A Case Study in Object-Oriented Knowledge-Base Design using the KIWI system

Els Laenens
John Snijders
Francois Staes

Philips KIWIS team. PASS-AIT.
Building HCM5, P.O. BOX 218, 5600 MD Eindhoven, The Netherlands

Dirk Vermeir

Dept. of Math. and Computer Science. University of Antwerp, U.I.A.
Universiteitsplein 1, B2610 Wilrijk, Belgium

Abstract

The KIWI system aims at intelligent interaction with large knowledge bases and databases. Its knowledge representation/manipulation formalism supports conceptual modeling in a unified way and is realized by OOPS+, an object-oriented database programming language. Optimization strategies for deductive query processing are applied to handle large amounts of knowledge and data. Moreover, a customizable window-interface (NAU), which is fully integrated with OOPS+, makes the system suitable for fast prototyping. This paper presents the KIWI system through the design process of a case study.

1. Introduction

Today, knowledge-based system technology lacks the means to provide efficient and robust knowledge bases, while database system technology lacks knowledge representation and reasoning capabilities [1] . The ultimate goal, a full integration of knowledge bases and databases, is not yet feasible. Much more experiment and experience is needed to specify both a complete set of requirements for such systems, and requirements that are mutually independent. We therefore can state that there is a need for flexible integration of data and knowledge from both databases and knowledge bases.

The KIWI system[2,3] supports a user-friendly and knowledgeable environment where the user can develop knowledge-based applications that may make use of data from a number of external databases. It has been designed and developed, by a consortium of industrial and research organizations, within the framework of ESPRIT.

The KIWI system uses a particular knowledge representation formalism, called OOPS+[4,5] , to represent in a uniform way both the schema and the data of the underlying databases. This formalism is also used to enrich the semantics of the databases by representing extra knowledge that is then stored locally, thus providing an integrated description of the complete knowledge-based system. While basically object-oriented, OOPS+ integrates database concepts as well as classical knowledge-representation techniques such as rule-based inference and demons. In addition, the language supports types as first-class objects, inheritance, imperative function definition, and query facilities based on logic programming.

The logic programming language supported by the database environment of the KIWI system is a simple extension of DATALOG which is based on the pure semantics of definite Horn clause queries. In order to improve the efficiency of query evaluation, advanced optimization techniques for the fixpoint computation or bottom-up evaluation are essential. The KIWI system incorporates among others magic counting methods[6] with very good results.

The user-interface to the KIWI system, called NAU: Not yet Another User-interface, combines ease of use with a powerful customization facility which greatly simplifies the development of end-user applications[7] . This is achieved by using the underlying object-oriented language (OOPS+) features to define a consistent and orthogonal set of graphical object classes that can be used to attach a taylor-made external representation of (a view on) objects and/or classes. In addition, using the same tools, it becomes a simple matter to define general purpose browsing and manipulation tools which otherwise would have to be developed separately.

The case study is an illustrative fragment of a genealogical information base. The information base represents the royal houses of Europe. In particular, it presents their families: persons, their parents, children, weddings, etc. The genealogical tree is hierarchical, but it is a directed acyclic graph rather than a tree, for X and Y, both great-grandchildren of Z may have a child C.

The aim of this paper is to present an overview of the KIWI system by means of this case study. Section 2 discusses the approach to knowledge representation (OOPS+). Section 3 describes the basic idea of the optimization methods for query processing. Section 4 shows the customizable window-interface (NAU).

2. OOPS+

In this section we discuss some OOPS+ features by using the royal houses case study. For a more complete description of the OOPS+ language, we refer to other papers[4,5] .

The notion of object is considered to be primitive in OOPS+. Because of the existence of complex objects, a single entity can be modeled as a single object: entity features need not be simple data values, but can be entities of arbitrary complexity. This should be contrasted with the traditional relational database approach, where an entity is represented as multiple tuples spread amongst several relations. For example, an object modeling a particular person, might look like this:

```
karelXVI =
    (
    name =          "Karel XVI Gustaaf";
    title =         "King of Sweden";
    sex =           male;
    birthDate =     1946;
    parents =       {gustaaf, sibylla}
    )
```

This *record object* or tuple object (referenced by `karelXVI`) is defined in terms of other already defined objects `sybilla` and `gustaaf`. We say that a record object is an *aggregation* of its attributes. `Name` and `birthDate` refer to primitive objects `"Karel XVI Gustaaf"` (a string object) and `1946` (an integer object) respectively while `parents` points to a set object.

