Visually Querying Object-Oriented Databases

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Manoj Chavda
1996

Supervised by
Associate Professor P. T. Wood

Abstract

As database requirements increase, the ability to construct database queries efficiently becomes more important. The traditional means of querying a database is to write a textual query, such as writing in SQL to query a relational database. Visual query languages are an alternative means of querying a database; a visual query language can embody powerful query abstraction and user feedback techniques, thereby making them potentially easier to use.

In this thesis we develop a visual query system for ODMG-compliant object-oriented databases, called QUIVER. QUIVER has a comprehensive expressive power; apart from supporting data types such as sets, bags, arrays, lists, tuples, objects and relationships, it supports aggregate functions, methods and subqueries. The language is also consistent, as constructs with similar functionality have similar visual representations. QUIVER uses the DOT layout engine to automatically layout a query; QUIVER queries are easily constructed, as the system does not constrain the spatial arrangement of query items. QUIVER also supports a query library, allowing queries to be saved, retrieved and shared among users.

A substantial part of the design has been implemented using the ODMG-compliant database system O_2, and the usability of the interface as well as the query language itself is presented. Visual queries are translated to OQL, the standard query language proposed by the ODMG, and query answers are presented using O_2Look. During the course of our investigation, we conducted a user evaluation to compare QUIVER and OQL. The results were extremely encouraging in favour of QUIVER.

Acknowledgements

Firstly, I would like to thank my supervisor Peter Wood. His patience, attention to detail and his ability to keep track of the big picture made developing QUIVER fun rather than phobic.

A big thank you to O_2 for providing us with a database and a subsequent upgrade. Thank you also to the various technicians at the online O_2 help line for their excellent e-mail support. Thanks to John Ousterhout and his team for developing `Tcl` and `Tk` [38], without which none of QUIVER would have been possible. Thanks to D. Richard Hipp for writing ET2C [22], which greatly aided the linking of `Tcl` and `C++`. Thanks to Don Libes for developing the `Tcl` debugger `tcldebug` [29] and to Andreas Zeller and Dorothea Lütkehaus for the `C++` debugger `DDD` [49]. Thanks also to J. Nijtmans for writing a patch [32] permitting dashed lines to be drawn in `Tk`. Thanks to Eleftherios Koutsofios and Stephen North for developing DOT [27], and especially to Stephen North for the e-mail support in sorting out various DOT problems. Thanks also to the ODMG for designing a standard, and for the email support in answering various technical questions.

For financial support I thank the FRD for funding my research over the past eighteen months.

Thank you to our system administrators Sandi Donno and Aleks Strez for keeping my computing environment running smoothly. Thanks also to the thirteen honours students who participated in the QUIVER user evaluation; your patience and enthusiasm is well appreciated.

Finally, a big thank you to my parents, brothers and sister. Their continual support and encouragement provided in so many ways inspired me much of the way; it is my privilege to share my successes with them.

Contents

List of Figures

Chapter 1

Introduction

As information requirements increase, the need for databases increases and the ability to construct queries efficiently in order to query a database becomes more important. The relational paradigm and relational databases are well established and are broadly used in order to store information. The traditional means of querying a relational database has been to write textual queries. In the relational context, the traditional means of querying has been to write SQL queries.

Visual query languages are an alternative means of querying databases. Visual query languages are interactive, and can provide feedback to the user during query construction. A visual query system can, for example, prevent syntactically incorrect queries from being constructed or can visually display where a query has been incorrectly constructed, thus aiding correction. A visual query language can also embody powerful abstraction techniques, using metaphors that the user identifies with in order to represent unfamiliar database and querying concepts. These factors make constructing queries in visual query systems potentially easier than constructing the same queries by writing them in a textual query language such as SQL. As more people without technical knowledge in computers are using computers, the need for an improved interface increases.

Object-oriented databases (OODBs) are, commercially, an emerging technology. An object-oriented paradigm is able to model the real world better than a relational paradigm, as it maps real world objects and associations between objects onto database objects. The widespread use of object-oriented databases, however, has been particularly hampered by the lack of a query language standard and database interface standards. In 1993, the Object Database Management Group (ODMG) released their first draft of an OODB standard, called ODMG-93 [6]. The "standard" consists of four components. They are the object definition language (ODL), an object query language (OQL), and bindings for the programming languages `C++` and `Smalltalk`.

Much of the work in visual query languages has been to design query languages for

relational databases. In this thesis, we develop a visual query system, QUIVER, for object-oriented databases. QUIVER is an acronym for "QUerying in an Interactive Visual EnviRonment". The system supports the ODMG data model and the query language supports objects, literals and structures as well as object relationships and methods. Aggregate functions, the collection types bags, arrays, lists and sets and the construction and use of subqueries are also supported. The query language is comprehensive as a large range of query concepts are represented by the visual constructs. The language is also consistent; similar query concepts are represented by similar looking and acting query constructs, reducing the number of basic query symbols.

We also describe a substantial implementation of the query language using the ODMG-compliant object-oriented database O_2 [34]. Queries constructed in QUIVER are translated into OQL, the object query language of O_2; the OQL query string is processed by the O_2 query engine and the query answers are displayed using the O_2 application O_2Look. QUIVER also communicates with a graph layout engine DOT [27]; the use of DOT aids the neat appearance of visual queries. QUIVER translates, with one minor exception, the full range of queries that can be constructed using its query language. The validity of QUIVER was tested in a user evaluation; the results were very encouraging and we describe the user evaluation in detail. Finally, QUIVER supports a query library, allowing queries to be saved, retrieved and shared among users.

1.1 Overview of the Thesis

Chapter 2 presents a brief tour of QUIVER. The usability of the QUIVER GUI is presented by demonstrating the step-wise construction of various queries. The chapter contains numerous screen snapshots in order to gain an understanding of the system.

In the last few years, many visual query systems have been proposed. Chapter 3 provides background information used to classify visual query systems, and a number of visual query systems (including QUIVER) are surveyed and classified according to these criteria. Chapter 4 provides background information on the ODMG and O_2. The ODMG data model, the modeling primitives and the relationships between primitives are introduced. The differences between the O_2 data model and the ODMG data model are discussed, followed by a discussion on the object query language OQL.

The QUIVER visual query language is discussed in Chapter 5. The motivating principles of the query language are mentioned, and the visual representation of basic data types is presented. Queries using only these basic data types are discussed, and the uses of further constraint conditions are then presented. There-

after, other types of queries and the representations of methods, functions, query output and subqueries and named queries are discussed.

Various implementation related issues are presented in Chapter 6. Implementation issues concerning the retrieval of database schemas are discussed, followed by a detailed discussion on the QUIVER GUI. The interface between QUIVER and O_2 is then presented, followed by the format used by QUIVER to save and retrieve queries to and from disk. Finally, the data formats used to communicate with the graph layout engine DOT are discussed.

Chapter 7 explains the query translation mechanisms used by QUIVER to translate a visual query to OQL. Initially, the translation is explained by means of examples. QUIVER identifies three different types of visual queries, and the translation of each of these types is explained; the translation of subqueries is then explained. Thereafter, the general rules pertaining to query translation are presented. Finally, alternative translations are discussed.

A user evaluation was conducted to determine the usability of QUIVER as a querying application; this user evaluation is presented in Chapter 8. The experimental setup and the two hypotheses tested are discussed. The results of the evaluation are presented, followed by possible sources of experimental error. Participants were also asked to comment on QUIVER and OQL; the discussion of the experiment results as well as these comments are included. The conclusions drawn from the experiment are then stated.

Appendix A contains the grammar of an O_2 ODL (O_2 schema definition) file in informal BNF notation. Appendix B contains further details on the user evaluation conducted, namely the schema and the queries used. This thesis does not provide any formal proof of the expressive power of QUIVER, but Appendix C provides a reasonable feel of the versatility of the query language. Appendix C provides QUIVER translation of queries from the visual query systems surveyed in Chapter 3.

Chapter 2

An Overview of QUIVER

2.1 Introduction

This chapter presents a brief tour of QUIVER. The design of the QUIVER graphical user interface (GUI) is crucial to the construction of correct queries, and its usability is presented in this chapter. The usability of the GUI is demonstrated by showing the step-wise construction of various queries and thereby introducing features that the GUI provides. Screen snapshots are provided at various stages.

2.2 Basic Operation

The QUIVER GUI consists of two main components, namely the *button bar* and the *canvas* (see Figure 2.2). The button bar is at the left of the QUIVER window and consists of two columns of buttons; each button is represented by an icon. The canvas is the large area to the right of the window and is the area in which query items are placed in order to construct a query. The scroll bars at the bottom and the right of the canvas allow it to contain an area larger than it occupies on screen. A *status bar* at the bottom of the QUIVER window provides useful messages to the user.

All query items are manipulated using the mouse. The only time the keyboard has to be used is to type in a literal value. Each button represents a query activity, and a button is activated by moving the mouse cursor over it and clicking the mouse; it is deactivated by clicking the mouse over it a second time.

When some buttons are activated, such as the *add property* button (), QUIVER first enters a *roaming* mode where it waits for the user to position the cursor over a suitable node. The node will then be highlighted; moving the cursor away from the node dims it. If the user clicks the mouse over a highlighted node an action

occurs. For example, when adding a property, QUIVER automatically creates new nodes to represent the property value. QUIVER then enters *placing* mode where the user drags the outline of the new node and when the mouse is clicked, the outline is replaced by the node itself. The outline appears as a hollow rectangle and its size indicates the size of the new nodes. If the user were to activate some other button, such as the *place class* button () then the *add property* button would automatically be deactivated before the *place class* button is activated.

2.3 Constructing a Constraint Query

Assume we have a university schema containing `Student`, `Course`, `Building` and `Module` objects; the named values `Students`, `Courses`, `Buildings` and `Modules` represent the extents of these object types. A course consists of a many modules, and a module is lectured in a particular building; modules and courses also have names. The method `courses_by_marks` of the `Student` class returns a bag of courses for which the student scored between *min* and *max*. The full schema is presented in Figure 4.5, page 47.

Assume we wish to construct the query "Are there any students who have scored between 75 and 100 in the `STA200W` course?". Examination of the schema suggests that we use the `courses_by_marks` method, supplying 75 for the parameter *min* and 100 for the parameter *max*. A suitable query entry point is the `Students` named value, which represents the extent of `Student` objects.

The *named values* button () produces a list of all the named values defined in the current schema (Figure 2.1). The user chooses `Students`, which is represented by a screened (grey) circle node surrounded by a blob node (representing a bag of `Student` objects) and QUIVER generates an outline of these nodes. QUIVER then enters *placing* mode, waiting for the user to select a position for these nodes. Once the outline has been placed, the *named values* button is automatically deactivated (Figure 2.2). The text labels of nodes contain a prefix and a suffix, separated by a colon symbol. The prefix indicates the named value (if one exists) associated with the node and the suffix indicates the type of the node. Next, the user wishes to place the `courses_by_marks` method of the `Student` object. The *add method* button () is activated, and QUIVER enters *roaming* mode. *Roaming* mode is terminated when the mouse is clicked over a class node. A list of the methods associated with that class, such as in Figure 2.2, is produced. From the figure we see that only one method has been defined for the `Student` class; the user selects this method. Method execution is represented by a closed (black) square since the method contents are hidden, and QUIVER remains in *placing* mode until the user places this node. An edge is automatically placed between the `Student` node and the method node; the edge is labeled `courses_by_marks` (see Figure 2.3).

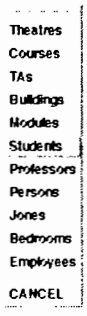


Figure 2.1: List of the named values

The user now wishes to specify the inputs to the method. The *method input* button was automatically activated (and the *add method* button automatically deactivated) after the user placed the method node. The user clicks the mouse while the method node is highlighted. The method requires two inputs (as it is defined with two parameters) and these inputs are represented by a single node which itself gets expanded – the input node appears as a structure consisting of two cells. The user clicks the mouse again to end the *placing* mode. The *method input* button remains activated and the user clicks on each of the cells in the input structure node in order to place the literal node associated with it. Each method parameter has an integer value, and is represented by an open (white) circle node. Directed edges labeled with the name of the parameter are drawn by QUIVER between the input node and each of the literal nodes (see Figure 2.3).

The user now sets the values of the parameters by activating the *change text* button (A); the *method input* button is automatically deactivated. The mouse is clicked over each of the literal nodes and an editing box allows entry of each of the text values 75 and 100. Figure 2.3 shows the parts of the query that have been constructed thus far.

In order to complete the query, the output of the method has to contain at least one `STA200W` object; this is specified by asserting that the `name` property of at least one of the objects within the method output has the value `STA200W`. The method output is placed by activating the *method output* button ( ). QUIVER again enters *roaming* and *placing* modes. The method output is represented by a screened circle (a `Course` object) surrounded by a blob (a bag of `Course` objects) – see Figure 2.4. In order to extend the `name` property, the user activates the *add property* button ( ) and clicks the mouse while over the `Course` object. The name is a string literal (represented by an open circle) and the value of this literal is set to `STA200W` using the *change text* button (A). The final form of the query is displayed in Figure 2.4. The query contains data-flow edges (the dashed edges) to and from the method node indicating the flow of data to and from the method.

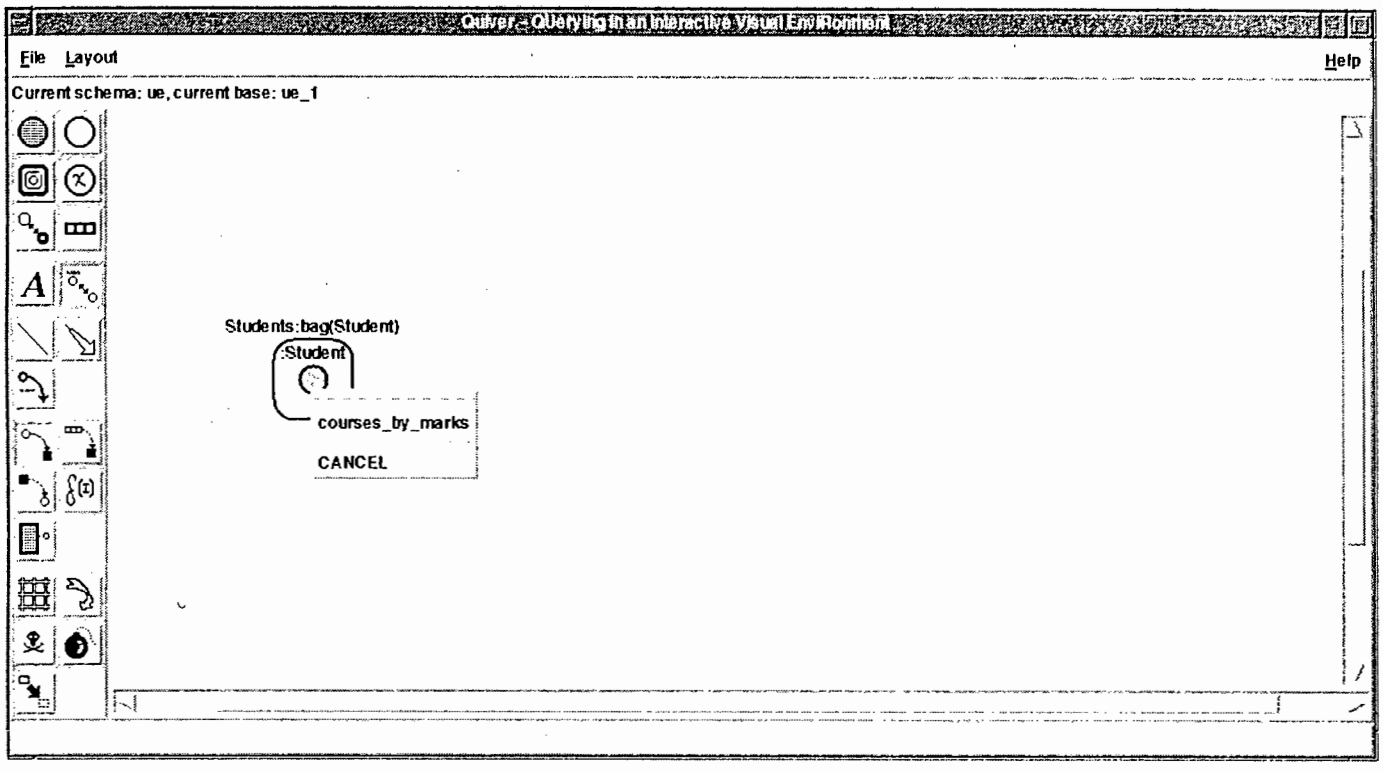


Figure 2.2: Methods associated with a class

The output of the query will be a boolean value – either TRUE or FALSE. The query translation engine is activated either through selection from the menus, or by pressing CONTROL-Q. O_2 is used as the underlying persistent store, and the query is sent to the O_2 query interpreter as an OQL query string; O_2 also generates a graphical representation of query answers. TRUE is represented by an activated radio button (which appears black), while a deactivated radio button (which appears white) represents FALSE. The answer window (Figure 2.5) shows a black radio button, indicating that the answer to the query is TRUE – there *does* exist at least one student who scored a first in the STA200W course.

