6

Flexibility of visual languages for data manipulation

Y. DENNEBOUY
Swiss Federal Institute of Technology
EPFL-DI-LBD CH 1015 Lausanne
(41 21) 693 5213 Fax (41 21) 693 51 95
Yves.Dennebouy@di.epfl.ch

Abstract

Quality of visual languages for data manipulation can be defined in terms of readability, direct manipulation, power of expression, adaptability, flexibility and level of abstraction. As some papers already defined methods to evaluate some of these qualities, flexibility is till now a fuzzy concept. This paper tries to quantify flexibility of such languages. For this purpose we define of a set of errors that can occur during the definition of a query. This grid of errors is applied to several tools or prototypes.

Keywords

Visual interfaces, visual languages, databases, query languages

1. INTRODUCTION

Two paradigms have gradually emerged in the area of user interfaces for database management systems (DBMS) : visual interaction and object support. The former became achievable thanks to the availability of powerful workstations and efficient window management systems, which is a standard nowadays for application developers in general, and for CASE or DBMS vendors in particular. The second paradigm, object support, is concerned with the combined evolution of database techniques and user requirements. On the one hand, database researchers have succeeded, via the object-oriented approach, in developing an operational alternative to the relational management of flat data structures only. In this way, complex

S. Spaccapietra et al. (eds.), *Visual Database Systems 3*

objects can be defined and manipulated. On the other hand, the database approach has entered many new application areas, with many new potential users who are normally unwilling to trade their usual view of application objects for flat representations but were obliged due to technological constraints. Both these phenomena lead us to the present situation where the support of complex objects is the standard for new interfaces and design methodologies.

Based on this evolution, and also considering that the complexity of textual languages for OO models will probably exceed the capabilities of average users, despite the SQL-like syntax, we expect that numerous visual interfaces, supporting definition and manipulation of complex objects, will be offered by CASE or DBMS vendors in the next decade. For the time being, however, an established know-how exists for data definition only. However, visual data manipulation is confined to some SQL generators which operate on the relational database and allow a query to be built using menus for selecting tables of interest and the operations to be performed on these tables. The design of visual interfaces supporting queries on objects is still an open research issue. While some of the available OO-DBMS offer visual browsing facilities and textual (SQL-like) assertional query languages, none of which supports visual query formulation. A limited number of prototypes support visual query formulation to some extent. PASTA-3 (Kuntz, 1989), QBD* (Angelaccio, 1990) and Graqula (Sockut, 1993) are among the most advanced. However, they generally fail in providing full support of the object paradigm since the relational structure of the underlying database influences the language. They also fail in providing a totally visual approach: they provide a visual formalism for underlying operators, in a procedural way of thinking, rather than visual specifications which could be termed as declarative because they do not presuppose a one-to-one mapping onto the underlying algebra. Moreover, they sofar have given very little attention to the complexity of the visual query formulation process, resulting in limited facilities for correcting the current formulation. These facilities will play a key role in future systems, where query engineering (the process where users will finally succeed in formulating a query after a number of correction steps) might become standard in the development of application programs.

This paper describes an approach which has been designed to offer a visual data manipulation interface, where the emphasis is on its visual nature, on supporting direct manipulation of complex objects and on providing for query re-formulation with minimal complications for users. A detailed description of the approach may be found in (Dennebouy, 1993). The current implementation is based on an objects+relationships data model, namely ERC+ (Parent, 1992), and the associated algebra. The database schema is displayed using ER-like diagrams (entity-relationship). Visual queries are translated into the ERC+ algebraic operators and executed on an ERC+ machine currently implemented on top of an OO-DBMS.

The paper focuses on the query formulation and re-formulation processes. Section 2 recalls the basic steps in query formulation. Section 3 analyzes the errors which may lead the user to backtrack the process to correct the current formulation. Section 4 checks how three existing prototypes support the correction process. Section 5 presents our approach. The conclusion points towards future work.

2. QUERY FORMULATION

There are two ways to extract data from a database. One way is extensional, by navigating through the database at the occurrence level, picking data of interest while flicking through.

Browsers are the tools for this. The other way is intentional, by formulating a query: asserting which properties have to be matched by objects and/or values in a database in order to be selected to be included into the resulting set of objects/values. Query formulation is normally in terms of the types in the database schema in order to define which object classes are relevant for the given query and in terms of attribute values and of links among objects in order to define which subsets of those populations are relevant. A query also defines which data from the relevant subsets has to be put into the result (typically a projection operation) and how these data have to be structured for the end result (unless the result is by definition a flat first normal form relation).

In a classical query language as SQL, we can find these different specifications expressed as SELECT-FROM-WHERE blocks: the FROM clause defines the relevant classes, the WHERE clause restricts these classes to the relevant subsets, finally the SELECT clause specifies the projected data.

