in: Proc. Int. Workshop on Interfaces to Database Systems, Glasgow, Scotland, 1992

A Graphical Interface to a Complex-Object Database Management System

Harald Schöning
University of Kaiserslautern
Kaiserslautern, Federal Republic of Germany

Abstract

The research in database systems for new application areas has created several new architectural designs for database systems, among them the database kernel architecture. We present a graphical interface which is located on top of such a database kernel system. It provides an interface to the molecule-atom data model, which is a complex-object data model that serves as the basis for the implementation of various applications from new areas such as VLSI design. Our interface is able to display the database schema graphically and allows the graphical formulation of queries. Furthermore, it displays the results of queries as sets of complex objects. Each of these objects corresponds to a directed graph. The level of detail in the presentation of the results may be changed interactively. Initially, for each element of the set one representative is shown. The graph (i.e. the complex-object structure) corresponding to one or several elements of the result can be visualised. For each node in a graph, its content (i.e., the attribute values) can be displayed. Furthermore, a type-oriented view to the components of the result set's complex objects is provided.

1 Introduction

Database management systems (DBMS) have become indispensable tools in many commercial applications. For new application areas such as VLSI design and CAD, there are a lot of additional requirements DBMS must fulfil before they can be expected to play a similar role. For this reason, a DBMS designed to serve these new areas is called nonstandard DBMS (NDBS). An analysis of the different application areas shows that each of them needs an NDBS with a specific functionality. However, some basic features are common to all nonstandard areas. Among these is the support of complex objects, which means that an object of the application which has a complex internal structure is to be handled as a whole by the NDBS. In conventional DBMS (using conventional data models, e.g. the relational model) these objects have to be decomposed into many parts, and the DBMS does not “know” about this decomposition. Hence, the objects have to be reassembled by rather sophisticated join operations. Furthermore, the semantics of the decomposition is not represented in the database and has to be coded in the join operations.

The observation that there are common features which have to be provided by all NDBS leads to the DBMS kernel architecture, where a general-purpose NDBS kernel provides all these features, among them a complex-object data model. So-called application layers serve the specific needs of an application, thereby exploiting the functionality of the NDBS kernel.

Such an NDBS kernel (called PRIMA [HMMS87]) has been developed at the University of Kaiserslautern. It provides the molecule-atom data model (MAD model [Mits88]) and its query language MQL as an interface, which allows for the dynamic definition of complex objects. In the MAD model, complex objects (called molecules) may be either disjoint or non-disjoint, and either recursive or non-recursive [BaBu84]. Several applications have been built on top of PRIMA, for instance tools for VLSI design [HüSu92] and a temporal DBMS [KäRS90].

During the development of application layers, and also in the later phases of an application program, there is a strong need for an interactive interface at the data model level, which allows for viewing the database at this level. To be universally applicable, the interface should be independent of a specific application. Having such an interface also opens new facilities for the use of PRIMA as a DBMS with ad-hoc query facility. However, such an interface is useful only if it is able to represent the complex-object properties in a natural way. Line-oriented output of database objects is not appropriate in our case (while it is not too bad for relational systems). Instead, we have to provide graphical representations of the database’s objects.

In the following, we will shortly sketch the MAD model, and will then present our graphical interface to PRIMA.

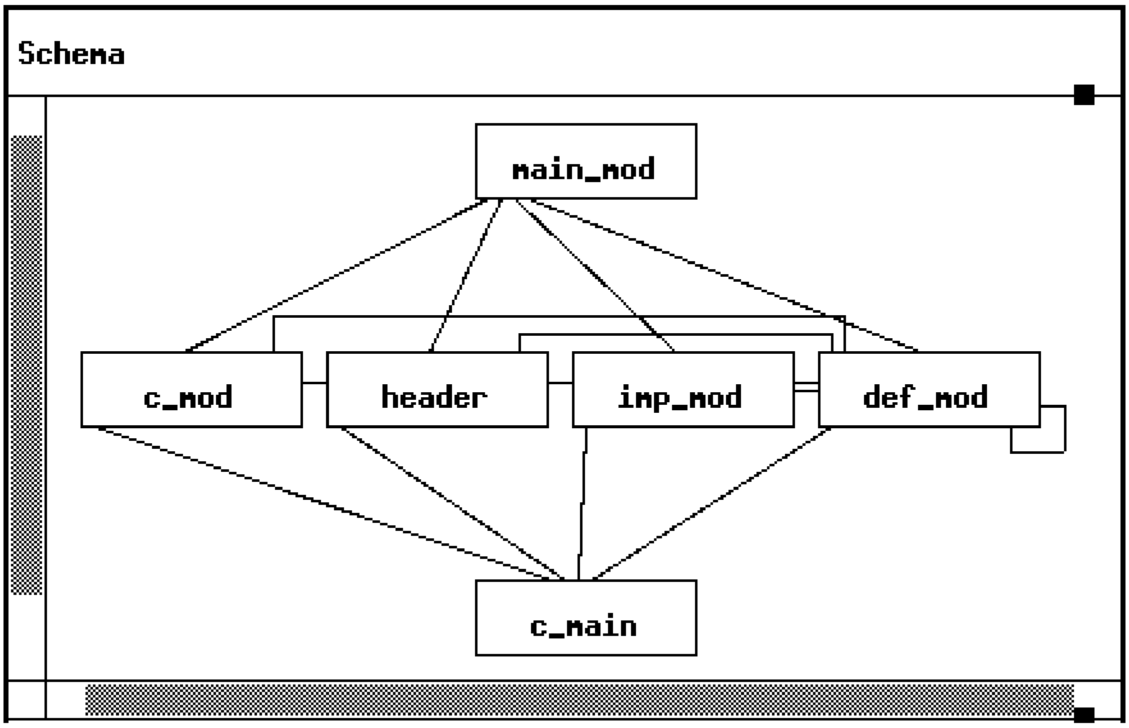
2 The Molecule-Atom Data Model

The basic building blocks of the MAD model are *atoms*, which may be compared to tuples in the relational model. Atoms belong to exactly one *atom type* and are uniquely identified by their *identifier*. An atom type has *attributes* of certain data types, and atoms consist of corresponding values.

Atom types may be connected to one another by link types. A link type represents a binary relationship type (1:1, 1:n or m:n). Analogously, atoms can be linked to one another by a link of such a link type. Thus, the schema of a database consists of a network of atom types, and the database consists of a network of atoms.

A link type is realised by a pair of attributes of the special type REFERENCE, one in each atom type involved. Attributes of this type have as value a set of identifiers of the atoms which are linked to the atom by this link type. Links are symmetric, i.e., if atom *a* is linked to atom *b* (of the same or another type), the atom *b* is necessarily linked to atom *a*, too.