As in mathematics, sets are defined either by enumerating their members, or by describing the unifying features that determine membership in the set. For instance, a set `persons` of all registered persons is a typical example of an enumerated set

```
persons = {oscarI, oscarII, gustaafV, victoria,
           gustaafVI, karelXVI, sybilla, gustaaf}
```

while the set of children of `gustaafVI` in the following example is a described one.

```
gustaafVI =
    (
    name =          "Gustaaf VI Adolf";
    title =         "King of Sweden";
    sex =           male;
    birthDate =     1882;
    deathDate =     1973;
    parents =       {gustaafV, victoria};
    children =      {kid | kid:persons and gustaafVI:kid.parents};
    weddings =      {gustaafV*margaretha, gustaafV*louise}
    )
```

Selection of a component of an aggregate object, also called *projection*, is done using '.'; for example `kid.parents` denotes the `parents` attribute of `kid`. The ':' denotes the *instance of* relation which captures both the notions of *having a type* and *explicit set membership*, and consequently serves two purposes, *type checking* and *referential integrity*. `kid:persons` stands for kid is a member of the set `persons`, i.e. kid must be an existing person. Thus the `children` attribute refers to a set of existing persons that have `gustaafVI` as a parent.

Note that we assume too hastily the existence of a `parents` attribute for each person (instance of `persons`). This inaccuracy can be solved by demanding that all persons conform to a certain standard (i.e. type checking). From learning about our specific examples, `karelXVI` and `gustaafVI`, we can construct an object which abstracts out what we believe is true about all individual persons. By specifying the properties shared by `karelXVI` and `gutaafVI`, we find the following abstraction:

```
personType =
    (
    name =        Str;
    title =       Str;
    sex =         {male,female};
    birthDate =   Int;
    parents =     [persons]
    )
```

This record object (labeled personType) acts as a *type* for both `karelXVI` and `gustaafVI`, which are in turn *instances* of the record object. Indeed, a record is an instance of another record if the instance-of relation holds for common attributes. For example `karelXVI : personType` boils down to

```
"karel XVI Gustaaf" : Str
"King of Sweden" : Str
male : {male,female}
1946 : Int
{sibylla,gustaaf} : [persons]
```

where `[persons]` is a *power object* denoting the set of all finite sets of instances of `persons`. In other words the members of `{sibylla,gustaaf}` must be instances of `persons`. Thus, `personType` can be looked upon as a specification of conditions for its instances: they need to have an attribute `name` referring to a string, an attribute `sex` referring to either `male` or `female`, etc. Additional attributes such as `children` are not restricted and can therefore not affect the instance-of relation. This explains why also `gustaafVI : personType`. On the other hand, `oscarII` is not an instance of `personType` because his mother is not an existing person (i.e. not in `persons`):

```
oscarII =
    (
    name =        "oscarII";
    title =       "King of Sweden and Norway";
    sex =         male;
    birthDate =   1829;
    deathDate =   1907;
    parents =     {josephine, oscarI}
    )
```

We can now demand that all instances of the set `persons` should conform to `personType`:

```
[personType]    persons = {..}
```

So, `gustaafVI` and `karelXVI` can be members of `persons` while `oscarII` can not.

As genealogical information changes, the information in our knowledge base needs to be updated: persons get married, get children, die, etc. So `deathDate`, `children` and `weddings` are no accidental attributes, instead they are of major importance in a genealogical information base. Therefore, we wish to extend `personType` in order to get some additional type checks for these optional attributes:

```
personType =
    (
    name =        Str;
    title =       Str;
    sex =         {male,female};
    birthDate =   Int;
    deathDate =   Int;
    parents =     [persons];
    children =    [persons];
    weddings =    [weddingType]
    )
```

But there is a problem: `karelXVI` can no longer be an instance of `persons` because according to the new `personType`, all persons must have a `deathDate`, a `children` and a `weddings` attribute, which is not acceptable.