This query is not very informative, as it does not tell us who the student is or students are. This query will now be modified so that it returns the object instances that satisfy the query constraints.

2.4 Generating Output

Assume that we wish to construct the following query: "Find those students who scored between 75 and 100 in any courses that contain the Graphics1 module". Recall that a course has a property called consists_of which is the bag of modules that a course consists of.

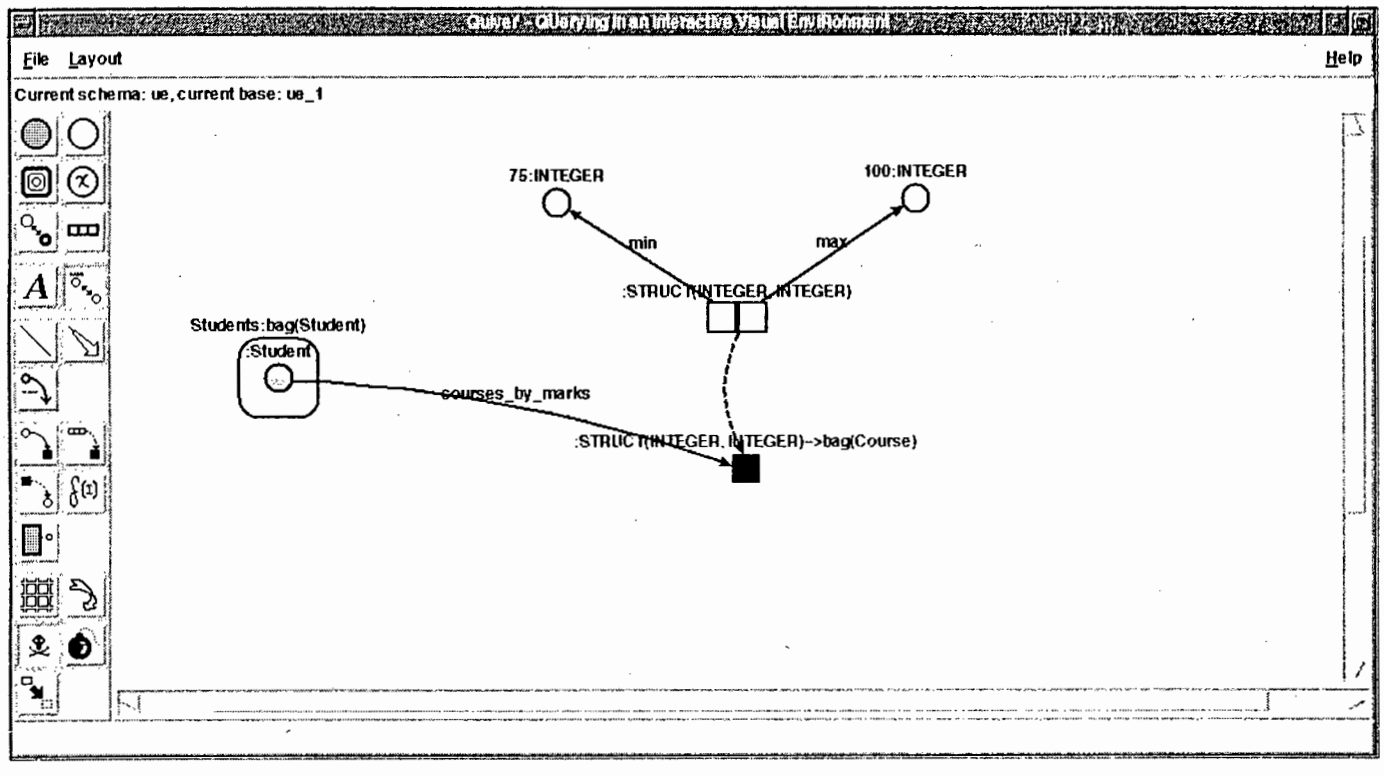


Figure 2.3: Specifying the inputs of a method

First, the node representing the name property value is deleted using the *delete* button ( ). Next the *add property* button ( ) is activated and the consists_of property of Course is selected. QUIVER generates the nodes to represent this property value – a screened circle and a blob (a bag of Module objects). The mouse is now clicked over the circle node representing the Module class, and the name property is selected. The name property has a literal value, and is represented by an open circle node. The *change text* button ( ) is activated and the literal's value is changed to Graphics1.

Next, the graph needs to be modified to return all the Student objects from the successful data patterns; **bold** representations indicate query output. The *modify to bold* button ( ) is activated and the mouse is clicked while over the circle node representing the Student object. The node's frame changes from a thin line to a thick line. Next, the *collecting blob* button ( ) is activated and the mouse is clicked while over the blob node surrounding the node representing the Student object. A bold blob (called the *collecting* blob) is placed around this node to become the new outermost blob of this *inclusion group*. The final query appears in Figure 2.6.

The bold items in the *output* inclusion group indicate the type of the query output. In this case, the answer type is a bag of Student objects (as both a

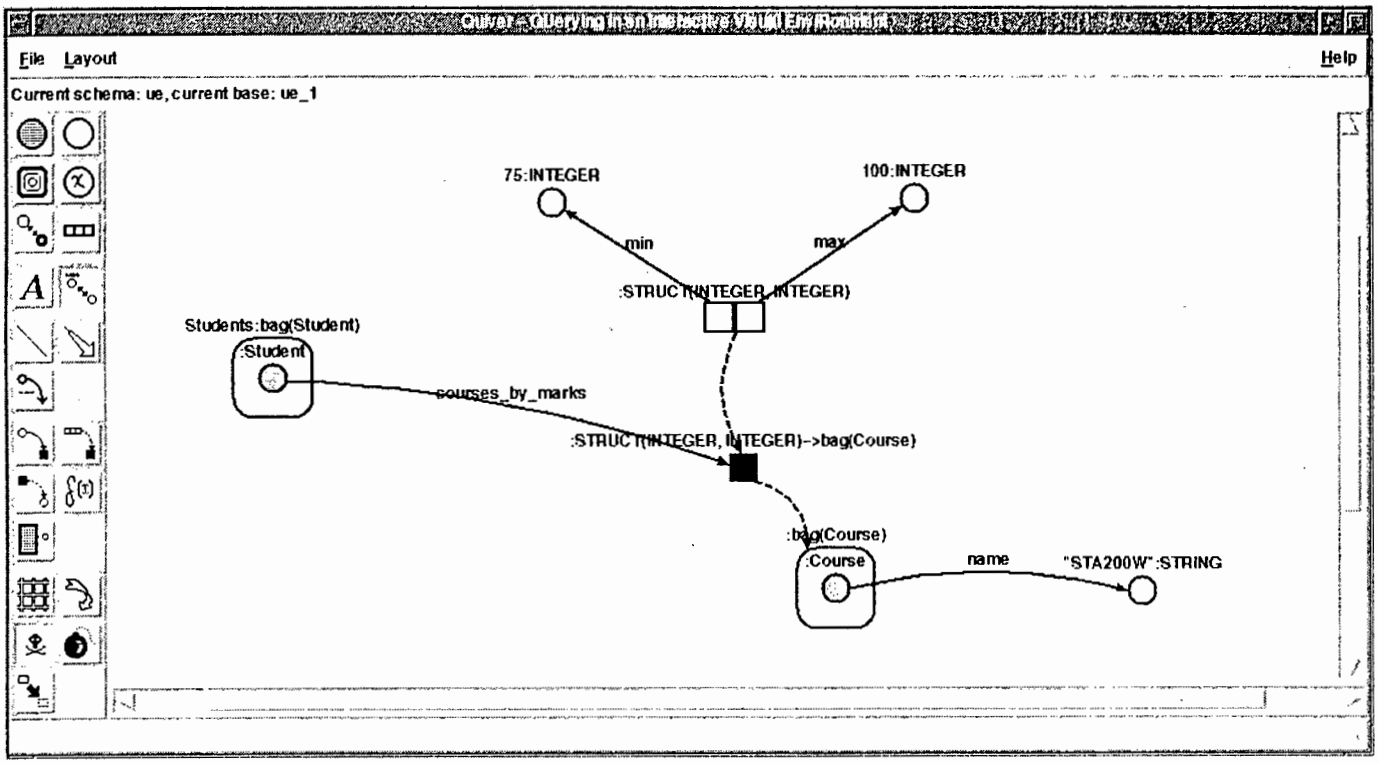


Figure 2.4: Are there students who scored between 75 and 100 for STA200W?

Figure 2.5: Graphical representation of TRUE

circle node and a blob node are bold). By marking the circle node as bold, QUIVER adds a Student object to the output for every successful match of the query pattern against the underlying data. The query answer (generated by O_2) is shown in Figure 2.7; two students scored firsts, namely Veena and Sailesh. Right-clicking the mouse over an object in the answer window generates a menu from which the user can select to view all the property values of that object.

2.5 Subqueries

Other QUIVER features are now introduced. Assume that we wish to construct the following query: "Which buildings are courses lectured in?". The old query is erased from the canvas by activating the *delete all* button ( ); the new query is shown in Figure 2.8.

Now assume that we wish to construct the following query: "Find those courses

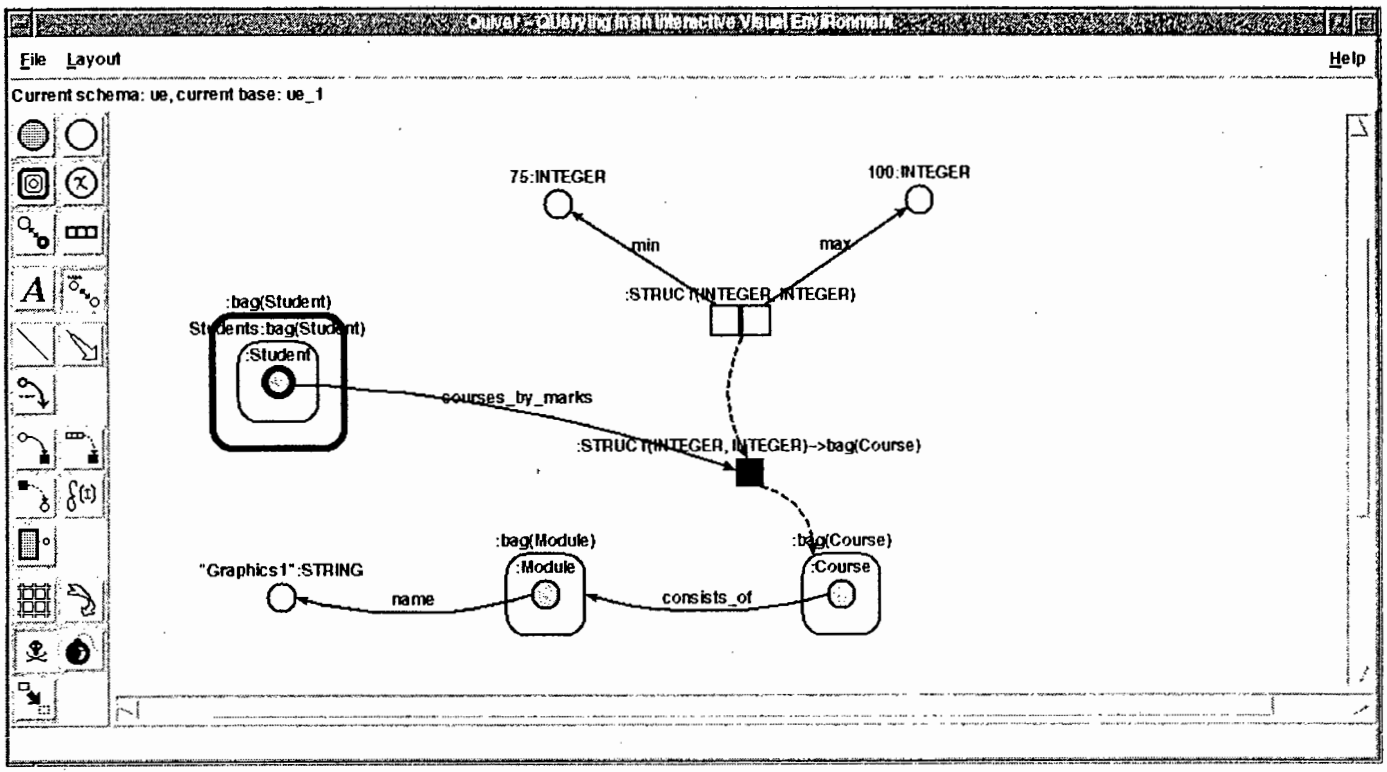


Figure 2.6: Generating Output

Figure 2.7: Output of the Query of Figure 2.6

whose modules are lectured across *all* the buildings on campus." Equivalently, if we accumulate all the buildings in which a course's modules are lectured, and if those buildings contain all the buildings on campus, then include the course in the output. This query will be constructed using a subquery – the three inclusion groups constructed in the previous query will form the inner query.

The *mark subquery* button ( ) is activated and all the inclusion groups are selected by dragging a rectangular-shaped rubber band. While moving the mouse, all nodes that the rubber band encloses become highlighted. The properties of a subquery are similar to the properties of a method. Both involve the execution of some type of code and both have output. Unlike methods, however, the "code" of a subquery (under the current implementation) is always displayed. A subquery thus appears as an open box (as in Figure 2.9) that surrounds the nodes that define it. The output of a subquery is represented the same way as the output of a method is indicated – by a dashed edge leaving it. In order to place a subquery's

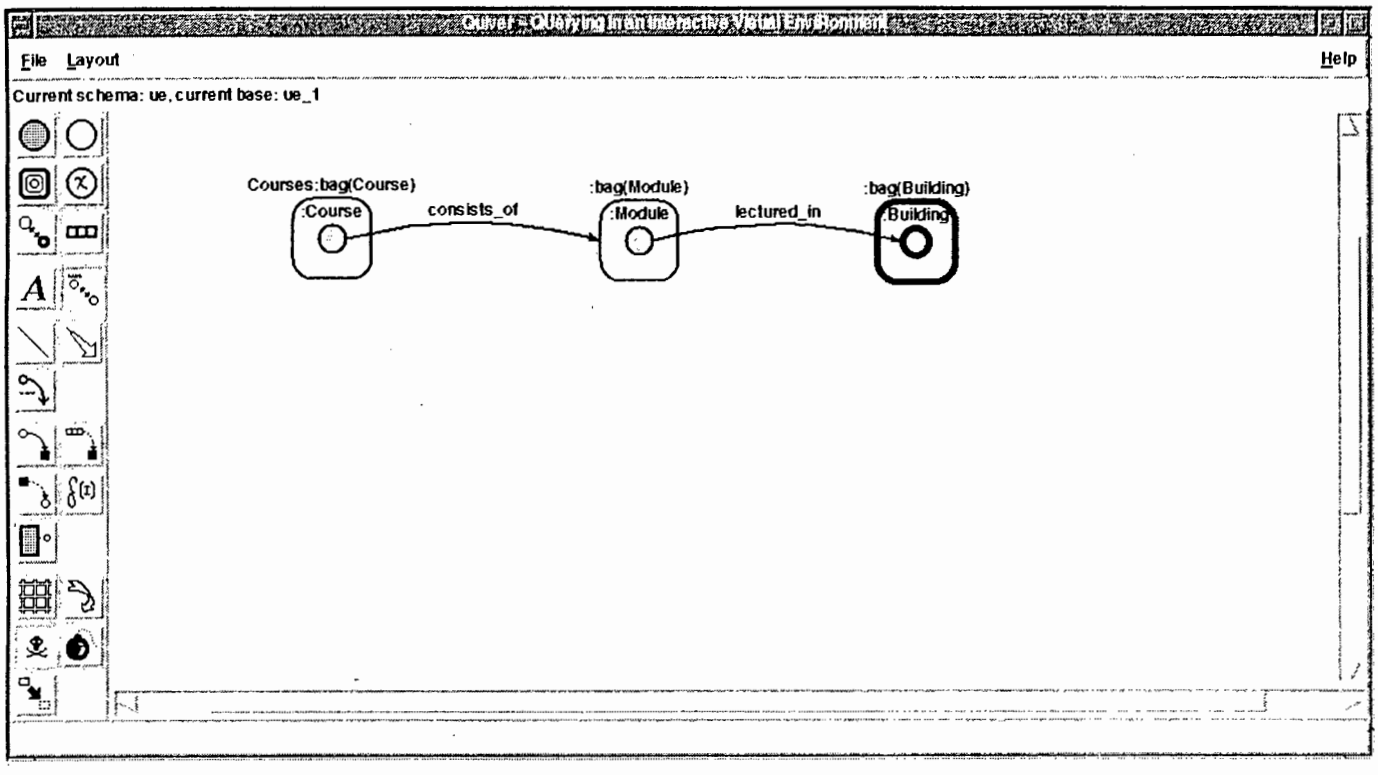


Figure 2.8: Finding buildings

output, the *method/subquery output* button ( ) is activated. QUIVER generates the output type of the subquery – a bag of `Building` objects (see Figure 2.9).