Example 1 shows a sample database schema taken from a software engineering database. It models several types of source code files and their dependencies. There are MODULA-2 IMPLEMENTATION MODULEs (**imp_mod**). Each of them has its own



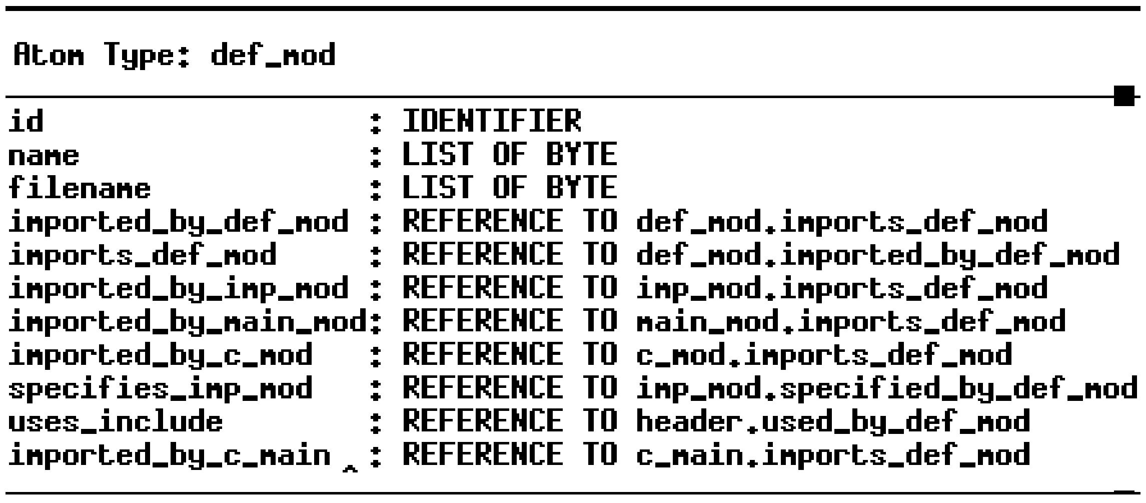
Example 1: A sample database schema

DEFINITION MODULE (**def_mod**). DEFINITION MODULEs may also be imported by other **def_mod** and **imp_mod**. Hence, there are two link types between **imp_mod** and **def_mod** (*has definition module* and *imports definition module*) and one connecting **def_mod** to itself (*imports definition module*). A MODULE (**main_mod**) forms the a main module of a program, i.e. there will be an executable program with the same name. A **main_mod** may import several **def_mod** (link type between **main_mod** and **def_mod**). When the program is linked, the compiled **imp_mod** are linked to form the program (link type between **main_mod** and **imp_mod**). We extend the MODULA-2 programming environment by allowing C modules (**c_mod**) to be added. They may also be linked to **main_mod**, and they may call MODULA-2 functions (link types between **c_mod** and **main_mod**, and **c_mod** and **def_mod**, respectively). Consequently, we allow C main programs (**c_main**), which may use MODULA-2 and C code (link types between **c_main** and **imp_mod**, **def_mod**, and **c_mod**, respectively). For all types of source code files, we allow the usage of header-Files (**header**) which may be included by a preprocessor prior to compilation.

The schema graph shown in Example 1 covers the link types mentioned above. Each link type is depicted by a line. The boxes correspond to atom types.

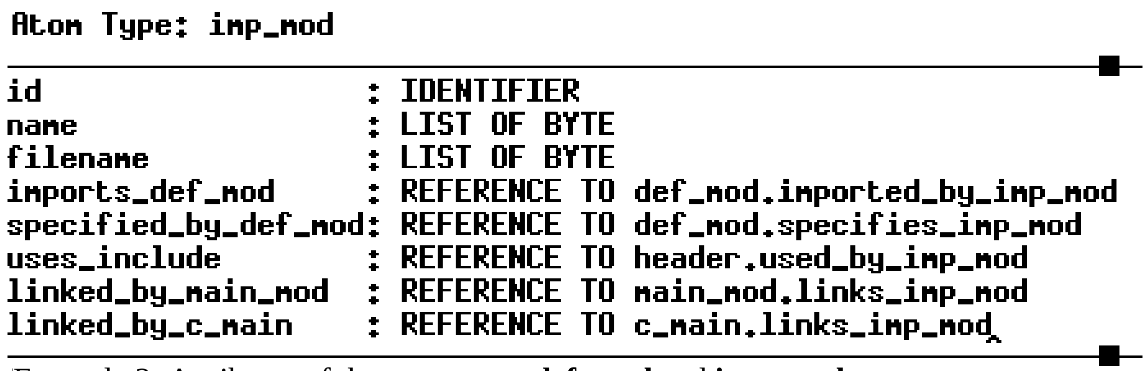
Each atom type of our example consists of several attributes. For each source code module, there is its name, and the file name where it is stored. Furthermore, there is an identifier attribute and a set of REFERENCE attributes to form the links. Example 2 shows the attributes of the atom types **def_mod** and **imp_mod**.

Based on the atom type network, queries may be formulated using the MAD model's query language MQL. In MQL, the retrieval statement has the following shape:



```
Atom Type: def_mod

id                     : IDENTIFIER
name                   : LIST OF BYTE
filename               : LIST OF BYTE
imported_by_def_mod    : REFERENCE TO def_mod.imports_def_mod
imports_def_mod        : REFERENCE TO def_mod.imported_by_def_mod
imported_by_imp_mod    : REFERENCE TO imp_mod.imports_def_mod
imported_by_main_mod   : REFERENCE TO main_mod.imports_def_mod
imported_by_c_mod      : REFERENCE TO c_mod.imports_def_mod
specifies_imp_mod      : REFERENCE TO imp_mod.specified_by_def_mod
uses_include           : REFERENCE TO header.used_by_def_mod
imported_by_c_main     : REFERENCE TO c_main.imports_def_mod
```



```
Atom Type: imp_mod

id                     : IDENTIFIER
name                   : LIST OF BYTE
filename               : LIST OF BYTE
imports_def_mod        : REFERENCE TO def_mod.imported_by_imp_mod
specified_by_def_mod   : REFERENCE TO def_mod.specifies_imp_mod
uses_include           : REFERENCE TO header.used_by_imp_mod
linked_by_main_mod     : REFERENCE TO main_mod.links_imp_mod
linked_by_c_main       : REFERENCE TO c_main.links_imp_mod
```

Example 2: Attributes of the atom types **def_mod** and **imp_mod**

SELECT *projection*
FROM *molecule type definition*
WHERE *condition*

The *molecule type definition* dynamically defines a molecule type (i.e., a complex object type). The *condition* allows to restrict the set of molecules in the result. The *projection* specifies which parts of the molecules are to be included into the result.

A molecule type (in the simplest case) corresponds to a directed coherent sub-graph of the database schema with exactly one root (the so-called root atom type). For example, the molecule type

main_mod-imp_mod

defines a molecule type which groups each **main_mod** together with the **imp_mod** linked when the program is bound. If there is more than one link type between two atom types, the name of the REFERENCE attribute implementing the link type has to be specified in the molecule definition, e.g.

def_mod.specifies_imp_mod-imp_mod

This definition forms molecules which consists of a **def_mod** and the **imp_mod** implementing it. Since this is a 1:1 relationship, each molecule of this type will consist of exactly two atoms.