Thus, it is desirable to be able to put type checking control on these attributes, taking into account that they might be missing. This is realized by allowing for *default* attributes in the type specification. This technique provides us with the notion of *inheritance* as used in object-oriented languages such as Smalltalk. Hence the change of the label, `personClass` instead of `personType`:

```
personClass =
    (
    name =      Str;
    title =     Str;
    sex =       {male,female};
    birthDate = Int;
    deathDate = Int default nil;
    parents =   [persons];
    children =  [persons] default
                    {kid | kid:persons and self:kid.parents};
    weddings =  [weddingType] default {};
    age =       default Int
                    {
                    if (self.deathDate==nil) then
                        return(currentDate.year-birthDate);
                    else
                        return (deathDate-birthDate);
                    }()
    )
```

Since both `karelXVI` and `gustaafVI` are instances of `personClass` (see below), we can require that the instances of `persons` should be instances of `personClass`:

```
[personClass]    persons = {..}
```

All instances of `personClass` inherit the default attributes `deathDate`, `children`, `weddings` and `age` while explicit occurrences of these attributes overwrite these defaults and should therefore conform to the type `Int`, `[persons]` and `[weddingType]` respectively. In other words, a default value is consulted whenever there is no explicit occurrence of the attribute at hand. For example, if in an instance of `personClass`, `deathDate` is mentioned explicitly it must conform to type `Int`, otherwise one can interpret it as being nil (the default). Similar, each member of `gustaafVI.weddings` should be an instance of `weddingType` while `karelXVI.weddings` refers to the empty set.

The default for the `children` attribute is a generalized version of `gustaafVI.children` where `self` denotes the owner of the attribute to be computed. Thus, occurrences of `self` in an operation definition refer to the object on behalf of which the operation is being executed rather than to the object in which the operation is textually defined.

Another powerful facility that exists only rudimentarily in most programming languages is *type coercion* or *type casting* which enforces instance-of conformance by possibly altering some objects. For instance, we can make `oscarII` an instance of `personClass` through casting, (denoted '<-'):

```
personClass <- oscarII
```

The casting operation boils down to the attributes:

```
Str <- "oscarII"
Str <- "King of Sweden and Norway"
{male,female} <- male
Int <- 1829
Int <- 1907
[persons] <- {josephine,oscarI}
```

It is obvious that no actions are needed for `T<-I` if `I:T`. Thus there is only one active cast left: `[persons] <- {josephine,oscar}` which in turn is equivalent to `persons <- josephine` (since `oscarI : persons`), the result of which will be the insertion of `josephine` in the set `persons` after which `{josephine,oscarI} : [persons]` and consequently also `oscarII : personClass`.

Besides the object-oriented paradigm, discussed so far, OOPS+ incorporates other powerful programming paradigms such as logic programming, rule-base programming, access programming and imperative programming each of them fully integrated with the object-oriented aspects of the language. A presentation of all of them however, is beyond the scope of this paper. We therefore describe logic programming which allows for a smooth warming up for the next section.

OOPS+ uses predicates in order provide powerful query facilities. In most object-oriented languages, it is rather difficult and tedious to reference objects by contents: the programmer has to 'hand code' support for such references explicitly by providing methods. In OOPS+, predicates can be used to query the object space.

The following defines a predicate `grandChild`. Recall that in our genealogical knowledge base, a child of a person `p` is a member of the set `p.children`.

```
grandChild(grandParent=_gp;grandChild=_gc)     :-
    _gp.children(self=_child),
    _child.children(self=_gc).
```

Note that labels are introduced in the predicate head in order to make both the interpretation of the argument positions and the arity transparent to the user.

`_gp.children(self=_child)` makes `_child` refer to an instance of (i.e. a member of) the set `_gp.children`. The result of the query

```
grandChild(grandParent=gustaafVI)
```

will be a set of record objects that satisfy the `grandChild` predicate, each having two attributes `grandParent` and `grandChild`. For example

```
{
(grandParent = gustaafVI; grandChild = karelXVI),
(grandParent = gustaafVI; grandChild = margaretha),
 ..
}
```

Let us now try to solve the following query: find all persons of the same generation as `karelXVI`.