For each `Course` object in the database, we wish to execute the subquery to obtain the buildings in which the course's modules are lectured. The *named values* button ( ) is activated, the `Courses` value is selected and its nodes are placed on the canvas. In order to set this `Course` object equal to the `Course` object within the subquery, the *equality edge* button ( ) is activated (see Figure 2.10). Next, we wish to assert that the output of the subquery contains at least all the buildings on campus. The extent of all `Building` objects is the named value `Buildings`, and this is placed on the canvas. The *subset* button ( ) is activated and is used to place the subset constraint that `Buildings` should be a subset of the output of the subquery.

Finally, the output of the query is set. We wish to add all the `Course` objects to the query output; the screened circle node representing the `Course` object is turned bold using the *modify to bold* button ( ) and a collecting blob is added as the new outermost blob, again using the *collecting blob* button ( ). The final query is displayed in Figure 2.10.

By saving and retrieving queries and by using subqueries, QUIVER encourages

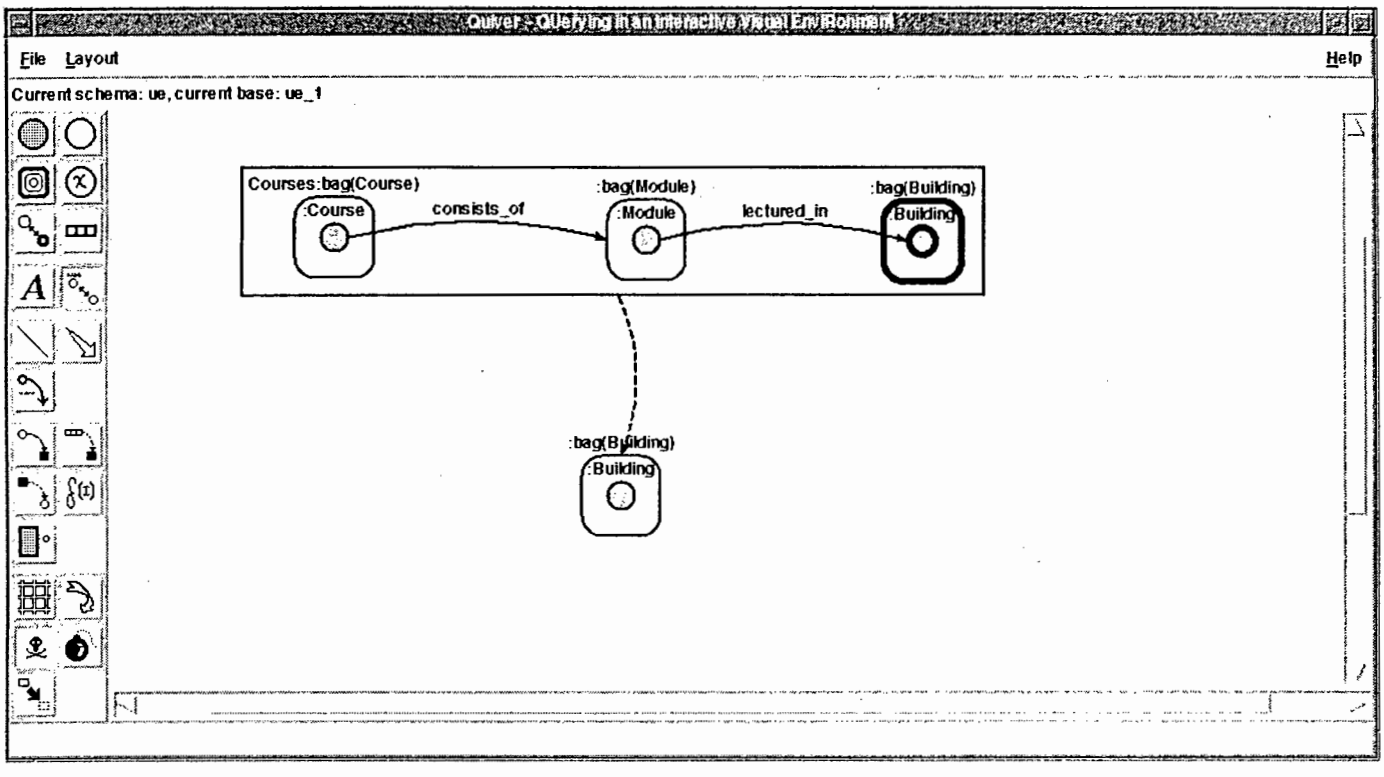


Figure 2.9: Drawing a subquery's output

the reuse and sharing of queries.

2.6 Layout

In order to aid the user in constructing neat, visually appealing queries, QUIVER uses the DOT graph layout engine [27]. The user calls the graph layout engine (the engine acts on the current query graph) by hitting the *layout* button ( ).

The query graph is translated into a series of nodes and edges, and DOT processes this information. DOT tries to minimize edge crossings and tries to keep edges short. The user may hit `CONTROL-O` to set the options for the graph layout engine; these are setting whether the graph should follow a left-right orientation, or a top-down orientation, and the space between the inclusion groups. Figure 2.11 shows the graph after the layout engine was activated; hitting the *undo layout* button ( ) returns the graph to its appearance before the layout engine was activated.

Note that the layout action does not alter the meaning of the graph; it simply alters the graph's appearance. The *move* button ( ) allows the user to freely

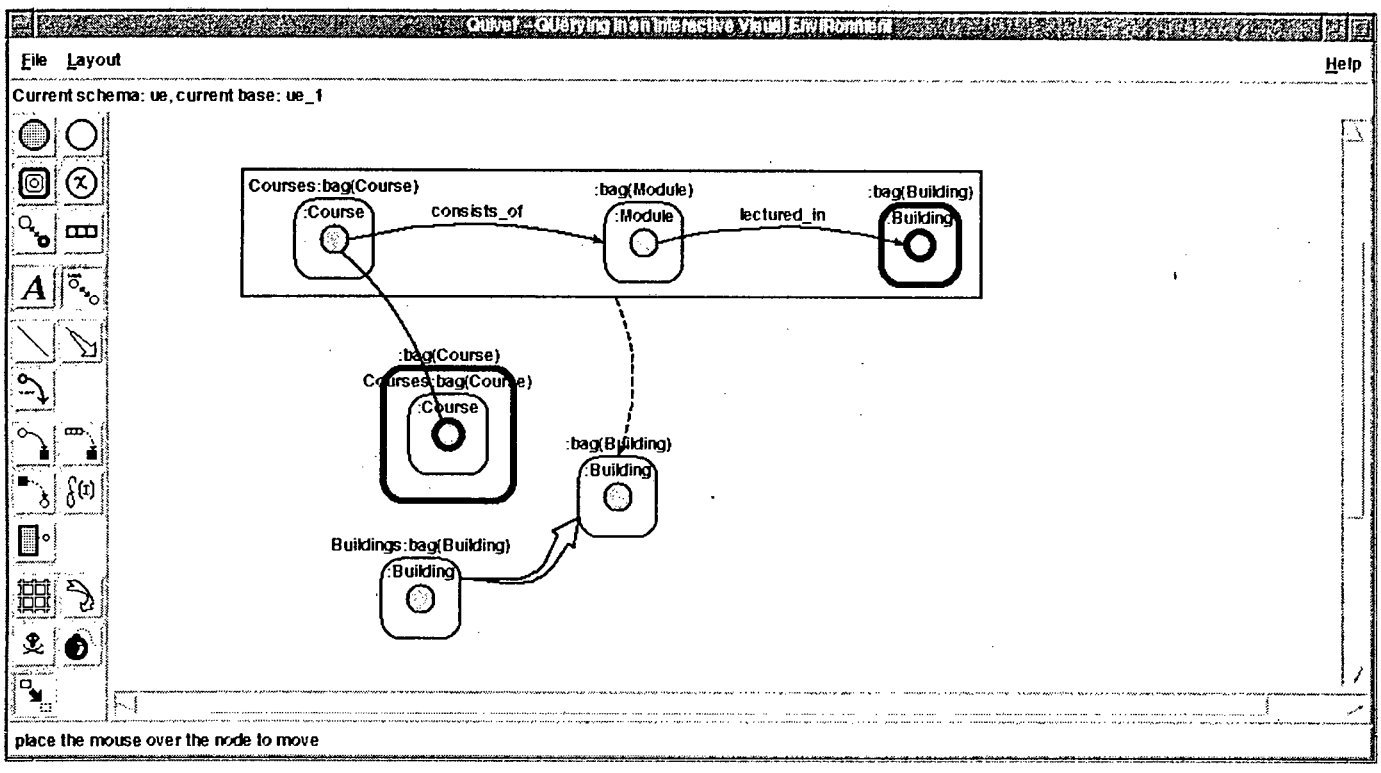


Figure 2.10: A query with a subquery

move inclusion groups of the graph. If an inclusion group is moved, the positions of the edges entering and leaving it are adjusted automatically.

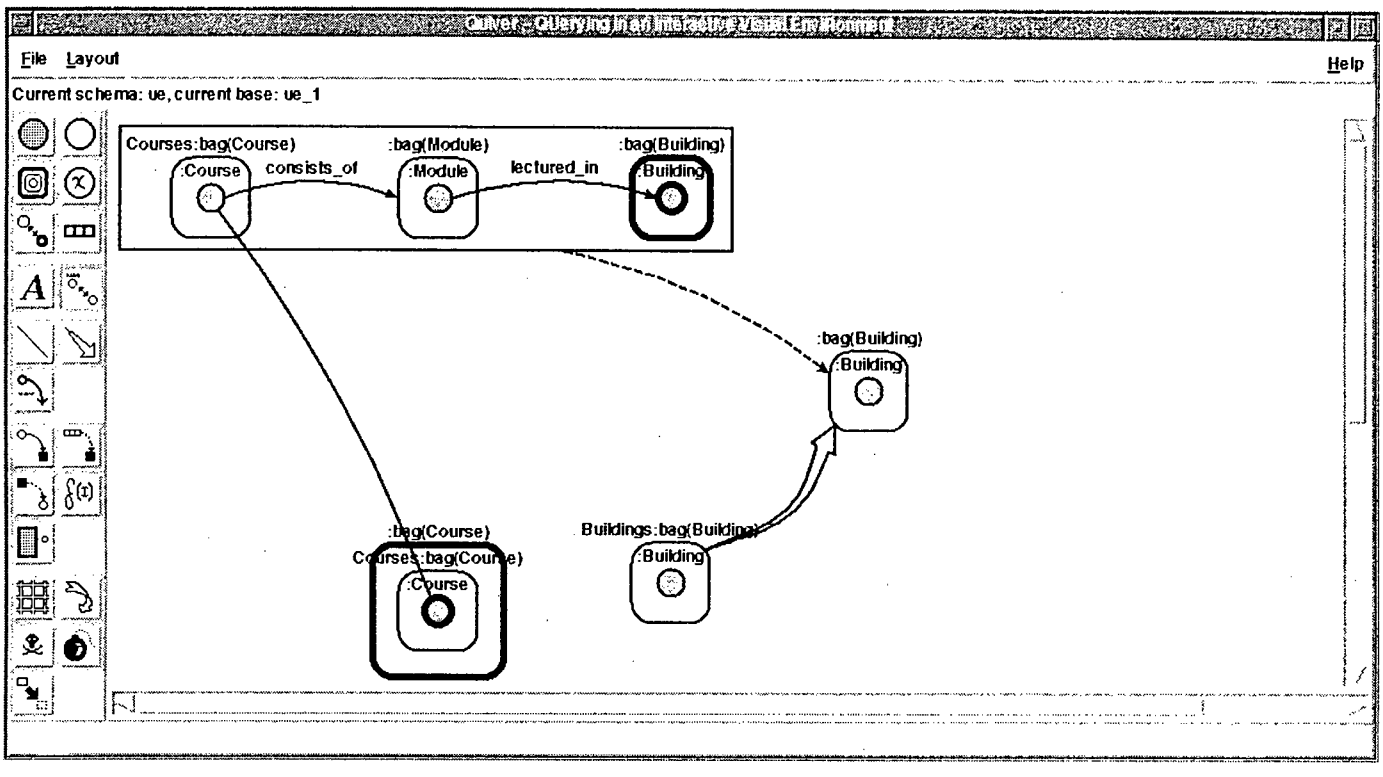


Figure 2.11: After calling the graph layout engine

Chapter 3

A Survey of Visual Query Systems

3.1 Introduction

In the last few years, many visual query systems (VQSs) have been proposed in the literature and they have been built adopting many representations and interaction strategies [8]. The goal of this chapter is to survey and classify some of these systems, exploring their features in solving typical problems of database accesses.

Distinction has to be drawn between languages handling visual objects that are visually presented (such as images) and languages whose constructs are visual. Chang [10] refers to the former as *visual information processing languages* and the latter as *visual programming languages*. Visual programming languages handle objects that do not necessarily have a visual representation. Within the class of visual programming languages, a subclass exists which deals with data in databases; this subclass is the class of *visual query languages* (VQLs). VQLs are of specific interest as they are the main components in VQSs. A VQS typically includes both a language to express the queries in a visual format and a variety of functionalities for facilitating human-computer interaction.

In Section 3.2 basic definitions are provided, while in Sections 3.3 to 3.5 a description of the basic components of VQSs is given. Section 3.6 documents a number of relevant VQSs, including QUIVER.

3.2 Basic Definitions and Classifiction Criteria

Visual representations use *visual formalisms* for communicating relevant concepts. In the object-oriented database context, these concepts refer to

- objects,
- classes of objects,
- relationships among objects,
- relationships among classes of objects, and
- operations on objects, classes and relationships

In the field of VQSs, four visual formalisms [8] may be distinguished:

forms: The form is the first attempt to leave the textual space, exploiting the bi-dimensionality of the screen. A form can be seen as a rectangular grid having components which may be any combination of cells and groups of cells (or a subform).

diagrams: Data contained in a form may be better understood if a graphical representation is used; graphical representations are better able to show the relationships among form data. A diagram uses visual elements that have a one-to-one association with specific concept types. For instance, rows of numbers can be transformed into a profile in which the height of each column is proportional to the value of the corresponding number.

icons: An *icon* can be defined as a segmented, stylized image [8]. The image *segmentation* implies the extraction of a single component from the background of the icon; an icon may be seen as an image with a foreground and a background component. An icon is generally small and simple in appearance; the word *stylization* refers to a representation made by a small number of significant lines that together constitute the icon.

