A Visual Interface and Navigator for the P/FDM Object Database

Ignacio Gil, Peter M.D. Gray and Graham J.L. Kemp

Department of Computing Science, University of Aberdeen,
King's College, Aberdeen, Scotland, UK, AB24 3UE
E-mail: {pgray|gjlk}@csd.abdn.ac.uk

Abstract

We have implemented a Java-based visual interface for P/FDM which has at its centre a graphical representation of the database schema. Users construct queries by clicking on entity classes and relationships in the schema diagram and constraining the values of attributes selected from menus. As this is done, the Daplex text of the query under construction is built up in a subwindow (the query editor window).

Queries are submitted to the database via a CORBA interface. Results satisfying the selection criteria are displayed in a table in a separate result window, together with the Daplex text of the query.

A particularly novel feature of the interface is a "copy-and-drop" facility which enables the user to select and copy data values in the result window and then "drop" these into the query editor window. When this is done, the selected values are merged into the original query automatically, in the appropriate place in the query text, to produce a more specialised query. This query can then be extended by the user, if required, and submitted to the database for execution. Thus, complex queries can be built up in stages by inspecting intermediate results and modifying the follow-on queries to have stronger selection criteria and additional navigation links.

1 Introduction

P/FDM [8] is a database management system which implements the functional data model (FDM) [18]. The basic concepts in the FDM are entities, which are used to represent real world objects, and functions, which are used to represent entity properties including scalar attributes and relationships between entities. The P/FDM database can be interrogated using the programming language Prolog or the query language Daplex. This paper is about the design of a graphical interface which can be used to formulate queries without the user having to learn to program in either Prolog or Daplex.

Queries are submitted to the P/FDM database via a coarse grain CORBA interface [12] . This interface handles queries expressed in a high level language, thus giving efficient bulk execution. Results satisfying the selection criteria are displayed in a table in a separate response window, together with the Daplex text of the query which gave the results. This is valuable to scientists as it reveals precisely what selection conditions and what relationships were used to derive the results. Otherwise, with exploratory browsing, it is all too easy to use quite complex conditions but be unsure what they were, which could upset one's conclusions.

In this paper we discuss previous work that has shaped the design of the current interface and relate its functionality to that of other systems. We then describe each of the interface's main features: the query window, the expression editor, the response window and the copy-and-drop facility. These are illustrated with screenshots showing the interface in use with our antibody database [11]. Finally, we summarise the novel features of the interface and discuss its use as a front-end to other remote database systems.

2 Background and related work

Visual query systems for databases are surveyed in [3]. However, query systems for object databases using OQL are relatively new [5, 13] and still developing.

In a previous project AMAZE [2] for our P/FDM object database we tried using a 3-D schema representation, with a graphic annotation for query restrictions represented as small boxes dangling off entity types in the schema [1]. This also explored 3-D representations of results in a novel way. While the result presentations were promising, and are being followed up, the form of query input felt over-complex, and ran

into the classic problems of positioning text labels on a rotating 3-D image.

Therefore we decided to revert to a 2-D schema representation for input in a kind of ER style with subtypes, as a variant of that used by many other groups. We also decided not to devise elaborate graphic representations for the query operations specifying quantified selection conditions, disjunctions or computed results. Instead, we decided to make use of the underlying query language Daplex, and express these in a textual representation with layout. In the previous design we had basically hidden the generated query language from the user, only displaying it in a window for a quick "sanity check" before sending it off to the database server. We now feel that this was a mistake, since the user then only saw the query during formation as a series of annotation boxes of various kinds, each of which had to be unpacked in order to see the query as a whole. By contrast, the query text has a concise uniform presentation.

We further concluded that really complex queries have to be formulated in Daplex anyway, since no graphic representation can easily cope with a complex combination of conjunction, disjunctions and quantifiers across a variety of entity types. However, there is a different problem in Daplex, since the end user usually has only a modest reading ability for simple Daplex queries, but no writing ability.

We decided to make this into a virtue by making the system function as a scribe to compose parts of the query, in response to mouse actions by the user highlighting parts of the schema and choosing options from generated menus. In this way the user can build up a query incrementally, knowing at each stage that the Daplex query on the screen is valid, and can be sent to the database server, if desired.