The molecules are built from the atom network as specified by the molecule type. For each atom of the root atom type, the links of the link types specified in the molecule type are followed leading to atoms of other atom types. For these, the procedure is repeated until all link types of the molecule type are covered. Thus, for each atom of the root atom type, one molecule is derived. Molecules are coherent directed graphs. Obviously, these molecules may overlap each other, and there may be multiple paths within one molecule which lead to the same atom. Molecules need not be “complete”. If for one link type there is no link for the atom under consideration, this link type is ignored. For instance, if an **imp_mod** does not import any **def_mod**, then there is a molecule of the type
imp_mod.imports_def_mod-def_mod
which consists only of one atom (of type **imp_mod**)

One can also define recursive molecule types [Schö89], which then construct the transitive closure of an atom. For example, the molecule type definition
def_mod RECURSIVE def_mod.imports_def_mod-def_mod
defines molecules which for each **def_mod** contain all directly or transitively imported other **def_mod** (transitive closure semantics, i.e., each **def_mod** is contained at most once in the result). In contrast to proposals for the inclusion of recursion in the relational model, recursive molecules do not only show the contents of the transitive closure, but also its structure. Hence, recursive molecules consist of several recursion levels.

The *molecule type definition* yields a set of molecules, i.e., all molecules of the specified type which can be derived from the database. The *condition* in the WHERE clause restricts this set of molecules. The molecules which qualify with respect to the condition undergo the *projection*. In the simplest case, the whole molecule is projected (indicated by the key word ALL in the SELECT clause). One may also enumerate the atom types to be projected. In this case, a coherent graph must be specified. Furthermore, it is possible to project only certain attributes of an atom type. For example, the following query lists the names of all **imp_mod** which import (among others) the **def_mod** “Global_Types.def”.

```
SELECT  imp_mod(name)
FROM    imp_mod.imports_def_mod-def_mod
WHERE   EXISTS def_mod: def_mod.name=’Global_Types.def’;
```

Of course, there are also statements allowing the manipulation of sets of molecules. In addition, there is a data definition sub-language for schema manipulations. The syntax of these parts of the MQL language will not be presented here.

This very short and incomplete presentation of the MAD model may suffice to illustrate the functionality of our graphical interface. More information about the MAD model may be found in [Mits88] and [Schö89].

3 GRIP - The Graphical Interface to PRIMA

As already mentioned, we need graphics to provide an appropriate interface to the MAD model. This fact is stressed by the notions of the MAD model itself. The database schema and the database are undirected graphs, and molecules are directed graphs.

We have implemented our interface GRIP in the X window systems. The master window (see Figure 3) consists of five subwindows:

- The *dialogue window* allows for the alphanumeric input of MQL statements.
- The *message windows* displays status information and error messages.
- The *result window* displays representatives of the molecules resulting from one query.
- The *table window* controls the table-oriented output of query results.
- The *command window* consists of several buttons to direct GRIP’s work. We will discuss their effects in the following.

3.1 Operations on the database schema

As already mentioned, the interface is application independent, and hence it is very important to be able to view the database schema. The command window button “Schema” creates a separate window displaying the schema as a graph of atom types and

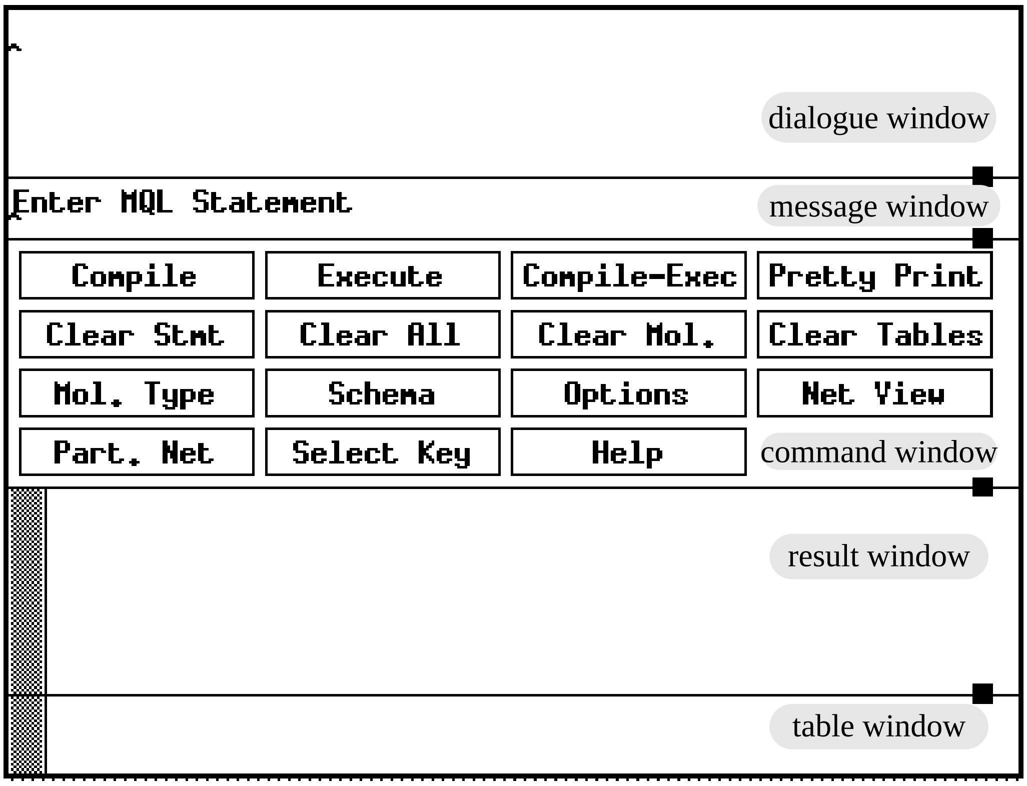
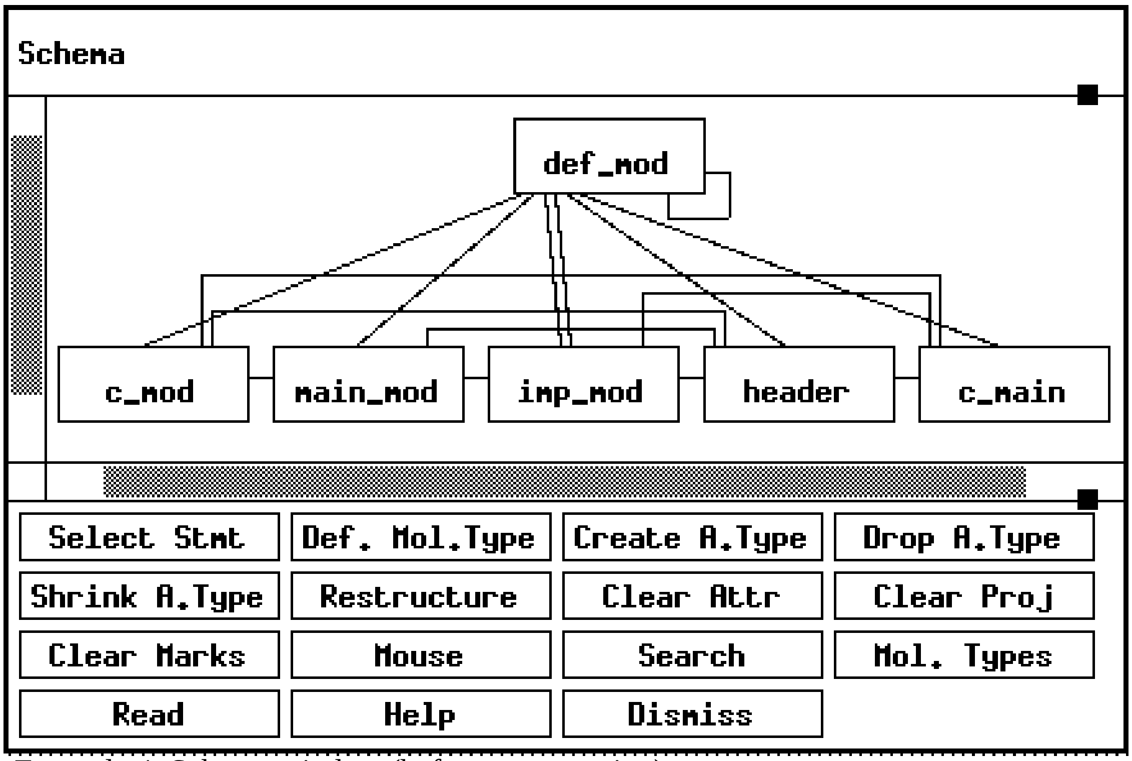