In a visual interface we need to represent not only images of real objects, but also abstract concepts or actions or processes. Sets of icons denoting both the entities of the real world and the available functionalities of the system are used.

pictures: At first, it may appear as though the most natural type of visual representation is the picture, whose main characteristic is the high resemblance with the represented object; a representation resembles an object if it physically appears similar to the object. Resemblance, however, is not necessary for reference; almost anything may stand for almost anything else. A picture that represents an object refers to that object and denotes that object – denotation is the core of representation. If resemblance between a picture and an object also occurs, then a particular kind of representation is referred to, i.e. pictographic representation. In a visual interface, it is usually desired to represent *prototypical* objects and not specific instances of

objects and thus resemblance between a picture and the object represented by the picture seldom occurs.

Each of these is discussed in more detail in Section 3.3. In order to classify VQSs, three criteria are considered [8]:

- the *visual representations* adopted to present the schema of interest, the applicable language constructors and the query results.
- the *expressive power* to take into account what can be computed using the language.
- the *interaction strategies*, i.e. the strategies the system allows for constructing the query.

These three criteria are described in Sections 3.3 to 3.5 and are used in Section 3.6 to classify various VQSs.

3.3 Visual Representation Paradigms

In this section different representation paradigms adopted in VQSs are described. VQSs are classified according to the representation paradigms.

3.3.1 Form-based representations

Initially, forms were used in the framework of the relational model [11], where queries were expressed by filling table fields with suitable operands. *QBE* [50] was the first visual query system, and adopted a form-based representation. Only the intensional (the schema) part of relations is shown, while the extensional part (the data in the database) is filled by the user in order to provide an example of the requested result. In more recent systems, both the intensional and the extensional part of the database may be manipulated by the user.

EMBS [45] aims to overcome the limitation of forms in visualizing associations among data. EMBS displays cells and buttons; the cells contain data values and are organized in a table. Selecting a button or changing a cell value triggers a data manipulation; after any action the cells are refreshed with new values so that any change is immediately reflected in the screen image.

3.3.2 Diagrammatic representations

Diagrams have emerged as the most popular representation paradigm used in existing VQSs. Queries are typically expressed by navigating on the diagram. The most simple form of diagram is a set of nodes, connected by edges. In this way it is possible to represent a set of elements and some binary relationship on them. Several types of objects and relationships among objects need to be visualized – different kinds of visual elements (usually simple geometrical figures such as circles, squares and rectangles) are used. Most relationships are represented by means of connections. Special relationships such as structural relationships are represented via inclusions of geometrical figures, such as in *Harel hygraphs* [21].

Another popular abstraction is the entity-relationship diagram [43]. Rectangles represent entities and diamonds represent relationships. Single-lined edges attached to labels are used to define attributes and double-lined edges specify certain cardinality constraints; cardinality is also specified by attaching numbers to the edges of a relationship diamond.

A popular technique for modeling in the object-oriented paradigm is *OMT*, or *object modeling technique* [44]. Like entity-relationship diagrams, OMT is a diagram-oriented modeling technique in which a rectangle represents a class (along with its attributes), and edges between rectangles represent associations between classes. Associations between classes are subject to multiplicity constraints; a multiplicity limit of one is indicated differently to a multiplicity limit of many between two objects classes. OMT also provides for the indication of relationships between classes, as well as aggregation and generalization between classes.

3.3.3 Iconic representations

An *iconic* representation uses sets of icons denoting both the entities of the real world and the available functionalities of the system. A query is expressed primarily by combining icons. Generally, in these systems, the database schema is not shown; these VQSs are mainly addressed to casual users who are not familiar with the concepts of data models and who may find it difficult to interpret, for example, an entity-relationship diagram.

When designing iconic systems, a crucial problem is how to construct icons that can express an objective meaning to humans; often the same image conveys different meanings to different people. It would be desirable to have some criteria to construct icons which could carry an unambiguous meaning. At the present time, however, no standard on the visual design of icons and their meaning can be imposed on the sets of icons used in different applications [8].

3.3.4 Pictorial representations

The pictorial representation does not possess the abstraction power typical of the other representations. It is, therefore, rarely used where classes of objects need to be identified by a prototypical image. VQSs using pictures are mainly used for accessing a database of images. For example, in *QVE* [24], retrieval may be performed by the user sketching a rough drawing of a painting.

3.3.5 Hybrid representations

A *hybrid* representation uses a combination of the above paradigms. Many systems use more than one visual formalism, but often one of them is predominant. In this case, the system can be classified as adopting that specific paradigm. In hybrid VQSs, the different visual formalisms share the same significance. The user is either offered alternative representations of the database and queries, or the different formalisms are combined into a single representation.

In *SKI* [26], for example, diagrams are used to describe the database schema and forms are used to show more detailed information and for formulating queries.

3.4 Expressive Power

Although traditional query languages are normally compared with each other in terms of their expressive power, a similar analysis has rarely been performed for visual query languages.

A significant class of queries inside the Chandra hierarchy was due to Codd [12] – namely, the class of first order queries. Codd showed two equivalent ways of querying a relational database: by using first order logic and by using relational algebra. The term *completeness* is used to indicate that a query language can express all first order queries. Queries that are only able to express queries less than first order are still important, since many queries used in database systems are simple.

Much of the discussion in this section is from [8].

3.4.1 Enriched Chandra hierarchy

Figure 3.1 is the *enriched Chandra hierarchy* [8], which graphically relates various classes of queries and orders them according to their expressive power. At the top of the hierarchy is the class of *computable queries*. *Fixpoint* queries are relational algebra queries enriched with the fixpoint operator, thus allowing, for

example, the expression of transitive closures. The class of *Horn clause* queries [9] is based on the logical paradigm and is not comparable with relational algebra, since queries can be found in either class that cannot be expressed in the other class. An extension of the Horn clause queries are *stratified queries* [31], that can express all first order queries and are a proper subset of fixpoint queries. *Generalized transitive closure queries* are the class of queries obtained by extending the relational algebra with the generalized transitive closure operator. The generalized transitive closure operator is a transitive closure operation where cycle conditions are replaced by boolean conditions with equality and inequality operators [46]. They have proven to be equivalent to *stratified linear queries* [13].

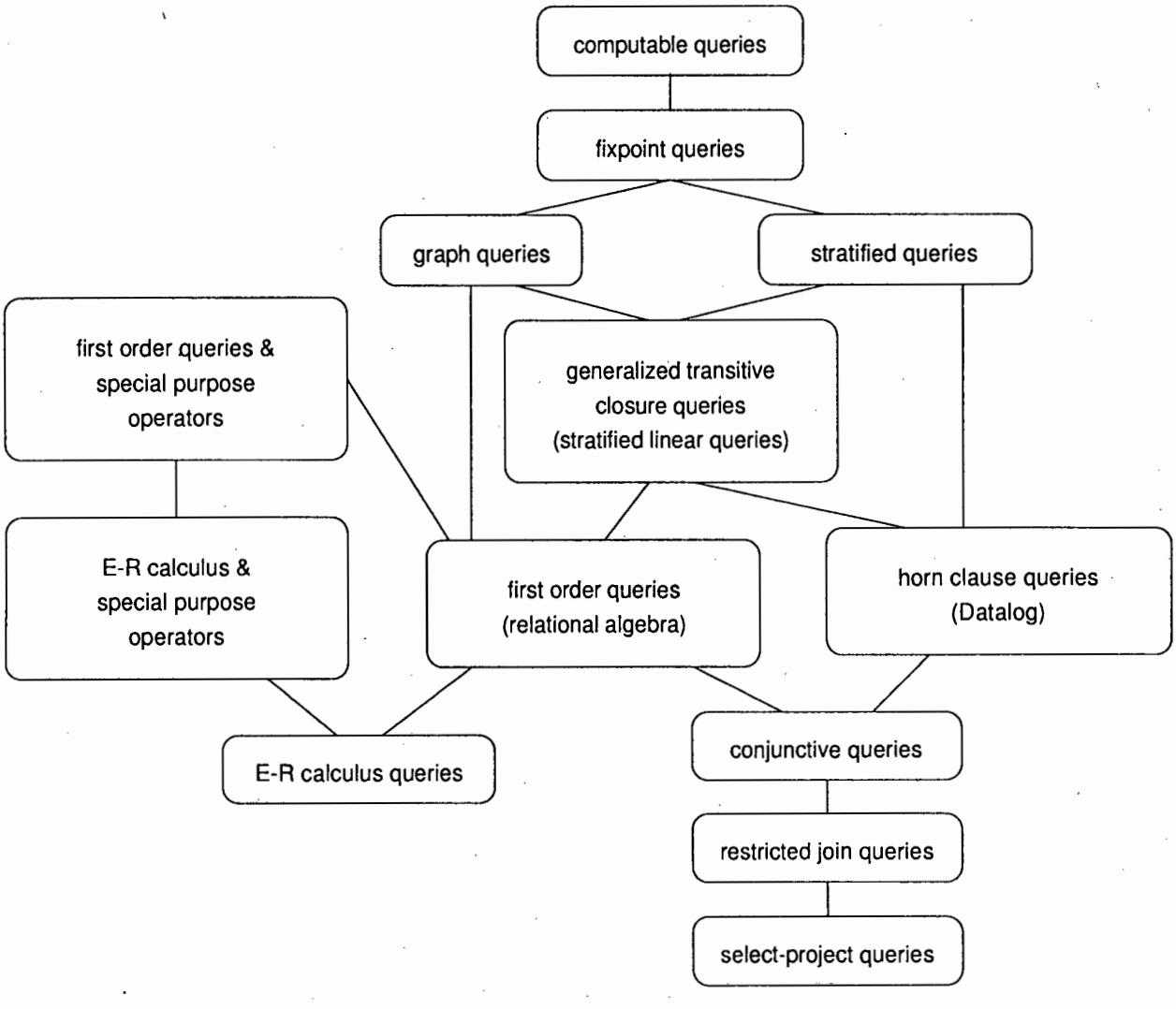


Figure 3.1: Enriched Chandra hierarchy

Graph queries correspond to the queries expressible by using the diagrammatic language *Graphlog* [13], and involve the computation of paths in directed graphs. *First order queries enriched with special purpose operators* are obtained by en-

riching relational algebra with operators that compute new database values by applying aggregation and counting functions. *Entity-relationship calculus queries* are obtained by using entity relationship (E-R) calculus [7].

E-R calculus queries enriched with special purpose operators are obtained by enriching E-R calculus operators with aggregation and counting functions, just as *first order queries enriched with special purpose operators* was obtained by enriching *relational algebra*. *Restricted join queries* are obtained by using the *select*, *project* and *join* operators.

With the exception of the graph language *Graphlog* most of the visual query languages developed so far are placed at the lower levels in the Chandra hierarchy. Iconic languages in particular fall into the lower level classes since they are directed to a casual user who is mainly interested in simple queries.

3.5 Interaction Strategies

The goal of interacting with the VQS is to formulate a query, and query formulation is composed of three steps [8]:

selection: the goal of this phase is the *query subschema*, or the fragment of the schema involved in the query. Generally, the database schema is much richer than the subset needed to formulate the query.

query formulation: The goal of query formulation is to express formally the available query operands involved in the query, with their related operators.

testing: In this phase the user may test the constructed query to make sure that it matches what was intended.

3.5.1 Strategies for selecting the query subschema

These strategies may be divided into three main groups:

1. *top-down:* General aspects of the schema are perceived first, and then specific details are introduced. This may be accomplished by iteratively refining the schema; each refinement is obtained from the previous one by means of primitive transformations on schema objects. A second approach is to selectively view the schema, such as only viewing objects above a certain importance level.

2. *browsing:* Browsing is a viewing technique aimed at gaining knowledge about the database. The main hypothesis is that the user has little knowledge about the database and the interaction techniques, and that the user has accessed the query system without having any predefined goal. The user starts the interaction by examining a concept and its adjacent concepts. A new element is selected by the user from adjacent concepts, and the process proceeds iteratively. An incremental enhancement of the user's knowledge, therefore, is obtained by exploring adjacent concepts. Browsing may be specialized into the following cases:

   (a) *intensional browsing:* This approach corresponds to querying the schema structure, such as asking for all existing paths connecting two concepts.

   (b) *extensional browsing:* This browsing is performed on the extension of the database. An example is to show the data as vertices of a graph where edges represent links between occurrences of data.

   Mixed browsing may also occur, in which the user browses on both the intension and the extension of the database.

3. *schema simplification:* A schema is simplified by building a user view, which is derived from aggregations and transformations of concepts of the original schema. In this approach the user may build a view of the schema which does not correspond to any detail level of the original schema.

3.5.2 Strategies for query formulation

Queries may be formulated in three ways:

1. *schema navigation:* This query strategy concentrates on a concept (or a group of concepts) and moving from it in order to reach other concepts of interest. One approach of schema navigation is to explore *arbitrary connected paths*. The user first navigates along an arbitrary path in the database schema, in order to select the concepts involved in the query. An example is OQL [3], an object-oriented query language. In OQL, a query is expressed by first browsing a diagram of object classes, and then selecting the classes of interest.

2. *subqueries:* A query is constructed by composing partial results, such as using query libraries – queries can be composed by using subqueries stored in a system library. Another approach is to compose the concepts of a query; this approach is followed in several iconic languages. Each operation, such as selecting an icon or overlapping one icon over another, is interpreted as a particular chain of operations on the database.

3. *matching:* This last strategy is based on the idea of the user presenting the structure of a possible answer; the structure is then matched against the database data. Two types of matching may be distinguished:

   (a) *by example:* The user provides an example of the answer and the system identifies the goal by generalizing the example.

   (b) *pattern matching:* In this approach the user provides a pattern (such as a regular expression) and the system looks for all fragments of the database matching the pattern.

3.5.3 Strategies for testing

The purpose of testing is to allow the user to verify that the query corresponds to what was intended. Testing strategies may be subdivided into two main categories: the first is based on rephrasing the query in some well known language and presenting the new version to the user for validation. The target language of the query rephrasing approach can be either a natural language or some standard query language, such as OQL. The second category consists of visualizing the result of the query; this concerns visualizing the structure of the query, the result of the query or both.

3.6 Visual Query Systems

In this section, various visual query systems are documented. Where possible, they have been documented according to their visual representations, their expressive powers and their interaction strategies as described above. Emphasis has been given to VQSs that have a strong visual component or that operate on object-oriented databases, or both.

Of the seven systems discussed in detail, three have quantified their expressive powers and three of the systems provide some form of query testing.

3.6.1 VQL (Vadaparty *et al*)

VQL [47] is a visual query language for object-oriented data models and has been implemented on the *ObjectStore* object-oriented database. VQL has a *form-based* representation and the input to the query system is a set of graphical tables within the forms, such as tables for accessing and constructing structures and collections, for producing different types of queries and for specifying various operators on queries. A single query, therefore, may be constructed over many tables.

In terms of querying ability, VQL supports recursion, nested output, schema querying, disjunctions, group by queries, min, max, count, some quantification queries, set operations, negation and parameterized queries. Most of these features are simulated through the use of other features; the authors thus claim to have a minimum of *ad hoc* operators and reserved words in the graphical query language. Group-by and quantification queries, for example, are simulated through the use of a *restricted universal quantifier.*[1]

Figure 3.2 is the visual representation of the restricted universal quantifier. If the *Depts* relation contains all the departments, the frame specifies that for every department d_j, $1 \leq j \leq k$, which appears in *Depts*, it is the case that *Sell*(d_j,*i*) holds. The construct thus finds the items that are sold by all departments.

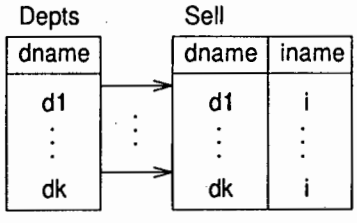


Figure 3.2: Visual representation of the restricted universal quantifier

VQL provides no graphic means of selecting the query subschema. The subschema is identified by the user selecting class names from a list; a primitive form of *intensional* browsing is thus provided. Once the subschema has been chosen, appropriate tables with the skeletons of the subschema are generated. Queries are specified by filling in tokens in the skeletons; the query system matches these tokens to each other and to any constraints specified and then formulates the query. Query formulation is thus done in a *by-example* matching strategy.

Figure 3.3 represents the query "*Get the models of all the blue cars that are driven by the president of the company that manufactures them*". The query consists of 4 tables; one to encode that c is a vehicle with colour "blue" manufactured by some manufacturer m; a second to encode that p is the president of m; the third table encodes that the same president, p, as an employee, owns a set of vehicles CA and the last table states that CA includes c.