In a visual environment, the definition of the relevant object classes (the FROM clause) is performed by visualizing the database schema on the screen (usually as a diagram, but also as a list in a menu) and by clicking on the desired types to lift them out from the schema into the query subschema (alternatively, by clicking on undesired types to remove them from the schema, which gradually reduces to the target query subschema). The query subschema is a subgraph of the original schema graph. It is easy to show that if the query subgraph includes a cycle, visual queries cannot be interpreted without ambiguity (Auddino ,1992). In these cases some transformation has to be done to obtain an acyclic graph. The latter is referred to hereafter as the **query frame**: the acyclic graph containing all object (and link) types relevant for the query. Unless the result is a flat Cartesian product of all objects in the frame, the graph has to be turned into a tree whose root denotes which object class is being queried.

For example, assuming the schema in figure 1, figure 2 illustrates the frame for the query "Give the information on persons, with the cars they own, for all persons who insure Ford cars that are all of the same color and either one of the Ford cars has been made after 1993 or all of them have been made before 1960" (note: this query is not meant to be semantically meaningful, but to illustrate the various issues we discuss in the paper). Since the query result will be in terms of people, the type person is the root of the query. The root type is visually denoted using bold lines.

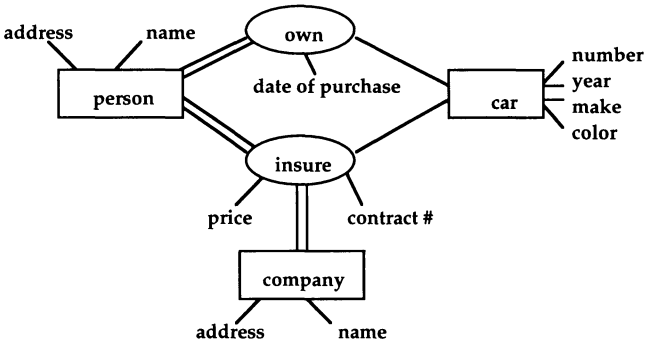


Figure 1: the example database schema.

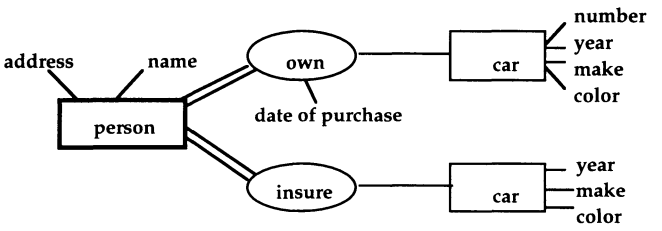


Figure 2: the frame for the example query.

The definition of the relevant subsets of the classes in the query (the WHERE clause) is expressed as conditions (predicates) on instances of the types in the frame. Only those objets that match the conditions will contribute to the result.

The definition of whose data is to be presented to the user, and of the manner in which it is to be presented (the SELECT clause) can be seen as the definition of a new, virtual object type. We will refer to this new type as the structure of the result (**query structure**, in short). The structure contains certain items (classes, links, attributes) from the frame, but not necessarily all of them. These items may be renamed. The query structure for the example query above is shown in Figure 3.

Although the condition on person in the example query could have been written as a single query, for the purpose of our discussion in sections 3 and 4, the following two predicates have been defined:

P1 that must be evaluated first, it states that the insured cars under consideration are restricted to Ford cars, it reduces the set of insured cars and does not reduce the set of persons.

P1: insure | insure.car.make = 'Ford'

And then P2,that reduces the set of persons, it states that there must be an insurance link (verifying P1) such that for all other insurance links for this person it is true that the corresponding cars have all the same color as the car in the first insurance link (i.e., all Fords of this person have the same color) and either all the cars insured by this person are made before 1960 (*forall* $insure_2$ ... $insure_2$.car.year < 1960) or one of them has been made after 1993 (*exist* $insure_1$ ... $insure_1$.car.year ≥ 1994).

P2: person | exist $insure_1$ forall $insure_2$
and (($insure_1$.car.color = $insure_2$.car.color
or ($insure_2$.car.year < 1960,
$insure_1$.car.year ≥ 1994))

The overall query formulation process, with its input and output, is depicted in figure 4.

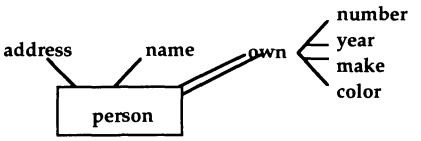


Figure 3: the query structure for the example query.

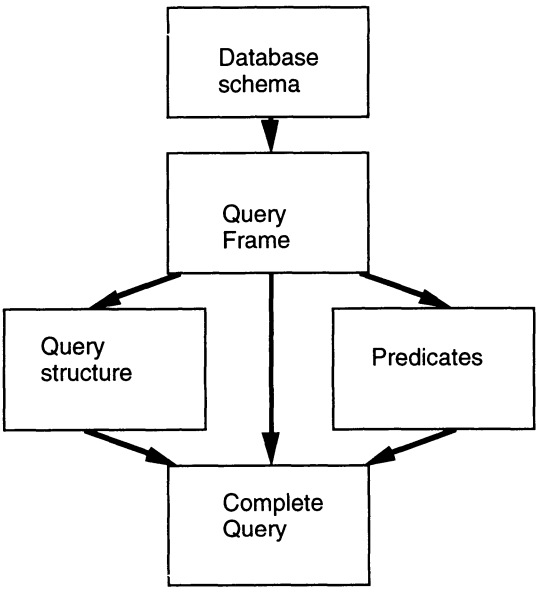


Figure 4: steps in the query formulation process.