Figure 3: The master window



Example 4: Schema window (before restructuring)

link types of the database. Example 1 and Example 4 show the output of the operation for our sample database.

Since the database schema is an undirected graph, the display of the schema has several degrees of freedom concerning the placement of atom types and link types. In general, a database schema consists of a set of coherent graphs (in our example, there is exactly one coherent graph). For each of the graphs, the number of lines crossing each other should be minimised in order to yield a clear representation. We try to achieve this goal by applying a placement heuristic. The display of the coherent portions of the schema graph is done in an row-oriented fashion, one beneath the other[1]. The first row only contains the atom type participating in the largest number of link types. Hence, the lines representing the corresponding link types do not intersect. The next row contains only those atom types which are connected to the atom type of the first row. There may be link types among atom types which reside in the same row (cf. Example 4). Obviously, it is preferable to place such atom types next to one another, because otherwise the lines of the inter-row link types intersect the link types connecting adjacent rows. Hence, an intra-row placement optimization has to take place. The following rows contain all atom types referenced by atom types of their predecessor row. Thus, lines representing link types connect only adjacent rows. The sequence of the atom types within one row determines the number of intersecting lines between the row and its predecessor, if there are connections to more than one atom type of the predecessor row. Unfortunately, the sequence minimizing the number of crossing lines between two rows may differ from the

[1] with the exception of graphs consisting of only one atom type, which are added to the top row

optimal sequence concerning in-row placement. By default, the dominant optimization criterion is the minimization of inter-row intersection, but the user may change it to intra-row optimization.

The choice of the atom type in the top row may contradict the user's intuitive view of the database. In this case, the user may select another atom type by mouse clicking and put it in the top row by clicking the "Restructure" button[1]. To support the display of a large schema, scroll bars are provided. Furthermore, there is a search function which changes the clipping displayed such that the specified atom type is contained.

In the schema window, only the names of atom types and the existence of link types is depicted. The attributes of an atom type are shown when the atom type is clicked with the help of the mouse. Example 2 shows the resulting output.

Clicking a line (i.e., a link type) displays the name of the two REFERENCE attributes which implement the link type. Optionally, the user may reduce the number of lines between two atom types to one. This is useful when there are many link types between the atom types, because in this case the graph becomes hard to survey. With this option, multiple link types are no longer graphically represented. However, all link types corresponding to a single line are listed when the line is clicked.

Schema modifications may be initiated from the schema window (command buttons "Create A.Type", "Drop A.Types", "Def. Mol.Type"). For these operations, an MQL statement is generated which may be compiled and execute.

As already mentioned, simple MQL retrieval statements just use a coherent directed sub-graph of the database schema graph as molecule type definition. Hence, such MQL statements may be specified graphically (cf. Example 6). Atom types and link types are highlighted with the help of the mouse, and a root atom is designated (marked by a dot). The button "Select Stmt" then generates a corresponding MQL query. Projection may also be specified by clicking to the attributes to be projected. For this purpose, a special projection menu is displayed (see Example 5).

The resulting statement is displayed in the dialogue window of the master window and may be further edited, e.g. to add a WHERE clause. Alternatively, a statement may be completely entered by typing in the dialogue window. Pressing "Pretty Print" button restructures the statement according to its syntactical structure.

3.2 Execution of statements

The "Compile" button in the command window submits the statement in the dialogue window to the compiler of the database system. If there are syntactical errors in the statement, an error message is displayed, and the statement may be edited to correct the errors. A successfully compiled statement is added to the statement clipboard. For retrieval statements, the structure of the resulting molecules may be displayed now. A click on the button "Evaluate" displays the statement clipboard. A statement may be

[1] The schema shown in Example 1 is the result of a "Restructure" operation applied to the schema shown in Example 4.

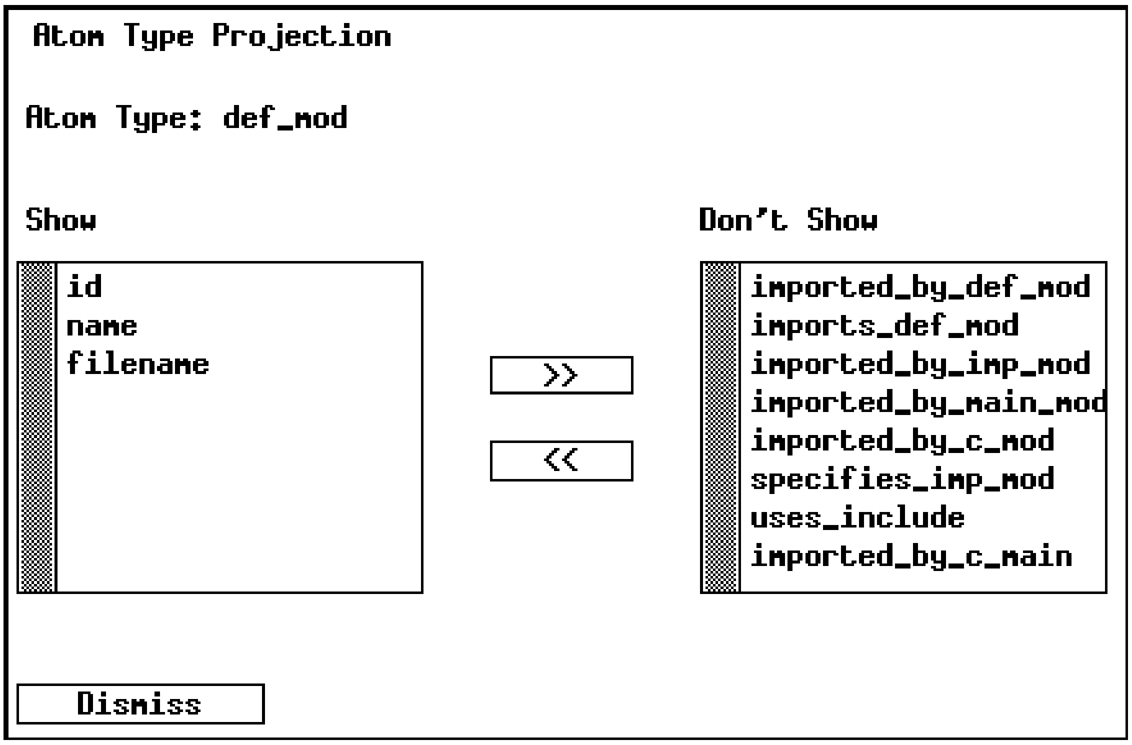
selected and sent to the database system for execution. In the case it is not a retrieval statement, the message window is used to report the success of the statement. The button “Compile-Exec” combines compilation and subsequent execution of a statement.