This mode of working has the advantage that it models the way users often work in Information Retrieval searches, by formulating a simple query, discovering there are too many answers, and then refining it. It also bears out the historical experience of the developers of SQL, who initially thought that all queries that users actually wished to ask could be expressed in a few lines with SELECT...FROM...WHERE, and then were surprised when users wished to modify their queries in ever more complex ways, which broke the original simplicity of the language.

An early Smalltalk prototype fulfilling these design aims was presented by Cole and Gray [6]. We have now completely re-implemented that early prototype in Java, with a number of detailed improvements. While the incremental generation of object queries is no longer novel, a significant contribution is the integration of a highlighting point and click editor with the displayed query text, so that a user can click on parts of a query and have relevant entities and relationships in the schema highlighted, as a prelude to menu selection. This helps the user to understand the text version of the query, by relating it to the diagram; it also leads on naturally to query refinement by menu selection. This has been further enhanced by the copy-and-drop facility which is described in Section 6.

We have thus adopted a very different approach from Papantonakis and King [17], who have developed a graphic query interface to a functional data model database, but who use a mix of elaborate graphic constructs to generate a query in their functional query language (FDL). Their problem is that the generated FDL is hard to read, for anyone but a mathematician, and it has an algebraic rather than a clausal syntax. We are able, by contrast, to capitalise on various features of Daplex. Firstly, it gives a natural sequence or *spine* to the query built up from nested loops. Secondly there are natural points in the query, which we can colour and highlight for selection, where the query can be extended by adding extra *such that* clauses, or conjoining or disjoining selections, or adding quantifiers.

We believe that by providing these visual cues we make it easy for the user to learn the language, and improve their reading ability. It also provides visual cues for legal modifications. Thus we believe we have "closed the control loop" in query formulation, by providing two representations, each of which is mouse active, with pointers into the other.

Daplex fits particularly well with extended entity-relationship models and semantic data models, since it was designed for them. There is a direct correspondence between entity names in the query and boxes representing entity types (or subtypes) in the schema.

The same goes for function names and relationships. Thus, although we can in fact generate SQL for tables with in-built object identifiers [10], we believe that if we were to try and teach the user to build up a complex SQL query by this technique, it would be much harder, particularly since SQL does not have the clean syntactic structure of Daplex. However, we can translate the Daplex queries we generate into the ODMG language OQL [4], so that the interface has more general applicability as a front end to other databases.

A closely related project is the Kaleidoquery visual interface [16] which has been developed to gen-

erate OQL. That interface is based around a 3D representation of the entity classes in the database. The classes are shown as icons but are not connected as in an Entity-Relationship diagram. Instead the relationships are shown on demand, as part of a list of attributes of a selected class. Also the user does not see the source version of the query developing. It is not clear how they could represent *self-joins* where two different variables independently range over the same entity class, which is a very useful feature of our system.

The Kaleidoquery interface uses a *pipeline metaphor* to show conjunctions and disjunctions. Disjunctive selections are shown by parallel sections of pipe, while conjunctive selections are shown as successive sections. It is well known that users find considerable difficulty in using these boolean connectives [15, 9], and this is an important practical issue. We explain our approach below, and we recognise the need to do systematic comparisons to establish the cases where the Kaleidoscope approach would be preferred to ours, and where not. However, as often happens, it would be difficult to isolate the handling of boolean connectives from other features of the interface when evaluating users' experiences.

Earlier work [7] allowed a user to choose between three different visual interfaces to the same object database: graph-based; forms-based and a textual interface (similar to Daplex). However, these were used singly and not in combination. It was possible to switch to a different interface partway through query formulation, and thus to see the Daplex-like text for a query that had been developed graphically. However, one was then faced with using a different editor if one continued in that mode. The results of this work showed that different users preferred different editors, depending on their experience and the complexity of the query. In fact, we believe that a web forms-based editor is best for simple queries, and we have developed such an editor, while using the Visual Navigator for more ambitious exploratory queries.

3 The query window

The basic concepts in the functional data model (FDM) are entities, which are used to represent real world objects, and functions, which are used to represent entity properties including scalar attributes and relationships between entities. The database schema is presented graphically as an entity-relationship (ER) diagram (Figure 1).