VQL does not provide any means for the user to test the query generated.

3.6.2 VQL (Mohan *et al*)

Mohan and Kashyap [30] have proposed a query language also based on an object-oriented data model – and also called VQL.

[1] A restricted universal literal is a formula of the form $\frac{\forall \vec{y}}{P(u,\vec{x})}(l(\vec{y}))$ where P is a predicate and $l(\vec{y})$ is a positive or comparison literal and $\vec{x}$ and $\vec{y}$ are vectors of variables and constants.

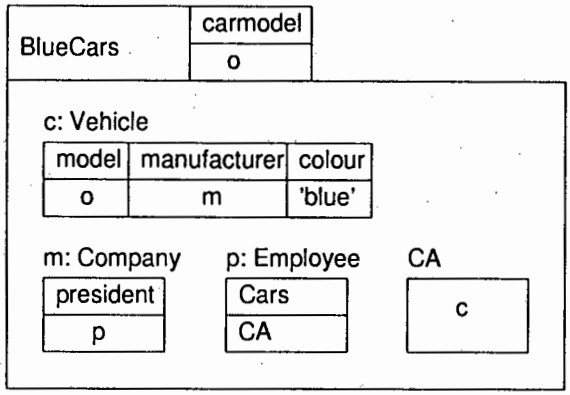


Figure 3.3: Representing a query in VQL

The system displays the intension of the database by displaying the classes in the database and the links between the classes. Classes are represented by labeled nodes and the links are represented by labeled edges between the nodes. The system thus uses a *diagrammatic* visual representation to display the database structure.

Formally, VQL is able to express the *generalized transitive closure queries* class of queries (Figure 3.1). The system is also able to express certain types of intensional queries, such as finding out which classes have interactions with a certain class or querying on the attributes of a class.

The query language is built from the three basic graphical primitives shown in Figure 3.4. The top portion of Figure 3.4(a) (or the "class icon" primitive) is for specifying the name of the class while the shaded portion is used for referring to any particular instance of the class. Figure 3.4(b) is the "attribute icon" primitive. Similar to (a), the top part specifies a name and the shaded portion specifies a particular instance – in this case an attribute name and an attribute value. Figure 3.4(c) is a "link icon".

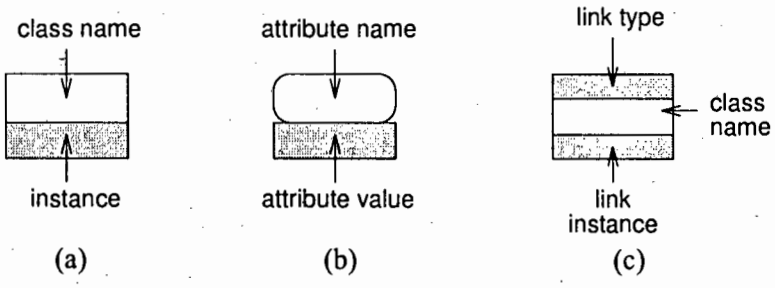


Figure 3.4: Visual components used in VQL

VQL represents the relationships between classes using four types of links; interaction links (indicated by **I**) – which are the most general type of link, liaison

links – which capture information about a physical connection between two objects (indicated by **L**), generalization links (**G**) which convey hierarchical information, and aggregation links (**A**) which indicate that one class is comprised of other classes.

The bottom shaded portion in Figure 3.4(c) is used for specifying an instance of a link, the middle portion specifies a link class name and the top shaded portion indicates the type of link (either **I**, **L**, **G** or **A**).

Intensional queries are used in order to select the query subschema of interest – the *intensional browsing* strategy is thus used. Graphic primitives may be duplicated, thus allowing reflexive queries to be constructed. VQL uses a number of reserved symbols within queries: "?" is used to denote the output of a query, "$\sim$" indicates set level negation and "$\neq$" indicates attribute-level negation. The depth of recursive queries may be limited by the user; to yield the entire transitive closure of a query, the user may use the symbol `M` instead of a depth limit.

By using the graphic primitives and the symbols mentioned above, the user specifies the *procedural* characteristics of the query. VQL thus uses the *schema navigation* strategy for query formulation.

The query in Figure 3.5 is a VQL query that finds those department names that sell all the items that are manufactured by the manufacturer "Ping". VQL does not provide any means for the user to test the query generated.

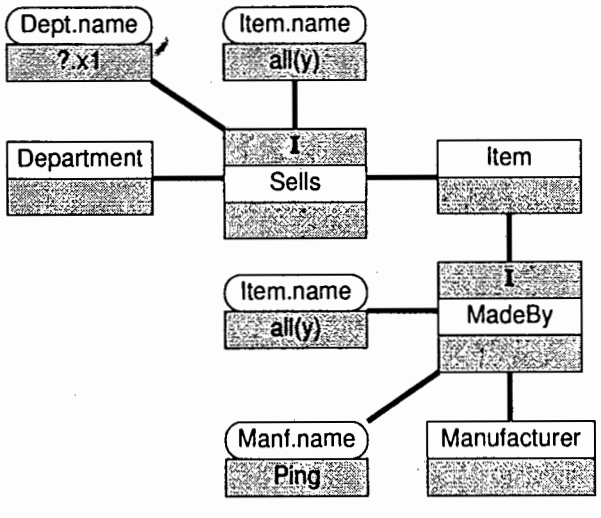


Figure 3.5: Finding manufacturers in VQL

3.6.3 GOOD

GOOD [18, 39] is an acronym standing for *Graph-Oriented Object Database*. GOOD is a graphical language which consists of operations that are sequenced to

build programs. All database operations are accessible through the construction of these programs. Because a GOOD database is stored in a relational database management system, the execution of GOOD programs requires that all operations and queries be translated into a relational database manipulation language.

Both the instance representation and the schema representation of a GOOD database are represented as directed, labeled graphs. The instance graph is composed of four kinds of icons – *ovals*, *rectangles* and *single-* and *double-headed* arrows. The ovals represent atomic values (such as strings and numbers) while rectangles represent objects. Relationships involving objects and atomic values are represented by edges; an edge with a single arrow head represents a functional relationship (such as the relationship between an object and another object), while an edge with a double arrow head represents a multi-valued relationship (such as the relationship between an object and a set of objects).

In the schema representation, the same four icons are used. Rectangles represent classes, ovals represent atomic types and edges once again represent relationships. GOOD, thus, uses a *diagrammatic* visual representation to display the schema and instance structure.

GOOD is able to express all computational queries [20].

GOOD provides two tools – *composition* and *decomposition* – that allow the user to alter the view of the schema in order to generate a query subschema. When composing two graphs, a node of the first also appearing in the second is deleted in the second and all its incoming and outgoing edges are connected to the node in the first. Figure 3.6 is an example of graph composition – composing graphs (a) and (b) produces graph (c). Conversely, decomposing graph (c) *can* produce the two graphs (a) and (b); the decomposition of a graph does not produce a unique set of graphs. The query subschema is thus selected through the *browsing* and *schema simplification* techniques.

The specification of all GOOD operators (in terms of data querying) relies on the notion of a *pattern*. A pattern is a graph used to describe the subgraphs to which an operation must be applied. Figure 3.7 is an example of a query which finds all data instances where a *person* has a *child* who, in turn, has another *child*. In order to view the result of the query, the user specifies the nodes to display and the result may either viewed as a graph or as text in the form of a relational table.

GOOD also allows the user to specify the manipulation of data graphically. Data may be manipulated using four basic operations: addition and deletion of nodes (corresponding to object addition and deletion) and addition and deletion of edges (corresponding to relationship addition and deletion). Elements that are being added to the database are indicated in bold; elements that are being deleted are drawn in outline. An example of data manipulation is given in Figure 3.8, where

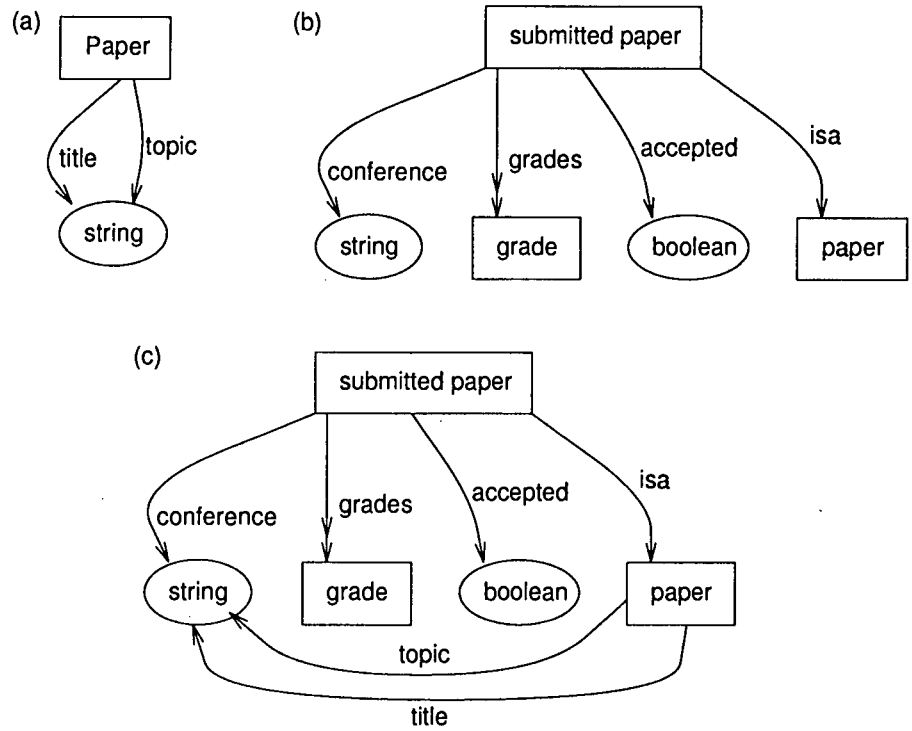


Figure 3.6: Composition of graphs

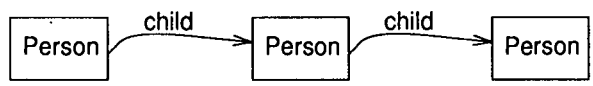


Figure 3.7: An example of a pattern

a *grandparent* relationship is added between the appropriate objects found using the query of Figure 3.7.

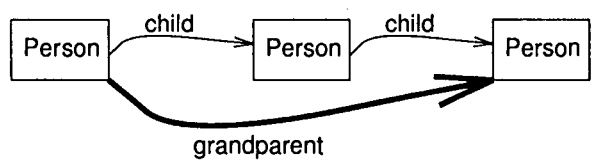


Figure 3.8: Data manipulation

Another example is in Figure 3.9, which deletes all objects of type *tag* that have a *tagged* relationship with author *Perot*; the tag object has been outlined.

GOOD also allows the user to define and call procedures, allowing the reusability of programs.

GOOD does not provide any means for the user to test the query generated.

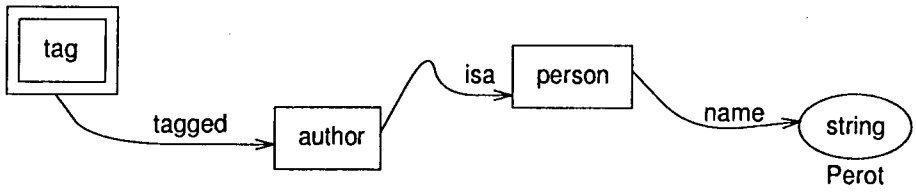


Figure 3.9: Object deletion

3.6.4 OdeView

OdeView [2, 16] is a graphical user interface for the Ode object-oriented database system [1]. It is implemented using O++, the database programming language of Ode; O++ is a subset of C++.

OdeView supports object instance browsing, and uses windows to display objects. Each window displays objects of one particular class only – however, many windows may be opened at a time in order to simultaneously view objects of many classes. Objects in a window are displayed individually; only objects within the *scanning range* of the window are displayed. Each attribute of an object is displayed within a frame and these frames are stacked vertically. A *selection* predicate (explained in greater detail below) may be used to restrict the number of objects that are within the scanning range of a window. The user may control the amount of information displayed in a window by *projecting* (also expanded on later) which attributes to display. OdeView supports synchronized browsing of objects across classes by maintaining a *display forest.*

If an object displayed in window W_1 refers to an object or a set of objects displayed in window W_2, then we say that W_2 *depends* on W_1; the set of windows and relationships between the windows form the display forest. Whenever the current object in window W_i changes, then all windows that depend on W_i are changed accordingly. This recursive propagation is called synchronized browsing.

OdeView uses a *form-based* representation in order to display the extension of the database. The system allows the user to view the intension of the database, displaying hierarchy information as a directed acyclic graph. If a class node is clicked on, it is expanded to display that class' structural information as well as its superclass and subclass names.

Constructing a query is broken down into selecting the *select* and the *project* predicates. Where possible, OdeView pops up menus with elements that may be selected by the user in order to select an operator or an operand. OdeView is, for example, able to pop up menus containing object member names, object attribute names and mathematical comparison operators. The user may either choose an item from the menu, or type text directly into the query boxes. Construction is done using the *schema navigation* strategy.

When defining a selection predicate, the user fills in operator and operand boxes which lie within a single *specification level.* Odeview assists the user in specifying arguments for a function by creating a temporary specification level. After the arguments for the function are entered, the argument specification box is removed and the arguments appear with the function label. Figure 3.10 shows the creation of a temporary specification box (containing the values 50 and 60) for the arguments of `age_between`.

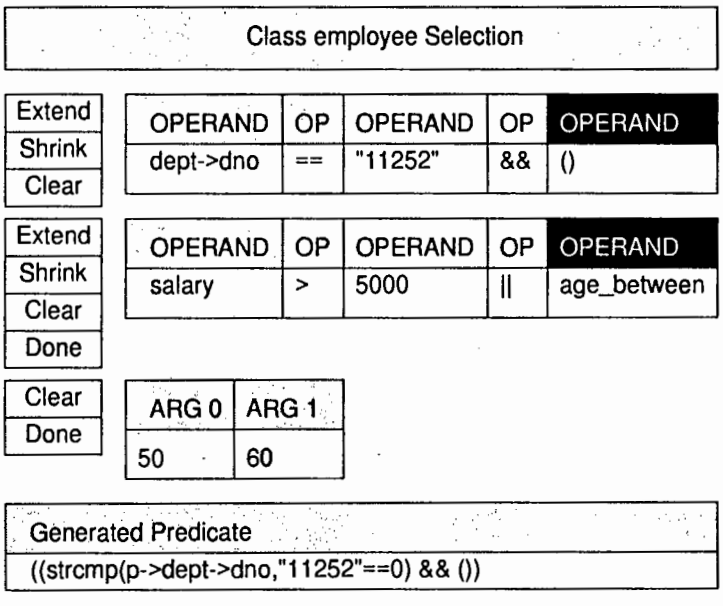


Figure 3.10: A selection window with multiple levels of specification

Each element of a specification is evaluated from left to right. The order of evaluation of the entire select predicate, though, may be controlled by nesting the specification levels. Nesting is indicated by setting an operand to the () symbols – if specification levels are nested, they are nested top to bottom.

When an operand is to be nested, a new level of specification is opened below the current one, such as in Figure 3.10. In the figure, the "`salary > 5000 || age_between(50,60)`" clause is logically nested within the () symbols. The *generated predicate* window at the bottom of the screen shows a textual representation of the current specification level. The query retrieves all employees in department 11252 who either earn more than 5000 dollars, or whose ages are between 50 and 60.

As mentioned, the user may control the amount of information displayed in an active window through the *project* button. Clicking this button brings up a specification window that allows the user to turn on or off the display of individual attributes. Each attribute is represented by a button and the user may set the state of this button to either *on* or *off.*

An object or a set of objects can be named; named objects or sets may be used in more complex queries. OdeView also allows the user to create, delete and update database objects.

In order to reference a particular object, that object may be *imported* into the select predicate by identifying it by name or by clicking on it if it is already displayed in the display forest. It or any of its attributes values may then be used in comparisons, just like any other constant value.

OdeView allows the user to perform queries on any class or object displayed on screen. It thus uses the *mixed browsing* technique to allow the user to select a query subschema – as both the intension and the extension of the database can be browsed.

