly constructed. The number of times it must be called depends on the degree of dependency between definitions. For the view definition example given in Section 5, the initialization function must be called three times until all view objects are fully constructed.

  Remark Errors occurring during the execution of the initialization function may be due to badly formulated queries (ill typed queries) and won't cease to occur regardless of the number of times the initialization function is called, whereas errors due to view object dependencies do cease to occur if there are no cyclic dependencies. Cyclicity is nevertheless allowed at virtual attribute definition level but not between class extension definitions.

- Generic views, as they are defined in [SDA94] are not implemented.

  A proper implementation of generic views would require some changes in the import mechanism of O_2. In spite of that, a simulation of generic views is under development, through which for each instantiation of a generic view a different implementation schema is generated. Generic views allow the definition of parameterized views and, keeping the analogy view-function, whereby a virtual base is the output of the function defined by the view, a generic view extends this analogy by allowing the definition of non-monadic functions. In that sense, the implementation of generic views under development (on top of O_2) can be seen as a temporary downgrade of such analogy whereby generic views are analogous to a macro definition instead of a function and where view activations are considered as a macro expansion instead of a function call.

- Class generalization is not implemented.

A proper implementation of class generalization, as it is defined in [SDA94] must be based on a rather powerful type inference mechanism and possibly extend the O_2 data model to cope with union types. Type inference is already performed by the O_2 system but this functionality is not available to the end user but used internally by the O_2C and O_2SQL compilers. A simple implementation of class generalization is under development in which the view designer must explicitly give the least common supertype of the generalized classes.

- Updates are performed on a limited basis.

  In addition to the view update problem, which is inherited from the relational context (although it acquires a different shape in the object-oriented context in the presence of object identifiers), some restrictions to update through the view in the current prototype have to do with the way objects and values imported from an external base can be updated in O_2, as discussed in Section 4.4.

- Automatic interface resolution, as it is discussed in [SDA94] is not implemented.

  The implementation of the automatic interface resolution implies radical changes to the resolution mechanism of O_2.

Many of the limitations pointed out above may be easilly overcome by pushing part of the implementation of O_2Views to lower levels of the O_2 system (having access to the O_2SQL compiler, extending the schema manager to cope with virtual schemas, using the type inference mechanism of O_2). In order to avoid redundant work, which can be more efficiently done in cooperation with O_2Technology, we postponed the inclusion of some features to a future upgraded version.

Much of the control left to the programmer can be automatically provided by the system by placing triggers to keep view objects consistent with respect to their root objects.

Optimization

Before concluding this section, a few words on optimization. As pointed out in the introduction, performance has not been an issue in the current implementation, but for a future version many optimizations are considered. We comment on some of them:

- The characteristic function must be more efficiently generated so as to avoid the evaluation of the defining query. This has been already discussed.

- Optimization techniques applied to nested queries and queries in general must be adapted to allow the generation of efficient characteristic functions. Nested views imply the evaluation of nested queries and must have their implementation heavily optimized in order to be of practical use.

- View objects should be materialized in a transparent way to the user so as to avoid their recomputation. This is already performed on a limited basis but not as transparent as it should be.

5 An Example: The View `special_promotion`

In this section, we briefly illustrate the mechanisms of O_2Views. A view is designed, then an application above this view. This section is a simplified version of an experience with a real insurance application. The principles of O_2Views are first recalled. Then a situation is given. An analysis is provided, together with the O_2Views code.

The view definition given here, together with the corresponding root schema and base, is part of the O_2Views package and can be run once O_2Views is installed.

5.1 Preliminaries on O_2Views (and O_2)

A view can be seen as a "filtering schema" through which the data of a root database schema are looked at. The point is to get part of the structure and the data of the root database, possibly restructured. Roughly speaking, the primitives and their role are the following. Virtual and imaginary classes design which are the objects in the view, selected from the root database or created in the view. Virtual attributes augment the interface of objects, and hidden attributes restrict it. The user can access the data through the extensions of classes.