Entity classes are shown as ellipses in the main panel of the query window. The Daplex text of the query being constructed is shown in a panel below the schema diagram. The user starts constructing a query by clicking on an entity class (e.g. ig_domain). The border of the ellipse is highlighted and the user can either press the right mouse button or the `accept` button below the schema window to confirm the selection. Confirming creates a *for each* loop in the Daplex query to iterate over instances of that class, and this line is added to the Daplex query in the lower panel.

Arcs between entity classes represent relationships. "Crow's feet" at the end of a relationship arc indicate a multi-valued relationship. A clickable circle is drawn on each relationship arc. When an entity class is selected, its ellipse is filled with colour, the border of the relationship circle and the relationship name of all relationships involving that entity class are highlighted, and again either the right mouse button or the `accept` button is used to confirm the selection.

The user selects the attributes to be printed by double-clicking on entity class ellipses in the main schema diagram. For example, double-clicking on the ig_domain ellipse will bring up a listbox with the names of the attributes defined on that class. Attributes selected from this list are added to the `print` line of the query. If the query involves two or more instances of the same class ("self join"), a menu listing the applicable instance variables is displayed when the user first double clicks. This enables the user to specify which instance's attribute values should be printed.

4 The expression editor

To place a condition on an instance variable, the user must first press the `add Modifier` button in the main query window (see Figure 1). This causes a small box icon to be drawn at the appropriate point in the Daplex code in the lower panel of the query window (Figure 1 actually shows box icons which are replaceable by integer expressions, because of the context). To instantiate the test, the user clicks on the box in the query to bring up the expression editor (Figure 2). This window allows the user to create and edit boolean expressions.

A function can be constrained to have a constant value by clicking on `Select Function` to bring up the Function Chooser window, selecting the function to be constrained and then typing a value into the entry box. A selection box is used to specify the comparison operator to be used.