When the execution of a retrieval statement has been finished, its result is displayed. Example 6 shows the graphical specification of a query (retrieve all **imp_mod** and their imported **def_mod** as well as the **main_mod** they are linked to) and its result set. In the dialogue window, the MQL code generated from the graphical representation of the query is to be seen. The result set consists of many molecules, represented by the name of the **imp_mod** in the result window. The length of the string may be adjusted by the user. The table window displays the name of all atom types forming the molecule type of the result. To cope with the complexity of the molecules and their large number, there are several levels of abstraction which may be chosen for the representation of the result.

3.2.1 Molecule set level

By default, for each molecule in the molecule set a representative is shown in the result window. There is scrollbars to cope with a large result set. GRIP chooses a representative from the attributes of the root atom according to the projection in the retrieval statement. The following rules are applied to determine the representative:

- If there is a one-attribute key declared on the atom type which has either a numerical data type or a string data type and is contained in the projection list, choose it as the representative.


Atom Type Projection

Atom Type: def_mod

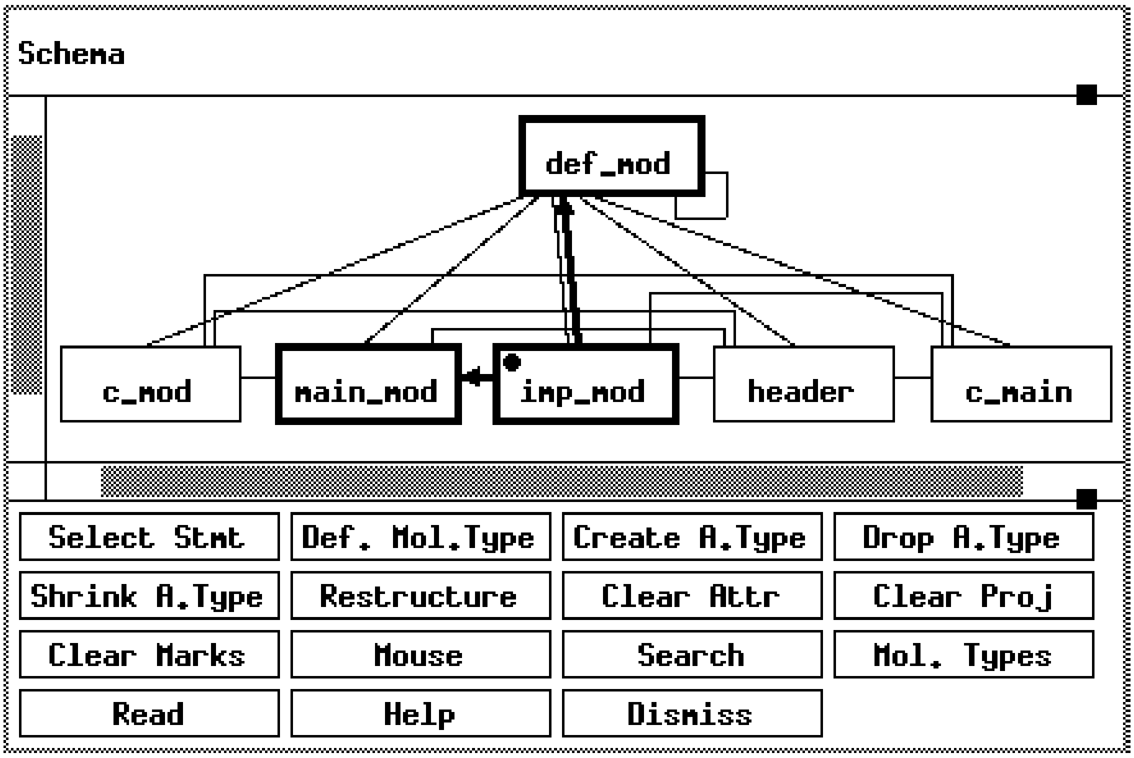
Show

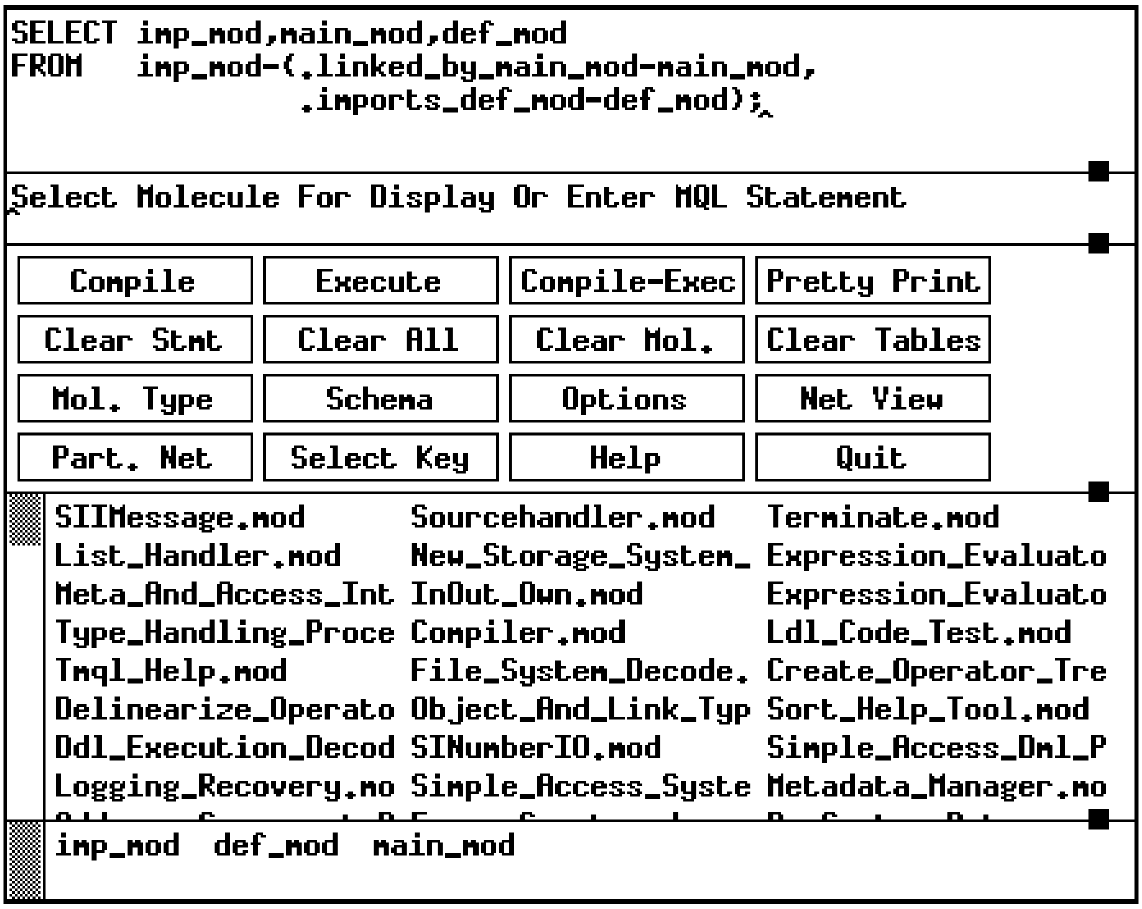
id
name
filename

Don't Show

imported_by_def_mod
imports_def_mod
imported_by_imp_mod
imported_by_main_mod
imported_by_c_mod
specifies_imp_mod
uses_include
imported_by_c_main