As mentioned earlier, OdeView displays, in text, the current specification level during query formulation. This is a limited form of *query testing*. To an extent, therefore, OdeView adopts the *rephrasing* strategy for query testing.

In order to support synchronized browsing, projection and selection of objects, OdeView requires that the user maintains a meta-data store. For each persistent class for which these operations are required, an object of type `class_decs` has to exist. Type information that has to be stored includes class name, superclasses, member functions and attribute names. Other information that has to be stored includes button names for the projection window and a function `display` which defines how to present the attributes of that object visually. An advantage of defining `display` is that attributes containing images or other complex data can be suitably displayed.

3.6.5 GraphLog

GraphLog is a query language which has been developed as part of the Hy^+ system [14]. The system is graph-based; data, queries and answers are all represented as graphs. Hy^+ interacts with a deductive database for query evaluation.

GraphLog has higher expressive power than SQL. It can, in particular, express queries that involve computing transitive closures. The language is also capable of expressing first order aggregate queries as well as aggregation along path traversals, such as shortest path queries. As mentioned, the system's expressive power may be formally characterized as the class of *graph queries* (Figure 3.1).

The query in Figure 3.11 is an example of a filter query in Hy^+. The `mem` edge in the pattern is thick, and is a visual way of distinguishing edges that the user wants to see after the match is found. The query finds subclasses of class `Object` that redefine some function `F` defined in `Object`. The subclasses identified are either direct subclasses or indirect subclasses, as they are found using a transitive closure

operation. The relation mem between between the subclass C and the inherited function F is displayed to the user.

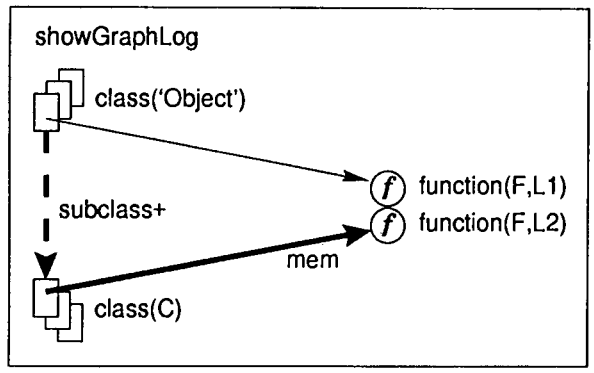


Figure 3.11: A filter query in Hy$^+$

In order to aid the visualization of instance level data, the system supports the technique of providing overviews at different levels of detail. One method is to display nodes as dots and to suppress the display of edges. The data representation may thus be simplified in this manner. Once displayed on screen, data may also be browsed. The use of filtering by using queries is, however, the most powerful tool in the management of large data visualizations. Complexity may also be managed through the use of *blobs*. Blobs are boxes that represent containment; blobs may be shown with their contents invisible – thus simplifying the display.

Queries are constructed on the graph representing the instance of the database. In order to construct a query, nodes of the graph are labeled by sequences of variables and constants and graph edges are labeled by regular expressions. Edges labeled by a regular expression containing a closure operator (+ or *) correspond to paths of arbitrary length in the database graph. The query evaluation process consists of finding in the database all instances of the given pattern and for each such instance performing some action, such as defining a new arc in the database. By placing a *distinguished* edge between two nodes, the user can select which nodes are to appear in the answer graph.

In the answer graph, variables on the nodes are substituted by constants. These constants correspond to the existence of a subgraph of the database graph that matches the pattern specified by the query graph.

Query testing is provided implicitly by the visualization of the answer graph. The user, by inspecting the number of query answers and visually inspecting the query answers is able to tell whether the query interpreted by the system matched the intended query.

3.6.6 QBD*

Query by Diagram (QBD*) [4] is a query language based on an entity-relationship data model. The system adopts a combination of two different graphic conventions for displaying the intensional part of the database. Querying is performed on an entity-relationship schema with attribute conditions specified on a form-like representation of entities; the display is not strictly form-based as it is modified by labeled edges.

When displaying the query subschema, entities are represented by rectangles and relationships by labeled edges between the rectangles. During querying, entity attributes are displayed as lists and relationships between these attributes are indicated by edges. QBD* thus uses a *diagrammatic representation* strategy for both browsing and querying.

The system is able to express all first-order queries, and a "reasonable" (in terms of practical use) class of recursive queries. In particular, queries involving the generalized transitive closure operator may be expressed. The query language is a visual navigation language on E-R schemata; a one-to-one correspondence between the graphical operations and the syntactic constructs of a textual query language has been defined. Subqueries generated in QBD* may be combined by means of union and intersection operators.

QBD* provides three means to allow the user to extract the query subschema. The first is by *direct extraction*, where the user "picks up" the interesting items. The second is by expressing queries on the database schema, and including items that satisfy the meta-query expression. The third means is by using the system-managed *library of schemata*; schemata (or selected parts) can be added to the current schema of interest using a "cut-and-paste" strategy.

The general structure of a query is based on the location of a *main concept* (an entity or a relationship) that can be seen as the entry point of one or more subqueries. These subqueries express possible navigations (or queries) from the main concept to other concepts in the schema.

Conditions on attributes are set by choosing the attribute names from a list and then choosing an appropriate operator. The system shows the results of the operation in the form-like interface as mentioned; labeled edges between attribute names correspond to operators between the attributes.

Figure 3.12 shows an example of the graphical solution proposed in order to compute a transitive closure. The schema contains information about flights and we wish to compute finite sequences of flights such that in each sequence:

1. the destination of each flight (except the last) is the source of the next, and

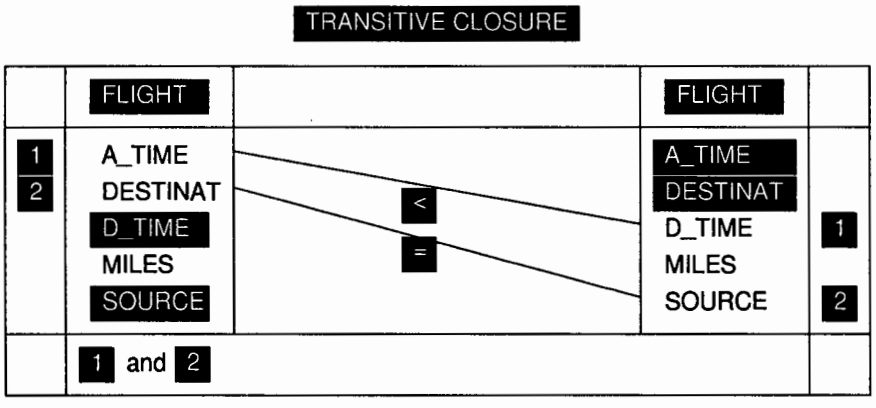


Figure 3.12: A query with transitive closure

2. the arrival time of each flight (except the last) occurs before the departure time of the next.

A double list of the same set of attributes is displayed and the user may specify the conditions comparing the attributes to each other. The conditions `DESTINAT` = `SOURCE` *and* `A_TIME` < `D_TIME` are set. The attributes appearing in the answer are `D_TIME` and `SOURCE` of the first `FLIGHT`, and `A_TIME` and `DESTINAT` of the last `FLIGHT`.

QBD provides a natural language echo for the user to test the query generated.

3.6.7 Other systems

In this section, we survey five other systems briefly, namely DOODLE, Fox, MoodView, Hyperlog and VDM.

DOODLE [15] is the first language to be proposed for querying object-oriented databases with *user-defined visual languages*; the system embodies concepts for declarative object-oriented databases. A DOODLE program defines a visual language in a *by-example* fashion. The user defines the visual language and relates the visual terms constructed (called *U-terms*) to *F-terms*, which are textual terms from *F-logic* [25]. Rules are generated from these visual terms and querying may be performed on the visual terms.

Consider the graph visualization of the components of a program. The U-terms on the left-hand side of Figure 3.13 are used to define the visual language. The program specifies that for database facts that make the F-terms true, graphical objects similar to the U-terms will be drawn on the screen. The first rule states that any object `M` in class *module* is to be displayed by a box. The second rule states that any object `P` in class *procedure* is to be displayed by a diamond.

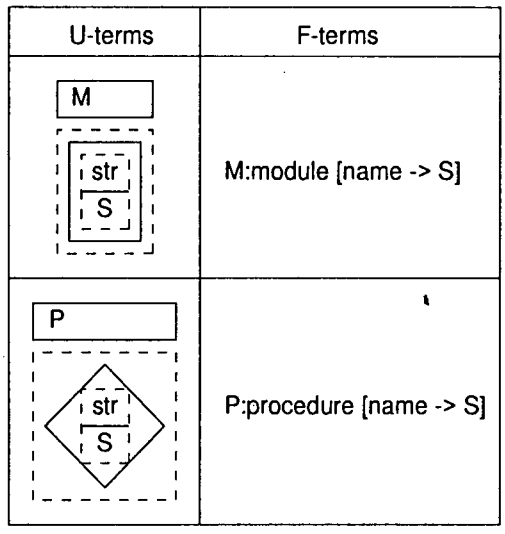


Figure 3.13: A DOODLE visual program

In order to construct a query, a mapping is specified between database objects and graphical objects. The query of Figure 3.14 asks for the nodes with diamond shape (procedure) that have a label (procedure name) "draw" to be displayed. DOODLE also supports meta-queries. Formally, DOODLE is able to express the *generalized transitive closure queries* class of queries (Figure 3.1).

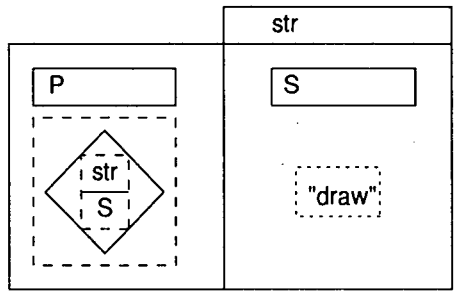


Figure 3.14: A query in DOODLE

Fox [48] is the declarative query language for Moose [23], an object-oriented data model supporting classes and collections (i.e. *sets*, *bags* and *arrays*) of classes. The schema of the database is represented as a graph; each class is a node and each relationship is a labeled edge in the graph. The schema editor contains features to make the visualization of large schemas more manageable – the schema editor allows parts of the schema to be made invisible, it allows the collapsing of subgraphs into single nodes and allows the use of special "reference" nodes to eliminate very long arcs. Reference nodes always appear in pairs, and clicking on the one element of the pair places the user at the other element in the pair.

Fox claims advantages over several other declarative query languages; it is able to

use complex path expressions to navigate any path in the schema graph in any direction, including paths through inheritance relationships. It also has the ability to associatively retrieve individual elements of an indexed-set given the corresponding elements of its keyset. There is, as yet, no graphical implementation of the Fox query language, and hence no graphical query language.

MoodView [5] is another graphical front end to an object-oriented database. It allows the definition of data in C++, SQL or the MoodView graphical environment. MoodView contains an SQL-like object-oriented query language called MOODSQL and all defined data can be transformed to C++ code or SQL statements; this enhances the system's compatibility with other systems. The MoodView query manager provides a query editor with facilities for accessing previous queries and for displaying objects graphically. The functional emphasis of the system, however, is not on querying the database as it does not contain a graphical query language.

Through the use of forms and graphs, MoodView displays objects, object method templates and inheritance diagrams, and the user is able to design object-oriented data types.

Hyperlog [28] encompasses a visual query system for a data model based on nested graphs (called *hypernodes*); schema information, data and query output are all represented as sets of hypernodes. Hyperlog has been implemented using the functional database programming language PFL [42]. Hypernodes have textual labels as nodes, and a hypernode can contain its own label as well as the labels of other hypernodes. Schema and data instances are represented by hypernodes. In a schema hypernode, identfiers represent types and constants, and edges represent relationships; edges do not cross hypernodes. When displaying data, types are replaced by instance values. Hyperlog claims, as one of its features, to have a very small number of query constructs.

Templates are used to construct Hyperlog queries. Templates may contain variables as nodes and may have their edges negated. Variables are preceded by a "!" symbol. Variables whose instances are to form the output are preceded by a "?" instead. A *program* in Hyperlog is a set of rules, and programs can be used to derive persistent information (such as when updating a database) or to derive transient information within a query. A program can consist of many hypernodes. The evaluation of a program consists of repeatedly matching the bodies of its rules against the current database state and updating this state with the information inferred until no more new information is inferred. In terms of expressive power, Hyperlog is computationally complete [41].

Orman has designed a Visual Data Model (VDM) and a Visual Data Language (VDL) for functional databases [37]. The VDM uses a representation consisting of geometric shapes; the major components of the model are entity sets, which are represented by rectangles, and labeled edges which represent functions defined

on entity sets. VDL is used in conjunction with VDM. The major component of VDL is a labeled arc connecting two points, where the points correspond to data items and the arc corresponds to a function relating the two data items.

Figure 3.15 models a university database. The STUDENT function returns the students in a course; the COURSE function is its inverse returning the courses taken by a student. The function INSTRUCTOR is defined on the data sets STUDENT and COURSE and returns the instructor of each student in each course. The other functions have obvious interpretations.

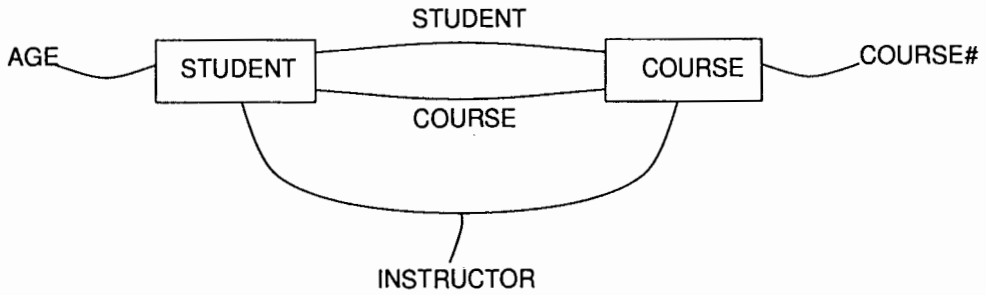


Figure 3.15: A university database

VDL expresses universal and existential quantification diagrammatically and each query is expressed by a dangling arrow from existing data. Figure 3.16, for example, finds the names of the students who take at least one course from each instructor. The connected double points " ▪ ", as in the INSTRUCTOR entity set, represent all objects in a data set.

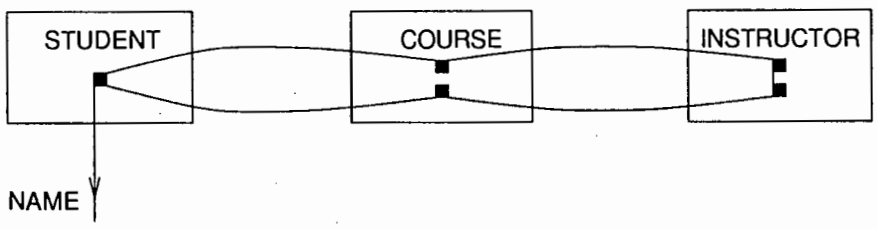


Figure 3.16: A VDL query

Database transactions are characterized by insert (" ▙ ") or or delete (" ▛ ") operations, such as in Figure 3.17, which modifies the department of SMITH from CS to MIS.

The model also defines the inclusion of arithmetic functions such as ADD and SUBTRACT and comparison operators such as "$\leq$" and "$\geq$" to expand the query language's expressive power. Formally, VDL's expressive power is equivalent to the *Horn Clause* form of first order logic (Figure 3.1).

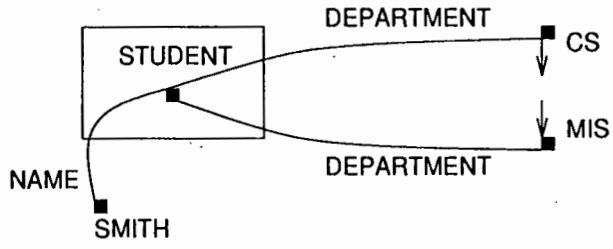


Figure 3.17: A database transaction

3.6.8 QUIVER

As discussed in Chapter 2, QUIVER is a querying application for object-oriented databases and is implemented using the O_2 database. The intensional part of the database is represented using a *diagrammatic* visual representation; the query subschema is displayed using a graph and the query is constructed on this graph. The visual query language consists of ten types of nodes and eight types of edges. Object types, literal types, structure types, methods, functions, subqueries, query output and collections of objects, literals and structures are represented by nodes. Properties, as well as various data constraints and the flow of data are represented by directed and undirected edges.