Two persons are of the same generation if they are siblings:

```
siblings(sibling1=_x;sibling2=_y)     :-
    _x.parents(self=_aParent),
    _y.parents(self=_aParent).

sameGeneration(person1=_x;person2=_y)     :-
    siblings(sibling1=_x;sibling2=_y).
```

or if their parents are of the same generation:

```
sameGeneration(person1=_x;person2=_y)     :-
    _x.parents(self=_aParentOfX),
    sameGeneration(person1=_aParentOfX;person2=_aParentOfY),
    _y.parents(self=_aParentOfY).
```

The answer to the query is then given by

```
sameGeneration(person1=karelXVI).person2
```

where the projection yields a new set object containing exactly the persons that are of the generation of `karelXVI`:

```
{solution.person2 | solution:sameGeneration(person1=karelXVI)}
```

Note that predicates can refer to both objects with instances (like `_x.parents` in `sameGeneration`) and other predicates (like `sibling` in `sameGeneration`). More general, in this context, an object with instances can be regarded as a predicate with extension the set of its instances. On the other hand, it is possible to look upon predicates as *virtual classes* where the

instances or a subset of the instances is only created when needed (and later destroyed when not referenced).

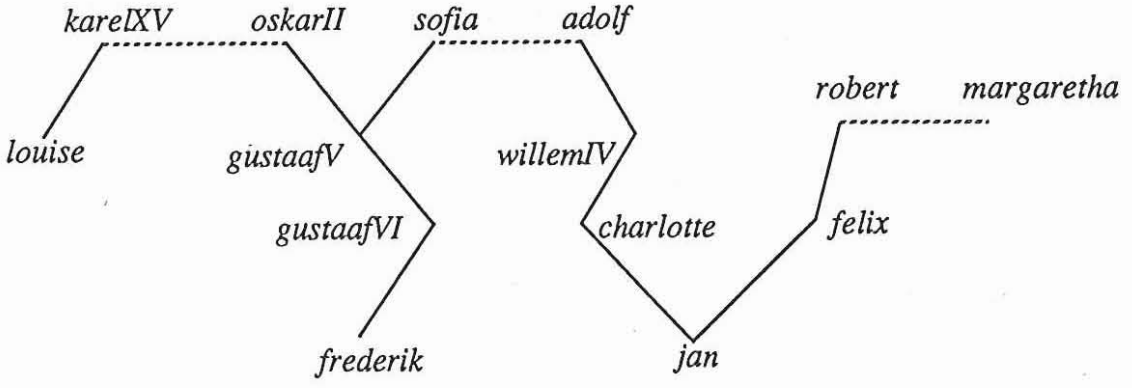
3. Deductive query optimization

In this section, we illustrate that the bottom-up evaluation of queries needs to be augmented with some optimization strategies in order to be of practical use.

Let us rewrite the sameGeneration (sG) predicate in a less verbose way in order not to overload the discussion below.

```
sG(X;Y) :- sibling(X;Y).
sG(X;Y) :- parent(X;X1), sG(X1;Y1), parent(Y;Y1).
```

This is a linear recursive Datalog program consisting of two rules. Let `sibling` and `parent` be base predicates; `sG` is a derived predicate. Consider the set of facts as shown in the next figure. The dashed lines denote `sibling` facts (e.g. `sibling(sofia;adolf)`) while the solid lines represent `parent` facts (e.g. `parent(frederik;gustaafVI)`).



The *bottom-up* evaluation of the query `sG(X;Y)` goes as follows. Let `sgSet` be a set that keeps track of the tuples for which the derived predicate `sG` holds. This set is initialized to the empty set and its members are generated by repeatedly applying the rules for `sG` to both the (`sibling` and `parent`) facts, and the tuples in sgSet:

```
begin
  initialize sgSet to empty;
  for each (X;Y) in sibling do
    add (X;Y) to sgSet;
  repeat
    for each (X;X1) in parent, (X1;Y1) in sgSet, (Y;Y1) in parent
      do add (X;Y) to sgSet;
  until no new tuples are generated;
end
```