Pure O_2 Classes and Named Sets We recall some differences between classes and named sets relevant in the present context. Design a class corresponds to the following. (1) design the structure of objects in that class, (2) provides them with oids, (3) attach methods specific to this class, and (4) inherit type and methods from the hierarchy. A named set (1) does not create a new type or class, (2) contains values or objects, (3) has no methods attached to it, and (4) is not connected to the class hierarchy. On the other hand, (5) its population is arbitrarily chosen by the programmer, for instance by a query, but it does not necessarily contain all the objects from a class.

Virtual Classes A virtual class (1) selects by a query objects existing in the root database, (2) is connected to the root hierarchy, and (3) provides a named set containing these objects: the extension of the virtual class. The aim is to provide the features of both usual classes and named sets described above. Moreover, since it is a class, (4) its interface is able to be modified by the view primitives of hiding an attribute or adding a virtual one.

Imaginary Classes An imaginary class (1) selects and restructures by a query data from the root database or the view, (2) turns them into objects, (3) is not connected to the root hierarchy (of course), and (4) provides an extension. As for virtual classes, the aim is to provide the features of both classes and named sets, as well as being combined with other view primitives. The oid of an object is defined by its key attributes.

Virtual Attributes A virtual attribute attaches to an object in the view (possibly restructured) data, selected by a query from the root database or the view. It augments the original interface of (virtual) objects. It also attaches to imaginary objects data not involved with their identity.

Attribute hiding It restricts the original interface of root objects or data. It hides the key attributes of an imaginary object not to be visible for an end-user.

Virtual Names select data irrelevant to be in a class.

5.2 The Situation

The marketing service of an insurance company finds out the specificity of some clients, likely to be interested in particular products. It suggests the creation of a full time position for the management of these clients and products. The company creates this position, and provides the employee in charge of it with a *view* of these clients coming from the general database. Aware only of this view, the employee will write himself the *applications* corresponding to the different products or treatments. The clients concerned are experienced single drivers (over thirty), and every family[6]. The management of the products depends only on the number of accidents of a client.

[6] A family is simply a couple of spouses, and people with the same name are married.

Here is one of the *treatments* performed by this employee. For Christmas, he decides to send a wish letter to the clients, at their address. A family receives only one letter, including the first name of each spouse. For families and singles involved in only few accidents — with less severity for dangerous Parisian districts — there is an additional congratulations paragraph in the letter, including the number of accidents. Moreover, the employee needs, by *ad hoc queries*, to display these clients, and to find for each district of Paris the average number of accidents per client[7].

5.3 An Analysis

The root database is composed of type *Address*, class *Personne*, and the set *RootPersonnes* of all clients. The schema is partially given below, i.e. only relevant definitions are given.

```
schema insurance;
base b_insurance;

import schema o2kit class Date;

type Adresse: tuple(
rue:  string,
code_postal: string,
localite: string );

class Personne
type tuple (
public nom: string,
public prenom: string,
public date_naissance: Date,
public adresse: Adresse,
        date_creation:          Date,
        date_maj:               Date,
        date_annulation:        Date,
public situation: list(Declaration)
)
method
/* affichage */
public afficher,
public get_mask: integer,
public get_icon_mask: integer,
        quitter,
/* securite */
public mettre_securite_en_instance,
public rejeter_pour_fraude,
public ouvrir_declaration,
/* manipulations */
public ajouter_declaration( d: Declaration ),
        /* generiques */
        title: string
end;

class Declaration
end;
```

[7]A family is one client.

```
name RootPersonnes: set( Personne );

/*-----------------------------------------------------------*/

method body title: string in class Personne
{
return(self->prenom + " " + self->nom);
};

export schema class Personne;
export schema class Declaration;
export schema name RootPersonnes;
```

Design of the View

Select certain persons (singles over thirty) needs a named set to gather them. To select all families, a relevant approach is to build them once by a query, and put them in a named set. For both, only postal coordinates and the number of accidents are relevant. On the one hand, named sets seem to be enough. On the other hand, as singles and families are the central data in the view, they are likely to be attached behaviors for future specific treatments and products. The same holds for addresses (e.g., format, printing). In this situation, we choose to model them as objects. As singles are simply objects selected from the root database, a virtual class is used. As families and addresses are new objects, imaginary classes are used.