Example 5: The projection menu





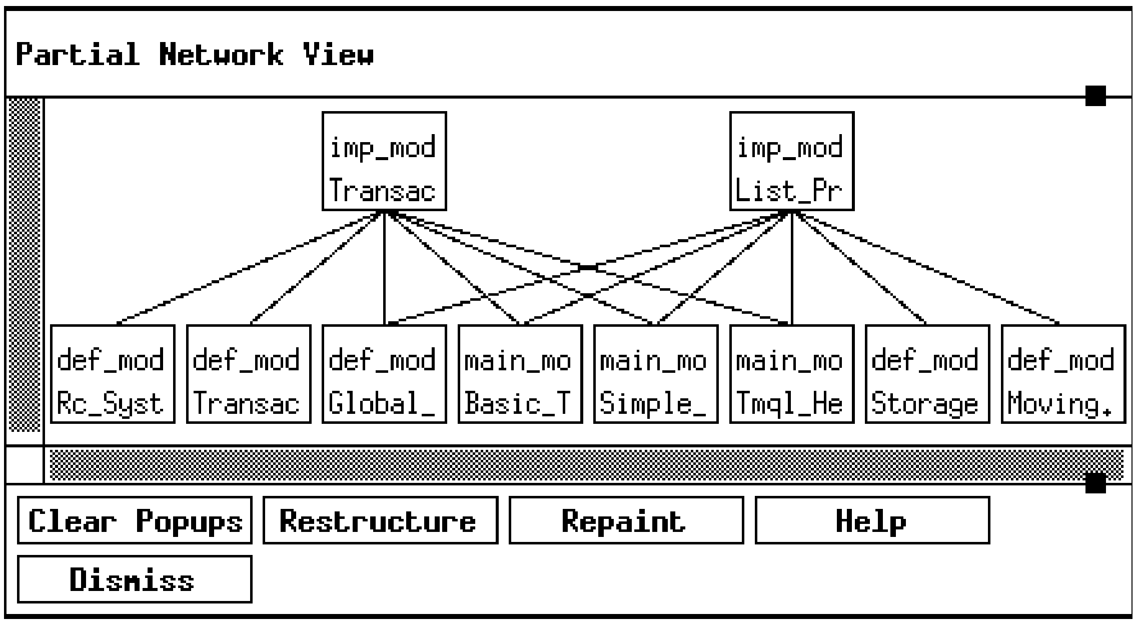
```
SELECT imp_mod,main_mod,def_mod
FROM   imp_mod-(.linked_by_main_mod-main_mod,
                .imports_def_mod-def_mod);
```

Select Molecule For Display Or Enter MQL Statement

```
SIIMessage.mod        Sourcehandler.mod     Terminate.mod
List_Handler.mod      New_Storage_System_   Expression_Evaluato
Meta_And_Access_Int   InOut_Own.mod         Expression_Evaluato
Type_Handling_Proce   Compiler.mod          Ldl_Code_Test.mod
Tmql_Help.mod         File_System_Decode.   Create_Operator_Tre
Delinearize_Operato   Object_And_Link_Typ   Sort_Help_Tool.mod
Ddl_Execution_Decod   SINumberIO.mod        Simple_Access_Dml_P
Logging_Recovery.mo   Simple_Access_Syste   Metadata_Manager.mo
```

```
imp_mod  def_mod  main_mod
```

Example 6: A graphically specified query and its resulting molecule set



Example 7: Two molecules from the result shown in Example 6

- Otherwise, choose the first attribute with data type string, if there is one contained in the projection list.
- Otherwise, choose the first numerical attribute, if there is one contained in the projection list.
- Otherwise, choose the identifier of the root atom (which is always available).

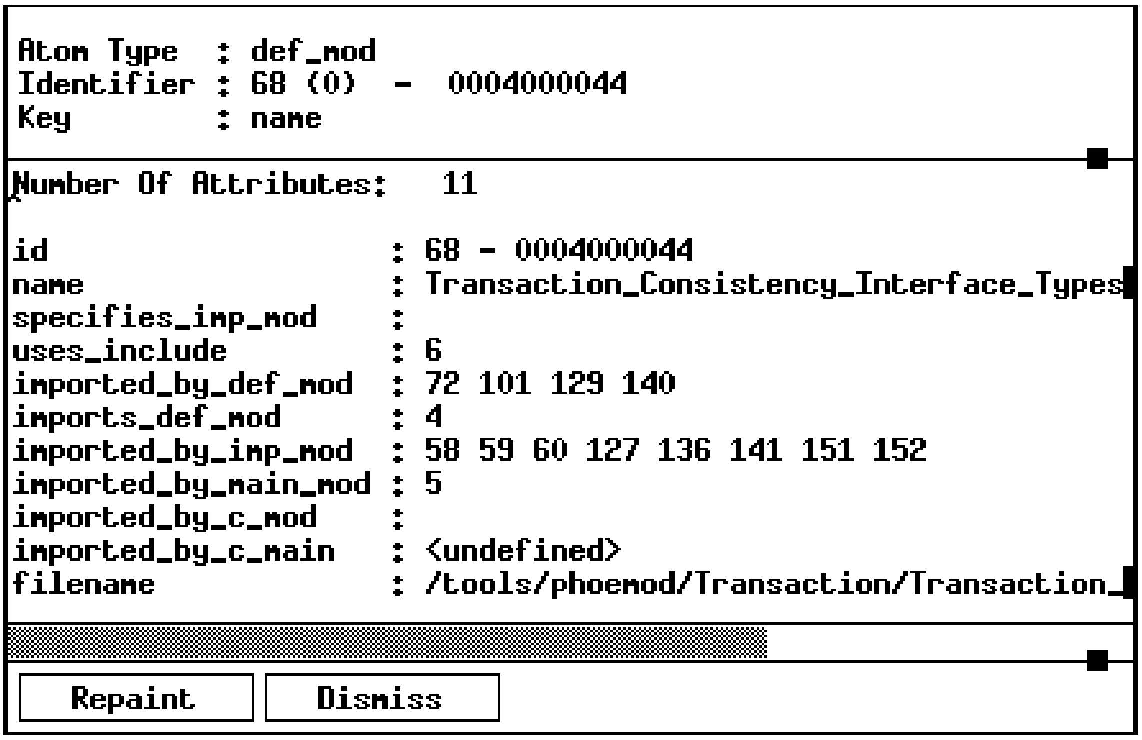
The system’s choice may be altered by the user via the “Select Key” button of the command window.