Thus, for our configuration, `sgSet` is constructed in four steps and contains the answers to the query `sG(X;Y)`:

sgSet							
first rule		first iteration second rule		second iteration second rule		third iteration second rule	
X	Y	X	Y	X	Y	X	Y
sofia	adolf	gustaafV	willemIV	gustaafVI	charlotte	frederik	jan
adolf	sofia	willemIV	gustaafV	charlotte	gustaafVI	jan	frederik
karelXV	oskarII	louise	gustaafV				
oskarII	karelXV	gustaafV	louise				
robert	margaretha						
margaretha	robert						

As another example, consider the query `sG(frederik;Y)` of which the first argument is *bound* to `frederik`. The bottom-up evaluation of this query is exactly the same, and the answer is given by

```
{Y | (frederik;Y) is in sgSet}
```

i.e. the singleton set `{jan}`.

Note that in this case, the algorithm is rather inefficient since we ask for the same generation of `frederik` and we compute the same generation of all persons. In other words, we consult more facts than actually needed. The problem is that the bottom-up evaluation was done without taking into account the additional binding information.

Since we know the answer set `{jan}` to the query `sG(frederik;Y)`, we can also compute the set of *relevant facts* for this query by making the derivations starting from `sG(frederik;jan)`:

sG(frederik;jan) =>

parent(frederik;gustaafVI)

sG(gustaafVI;charlotte) =>

parent(gustaafVI;gustaafV)

sG(gustaafV;willemIV) =>

parent(gustaafV;sofia)

sG(sofia;adolf) =>

sibling(sofia;adolf)

parent(willemIV;adolf)

parent(charlotte;willemIV)

parent(jan;charlotte)

The set of relevant facts consists of the facts that appear in these derivations. It is indeed a subset of the set of consulted facts. Thus, whenever a query contains bound arguments, the bottom-up method tends to become inefficient because the set of facts consulted during evaluation may be much larger than the set of facts that are relevant to the query.

So why not using the set of relevant facts instead of all facts? Observe that we need the answer to the query in order to derive the set of relevant facts, therefore it is not feasible to use this set in the query evaluation (since the latter would already be done).

A usual approach in a query optimization strategy is to attempt to rewrite the program so that bindings in the query can be used to restrict the set of facts to which the rules will be applied. For instance, for the query `sG(frederik;Y)`, we want to use the binding (`frederik`) in the determination of a set of facts, which should be a subset of all facts, containing all relevant facts for the query and which will then be used in the query evaluation.

The following shows how the *magic sets* optimization strategy deals with this idea. The method uses some auxiliary predicates, called *magic predicates*, that compute sets, called *magic sets*. The rules for these magic predicates are derived from the original rules taking into account the query bindings and propagating the binding information down to the facts. These magic predicates are put in the bodies of the original predicates so that the latter will fire only for members of the magic sets.

For example, for the query `sG(frederik;Y)` the magic predicate for `sG` is:

```
magic.sG(frederik).
magic.sG(X1) :- magic.sG(X), parent(X;X1).
```

while the original rules for `sG` are modified as follows:

```
sG(X;Y) :- magic.sG(X), sibling(X;Y).
sG(X;Y) :- magic.sG(X), parent(X;X1), sG(X1;Y1), parent(Y;Y1).
```

The magic set computed by `magic.sG` is the set of ancestors of `frederik`. In the bottom-up evaluation of the modified `sG` predicate, the fact `sibling(X;Y)` will only be used if `X` is in the magic set. A similar thing holds for the first occurrence of `parent` in the recursive rule.

There is a wide variety of optimization methods each trying to compromise the tightness of the set of potentially relevant facts and the overhead caused by its computation. The two extremes are obviously the set of all facts, for which no computation is needed, and the set of relevant facts, which is the tightest possible and also the most expensive one.

The database environment of the KIWI system uses a combination of different optimization strategies, some of which are based on the idea underlying the *magic sets* method: the *minimagic* method, the *counting* method, the *magic counting* method, etc.[6,8,9] . Applying these techniques, Datalog programs are compiled into efficient code: safe and efficient prolog programs where the bottom-up computation is enforced. Such a compiled query runs up to 20 times faster than the direct prolog execution of the original query.