Note that in such a situation, the design of the view itself is somehow related to the future use of the view.

Singles are selected based on the age, and if not belonging to a family. Root attributes relevant to the view are renamed from french to english names (first name, family name). Other attributes are hidden. The number of accidents is computed from the root object by a virtual attribute. The address comes from the view.

A Family is built from the root database. It is characterized by the two spouses together (there is no information about sex in the root database). That is, $\forall f, f' \in families, f = f' \Leftrightarrow f.spouses = f'.spouses$. The only spouse attribute satisfying this property is the set of the two spouses. This will be the key attribute, which determines the oid. Once the oid defined, this attribute is hidden (irrelevant to be visible for an end-user). Only name and first names are relevant. They are given by virtual attributes, as they are not involved in the oid. As for singles, the address and the total number of accidents are added.

The Christmas Treatment

Methods formatting the address and computing a danger coefficient for special districts are attached to class *Address*. Method *wishes*, for both singles and families, compute the wish letter. The treatments consist in displaying all clients in the view, and sending the letters.

5.4 O_2Views Code

We give below the entire view definition, i.e. the contents of a view definition file, with explanatory comments.

```
/* -------------------------------------------------------------- */
/* VIEW special_promotion */
/* -------------------------------------------------------------- */

virtual schema special_promotion from insurance;

/*
  Are put in the design of the view every information involved in the
  definition of the view.
*/

/*
   The following are the real types and classes used in the view,
   called the "type closure". This will be built automatically.
   In the current release, it relies heavily on the O2 import mechanism,
   and is done by hand. Moreover, all the following has to be exportable
   from the real schema (insurance).

   begining of type closure
*/

import schema o2kit class Date;
import schema insurance class Personne;
import base b_insurance real name RootPersonnes;
import schema insurance class Declaration;
type Adresse:tuple(rue:string, code_postal:string, localite:string);

/* end of type closure */

/* Family:
   The identity of the family depends on key attributes. Thus, these must
   be chosen so that the family changes (is no longer the same family) iff
   any of these changes. In particular, the family may move, its name change,
   it may have more accidents, without being a different family. More
   precisely, it is the same family iff the spouses are the same. Hence, the
   key attributes must be exactly these, as being of class Personne. The
   others must be virtual. As there is no information about sex, we use the
   set of the two spouses.
*/

imaginary class Family
  key attributes spouses: set(Personne)
  virtual attribute
          family_name : string =
                 element(select distinct p.nom from p in self.spouses);
          first_names : set(string) =
                 select p.prenom from p in self.spouses;
          address : Address =
                        element(
                        select distinct a
                        from a in Addresses,
                             p in self.spouses
                        where a.street = p.adresse.rue
                          and a.code   = p.adresse.code_postal
                          and a.town   = p.adresse.localite);
          /* address: spouses are required to live together. */
```

```
          nb_accidents : integer =
                      count(flatten(
                      select p.situation
                      from p in self.spouses));
  hide attribute spouses
  with extension Families =
    select distinct tuple(spouses: set(h,w))
    from h in RootPersonnes,
         w in RootPersonnes
    where h.nom = w.nom
      and h <> w;
end;

virtual class Single from Personne
  virtual attribute
          nb_accidents : integer = count(self.base_object.situation);
          address : Address =
            element(
            select a
            from a in Addresses
            where a.street = self.base_object.adresse.rue
              and a.code   = self.base_object.adresse.code_postal
              and a.town   = self.base_object.adresse.localite);
  hide attribute
       situation, date_naissance, adresse,
       date_creation, date_maj, date_annulation
  rename attribute prenom as first_name,
                   nom as family_name
  with extension Singles =
    select distinct p
    from p in RootPersonnes
    where not (p in flatten(
                    select f.spouses
                    from f in Families))
      and age(p.date_naissance) > 30;
end;

/* Address: care of the distinct! */
/* Exactly one object per address */

imaginary class Address
  key attributes street:string, code:string, town:string
  with extension Addresses =
    select distinct tuple(street: p.adresse.rue,
                          code: p.adresse.code_postal,
                          town: p.adresse.localite)
    from p in RootPersonnes,
         s in Singles,
         f in Families,
         q in f.spouses
    where p = s.base_object
       or p = q;
end;

virtual name AllYears: set(integer) =
   select p->date_naissance->get_year
```

```
   from p in RootPersonnes;

end virtual schema;

/* ----------------------------------------------------------------- */