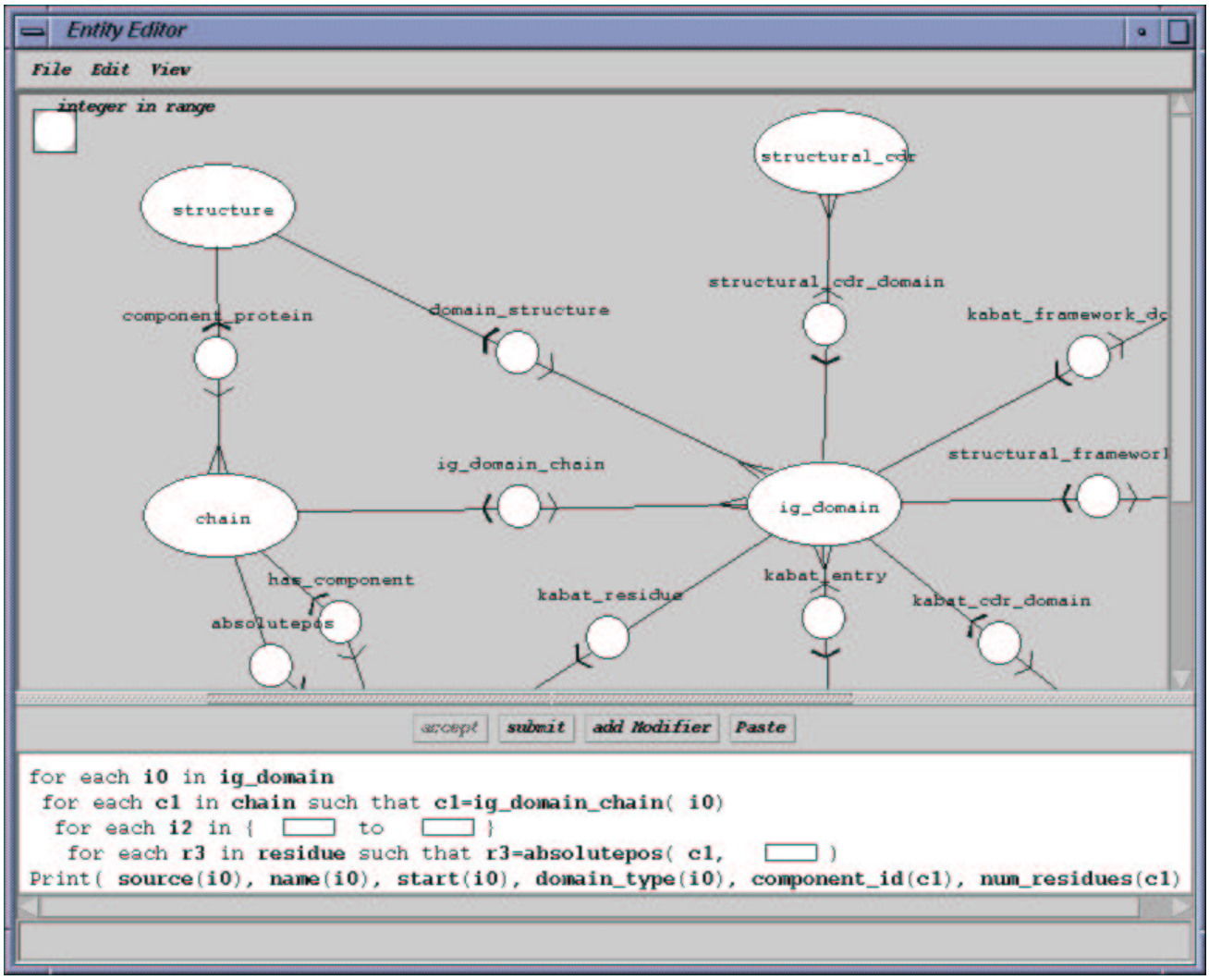

Figure 1: The query window

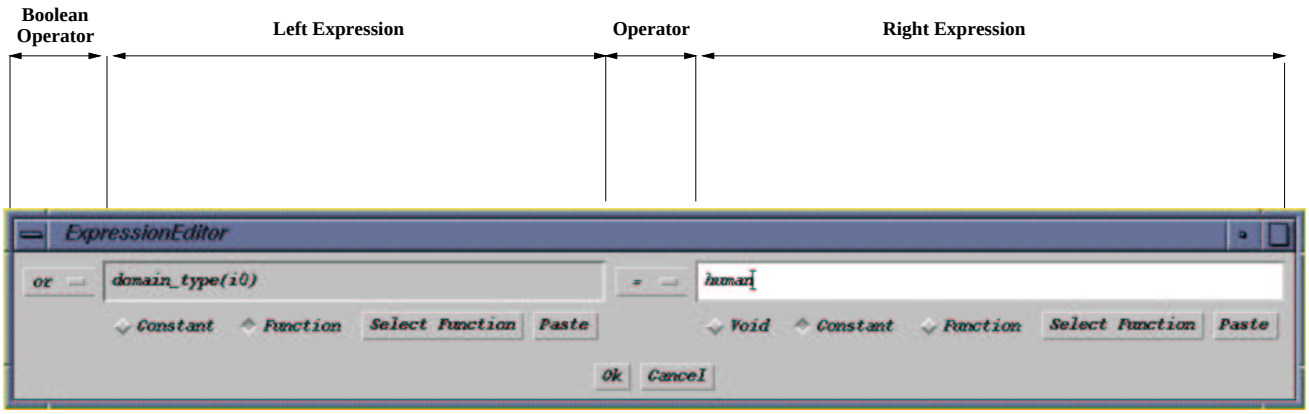

Figure 2: The expression editor

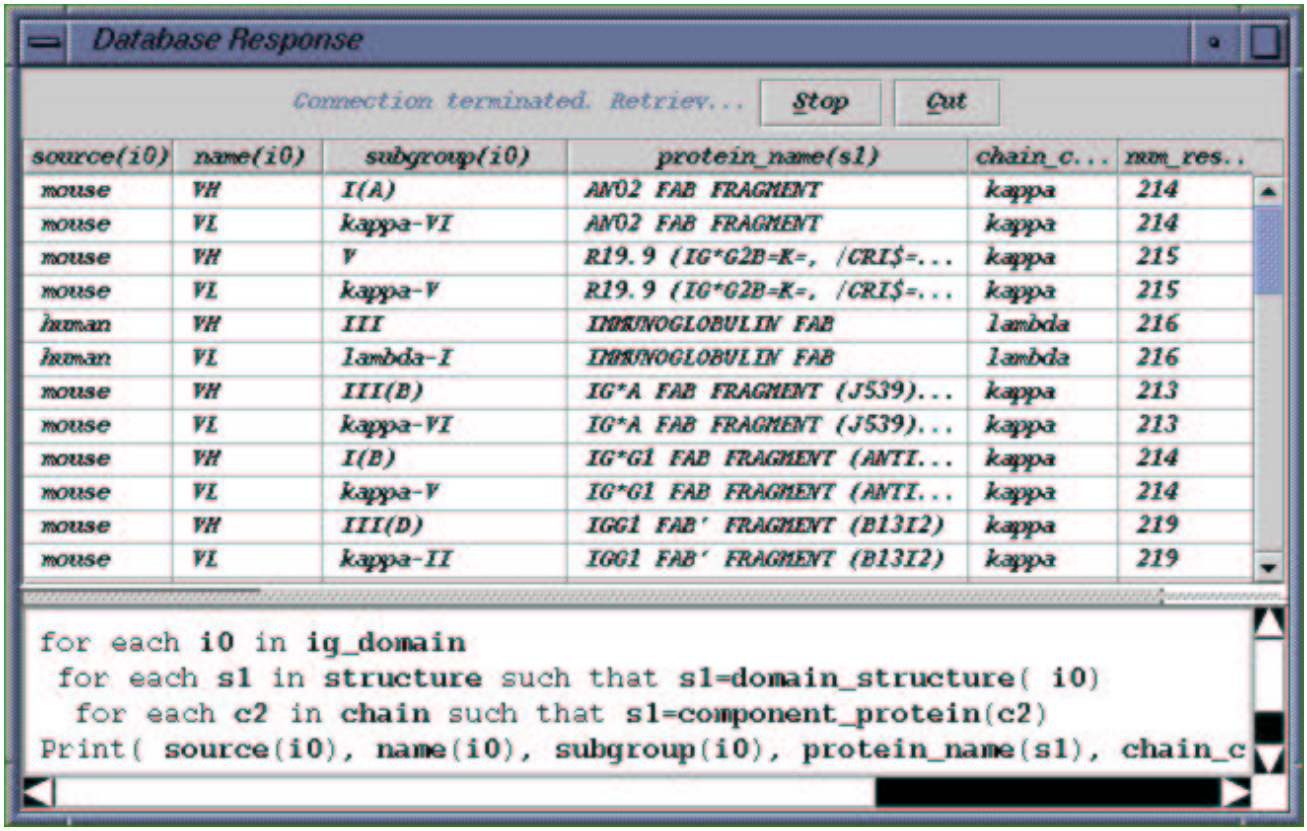

Figure 3: Response window

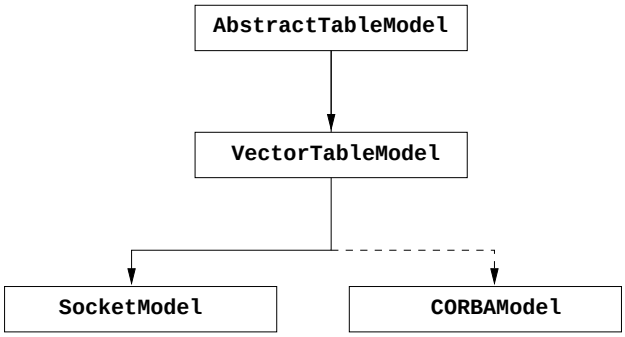

Figure 4: Vector table model inheritance

Additional selection criteria can be added to the query. A selection box at the left of the expression editor specifies whether a conjunction or disjuncion will be made. When the `OK` button is pressed, the query in the main window is updated with the test replacing the box icon in the query text.

5 The response window

Query results are tabulated in a separate window (Figure 3). The upper part of this window contains the query results. The heading for each column shows the attribute name and the variable used in the originating query to refer to an object identifier. The text of the originating query is shown in a panel at the bottom of the results window. This is useful since it enables the user to check the query that generated that response, and because the response windows for several queries may be present on the screen at the same time.