4. NAU

Currently, there exist three user-interfaces to the KIWI system, each aimed at a different kind of user. The naive user interface offers simple browser functions. The knowledge engineer's interface (KEI) has been developed in an attempt to provide a user-interface that is in full harmony with the OOPS+ language and provides facilities for browsing, updating, and activating functions in the knowledge base. Thus this interface supports a full view on the knowledge representation formalism in a window-based format. Finally, it was realized that none of the above interfaces is likely to be acceptable to an end-user if he wants to do more than the most trivial operations. So an application would still need to develop its own interface. To facilitate this, an experimental applications user-interface (NAU, "Not Yet Another User interface") has been developed.

NAU extends the underlying object-oriented language (OOPS+) with a limited number of predefined interface objects and types (classes) that have an interpretation in terms of the I/O facilities (mouse, windows, etc.).

Using these interface objects, it becomes straightforward to define a customized representation of an object (or a class of objects), which also includes dialogue features that define the possible interaction with an object through buttons, edit windows etc.

This is further facilitated by using other object-oriented features of the underlying language such as polymorphism and default values (inheritance). This allows e.g. to limit the number of functions that belong to the interface to just a few, which benefits the readability and correctness of the application.

These interface tools have now been tested for a number of applications and the results indicate that programmer productivity is greatly increased and it becomes possible to develop prototype knowledge-base applications with attractive user interfaces in a very short time.

Typically, applications developed using this interface can be regarded as customized browsers. In addition to the usual browsing activities, a number of predefined transactions that constitute the application are associated as methods with the proper objects. They can be called directly from the browser by activating the corresponding visualization, e.g. a button.

In order to facilitate the incremental development of such a customized interface to a knowledge base, we have provided a default browser (mostly written in OOPS+) that associates a standard visualization with each object. It is then possible to customize the presentation of the objects that are more relevant to the application. Due to inheritance, the browser run-time system will use the customized visualization whenever it exists and otherwise default to the standard one.

In the rest of this section we will show some customization facilities.

There are a number of different kinds of windows supported by NAU. Some information, such as the size, which is common to all kinds is stored in a *frame*, i.e. a window without contents. The type `FRAME` is defined in OOPS+ as follows :

```
FRAME = (label = Str default nil;
         size = COORDINATE default nil;
         location = LOCATION default ( X = 0; Y = down );
         action = EVENT -> nil default ..;);
```

The meaning of the various attributes is as follows: if `label` is not `nil` then the window will be displayed with a labeled border. If `size` is not specified (as a `width` and/or `height`), the window will fit its contents. Specifying the size may cause a scrollbar to be automatically added whenever the size of the contents would exceed the given size. The location attribute may be used to specify the position of the window frame within an encompassing superwindow. Besides an absolute location `(X=..;Y=..)`, it is also possible to specify a location relative to both the previous window in a layout sequence (down, up, left, right) or to the encompassing superwindow (top, center, bottom, etc.).

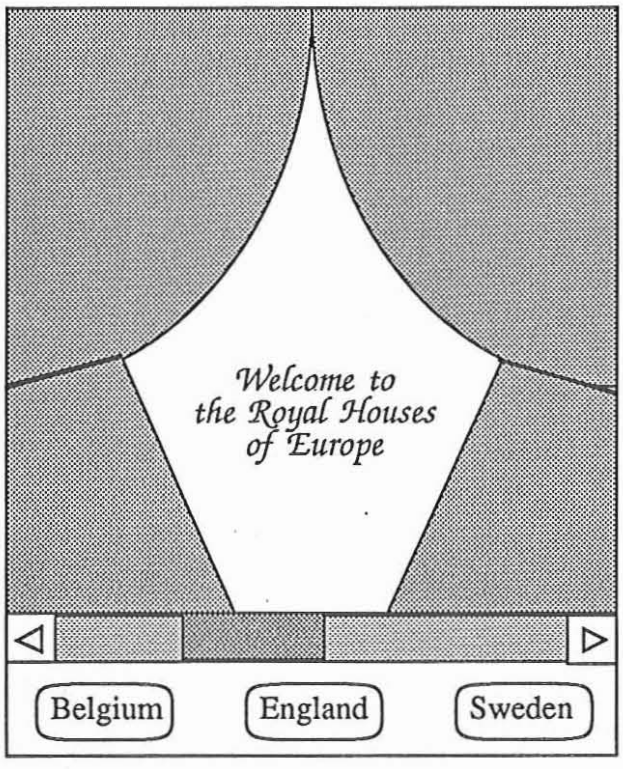
The action function specifies what happens upon the occurrence of an event that is to be handled by the window. Relevant information about this event will be stored in an EVENT object that is passed to the action function.