/* ----------------------------------------------------------------- */
/* Auxiliary functions for building the view */

function today : Date;
function body today : Date
{ o2 Date d = new Date(0,0,0);
  d->set_to_current_date;
  return(d); };

function age (birth: Date) : integer;
function body age (birth: Date) : integer
{
  o2 Date today0 = today();
  o2 integer year = today0->get_year - birth->get_year;
  o2 integer month = today0->get_month - birth->get_month;
  o2 integer day = today0->get_day - birth->get_day;

  if (month<0 || (month=0 && day<0))
    year -= 1;
  return(year);
};


/* ----------------------------------------------------------------- */
/* AN APPLICATION ON special_promotion */
/* ----------------------------------------------------------------- */

application special_promotion
  program public display_singles,
          public display_families,
          public send_wishes
  end;

method public address_text : string in class Address;
method public coefficient : integer in class Address;
method title : string in class Single;
method public wishes in class Single;
method public quitter in class Single;
method title : string in class Family;
method public wishes in class Family;

function latex_xdvi_letter (family_name : string,
                            first_names : set(string),
                            address : Address,
                            nb_accidents : integer);

function letter_text       (family_name : string,
                            first_names : set(string),
                            address : Address,
                            nb_accidents : integer) : string;
```

```
/* ---------------------------------------------------------------- */
/* programs */
/* ---------------------------------------------------------------- */

transaction body display_families in application special_promotion
{ display(Families); };

transaction body display_singles in application special_promotion
{ display(Singles); };

transaction body send_wishes in application special_promotion
{
  o2 Single s;
  o2 Family f;

  for (s in Singles)
    s->wishes;
  for (f in Families)
    f->wishes;
};

/* ---------------------------------------------------------------- */
/* Address */
/* ---------------------------------------------------------------- */

method body address_text : string in class Address
{ return(self->street + " " + self->code + " " + self->town); };

method body coefficient : integer in class Address
{
  if ((self->town == "Paris")
      && (self->street == "Place de l'Etoile"))
    return(5);
  else
    return(3);
};

/* ---------------------------------------------------------------- */
/* Single */
/* ---------------------------------------------------------------- */

method body title : string in class Single
{ return(self->family_name); };

method body wishes in class Single
{
  latex_xdvi_letter (
    self->family_name, set(self->first_name), self->address, self->nb_accidents);
};

method body quitter in class Single
{ display("Bye, bye..."); };

/* ---------------------------------------------------------------- */
/* Family */
/* ---------------------------------------------------------------- */
```

```
method body title : string in class Family
{ return(self->family_name); };

method body wishes in class Family
{
  latex_xdvi_letter (
    self->family_name, self->first_names, self->address, self->nb_accidents);
};

/* ----------------------------------------------------------------- */
/* function */
/* ----------------------------------------------------------------- */

function body letter_text (family_name : string,
                           first_names : set(string),
                           address : Address,
                           nb_accidents : integer) : string
{
  o2 string n, s, letter, string_nb_accidents = "0";

  for (n in first_names)
    s += n + ", ";
  string_nb_accidents[0] = nb_accidents + 48;
  letter = "{" + family_name + " " + s + "\n\n" +
           address->address_text + "\n\n";
  letter += "}\n\n\\today\n\n\\vspace*{1cm}\n\n";
  letter += "Dear " + s + "\n\n";
  letter += "Merry Christmas!!!";
  if (nb_accidents<address->coefficient)
    letter += "You had only " + string_nb_accidents + " accidents.\n" +
              "Congratulations!\n\n";
  return(letter);
};


/*
  Everything happens in the directory where o2 was launched.
  Recall that /tmp may be mounted from another machine.
  Note (1) $DISPLAY for xdvi, (2) latex commands, etc.
*/

function body latex_xdvi_letter (family_name : string,
                                 first_names : set(string),
                                 address : Address,
                                 nb_accidents : integer)
{
#undef NULL
#include <stdio.h>

  FILE *out;

  out = fopen("/tmp/special_promotion_display_file.tex","w");

  fprintf(out,"%s\n","\\documentstyle[12pt]{letter}");
  fprintf(out,"%s\n","\\signature{United~Insurances~Inc.}");
```

```
  fprintf(out,"%s\n","\\begin{document}\\begin{letter}\n");
  fprintf(out,"%s\n\n",
    letter_text(family_name, first_names, address, nb_accidents));
  fprintf(out,"%s\n","\\closing{Best regards}");
  fprintf(out,"%s\n","");
  fprintf(out,"%s\n","");
  fprintf(out,"%s\n","\\end{letter}");
  fprintf(out,"%s\n","\\end{document}");
  fclose(out);

  system("latex /tmp/special_promotion_display_file.tex");
  system("xdvi -display beaugos:0 special_promotion_display_file.dvi");

  system("rm special_promotion_display_file.dvi");
  system("rm special_promotion_display_file.log");
  system("rm /tmp/special_promotion_display_file.tex");
};
```