The query subschema is selected using *intensional browsing*. The user chooses from menus containing intensional information, reducing the intensional details the user has to remember. Queries are formulated using *pattern matching*. The user places restrictions on the data on the intensional graph and QUIVER either returns a boolean value indicating if the pattern was found or returns all the data instances that satisfy the pattern. In the latter case, the data may be browsed and explored using a *form-based* representation. Queries may also be formulated using queries stored in a library; any query constructed may be added to the library.

By default, an object instance in the answer of a query appears using all its property values. If any of these values are objects, that property may be explored further. O_2 allows the user to define methods in order to refine the appearance of object instances, similar to the O++ database used by OdeView (Section 3.6.4). Bitmaps and icons may be included in the presentation of object instances.

QUIVER provides a limited form of testing by displaying the OQL generated. It does not permit the updating of information; it exists solely as a querying tool. As yet, the expressive power of QUIVER has not been determined.

Chapter 4

ODMG and O_2

4.1 Introduction

This chapter introduces the ODMG and O_2. The ODMG data model is discussed in Section 4.2 and the modeling primitives and relationships between primitives are introduced. O_2 was the first ODMG-compliant database, but differs to the ODMG specification in some ways – the O_2 database system and the differences between the O_2 implementation and ODMG are discussed in Section 4.3. O_2's OQL implementation is equivalent to the ODMG OQL specification, and OQL is discussed in Section 4.4.

The Object Database Management Group (ODMG) is a consortium of object-oriented database management system (OODBMS) vendors and other interested parties. The ODMG is working on standards that allow the portability of software across OODBMS products. The objectives of the ODMG are to develop a standard set of specifications, and to make the definitions developed available to others for general use. The ODMG produced its first draft of the standard (called ODMG-93) in 1993 [6]; ODMG-93 is currently in Release 1.2. The next major update to the specification will be Release 2.0, with publication expected sometime in 1997.

The "standard" consists of four components. They are the object definition language (ODL), an object query language (OQL), and bindings for the programming languages `C++` and `Smalltalk`. The ODL is a specification language, defining the interfaces to object types. ODL is not a programming language; it is merely an interface that supports data definition. OQL is the query language, and is an object-oriented database equivalent of the SQL (structured query language) of relational databases. The language bindings define the communication protocols between a computer program and the object-oriented database. The protocols define how the ODL is expressed in the programming language, and how objects are retrieved and modified.

4.2 The ODMG Data Model

The basic modeling primitive in the ODMG data model is the *object*. Objects have state and behaviour, and objects can be categorized into *types*. All objects of a certain type have common behaviour and a common range of states. The state and behaviour of an object are collectively referred to as the *characteristics* of that object. The characteristic hierarchy is displayed in Figure 4.1. The *instantiable* types are shown upright and the *abstract* types are shown *italicised.*

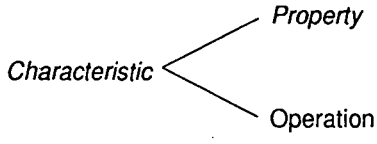


Figure 4.1: Type hierarchy of *characteristic*

An abstract type exists only for the benefit of its subtypes and aids in the design of an inheritance hierarchy. Variables of an abstract type may not exist as they do not have any functionality. An abstract type is thus termed *non-instantiable.* In Figure 4.1, operation is the only instantiable type.

The state of an object is defined by the *values* of a set of *instance properties.* Instance properties are the properties for which objects of the type carry values. The behaviour of an object is defined by a set of *operations* that can be executed on it. Operations are mentioned again in Section 4.2.1, while properties are discussed in greater detail in Section 4.2.3.

An object type has one *interface* and one or more *implementations*. The interface defines the properties of the object and contains the signatures of its operations. The implementation defines the *data structures* in terms of which the type is physically represented and the *methods* that operate on those data structures to support the interface.

Figure 4.2 is a sample of an ODMG ODL schema; the schema is the university schema described briefly in Chapter 2. The O_2 version of this schema (in O_2 ODL) appears in Figure 4.5.

4.2.1 Object types and instances

Objects may be organized into a graph of subtypes and supertypes. An object subtype inherits all of the characteristics of its supertypes, and may define additional characteristics; the subtype supports the characteristics of the supertype as well as its own characteristics. Thus, if type b is a supertype of type a, then an instance of type a is also an instance of type b. An object type may inherit from more than one supertype. This raises the possibility that a type will inherit

```
typedef struct<Integer date, Integer month, Integer year> Date;
typedef struct<Integer number, String street> Address;

interface Person {
        extent Persons;
        attribute String name;
        attribute Date birthdate;
        attribute Person spouse;
        attribute Bag<Address> lives_in;
        attribute Bag<Person> children;
};

interface Professor: Person {
        extent Professors;
        attribute Bag<Module> teaches;
};

interface Student: Person {
       extent Students;
        attribute Bag<Course> takes;
        courses_by_marks (in min: Integer, in max: Integer, out: Bag<Course>);
/* the method returns the Courses taken by this student, for which the  */
/* student has scored between min and max */
};

interface Employee: Person {
        extent Employees;
        attribute Employee boss;
};

interface TA: Student, Employee {
        extent TAs;
        attribute Bag<Module> assists;
};

interface Course {
        extent Courses;
        attribute String name;
        attribute Bag<Module> consists_of;   /* course consists of many modules */
        attribute Integer level;                          /* level=3 means 3rd year, etc */
};

interface Module {
        extent Modules;
        attribute String name;
        attribute Building lectured_in;
        attribute Integer enrollment;
        attribute String dept;
};

interface Building {
        extent Buildings;
        attribute String name;
};

/* Jones is a reference to a particular Professor object */
Professor Jones;
```

Figure 4.2: University database schema in ODMG ODL

properties that have the same name from two different supertypes. This name clash is handled by requiring the inheriting type to rename one of the inherited characteristics. The model defines a base object type called `Object`; all object types are inherited from `Object`.

The schema in Figure 4.2 defines eight classes; each class definition is preceded by the keyword `interface`. If a class has superclasses (other than `Object`) associated with it, the superclass name or superclass names appear after the class name. `TA`, for example, has superclasses `Student` and `Employee`. The schema also defines a *named* `Professor` object `Jones`; names are discussed in Section 5.2.1.

An *operation signature* is included in the object interface for each operation. The signature includes information such as argument names and types and return values. Operations are always defined on a single object type and operation names have to be unique within a single type definition. This raises the possibility (as with properties earlier) that a type defines an operation with the same name as one that it inherits. These operations are called *overloaded operations*. When an operation on an object is invoked, the operation defined on the most specific type of the object is executed.

The *extent* of a type is the set of all instances of that type. An extent is named when defining objects of that type; the extent is automatically maintained by the OODBMS as objects are created and deleted. If an object is an instance of the type a then it is a member of the extent of a. If type a is a subtype of type b, then the extent of a will be a subset of the extent b. The declaration of an extent is not compulsory.

The schema in Figure 4.2 defines an extent for each class. The extent of `TA` objects, for example, is called `TAs`.

Only object types have user-definable implementations. As mentioned, an implementation consists of a representation and a set of methods. The representation is a set of data structures and the methods are procedure bodies. The combination of the type interface specification and one of the implementations defined for the type is termed a *class*. The ODMG class model is richer than the `C++` model in that it allows multiple implementations. The implementation to be used by an object is specified at object creation time. The model does not provide any means for dynamically changing the implementation of an object.

4.2.2 Data types

The hierarchy of data types is rooted at the type `Denotable Object`, as in Figure 4.3. Denotable objects may be mutable (such as objects) and immutable (such as literals); these terms are explained later. Each of the mutable and immutable types may be atomic or structured. All denotable objects have identity, but the representation used to maintain that identity is different for objects and literals. Again, the data types shown upright are instantiable and data types that appear *italicised* are abstract data types. In the figure, only the `atomic object` data type may be instantiated.

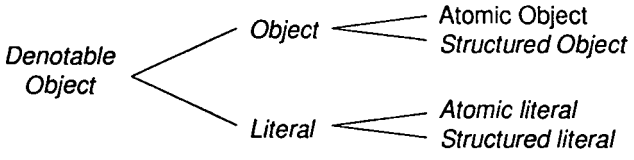


Figure 4.3: Root of Data Types

Objects

Instances of type `Object` are mutable. The values of their properties may change, and the properties in which they participate may change, but the identity of the object never changes – it remains the same object. The representation of the identity of an object is an *object identifier* (OID). An OID is a unique bit pattern

generated *solely* for the purpose of uniquely identifying a particular object. OIDs are never reused, even if an object is deleted.

Individual (atomic) objects and collections of atomic objects may be given names. An object will have one OID, but may have many names; a name refers to a single object. Given an object, it is possible to determine at runtime all the names referencing that object.

The properties of an object can be set when the object is created. If an attempt is made to retrieve an object-valued property that has not yet been initialized or assigned a value, the value `nil` will be returned.

Structured Objects

The type `structured object` has two subtypes – `structure` and `collection`.

A structure is an unnamed group of a fixed number of elements. Each element is a *(name, value)* pair, where the value may be any subtype of type `denotable object`. The types of the elements in the slots may be different and the slots are referred to by using the name of the slot.

Collections, by contrast, contain an arbitrary number of elements, do not have named slots and contain elements that are all of the same type. It is possible to have elements that are instances of different subtypes of the type over which the collection was defined. This is not a violation of any rule, since (as was mentioned in Section 4.2.1) an instance of type a is also an instance of type b if b is a supertype of a. Collections may be defined over any instantiable subtype of type `denotable object`.

The model supports the following collections: `set`, `bag`, `list` and `array`. Structured objects may be freely composed; the model, for example, supports sets of structures and arrays of structures.

`Sets` and `bags` are both unordered collections, the difference being that `sets` do not allow duplicates. On the other hand, `lists` and `arrays`, are ordered collections. `Arrays` are one dimensional, indexed collections. An initial size for the dimension is specified at array creation time, but the length can be changed after its creation. Removing an element from an array differs to removing an element in the other three collection types. The corresponding cell in an array is set to `nil`, and the length of the array does not change.

The structure type and collections mentioned are all mutable subtypes of `structured object` – the collection or structure retains its identity even if its members change. A `set`, for example, remains the same set even if a new object is added to it. This is in contrast to subtypes of `structured literal`; structured literals are discussed below.

Literals

Literals are immutable data types. Instances of `atomic literal` types have unique identity, but do not have OIDs. Examples of atomic literals are numbers and characters. The representation of the identity of a literal is typically a bit pattern of its value. The model supports the following types of atomic literals: `integer`, `float`, `boolean` and `character`. Literals and collections of literals may also be named.

Structured Literals

Literals are the immutable "versions" of objects; the type `structured literal` (orthogonal to `structured object`) has two subtypes – `immutable structure` and `immutable collection`. The model defines a few built-in immutable structures, such as `Date` and `Time`, and a few immutable collections, such as the collection types `string` and the `enumeration` type. An *enumerated* type is a generated type. Each enumeration declaration defines a type that has only the names mentioned in the declaration as its states. These names do not have any properties or operations.

The immutable forms of the mutable collections `set`, `bag`, `list` and `array` are `immutable set`, `immutable bag`, `immutable list` and `immutable array`. The treatment of the identity of immutable collections differs to that of mutable collections; two immutable sets are the same if and only if they have the same members. If two sets are both empty (i.e. they have no members), they are not necessarily the same set in the mutable treatment, but are always the same set in the immutable treatment.

Figure 4.4 shows an expanded view of the hierarchy shown in Figure 4.3. All instantiable types are shown `upright`.

4.2.3 Properties

As mentioned, properties are defined on object types only. The model supports only binary property relationships, not n-ary relationships. One-to-one, one-to-many and many-to-many property relationships are supported. Properties themselves have no names. *Traversal paths* are defined for each direction of a traversal of a property. An example is that a student *takes* a set of courses and a course *is_taken_by* a set of students. These paths are inverses of each other. Inverses are only possible when the value of a property is an object type.

Property names and traversals appear in the interface definition of an atomic object. The inverse path for a property does not have to be declared; in that

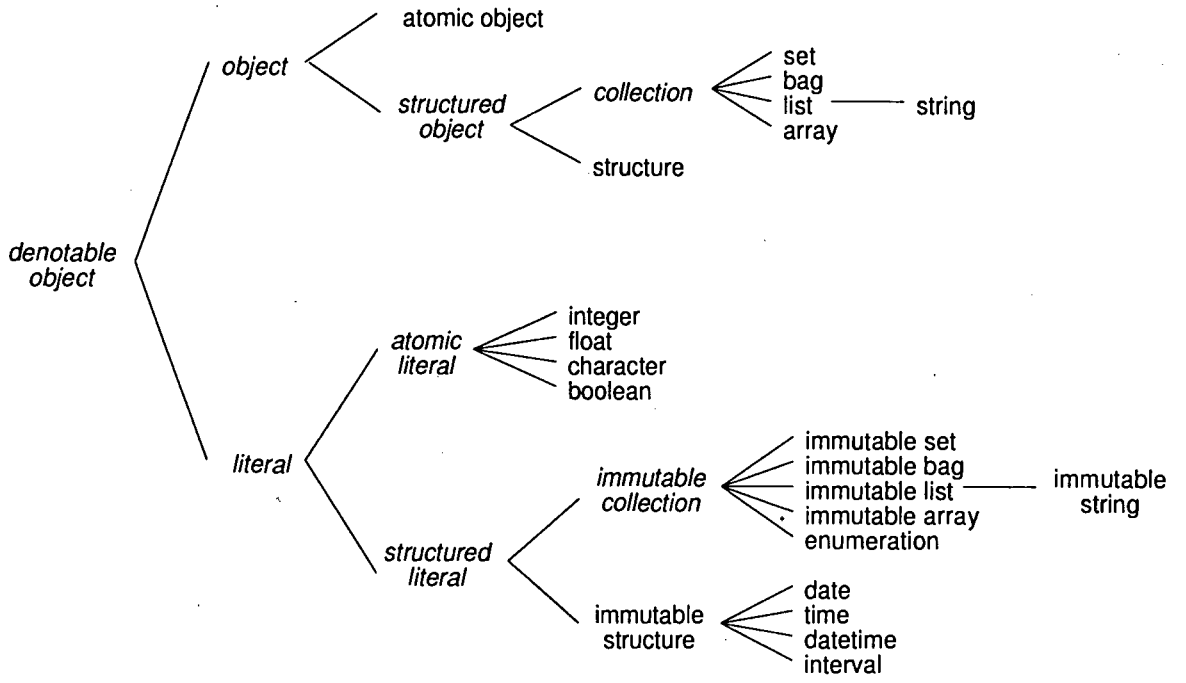


Figure 4.4: Expanded built-in hierarchy

case only one direction of traversal will be possible. Properties in the object model maintain referential integrity. If an object that participates in a property is deleted, an attempt to traverse the property in either direction will raise an exception.

The schema in Figure 4.2 does not define any *relationship* properties; relationships require inverses and none have been defined. Instead, all properties are *attribute* properties and are defined using the keyword `attribute`.

4.2.4 Equality

Two objects are equal only if their OIDs are equal; object a is thus equal to object b only if a and b refer to the *same* object, regardless of whether they have identical state or not. Two objects are thus equal only if they are the *identical* object. Equality between literal types is more relaxed. The identity of an immutable type is a function of its value. Two literals are thus equal if they have the same value. Other mutable and immutable types, such as mutable structures and immutable structures, mutable arrays and immutable arrays, carry similar properties.

4.3 The O_2 Database System

O_2 is a distributed object-oriented database management system, and was the first ODMG-compliant object-oriented database [34]. O_2 has a client/server architecture. The server provides persistence, disk management, concurrency control, data recovery and database security. The client manipulates O_2 objects and values.

The various definitions of an O_2 data model are grouped together into a schema; any number of schemas can be created. The schema is made up of the class definitions, named value definitions, class methods and program functions. A schema has one or more bases associated with it; a base contains the objects and values of the structures described in the schema. Any number of bases may be associated with a schema, but a base can be associated with one schema only.