A complete window consists of a frame and a specification of its contents. Below is a summary of the kinds of windows that are available in NAU.

- A `form` window serves as a container of other windows. It has a layout attribute that consists of a sequence of windows, any of which may again be forms. Upon display, the form is filled with the subwindows as specified in the layout attribute.
- `Text` windows for displaying text.
- `Edit` windows that allow update of text.
- `Pictures` that display raster images.
- `Buttons` which are typically associated with actions that perform some application function.

- `Tables` that provide automatic alignment and resizing (both horizontally and vertically) between rows of subwindows. Note that there is no restriction on the kind of subwindows that are supported by tables.
- `Choice` windows which allow an event that corresponds to the selection from a collection of alternatives.

For example, let us consider the opening window for our case study :

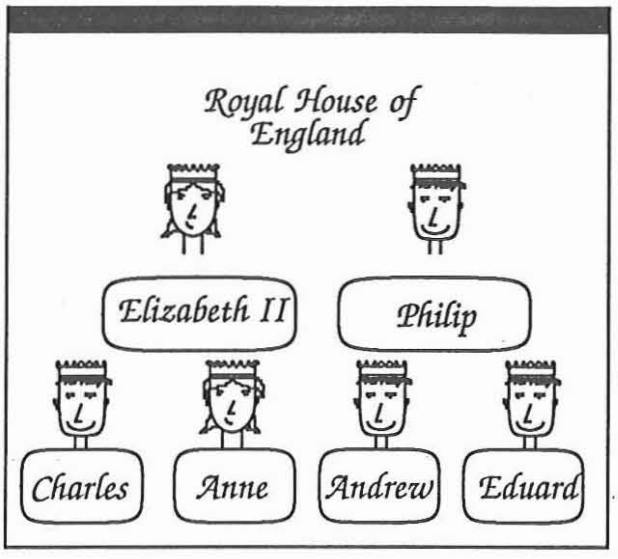


This window uses the default attributes of FRAME and has two subwindows: a picture window, and a table window which in turn has subwindows (text-windows):

```
WINDOW welcomeWindow =
    (
    layout = <
       PICTURE <- (pic = welcome.pic),
       TABLE <- (size = (width = 20);
                 rows = < <
                    BUTTON <- (txt = "Holland";
                               action=display(window=hollandWindow))
                    ..
                    BUTTON <- (txt="England";
                               action=display(window=englandWindow))
                    ..
                        > >)
             >)
```

where `welcome.pic` denotes a picture; `display` is a function with one argument labeled `window`, and `< >` denotes the sequence constructor in OOPS+.

Whenever a button or a text is selected, the associated function (`action`) gets activated. For instance, selecting 'England' in the opening window yields:



which is specified in OOPS+ as follows :

```
WINDOW englandWindow =
  (
  label = "";
  font = zapfChancery;
  layout = <
      TEXT <- (txt="Royal house of @ England";
               location = (X=center));
      TABLE <- (location = (X=center);
                rows = < <
                   PICTURE <- (pic=queen.pic),
                   PICTURE <- (pic=king.pic)
                   >,
                   <
                   BUTTON <- (txt = "Elizabeth II";
                    action=display(window=
                                   personWindow(person=elizabeth))),
                   BUTTON <- (txt = "Philip";
                    action=display(window=
                                   personWindow(person=philip))),
                   > > ),
```