5.5 Demonstration Scenario

Display together (two O_2 processes), one on the root database and another one the view. Compare the structure, oids and behaviors. Note in particular that virtual objects *are* root objects, seen through the view.

6 View Administration

6.1 Installing O_2Views

The procedure for installing O_2Views is very simple. The instructions below assume the user is acquainted with the procedure for installing a new O_2 system.

O_2Views version 2 is currently available for versions 4.4 and 4.5 of the O_2 system running on Sun Sparc stations. The whole O_2Views package is available via anonymous ftp at ftp.inria.fr under the directory INRIA/Projects/Verso/Goodstep/O2Views-v2.0. The following can be found in this directory:

- `README` - Reports the last changes to the prototype or to the installation procedure which were not yet taken into account by the manual, as well as the synopsis of reported bugs, if any, with corresponding work arounds.
- `o2views.tar.Z` - The compressed tarfile where the whole package is assembled.

File `o2views.tar.Z` contains the directory o2views with the following structure:

```
o2views/
   README
   demo/
     insurance.o2
     special_promotion.o2v
   doc/
     o2views_manual_v2.0.ps.Z
   install
   src/
```

```
Makefile
o2views.o2
o2views_LKdefaults
o2views_batch.c
```

The procedure for installation is given in file `README`.

6.2 View Maintenance

As discussed in page 9, views can be stored in many different bases instantiated from schema `o2views`, according to different partitioning criteria. A base instantiated from schema `o2views` must have its contents initialized by running the function `Initialize()` on it. This function must be run only once, when a new base is created and before running the application `View_Management` on it.

A view must be recompiled whenever its root schema changes, unless the view designer is sure that the change to the root schema is conservative and does not affect the view. Also, imported definitions can be updated and the view must be updated accordingly. If the view definition itself does not change due to a schema update, then only the view implementation schema has to be generated again. Otherwise, the view definition has to be recompiled before generating the implementation.

As for virtual bases, it's up to the view application designer to guarantee the consistency of virtual data with respect to its root base. Update functions (for class extensions and virtual names), as well as update and consistency check methods (for virtual and imaginary objects) are supplied for that end.

References

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 238–247, 1991.

[BDK92] F. Bancilhon, C. Delobel, and P. Kannelakis. *Building an Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, 1992.

[dS94] Cassio Souza dos Santos. Design and Implementation of an Object-Oriented View Mechanism. In *10èmes Journées Bases de Données Avancées, Clemont-Ferrand, September 1994*, Rocquencourt - France, 1994.

[SDA94] Cassio Santos, Claude Delobel, and Serge Abiteboul. Virtual Schemas and Bases. In *International Conference on Extending Data Base Technology*, Cambridge, March 1994.

[Tec94] O_2 Technology. *The O_2 User's Manual, version 4.5*, November 1994.