3. ERRORS IN QUERY FORMULATION

A logical way of writing a query is to define in a sequence the three facets of the query. This is the 'methodology' provided and enforced in most of the existing tools. This sequence may not exactly correspond to the user needs. As shown in figure 4, there is no inherent ordering between the definition of the query structure and the definition of the predicates. Moreover, users can have a mental image of a query as a composition of subqueries or as a refinement of increasingly complex queries. Users can also (and will most probably) make some mistakes, which they will notice later, maybe after the first evaluation of the query.

Thus, a metric for a visual language cannot be defined by the expressive power of the underlying language only (i.e., by the set of all possible queries) (Batini, 1991). The quality of a visual language should also include a flexibility measure: 1/ to what extent the language frees

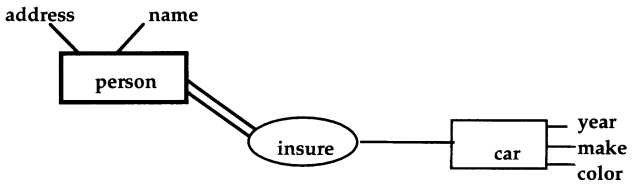
users from having to follow a specific sequence in query formulation, and 2/ which facilities are provided to correct errors.

In order to help quantify the flexibility of a visual language, we list below all possible errors that the user may make during the query formulation process. Errors are performed on one of the three facets. This checklist will be confronted in the next section with the facilities provided by current prototypes. Illustrations are based on the example schema and the example query from section 1.

1 - Frame definition errors

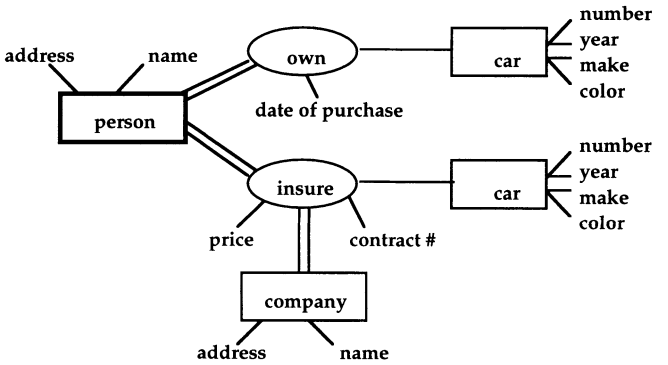
11 - Missing objects in the frame

The user forgot to include owned cars in the frame. The frame reduces to:



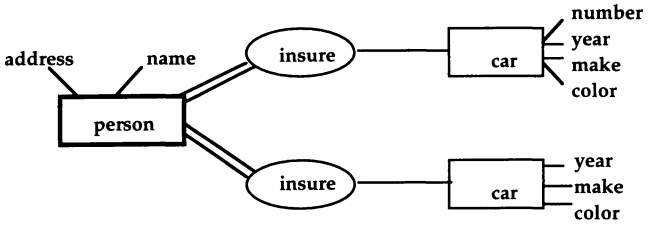
12 - Having non relevant objects in the frame

If the user forgot to take out the company, the frame will be:



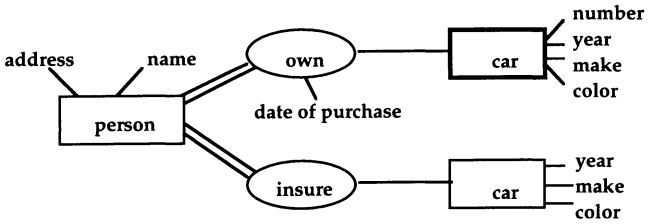
13 - Using a wrong path in the frame

The user selected the 'insure' relationship instead of the 'own' relationship:



14 - Getting a wrong root object type

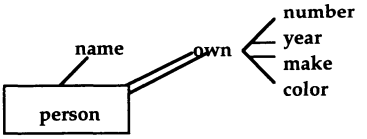
The user designed the frame to obtain cars instead of persons:



2 - Structure errors

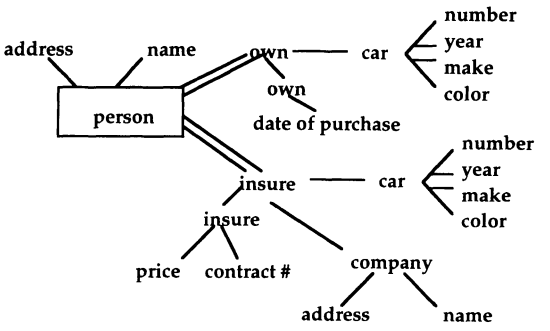
21 - Missing attributes in the structure

The user deleted the 'address' attribute to the 'person' entity type:



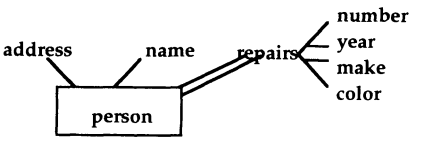
22 - Having undesired attributes in the structure

The user omitted to delete the 'insure' and 'company' attributes within the insurance information:



23 - Bad (re)labeling in the structure

The user erroneously renamed the car attribute as repairs:



24 - Creating a predicate on wrong objects

The following predicate defined on the attribute insure, instead of the one normally defined on *person*, will keep all persons and for them their 'special' Ford car (as previously defined).

insure | forall insure$_2$
and (insure.car.color = insure$_2$.car.color,
or (insure$_2$.car.year < 1960,
insure.car.year ≥ 1994))

3 - Inner predicate errors

31 - Misquantified variable in a predicate

The user can make a mistake in the choice of a quantifier:

Person | *exist* insure$_1$ *exist* insure$_2$
and (insure$_1$.car.color = insure$_2$.car.color,
or (insure$_2$.car.year < 1960,
insure$_1$.car.year ≥ 1994))

32 - Wrong order in quantification

The user can make a mistake in the ordering of the quantifiers:

Person |*forall* insure$_2$ *exist* insure$_1$
and (insure$_1$.car.color = insure$_2$.car.color,
or (insure$_2$.car.year < 1960,
insure$_1$.car.year ≥ 1994))

33 - Wrong comparison type

Person |*exist* insure$_1$ *forall* insure$_2$
and (insure$_1$.car.color = insure$_2$.car.color,
or (insure$_2$.car.year = 1960,
insure$_1$.car.year ≥ 1994))

34 - Wrong value for a constant

Person |*exist* insure$_1$ *forall* insure$_2$
and (insure$_1$.car.color = insure$_2$.car.color,
or (insure$_2$.car.year = 1960,
insure$_1$.car.year ≥ 1994))

35 - Missing variables in a predicate

The user can forget one condition concerning the year of the first quantified insured car

Person |*exist* insure$_1$ *forall* insure$_2$
and (insure$_1$.car.color = insure$_2$.car.color,
insure$_2$.car.year < 1960)

36 - Introducing too many variables in a predicate

Person |*forall* insure$_2$ *exist* insure$_1$
and (insure$_1$.car.color = insure$_2$.car.color,
or (insure$_2$.car.year < 1960,
and (insure$_1$.car.year ≥ 1994,
insure$_2$.car.make = 'Ford')))

37 - Comparing two variables to each other instead of comparing each of them to a constant

Person |*forall* insure$_2$ *exist* insure$_1$
and (insure$_1$.car.color = insure$_2$.car.color,
insure$_2$.car.year = insure$_1$.car.year)

38 - Comparing two variables to a constant instead of comparing them to each other

Person |*forall* $insure_2$ exist $insure_1$
and (and ($insure_1$.car.color = 'red', $insure_2$.car.color = 'red'),
or ($insure_2$.car.year < 1960,
$insure_1$.car.year ≥ 1994))

39 - Wrong combination of logical operators

Person |*forall* $insure_2$ *exist* $insure_1$
or (*and* ($insure_1$.car.color = $insure_2$.car.color,
$insure_2$.car.year < 1960),
$insure_1$.car.year ≥ 1994)

4. EVALUATION OF SOME GRAPHICAL QUERY EDITORS

Principles

In this section we analyze three graphical tools and check how they allow the user to recover from the errors listed above. There are three ways to recover errors. The best one is to make the change directly where the error appears and to preserve the work that has already been done. The second is to backtrack in the tool to the state where the mistake occurred, correct the error and redo the steps that followed the erroneous specification. Finally, the worst case is when the user has to redo the entire query formulation from scratch.

The tools we analyze below are defined on first normal form models, either relational or ER. With first normal form models, the structure of a query is flat, and so very different from its frame. The predicates are notably easier to write. As in SQL, there is no need to define several variables on the same attribute. The use of several variables in the same table is described by the use of several copies of the same table in the frame. As we shall see in the section presenting our visual language, which is defined on a complex object's model, this trick cannot be used with non first normal form models.

As these models do not include the notion of object identity, there is no concept of root object type of a query, so error 14 will not be discussed.

QBD*

The query process in QBD* (Angelaccio, 1990) is a transformation of the database schema, performed as a sequence of several predefined steps. The query representation is unique, and many operations are non reversible. Errors leading to unwanted objects in the schema (12, 22) can always be corrected by removing these objects, but, as the original schema is no more available, errors due to the lack of some objects in the query workspace (11, 13, and 21) cannot be corrected and can lead to the complete rewriting of the query. Attributes can be

renamed, so error 23 can easily be corrected. As predicates are not related to a particular object, error 24 is not relevant .

As the QBD* paper only shows simple predicates, without any universal quantification, we could not check how the quantification order is defined, so errors 31 and 32 are undetermined. If a variable is missing in a predicate, the correction may require to use another copy of a table in the query so error 35 cannot be corrected. Edition of a predicate that include all necessary variables can be done easily (33, 34, 36, 37, 38). We also cannot draw a conclusion in case of wrong combinaison of logical operator (error 39), because of the lack of examples showing a complex combination of logical operators.