The results are presented in a table which is implemented using the swing class `JTable`. This provides a table with cells that can be selected, resized and moved. The Java library swing uses the full MVC architecture (as in Smalltalk). The JTable class acts as view-controller of an instance of the class TableModel. In order to override the behaviour of the default table, the TableModel class has to be inherited and some methods redefined. The class `VectorTableModel` is an abstract class, the method `submit` has to be redefined to access the database and get the result. Figure 4 shows the object hierarchy of the model. New lines are added to the model with the method `addRow` which has as parameter a vector of strings (one entry per column). It sends the MVC event to the JTable view.

An early version of the Virual Navigator used a socket connection to access the database, but this has now been replaced by a CORBA connection.

Communication with the server is done in the background, and the user can browse the data once the first row has been retrieved. This is convenient for large queries since the user can begin inspecting the results before the query has run to completion.

6 Copy-and-drop facility

An important aim in designing the interface was to enable the user to use the results on one query in refining a follow-on query. The copy-and-drop facility enables the user to select and copy values from the results window and then drop these into the query editor window. When this is done, the selected values are merged automatically into the right place in the original query to produce a more specialised query.

6.1 Operational description

Cells are selected by clicking on them with the left mouse button. When a **single cell** is pasted, the program simply adds a modifier to the query line that has the function of the selected value. Figure 5 shows original query, the selection of a single cell containing *human*, and the modified query after pasting.

When the user selects **multiple cells**, the selection must form a rectangular pattern. If a cell is selected and the user drags the mouse, then a block of adjacent cells will be highlighted. Non-adjacent cells can be selected by holding down the control key while selecting cells with the mouse. The program will select automatically any additional cells required to form a rectangular pattern (Figure 6). The satisfying result is that cells from the same row are connected with an AND operator, and different rows are connected with OR operators. Figure 7 shows the result of pasting values from two columns into a query.

If attribute values for several different object instances are selected from the response window, then the corresponding tests are placed in the innermost relevant loop of the query. While an experienced Daplex programmer might normally place each closer to the *for each* loop introducing its instance variable, placing these in the innermost loop works just as well and the database's query optimiser will move the tests automatically when the query is submitted, if this will improve performance.

If the existing query contains a test, e.g. `s1=domain_structure(i0)`, then the pasting the selection shown in Figure 6 produces a query that is interpreted as follows:

```
for each i0 in ig_domain
Print( subgroup(i0), source(i0), end(i0) );
```

subgrou...	source(i0)	end(i0)
kappa-VI	mouse	109
I(A)	mouse	115
kappa-V	mouse	108
V	mouse	125
lambda-I	human	113
III	human	126
kappa-VI	mouse	107

```
for each i0 in ig_domain such that source(i0)="human"
Print(  );
```

Figure 5: Copy-and-drop a single cell

```
for each i0 in ig_domain
Print( subgroup(i0), source(i0), end(i0) );
```

subgrou...	source(i0)	end(i0)
kappa-VI	mouse	109
I(A)	mouse	115
kappa-V	mouse	108
V	mouse	125
lambda-I	human	113
III	human	126
kappa-VI	mouse	107
III(B)	mouse	118
kappa-V	mouse	108

```
for each i0 in ig_domain such that source(i0)="human" and end(i0)=113
                                or source(i0)="mouse" and end(i0)=118
 Print(  );
```

Figure 7: Copy-and-drop multiple cells

name(i0)	subgroup(i0)	resolution(s1)
VL	kappa-V	2.8
VH	III	1.9
VL	lambda-I	1.9
VH	III(B)	2.6
VL	kappa-VI	2.6

Figure 6: Selecting non-adjacent cells. If the user selects the cell containing `VH` and the cell containing `2.6` then the cells containing `VL` and `1.9` are selected automatically to complete a rectangular pattern.

```
for each i0 in ig_domain
  for each s1 in structure such that
    s1=domain_structure(i0) and
      (name(i0)="VH" AND resolution(s1)=1.9
      or
       name(i0)="VL" and resolution(s1)=2.6)
```

While pasting result values from the response window into the main query window is the principal use for the copy-and-drop facility, it is also possible to paste a single value into the expression editor. Thus there is a button named `paste` for each expression entry box (left part and right part of the expression, Figure 2). When the user clicks on the `paste` button, the value of the selection is copied to the entry box in the expression editor.