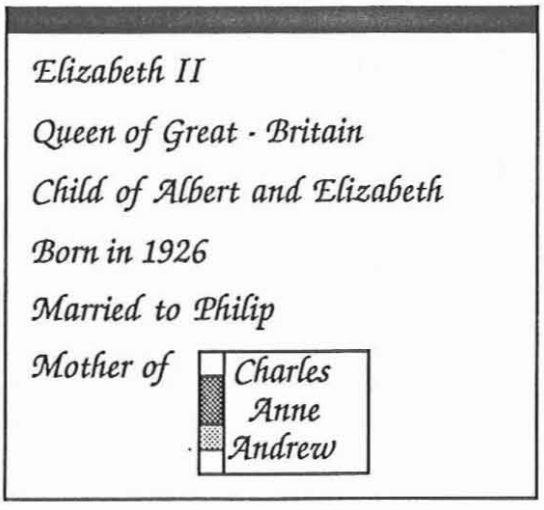
```
    TABLE <- (location = (X=center);
              rows = < <
                 PICTURE <- (pic=prince.pic),
                 ..
                 >,
                 <
                 BUTTON <- (txt = "Charles";
                 action=display(window=personWindow(person=charles))),
                 ..
                 > > )
    > )
```

Note that in this example, all of the actions associated with buttons call the function `display` with as `window` argument the result of another function call `personWindow(person = ...)`. The latter function creates and returns a window that represents a number of features of the person at hand :

```
personWindow = WINDOW
    {
    return (
        label = "";
        layout =
            <
            TEXT <- (txt = person.name);
            TEXT <- (txt = person.title);
            TEXT <- (txt = "Child of " + father(p=person).name +
                           " and " + mother(p=person).name),
            TEXT <- (txt = "Born in " + person.birthdate),
            TEXT <- (txt =
                {
                if person.sex == "male" then
                    return("Father of")
                else
                    return("Mother of")
                } () () ),
            TABLE <- (size = (height = 3),
                      location = (X = right) ;
                      rows = .. )
            >
            )
    }(person = personClass);
```

This function should be called with one argument `person` that must refer to an instance of `personClass`.

Thus selecting the ElizabethII button in the previous window will result in :



5. Conclusion

Many interesting results have been achieved in the KIWI project, some of which are outlined in this paper. The main issues of the project are knowledge representation languages with object-oriented flavor, graphical user interfaces and tight combination of logic programming and databases. Prototypes have been developed using C-language and are running on a Unix workstation.

While KIWI was mainly a research project, its results will now be used and further developed in a more industrial Esprit II project, called KIWIS.

6. Acknowledgement

Many thanks to the KIWI team. We enjoyed working with them very much.

This research was supported in part by the EEC Esprit program under contracts P1117 and P2424 (Esprit II).

References

1. *On Knowledge Base Management Systems,* Springer-Verlag, 1985.
2. D. Sacca, D. Vermeir, A. D'Atri, J. Snijders, G. Pedersen, and N. Spyratos, "Description of the overall architecture of the KIWI system," in *Proceedings of the Esprit Technical Week*, 1985. Elsevier Publ. Co.
3. The KIWI team, "A System for Managing Data and Knowledge Bases ," in *Proceedings of the Esprit Conference*, pp. 826-844, North-Holland , 1988.
4. E. Laenens and D. Vermeir, "An overview of OOPS+, an Object-Oriented Database Programming Language," *Proceedings of the ECOOP'88 conference*, pp. 350-373 , Springer-Verlag , Oslo, August 1988.
5. E. Laenens and D. Vermeir, "A Language for Object Oriented Database Programming," *Journal of OBJECT-ORIENTED Programming* , 1989. to appear

6. D. Sacca and C. Zaniolo, ''Magic Counting Methods,'' *ACM SIGMOD Proceedings*, pp. 49-59, 1987.

7. E. Laenens, F. Staes, and D. Vermeir, *A Customizable Window-Interface to Object-Oriented Databases*, 1989. Submitted for publication

8. D. Sacca and C. Zaniolo, ''The Generalized Counting Method for Recursive Logic Queries,'' *ICDT '86 Proceedings*, 1986.

9. D. Sacca and C. Zaniolo, ''Implementation of Recursive Queries for a Data Language based on Pure Horn Logic,'' *Proc. Fourth Int. Conference on Logic Programming, Melbourne, Australia*, 1987.