In conclusion, four errors cannot be corrected, eight can be corrected, three are not relevant to the QBD* model and for the last three we do not have enough information to conclude.

PASTA-3

PASTA-3 (Kunt, 1989) offers a multiwindow environment. The objects used in a query are highlighted in the database schema (which is presented in two different windows: one for the relationships graph and one for the specialization hierarchy). PASTA–3 allows almost all possibilities of error recovery, but mainly by the use of menus. Frame definition errors (11, 12, 13) are easily corrected mainly due to the fact that database schema is still available.Structure errors that didn't implies predicates (21, 22, 23) can be easily recovered. As it is impossible to define several predicates on the same query, it requires the use of subqueries to write queries like our example query. The subquery is the selection of Ford cars, this subquery is reused in the main query. So it is quite difficult to recover from error number 24, a predicate defined on the wrong object. Conditions in the predicate can be moved to restruture the predicate, it permits correction a wrong combination of logical operator (error 39). Quantification is by default existential but maybe changed (31) and quantification order can be change in moving the first left or up the second (32), comparison type are choosen in a menu (33) and values are choosen in the set of possible value or directly typed by the user (34), condition can be added or removed to a predicate and so correct errors from 35 to 38.

In conclusion, PASTA-3 is a very flexible environment, even if most of the manipulations is made by menus instead of direct manipulation.

Graqula

Graqula (Sockut, 1993) is also a multiwindow environment. It always contains a display of the database schema. This allows the representation of the frame of the query. Errors on the definition of the frame can be corrected (11, 12, 13). Attributes' member of the structure are presented in reverse video: they can either be removed or added to the structure. This permits the correction of errors 21 and 22.

Operation to rename attributes is not explicitly described. It seems that the renaming of entity types is automatic, so we cannot give a definite answer for the correction of error 23.

The construction of complex predicates is done by making frames, which can always be removed and created. This allows correction of errors 31 and 32, with some rewriting. A comparison operator can be changed in the 'OP' row (it corrects error 33). In the same way, the user can change the value in the VALUE row to correct error 34. Adding or removing variables in a predicate seems to be easy, so errors 35 and 36 only have minor consequences.

The comparison of variables is realized by drawing a vertex between the variables: this can always be drawn or removed; the comparison to a constant can also be created or removed. Errors 37 and 38 can be corrected. A wrong combination of logical operators can also be corrected. This can sometimes be complex and lead to the introduction of new objects in the query, so error 39 can be corrected but again with some rewriting.

In conclusion, all errors can be corrected, eight easily, four with some rewriting, one is not relevant to this kind of models (relational and ER) and for another one we do not have enough information.

5. SUPER MANIPULATION TOOL

Short description

Our manipulation tool is part of a set of visual interfaces which together form a conceptual front end, called SUPER, to existing OO or relational DBMSs (Auddino, 1993). SUPER relies on a semantic data model, ERC+, supporting complex objects and object identity (Parent, 1993). The manipulation editor allows formulation of retrieval and update queries, but only the former will be discussed here. Firstly we shall briefly describe our tool and then describe more precisely the construction of predicates, which are, as we have already stated, more complicated in an NF2 model.

User interaction with the manipulation editor uses three windows. The first window contains the diagram of the database schema (figure 5). The second window, named the Working Schema window (WS), allows the user to create the query frame from parts of the database schema (figure 6). The third window contains the structure of the query.The structure of an ERC+ query is still a tree (as the frame). This make a big difference with first normal form models, in which structure is transformed in a flat join of all attributes.

A temporary structure is automatically built by the tool as soon as the root of the frame has been defined by the user. The final structure will differ from this one only if the user specifies that some attributes, which have been kept because they are involved in a predicate, have to be hidden in the result. Predicates are expressed against the structure, not against the frame, because of ergonomic reasons: 1/despite they are equivalent, the structure gives a more intuitive visual image of the resulting objects than the frame, where the resulting objects still appear as a net of components, and 2/ this allows the user to visually separate the mental task of designing the frame from the task of defining the predicates.

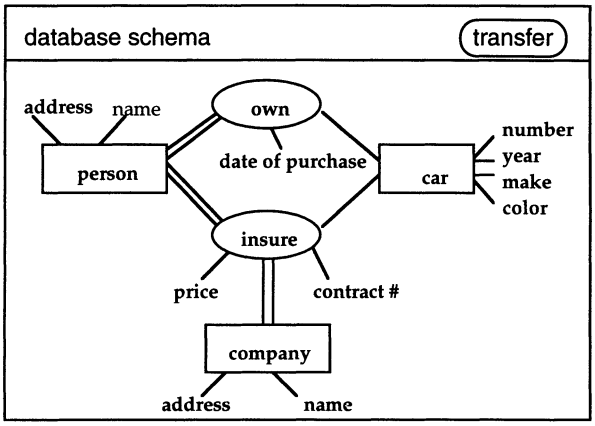


Figure 5: a SUPER database schema window.