6.2 Formal definition

In general, the generation of extra selection clauses works as follows:

- If the selected items $value_1, value_2, ...$ are all in one column referring to attribute a of the entity instance e, then, on the line in the query iterating over that entity instance we add a selection:
  and $(a(e) = value_1$ or $a(e) = value_2$ or ...).
- If the selected items are all in one row, referring to attributes $a1(e1)$, $a2(e1)$, $a3(e2)$,... then, on the most deeply nested line in the query iterating over one of the listed entities $e1, e2, ...$, we add a selection:
  and $(a1(e1) = value1$ and $a2(e1) = value2$ and $a3(e2) = value3$ and etc.)
- Otherwise the selected items may be in repeated rows each referring to the same combination of columns $a1(e1)$, $a2(e2)$, $a3(e3)$, etc.. Here we add a selection, again on the most deeply nested entity instance:
  and $((a1(e1) = value1_1$ and $a2(e2) = value2_1$ and $a3(e3) = value3_1$ and etc.)
  or $(a1(e1) = value1_2$ and $a2(e2) = value2_2$ and $a3(e3) = value3_2$ and etc.)
- Where the selection is the first on a given line, the leading *and* is replaced by *such that*.
- Note that two attributes may refer to the same instance of an entity type (e.g. $a1(e1), a2(e1)$). Also, two different entity variables $(e1, e3)$ may refer to different instances of the same entity type, as in a self-join.

Once the selections are in place they may be individually highlighted and then edited by the expression editor, or else deleted. One may of course repeat the exercise after generating a new results window, and thus add extra selections. However, one must beware of over-complicating the query to a degree which the user would find difficult to understand! Thus we only generate a limited combination of conjunctions and disjunctions, but we think these are the ones that are most useful. If an experienced user wishes a more complicated disjunction than is provided by the copy-and-drop facility, then they could formulate this directly by using the expression editor.

7 Conclusions

The graphical interface described in this paper enables the user to construct queries against a functional data model database. Complex queries are built incrementally; the user does not have to foresee the final query before starting to construct it. The user can refine the query using intermediate values from the results window. The Daplex query is generated automatically, so the user does not have to remember the query language syntax. The user can also undo choices selectively, by deleting parts of the query. A full and exact description of the query is displayed with the results.

The Daplex language was developed in the 1970's for use with the Functional Data Model in the MULTIBASE project [14], and is being used in our current work with the P/FDM database. The Object Database Management Group (ODMG) are promoting OQL as a standard query language for object-oriented databases. We have implemented an OQL code generator so that the P/FDM mediator can produce OQL code to be run against remote ODMG-compliant databases. We have found it straight-

forward to map an existing schema for a database on protein-protein interactions onto the FDM, and have then used the graphical interface described in this paper to generate queries against that database.

This interface is a complete re-implementation in Java 1.1 (with Swing classes) of an earlier Smalltalk prototype [6]. In the process a lot of detailed improvements were made, based on the experience of using the earlier prototype with a class of students, and on Ignacio Gil's own experience with it. In doing so, we kept the basic concept of using a schema diagram and the textual form of the query in combination. The query is always syntactically well formed, and can only be edited and revised in conjunction with the schema diagram. Well formed sub-expressions involving comparisons can be edited using an expression editor with menus. Whole expressions can be added or deleted. Highlighting variables in the query causes corresponding entities in the schema diagram to be highlighted, together with paths to immediately related entities in order to help with the choice of navigation path.

The important step forward has been to integrate the results window with the query and schema. This supports a natural process of alternately making the queries more selective, based on displayed results, and then extending the query to pick up related information by navigation. In the course of doing this we have found a very easily remembered way of expressing conjunctions and disjunctions, which is a novel alternative to "pipe-flow". The resultant query text then serves to document the results. Thus we believe that scientist end users can read the queries (including "and" and "or" operations) as text but need help with writing, which the interactive system provides. Seeing the queries built up in stages with displayed results strongly promotes understanding of the query language.

We are working on further improvements before making an end-user evaluation, but informal feedback from users has been encouraging. We also need to tackle the problem of quantified sub-queries (e.g. "some x such that ...") and computation of aggregates over sets specified by sub-queries. The challenge is how to do this without presenting a naive user with too many complex options before they need this functionality.