3.2.2 Molecule level

One can view the structure of the molecules in the result

- for a single molecule by clicking its representative,
- for all molecules by clicking the “Net View” button, or
- for a subset of the molecules by clicking the “Part. Net” button and selecting several representatives.

In the latter two cases molecules are merged, i.e. atoms shared by several molecules are shown only once. However, one can highlight the structure of a single molecule by clicking on one atom of the root atom type. Since this is a directed graph, there is less choice in the display layout: the root atom is placed on the top of the molecule, the atoms referenced in the next row, and so on. Again, for each atom a box is shown, which contains an atom representative, and each link is represented by a line. The user can force the atom type name (cf. Example 7) and / or the identifier of the atom to appear inside the atom box. If the molecule type is not linear, there will be atoms of several types within one row of the molecule representation. To facilitate the distinction among them, different colours or patterns may be assigned to atoms of different types (this is particularly useful



```
Atom Type  : def_mod
Identifier : 68 (0)  -  0004000044
Key        : name
```

```
Number Of Attributes:   11

id                     : 68 - 0004000044
name                   : Transaction_Consistency_Interface_Types
specifies_imp_mod      :
uses_include           : 6
imported_by_def_mod    : 72 101 129 140
imports_def_mod        : 4
imported_by_imp_mod    : 58 59 60 127 136 141 151 152
imported_by_main_mod   : 5
imported_by_c_mod      :
imported_by_c_main     : <undefined>
filename               : /tools/phoemod/Transaction/Transaction_
```

Example 8: Attribute values of an atom

when the name of the atom type is omitted). Example 7 shows two molecules displayed by clicking on the “Part. Net” button.

When displaying sets of molecules which were derived by recursive queries, atom sharing cannot be represented by showing the atom only once, when the atom occurs within several molecules are at different levels of recursion. In this situation, clicking on an atom highlights all representations of the same atom.

3.2.3 Atom level

Clicking on an atom yields another window which displays the values of the atom’s attributes. Example 8 shows the attribute values of the root atom. On the top of the window, there are several general data about the atoms: its identifier, the atom type, and the key defined in the database schema. The user can choose if he wants to see the REFERENCE attributes or not using the “Options” button of the command window.

3.2.4 Table level

The display modes described so far were tailored to molecule-oriented display. Sometimes, however, one might be interested in viewing all atoms of a certain atom type regardless of their position in the molecule structure. For this purpose, one can click the atom type’s name in the table window which results in the display of a table containing all atoms of the selected type. In this table, the width of the columns may be adjusted to the characteristics of the data. A column may be omitted by assigning a width of 0 to it.

Example 9 shows a table containing atoms of type **imp_mod**. Only three columns have been selected for display.

3.3 Implementation

The interface consists of two cooperating processes. One of them is responsible for the display management, i.e. is an X client. The other one communicates with the database. From an architectural view, GRIP is just another application layer on top of the NDBS kernel.

We had to choose the two-process implementation, because on one hand, we have to wait for X events, and on the other hand we have to wait for responses of the database system.

The properties of the representation may be manipulated by X resources. Some properties may also be adjusted interactively by menus displayed when clicking the "Options" button in the command window. A help function is associated with each window to indicate which choices the user has.

4 Comparison to other work

Graphical interfaces to database systems have been studied for several years. In particular, the graphical nature of the Entity-Relationship model [Chen76] has lead to the development of several tools for automatic drawing of Entity-Relationship diagrams [TaBT83], [BaFN85]. The layout is based on a rectangular grid and is optimised

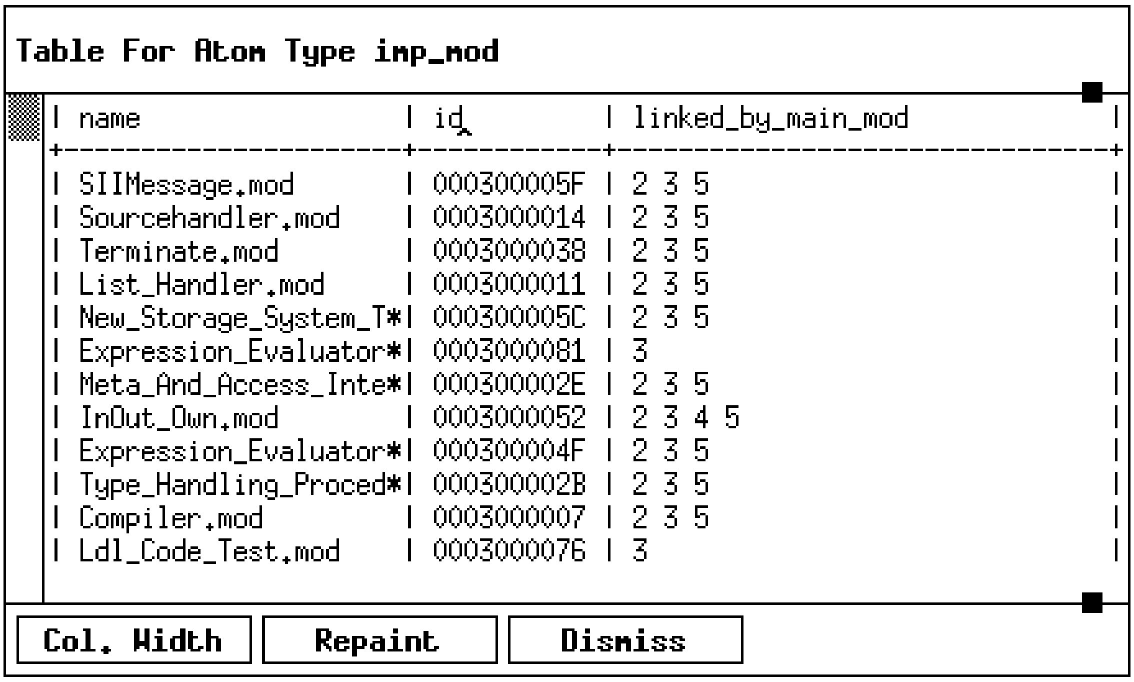


Table For Atom Type imp_mod

```
| name                 | id         | linked_by_main_mod              |
+----------------------+------------+---------------------------------+
| SIIMessage.mod       | 000300005F | 2 3 5                           |
| Sourcehandler.mod    | 0003000014 | 2 3 5                           |
| Terminate.mod        | 0003000038 | 2 3 5                           |
| List_Handler.mod     | 0003000011 | 2 3 5                           |
| New_Storage_System_T*| 000300005C | 2 3 5                           |
| Expression_Evaluator*| 0003000081 | 3                               |
| Meta_And_Access_Inte*| 000300002E | 2 3 5                           |
| InOut_Own.mod        | 0003000052 | 2 3 4 5                         |
| Expression_Evaluator*| 000300004F | 2 3 5                           |
| Type_Handling_Proced*| 000300002B | 2 3 5                           |
| Compiler.mod         | 0003000007 | 2 3 5                           |
| Ldl_Code_Test.mod    | 0003000076 | 3                               |
```

Example 9: Table view of atoms of type **imp_mod** according to the following criteria: minimization of crossings between connections, minimization of the total number of bends in a connection, minimization of the length of connections and minimization of the overall area of the diagram.