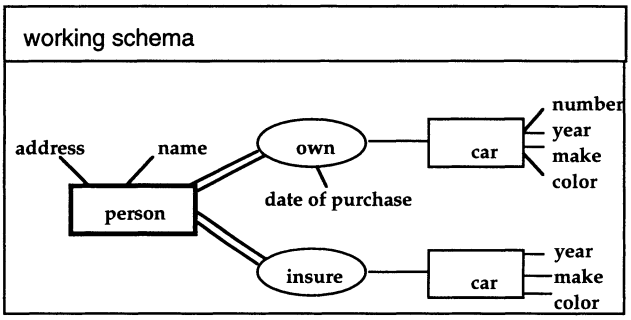


Figure 6: a SUPER working schema window.

The definition of the frame and of the structure of a query are quite easy. Attributes that do not belong to the structure are shaded (as insure in figure 7).

The definition of conditions is, itself, rather complex.

One variable is implicitly defined for each attribute of the structure and can be used in predicates. If the user needs to use a second variable on the same attribute, (s)he can create a variable that is shown in italic, linked to the attribute it is defined on. In our example (figure 7), the user has created, from the attribute insure_ford, a new variable *insure_ford1* that (s)he can use in the definition of P2.

A predicate can be attached to the root object or to one of its attributes. If attached to the object (P2 in our example) it selects instances of the object. If attached to an attribute, it selects values of the attribute in all instances of the object. For example, P1 restricts, for each person, the values of insure to those values that match the predicate. As in our example, this may be used to define nested predicates. To this purpose, a copy of the restricted attribute is

shown to denote the restricted set of values: insure_ford in figure 7 denotes the result of P1. In this figure, the variable insure ranges over any of the insured cars, while variables insure_ford and insure_ford$_1$ range over only insured Fords (which is needed in P2).

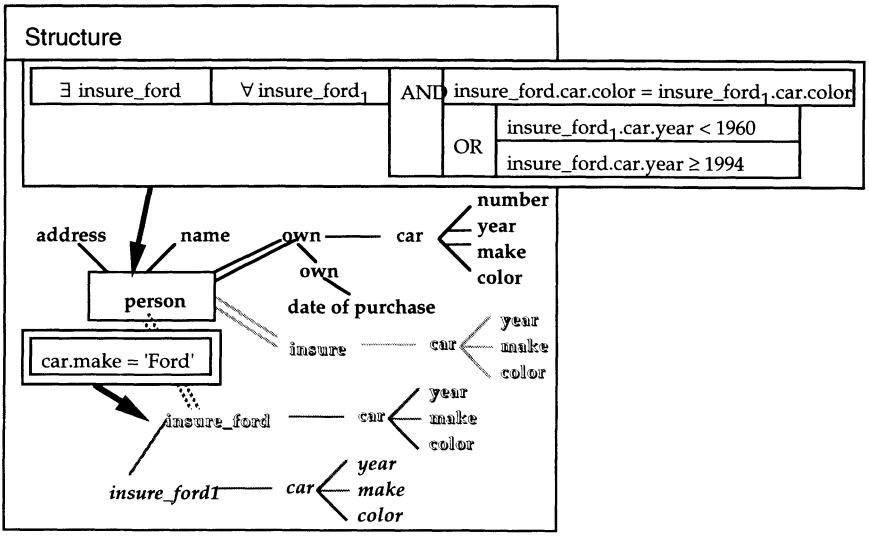


Figure 7: state of structure window corresponding to our example.

Error recovery with SUPER

We now check the way to recover the 18 errors describe above.

1 -Frame definition errors

-11 The user has forgotten owned cars in the frame. (S)he has to copy the relationship own into the working schema, where it will appear with both related object types, person and car. Then (s)he drags one of the two person types over the other one, to unify them into a single copy. The query is corrected without any loss of previous work.

-12 The compagny is still in the frame. The user only has to select in the WS the non relevant objects and delete them, using either the delete key on the keyboard or the delete command in a menu.

-13 The user selected the 'insure' relationship instead of the 'own' relationship, (s)he has to combine the two previous corrections: remove irrelevant objects, copy the relationship own and unify the two copies of the object type person.

-14 The query selects cars instead of persons, The user just has to double click on the object type person to make in become the root object type.

2 - Structure errors

-21 The result of the query does contain the address of persons. This address has not been physically removed from the structure. It appears as shaded items and a simple click toggles its state from hidden to shown (black).

-22 In the same way if the attributes 'insure' and 'compagny' are still in the result, a simple click on these desired attributes will hide them.

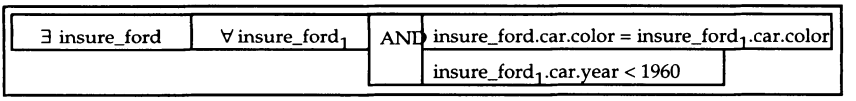
-23 If the attribute car has been erroneously renamed to 'repairs', the user only have to select the wrong name and type the good one, 'cars'.