We are currently using this interface to provide access to data on antibody structures and sequences, and we intend using it to express multi-database queries against distributed biological databases through our mediator, which is being developed with support from the BBSRC/EPSRC Bioinformatics Programme.

Acknowledgements

This work is supported in part by a grant from the BBSRC/EPSRC Joint Programme in Bioinformatics (Grant Ref. 1/BIF06716). It builds on an earlier Smalltalk Browser by Nicholas Cole [6] and an extension to this by Christian Bruder.

References

[1] J. Boyle, J. Fothergill, and P. Gray. Design of a 3D Interface to a Database. In J. Lee and G. Grinstein, editors, *Database Issues for Data Visualization*, pages 173–183. Springer Verlag, 1993.

[2] J. Boyle, S. Leishman, and P. M. D. Gray. From WIMP to 3D: the development of AMAZE. *Journal of Visual Languages and Computing*, 7:291–319, 1996.

[3] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual Query Systems for Databases: A Survey. *Journal of Visual Languages and Computing*, 8:215–260, 1997.

[4] R.G.G. Cattell, editor. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, 1997.

[5] M. Chavda and P. Wood. Towards an ODMG-Compliant Visual Object Query Language. In M. Jarke, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *Proceedings of the 23rd International Conferences on Very Large databases*, pages 456–465, 1997.

[6] N. Cole and P. M. D. Gray. Smalltalk Graphics as a Visual Aid to acquiring Query Language Skills over a Semantic Data Model. In *Proceedings 3rd International Workshop on Interfaces to Databases*, 1996.

[7] D. K. Doan, N. W. Paton, A. C. Kilgour, and G. al Qaimari. Multi-paradigm query interface to an object-oriented database. *Interacting with Computers*, 7:25–47, 1995.

[8] P. M. D. Gray, K. G. Kulkarni, and N. W. Paton. *Object-Oriented Databases: a Semantic Data Model Approach.* Prentice Hall Series in Computer Science. Prentice Hall International Ltd., 1992.

[9] S. Greene, S. Devlin, P. Cannata, and L. Gomez. Mo IFs, ANDs, or ORs: A Study of Database Querying. *International Journal of Man-Machine Studies*, 33:303–326, 1990.

[10] G. J. L. Kemp, J. J. Iriarte, and P. M. D. Gray. Efficient Access to FDM Objects Stored in a Relational Database. In D.S. Bowers, editor, *Directions in Databases: Proceedings of the Twelfth British National Conference on Databases (BNCOD 12)*, pages 170–186. Springer-Verlag, 1994.

[11] G. J. L. Kemp, Z. Jiao, P.M.D. Gray, and J.E. Fothergill. Combining Computation with Database Access in Biomolecular Computing. In W. Litwin and T. Risch, editors, *Applications of Databases: Proceedings of the First International Conference*, pages 317–335. Springer-Verlag, 1994.

[12] G. J. L. Kemp, C. J. Robertson, and P. M. D. Gray. Efficient access to biological databases using CORBA. *CCP11 Newsletter*, 3(1), 1999. http://www.hgmp.mrc.ac.uk/CCP11/newsletter/vol3_1/.

[13] E. Keramopoulos, P. Pouyioutas, and C. Sadler. GOQL, a Graphical Query Language for Object-Oriented Database Systems. In *Basque International Workshop on Information Technology*, pages 35–45, 1997.

[14] T. Landers and R. L. Rosenberg. An Overview of MULTIBASE. In H.-J. Schneider, editor, *Distributed Data Bases*. North-Holland Publishing Company, 1982.

[15] A. Michard. Graphical Presentation of Boolean Expressions in a Database Query Language: Design Notes and an Ergonomic Evaluation. *Behaviour and Information Technology*, 1:279–288, 1982.

[16] N. Murray, N. Paton, and C. Goble. Kaleidoquery: A Visual Query Language for Object Databases. In *Proceedings of Advanced Visual Interfaces*, pages 25–27, L'Aquila, Italy, May 1998.

[17] A. Papantonakis and P. J. H. King. Syntax and Semantics of Gql, a Graphical Query Language. *Journal of Visual Languages and Computing*, 6:3–25, 1995.

[18] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.