The rectangular representation seems adequate, since an Entity-Relationship diagram is an undirected graph. In our case, however, we display the schema graph (which is also undirected and can directly be mapped onto an Entity-Relationship diagram) in a hierarchical way, because this lays emphasis on the molecule type selection mechanism, which is understood as selecting a directed graph from the schema network. For this purpose, the user may restructure the schema graph such that the root of his molecule and the top atom type of the schema graph coincide. Two of the optimization criteria mentioned above are automatically covered by our layout. Connections never span more than two rows and consist only of straight lines (in a connection within a row there may be two bends). While we do not try to reduce the area occupied by the schema graph, we also strive to minimise the crossings between connections.

There are several approaches to graphically formulate a query [ZhMe83], [LeSN89]. The answer, however, is usually given in a tabular form, which to our opinion is caused by the relatively simple structure of the result. In our approach, only basic queries may entirely be formulated graphically. More complex queries need additional editing. We are in doubt that a graphical interface which allows to express arbitrarily complex queries (like queries containing nesting or a Cartesian product) would be manageable by the user.

Recently, there were some efforts to enhance object-oriented database systems by a graphical interface [Wegn89], [BMPP92], [ZoBa92]. In these cases, however, the complex structure of the data is mapped to nested tables. While this representation is sufficient for hierarchical data, it does not reveal sharing of components as it often occurs in the MAD model.

In [MoNK91] the database schema is shown as a directed graph (the direction represents a specialization hierarchy). While interactive schema manipulation is supported, there are no facilities for querying the database.

In the GOOD system [PaVA92] databases are defined as being graphs. The query language is a graph transformation language. However, we do not see any abstraction mechanism which allows the handling of large data sets. In the visual query system described in [CoCM92], abstraction is done by representing nodes of graphs as points and omitting the edges, which seems not to be a very helpful mechanism because the remaining points do not carry any information except their position in the data space (which may not be very informative without the connecting edges). Our abstraction mechanism carries the appropriate information at every level of abstraction.

5 Summary

The DBMS kernel architecture has been developed to serve for the implementation of nonstandard applications. One system which has been implemented according to this architectural model is the NDBS kernel PRIMA. At its interface, it provides the MAD model, which is a general-purpose complex-object data model.

During the development of applications, it is essential to be able to query the database at the MAD model level. Furthermore, PRIMA may be used as an ad-hoc database system. For these purposes, the graphical interface GRIP has been built.

One main function of this interface is the display of the database schema as a network. The schema presentation may be tailored to the user’s needs. Retrieval queries may be graphically formulated by navigating in the schema graph, so that querying the database does not require the knowledge of the syntax of the MAD model’s query language MQL.

The result of a retrieval statement is a set of molecules which may be viewed at several levels of abstraction. The highest level of abstraction is the *molecule set level* where each molecule is represented by just one value. At the *molecule level*, the structure of a set of molecules is shown as a graph consisting of atoms and links, while at the *atom level* the attributes of atoms are shown. All atoms of one type may be viewed at the *table level*.

The system has been implemented on Sun workstations using the X window system.

Acknowledgements

The author wishes to thank Uwe Battschinger for his work in the implementation of the tool described here. The helpful comments of Prof. Dr. Richard Cooper and the anonymous referees are gratefully acknowledged.

References

[BaBu84] Batory, D.S., Buchmann, A.P.: *Molecular Objects, Abstract Data Types and Data Models: A Framework*. In: Proc. 10th Int. Conf. on Very Large Data Bases, Singapore, 1984, pp. 172-184.

[BaFN85] Batini, C., Furlani, L., Nardelli, E.: *What is a good diagram? A pragmatic approach.* In: Proc. 4th Int. Conf. on Entity / Relationship Approach, Chicago, Ill., 1985, pp. 312-319.

[BMPP92] Borras, P., Mamou, J.C., Plateau, D., Poyet, B., Tallot, D.: *Building user interfaces for database applications: The O_2 experience.* SIGMOD RECORD 1992; 21: 32-38.

[CoCM92] Consens, M.P., Cruz, I.F., Mendelzon, A.O.: *Visualizing Queries and Querying Visualizations.* SIGMOD RECORD1992; 21: 39-46.

[Chen76] Chen, P. P.-S.: *The Entity-Relationship Model - Towards a Unified View of Data.* ACM Transactions on Database Systems 1976; 1: 9-36.

[HMMS87] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: *PRIMA - A DBMS Prototype Supporting Engineering Applications.* In: Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, 1987, pp. 433-442.

[HüSu92] Hübel, C., Sutter, B.: *Supporting Engineering Applications by New Data Base Processing Concepts - An Experience Report.* Engineering with Computers 1992; 8: 31-49.

[KäRS90] Käfer, W., Ritter, N., Schöning, H.: *Support for Temporal Data by Complex Objects*. In: Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, 1990, pp. 24-35.

[LeSN89] Leong, M.-K., Sam, S., Narasimhalu, D.: *Towards a Visual Language for an Object-Oriented Multi-Media Database System*. In: Kunii, T.L. (ed.): Visual Database Systems, North-Holland, 1989, pp. 465-495.

[Mits88] Mitschang, B.: *Towards a Unified View of Design Data and Knowledge Representation*. In: Kerschberg, L. (ed) Proc. 2nd Int. Conf. on Expert Database Systems, 1988, pp. 33-50.

[MoNK91] Morsi, M. M. A., Navathe, S. B., Kim, H.-J.: *A Schema Management and Prototyping Interface for an Object-Oriented Database Environment.* In: Proc. IFIP working group 8.1 Conference on The Object Oriented Approach in Information Systems, Quebec City, Canada, North Holland, 1991, pp. 157-180.

[PaVA92] Paredaens, J., Van den Bussche, J., Andries, M., et al.: *An Overview of GOOD*. SIGMOD RECORD 1992; 21: 25-31.

[Schö89] Schöning, H.: *Integrating Complex Objects and Recursion.* In: Proc. 1st Int. Conference on Deductive and Object-Oriented Database Systems, Kyoto, 1989, pp. 535-554.

[TaBT83] Tamassia, R., Batini, C., Talamo, M.: *An Algorithm for Automatic Layout of Entity Relationship Diagrams.* In: Proc. 3rd Int. Conf. on Entity / Relationship Approach, Anaheim, Ca., 1983, pp. 421-439.

[Wegn89] Wegner, L.: *ESCHER - Interactive, Visual Handling of Complex Objects in the Extended NF^2-Database Model*. In: Kunii, T.L. (ed.) Visual Database Systems, North-Holland, 1989, pp. 277-297.

[ZoBa92] Zoeller, R.V., Barrty, D.K.: *Dynamic Self-Configuring Methods for Graphical Presentation of ODBMS Objects.* In: Proc. 8th Int. Conf. on Data Engineering, Tempe, Az., 1992, pp. 136-143.

[ZhMe83] Zhang, Z.-Q., Mendelzon, A. O.: *A Graphical Query Language for Entity-Relationship Databases.* In: Proc. 3rd Int. Conf. on Entity / Relationship Approach, Anaheim, Ca., 1983, pp. 441-448.