-24 The predicate is linked by an arrow to the attribute insure instead of person, the user selects the arrow, drags it from insure to person and the predicate is automatically modified to keep a consistent quantification.

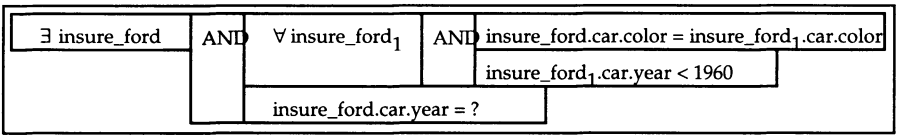
3 - Inner predicate errors

-31, 32, 33, 34 A bad quantifier is simply corected, quantifiers are graphic objects that react in the same way as radio buttons: double click on them switches their current value to the other value. To correct a wrong order in quantification , the user only has to drag them where (s)he wants to put them. The tool checks the correctness of the operation and, if it is possible, it makes the change. if a comparison operator is wrong, a double-click on a pops up a menu showing all operators of the domain: the user only has to choose the good one. Constants in predicates are editable text boxes.

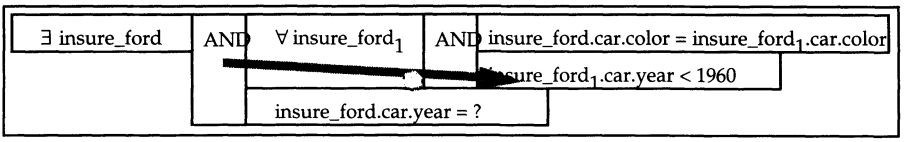
-35 The user forgot one condition concerning the year of the first quantified insured car, (s)he gets the following predicate box:



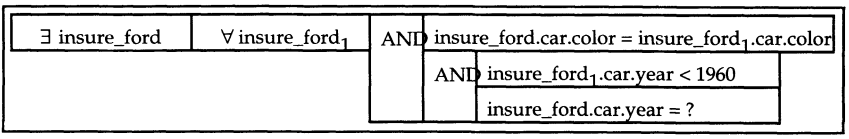
To add a new variable to a predicate the user drags it to the predicate box: the variable will appear in the predicate box, after its quantification. Here, the user drags the variable year (insure_ford.car.year) in the predicate and (s)he gets:



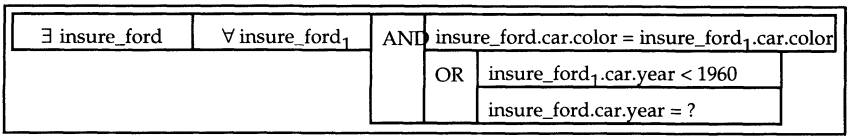
The structure of the predicate is as the user wanted: (s)he has to drag the AND box down in the tree (the drag action is represented by the gray arrow above)



(S)he obtains the following result:



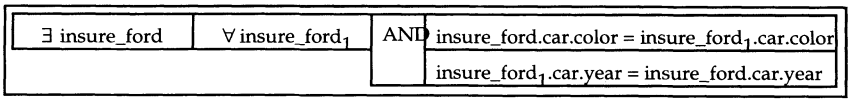
The user now has to double-click on the AND statement to make it toggle to an OR statement, which finally produces the good structure:



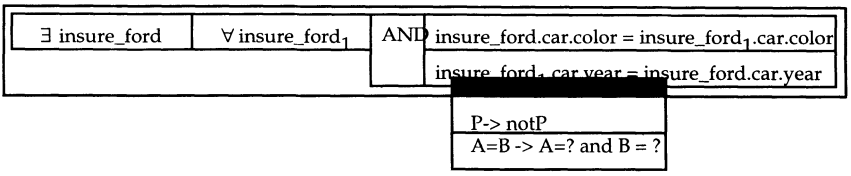
To finally get the complete predicate, the user has to change the comparison operator and introduce the value of the year as shown in correction of errors 33 and 34. This example also shows how to correct a wrong combination of logical operators (error 39).

-36 If a extra condition has been accidentally introduced in the predicate, the user only has to select the box containing this condition and delete it. The conjunction (or disjonction) is automatically removed and the user gets the good predicate.

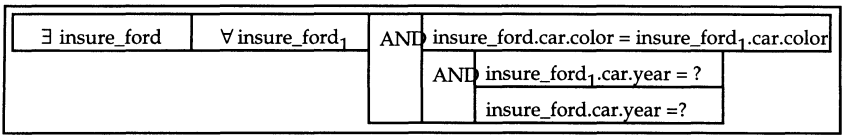
-37 The two conditions related to the year of production of cars have been accidentally transformed into one comparing the two variables to each other:



A double click on the wrong condition shows a menu, where the user can choose between all possible evolutions of the condition. In this case the proposed choices are the negation or a split of the condition:

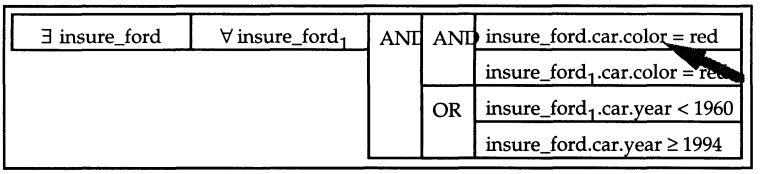


The user chooses the split and gets the following:



(S)he has to change the AND in OR as above, to type the values of the constants in the comparisons and to change with a pop-up menu the type of the operator.

-38 The user thought that all cars have to be red instead of beeing of the same color, (s)he wrote the following predicate:



The user has to drag one comparison to the other to get the good predicate.

Conclusion

SUPER allows correcting of all the errors we listed without any loss of previous relevant work. These corrections are made in the same way the user normally writes the query, mostly by direct manipulation. Only twice a pop-up menu and twice the delete key had to be used. Pop-up menus are quasi direct manipulation and the use of the delete key is quite natural. The only possibilities to avoid the use of that key is to introduce a waste basket, which seems quite inappropriate, or to match the delete action with the action of dragging the object outside the window, which is difficult to do with scrollable windows, which are needed for large schemata.

Possible extensions

Our manipulation paradigm can be used with any other model and with other languages. We particularly have in mind the possibility of defining such a language for an object oriented model. The expressive power can also be improved by the definition of new operators, like transitive closure. This operator is not in our language for two reasons, the first is naturally safety and the second is more philosophical. As we have seen before, the structure of a query is isomorphic to its frame, there is no loss of information. The use of an operator like the transitive closure leads to information loss. It maintains departure and arrival but looses the path.

6. CONCLUSION

The importance of visual interfaces is nowadays well recognized. Despite many commercial tools and research prototypes, significant advances are still needed to extend the visual paradigm from data definition to data manipulation and from relations to objects. As far as we

know, browsers are the only tools that succeeded in combining both extensions. However, they only support navigational access.

We have developed a visual language for assertional object manipulation (Dennebouy, 1993). It has been implemented as part of a conceptual front end (the SUPER prototype) to existing DBMSs. A description of the main features of our language can be found in (Dennebouy, 1995). Besides the basic functionalities, one of our aims has been to allow for easy correction of the query being formulated. The experience with SQL shows that query formulation on objects will probably be a non trivial task for users for whom visual interfaces are intended. If we compare the process of query design and the process of program writing, we know that a programmer who forgets to define a variable or a loop in his program in a high level language will not accept to rewrite the program due to this error. Neither (s)he will accept to correct the program by patching the object code at machine level language. We believe that a query designer needs to debug a query the same way (s)he wrote it.

While some evaluation criteria (expressiveness, ...) for visual languages have been discussed elsewhere (Batini, 1991), this paper focused on flexibility of these languages, mainly intended as the provision for easy re-formulation of an incorrect query. As a metric for flexibility we have proposed a list of typical errors during query formulation. We have analyzed how some visual manipulation languages, including ours, behave concerning correction of these errors. We hope that this work will help visual languages designers to take into account this main feature, flexibility.

REFERENCES

Angelaccio, M.,Catarci, T., and Santucci, G. (1990) "Query by Diagram*: A Full Visual Query System", *Journal of Visual Languages and Computing* (1) pp. 255-273.

Auddino, A., Dennebouy, Y., Dupont, Y., Fontana, E., Spaccapietra, S., and Tari, Z. (1992) Super-Visual Interaction with an Object-Based ER Model, *Lecture Notes In Computer Science* 645, pp. 340-356

Batini, C., Catarci, T., Costabile, M.F., and Levialdi S. (1991) Visual Query Systems, RAP 04.91 , Universita di Roma "La Sapienza"

Dennebouy, Y. (1993) Un langage visuel pour la manipulation de données *Ph.D. Thesis #1182*, Swiss Federal Institute of Technology .

Dennebouy, Y., Andersson, M., Auddino, A., Dupont, Y., Fontana, E., Gentile, M., and Spaccapietra S. (1995) SUPER: Visual Interfaces for Object+Relationship Data Model", *Journal Of Visual Languages and Computing* , Mars 95

Kuntz, M. and Melchert R. (1988) Pasta-3's graphical query language: Direct manipulation, cooperatives queries, full expressive power *Proceedings of the. 15th International Conference on Very Large Data Bases* pp. 97-105

Parent,C. and Spaccapietra, S. (1992) ERC+: An Object-based Entity-Relationship Approach, *Conceptual Modelling, databases and CASE: An integrated View of Information Systems Development*

Sockut, G.H., Burns, L.M., Malhotra, A, and Whang, K.Y. (1993) GRAQULA: A graphical query language for entity-relationship or relational databases, *Data & Knowledge Engineering* 11 pp. 171-202

BIOGRAPHY

Yves Dennebouy received his DEA in computer engeneering from the university of Rennes, France in 1985. After two years spent in a software compagny in Paris he joined the database lab of the Swiss Federal Institute of Technology (Lausanne) where he received his PhD in 1993. He is now first assistant and leads the interface research activity of the laboratory.