QUIVER uses O_2 as its persistent store and the associated O_2Look application to view the data instances returned from a query. In O_2Look, each data instance is represented by a text value, and all the data instances appear in a single window; the data instances can be expanded and explored by the user to view all properties. The default text representation of each object can be redefined using the `title` and `bitmap` methods [35].

For the query examples presented here and in subsequent chapters, the university schema presented in Figure 4.5 is used; the schema was also used in Chapter 2. The schema is in O_2's ODL format, while the ODMG ODL version was listed in Figure 4.2. As mentioned, the schema defines eight classes (with their extents) and an additional named value `Jones`.

The schema contains different types of people: students, professors, employees and teaching assistants. Students take many courses, and each course consists of many modules. A building has a name and a module is lectured in a building. Professors teach many modules, and the `Jones` value is a named `Professor` object. The method `courses_by_marks` of the `Student` class returns the collection of courses for which the student scored between a specified range of marks.

In some cases, O_2 terminology differs to ODMG terminology [33]. When defining schemas and manipulating objects in any programming language bound to O_2, `set` refers to an unordered collection containing duplicates (called a `bag` by ODMG). The type `unique set` refers to an unordered collection that does not contain duplicates (called a `set` by ODMG). In addition, an *array* is interpreted as a *list* in O_2. ODMG's OQL standard is based on O_2's OQL [36], thus O_2 OQL is ODMG-compliant.

In O_2 ODL, inverses are not explicitly defined. An inverse is implemented by defining methods for both object types participating in the traversal path and its inverse. Also, extents are not implemented in an ODMG-compliant way; in

```
type Date : tuple(day: integer,
        month: integer,
        year: integer);

type Address : tuple(number: integer,
        street: string);

class Person public type
        tuple(name: string,
                        birthdate: Date,
                        spouse: Person,
                        lives_in: set(Address),
                        children: set(Person))
end;

class Professor inherit Person public type
        tuple(teaches: set(Module)) end;

class Student inherit Person public type
        tuple(takes: set(Course))
        method
                courses_by_marks (min: integer, max: integer): set(Course)
/* the method returns the Courses taken by this student, for which the  */
/* student has scored between min and max */
end;

class Employee inherit Person public type
        tuple(boss: Employee) end;

class TA inherit Student, Employee public type
        tuple(assists: set(Module)) end;

class Course public type
        tuple(name: string,
                        consists_of: set(Module),       /* course consists of many modules */
                        level: integer)                 /* level=3 means 3rd year, etc */
end;

class Module public type
        tuple(name: string,
                        lectured_in: Building,
                        enrollment: integer,
                        dept: string) end;

class Building public type
        tuple(name: string)
end;

/* named sets follow.  They are assumed to be the extents of their types */
name Persons: set(Person);
name Professors: set(Professor);
name Students: set(Student);
name TAs: set(TA);
name Courses: set(Course);
name Modules: set(Module);
name Employees: set(Employee);
name Buildings: set(Building);

/* Jones is a reference to a particular Professor object */
name Jones: Professor;
```

Figure 4.5: University database schema in O_2 ODL

O_2, extents are implemented by the programmer. Named values are defined to hold collections of different objects types, and each time an object of that type is created, it is added to its respective named value and to the named values representing the extents of its supertypes. In Figure 4.5, the named values `Persons`, `Professors`, `Students`, `TAs`, `Courses`, `Modules`, `Employees` and `Buildings` are implemented as the extents of their respective types.

4.4 Object Query Language

The Object Query Language (abbreviated OQL) is a query language which supports the ODMG data model. OQL is very close to SQL-92 [17], and contains extensions to support object-orientation. The extensions include object identity, path expressions and operation invocation. In this section, we provide only a brief overview of OQL. For more details, the reader is referred to [6].

OQL can be used as a function called from a programming language bound to the

database (such as C++) or as an ad hoc query language. As a stand-alone language, OQL allows the querying of denotable objects starting from their names, which act as "entry points" into a database. The simplest queries are those comprising of names only. The query "Persons" returns all persons and the query "25" returns the literal value 25.

The "select from where" clause is used to extract those elements meeting a specific condition from a collection. The select clause defines the structure of the query result, from introduces the collections against which the conditions are checked and where sets the conditions to filter the collections. The following query returns all those people who have spouses:

```
select x
from x in Persons
where x.spouse != nil
```

The query returns a bag of objects, as duplicates within the answer would be allowed. If the keyword distinct appears after select, duplicates are eliminated and a set is returned. The distinct keyword can also be used to eliminate duplicates from lists. A join is performed by the query processing system if more than one collection appears in the from clause. The following query returns those people who have the same names as any of their children:

```
select x
from x in Persons, y in x.children
where x.name = y.name
```

A query result can be constructed using the struct keyword. Consider the following query:

```
select struct (parent: x, children: x.children)
from x in Persons
where count(x.children) != 0
```

The query constructs a structure containing two elements for each person object satisfying the where condition. The structure contains the parent object (this component is named parent) and a bag of children, provided the parent has at least one element in its children property value (this second component is named children); the count operator returns the number of elements in a collection. Other aggregate operators are min, max, sum and avg. If a bag or a set contains a single element, the element can be extracted by using the element operator; if

`element` operates on a collection with one element x, it returns that element x; the operator fails if the collection does not contain exactly one element.

The `exists` clause returns `TRUE` if its specified conditions succeed, and returns `FALSE` otherwise. The following query returns all the people who have at least one *non*-`nil` element in their `children` property value:

```
select x
from x in Persons
where exists y in x.children: (y != nil)
```

This query answers the question "Find those people that have at least one child". The previous query is only partially correct, as it would have returned a person and their children even if all the elements within the person's `children` property were `nil`.

OQL has a flexible structure, and entire queries may appear with the `select`, `from` and `where` clauses of a query. The following query is similar to one presented earlier, but uses nested OQL; the query returns those people who have the same names as any of their children:

```
select x
from x in Persons
where x.name in (
   select y.name
   from y in x.children
)
```

The `group by` operator groups together objects of a collection with the same value. Consider the following query:

```
group x in Persons by (date: x.birthdate)
with (number: count(partition))
```

The query returns a bag of two-component tuples, containing a birthdate and the number of people sharing that birthdate. The keyword `partition` refers to those elements that form the partition after the grouping takes place. OQL also allows data to be sorted:

```
sort x in Persons by x.name
```

The result of a sort operation is always a list, even if the objects to sort may be a bag or a set.

The `for all` clause returns TRUE if a condition holds for *all* the elements within a collection, FALSE otherwise:

```
for all x in Persons: (x != nil)
```

This query returns TRUE if all the elements in Persons are *non*-nil. If the Persons collections is empty, the query returns TRUE as well.

The condition clause of a `for all` clause may contain any clause that returns a boolean value. By combining the `for all` and `exists` clauses, the following query simulates a *subset* operation. The query returns TRUE if all the elements in the CSCStudents collection are contained in the Persons collection:

```
for all x in CSCStudents: (
   exists y in Persons: (
      x = y
   )
)
```

For a full discussion on OQL and its constructs, the reader is referred to [6].

Chapter 5

The QUIVER Query Language Design

5.1 Introduction

Graphs are a natural representation for the data in several application domains, such as transport networks, project scheduling, parts hierarchies and hypertext systems. In these applications, visualizing the extensional part of the database as a graph is the most intuitive representation from the user's perspective.

We also believe that graphs are an ideal way to query the extension of a database. This chapter formally describes the QUIVER query language. Queries are represented by graphs, and these graphs represent constraints to be matched against the extension of the database.

One of the principles that guided the design of the query language was that it had to be consistent – similar query concepts had to be represented by similar looking and similar acting query items. The language developed also makes minimal use of colour; this makes QUIVER usable across a broad range of hardware. The query language presents no problems in visual scaling either, as even if the query graph is zoomed into, there is no ambiguity as regards the identification of query items. Such a problem would occur if, for example, large circles represent one concept and small circles represent another.

Note that all figures in this chapter, unless otherwise stated, are postscript output generated by QUIVER. In the queries presented, all named values that appear as plurals represent the extents of those objects. More specifically, the named value `Students` represents the extent of `Student` objects, and the named value `Courses` represents the extent of `Course` objects.

We now introduce some terminology that will be used in this chapter:

Definition 5.1 *A* node *is a graphical query element used to represent a basic data type. Each node has a visual representation: a circle, a square, a rectangle, or a rounded rectangle.* □

Definition 5.2 *An* edge *is a graphical query element that connects two nodes. Each edge has a visual representation: a solid line, a solid line with an arrowhead, a dashed line with an arrowhead or a double-line with an arrowhead.* □

Certain data structures are represented by many nodes. A bag of `Course` objects, for example, is represented by a circle enclosed by a rounded rectangle (a circle enclosed by a *blob*) – the rounded rectangle *contains* the circle. The query language has been designed such that nodes never overlap each other.

Definition 5.3 *A* maximal node *is a node that is not contained in any other node.* □

Definition 5.4 *An* inclusion group *of nodes consists of a maximal node, and the closure of all the nodes that it contains.* □

5.2 Representation of Basic Data Types

Throughout the remaining text of this chapter, reference will be made to features that have not been implemented, or features that would be an extension to the current implementation of QUIVER. These features are part of the design, but have not yet featured in the implementation. We believe that we have achieved an implementation that demonstrates sufficient functionality.

5.2.1 Objects and literals

As mentioned, the two basic data types of an object-oriented database are *class* instances (or objects) and *literal* instances. They are both represented by the same primitive shape, a circle, but with different shadings of the circle.

The terminology used is that an *object node* or a *literal node* is represented by a *circle*. Each of the types below has a visual representation. A structure node, for example, is represented by a rectangle. Nodes and edges also have labels associated with them.

In order to construct a query on an object, the user can constrain the properties and methods of the object. The user, however, does not *need* to constrain any of the properties or methods. The user is also never aware of the order in which

the properties and methods of a class are defined, as this has no effect on query construction.

Each object also has a unique *object-identifier* (abbreviated *OID*) associated with it. OIDs are maintained by the database system and consist of a bit pattern generated solely for the purpose of uniquely identifying each object. OIDs are never displayed nor modified in QUIVER.

A significant part of the state of an object is thus hidden from the user. Part of the state may be displayed, namely the properties and methods necessary for constructing the query. An adapted form of the *grey-box* view (as discussed in [40]) was thus chosen to represent objects. Objects are displayed with their contents *screened*, or grey in the case of a black foreground and a white background (see Figure 5.1).

Literals are distinguished by their values, which can be displayed on screen. There are no unique identifiers as there are with objects, and there is thus no need to indicate any form of hiding. An adaptation of the *white-box* view [40], called *open* here, is thus used to represent them. An adaptation of the *black-box* view (called *closed*) was chosen to represent methods (methods are discussed in Section 5.5), as the code associated with methods is always hidden.

Objects may also be named; a name allows the object to be referred to directly during a query. An *unnamed* object is shown in Figure 5.1(a). The prefix of a circle's label (the part of the label preceding the colon) is blank, and the suffix displays the type of the object – in this case the object is of type `Professor`. In the case of a named object, the label prefix displays the object name, and suffix displays the type of the object. Figure 5.1(b) shows the named object `Jones` which is of type `Professor`.

Figure 5.1: (a) An unnamed object and (b) a named object

An extension to the implementation of QUIVER would be to allow the user to choose the appearance of objects and literals. A `Person` object could, for example, be represented by a stick-figure, and an `Address` object could appear as a house. It would then be unnecessary to display the name (if one existed) *and* the type in the label – the type would be obvious from the visual display.

Literals could also be replaced by icons, with different icons representing integers, characters, real numbers and strings. Providing this would simply require adding a graphic icon editor to QUIVER.

An alternative display of labels has also been implemented. If labels are to be displayed in full (*full-labels* turned on), they appear as described above, with the

name and the type separated by a colon. If an object's label is to be displayed abridged (*full-labels* turned off), only the prefix (without the colon separator) is displayed. The label is thus blank in the case of an unnamed object, while in the case of a named object, only the object's name is displayed.

In the discussion and examples of the query primitives that follow, the naming convention refers to primitives with the full-labels turned *on*, unless otherwise stated.

Just as with objects, literals may be named or unnamed. The label of an open circle indicates either the *value* of the literal and its type (in the case of an unnamed literal) or the *name* of the literal and its type (in the case of a named literal). The value of a named literal is never displayed; it is always referred to by its name only. The type of a literal may appear as `CHAR`, `INTEGER`, `STRING`, `BOOLEAN` or `FLOAT`, representing a character, an integer, a string, a boolean value or a floating point number. Only characters, integers and strings have been implemented.

If a string literal's value is displayed, it is always surrounded by double quotation marks, while a character literal's value is surrounded by single quotation marks. There is thus never any ambiguity in the prefix of a literal label as to whether it refers to the literal name (i.e. that it represents a named literal) or to the literal's value (i.e. that it represents an unnamed literal).

As with objects, a literal's label contains just the prefix if full-labels are turned off. Thus an unnamed literal has just its value displayed and a named literal has just its name displayed.

Figure 5.2: (a) & (b) Unnamed literals and (c) a named literal

Figure 5.2(a) represents an unnamed character, with value `x`, (b) is an unnamed integer, with value `1996` and (c) is a named string, with name `User`.

5.2.2 Structures

Structures are data types that contain groups of elements, where each element is a (*name*, *value*) pair. The element value may be of any type – a structure may thus contain other structures.

Just as with literals, structures do not have unique identifiers. A structure is, however, similar to an object in that the user does not need to display all of its properties in order to query on it. In addition, the order in which a structure's properties were defined is, by default, not displayed.

To represent structures, we chose a screened view once again. Structures are represented by a screened rectangle, as in Figure 5.3, and the label of the structure node indicates its type. Structures may not be named, thus the prefix of a structure's label is always blank. The type of the structure is a comma-delimitered list containing the type of each of its components.

Figure 5.3: A structure

The structure in Figure 5.3 might be used to model a person's birthdate, storing the day, month and year that the person was born. If full-labels are turned off, a structure's label is blank.

5.2.3 Collection types

Collections represent the inclusion of elements within other elements. In order to represent this logical inclusion, we chose to represent collections using visual inclusion.

Collections contain elements of the same type. A bag collection is represented by an *open rounded rectangle* (or an *open blob*) and an instance of the type of its elements is always displayed within the blob – a single instance is displayed even if the bag contains many elements. Of the four mutable collection types defined by the ODMG, only the bag collection has been implemented; immutable collection types have not been implemented.

Just as with objects and literals, collections may be named or unnamed. The convention used in constructing the label of a collection node is identical to that already mentioned – the prefix of the label contains, if it exists, the collection name and the suffix contains the type of the collection. Figure 5.4 represents two collections, both of which are bags of `Student` objects. Figure 5.4(a) represents an unnamed collection and (b) represents a collection named `Students`.

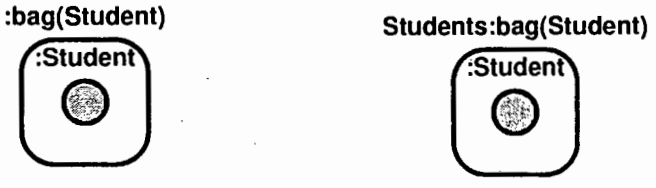


Figure 5.4: (a) An unnamed bag and (b) a named bag

As mentioned, collections contain elements of any type, thus a bag may contain other bags. Figure 5.5 represents a bag of bags of `Course` objects. In the figure,

the *inner* blob (representing the inner-bag node) lies within the *outer* blob (the outer-bag node). It must be stressed that visual inclusion does not indicate that the inner collection is a subset of the outer one, rather that the inner collection is an element of the outer collection.

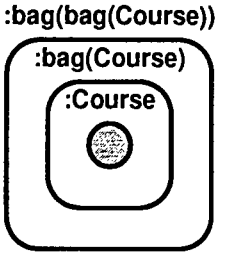


Figure 5.5: A collection of collections

Figure 5.6 represents the other three collection types defined by the ODMG, namely *set*, *list* and *array* collections. A set node appears identical to a bag node, except that it is surrounded by the braces "{" and "}" – such braces are commonly used to denote sets. Lists and arrays are both ordered collections – their visual representations thus both contain an ellipsis ("..."), indicating that the collection elements exist in a particular sequence. We chose to represent the list collection as growing *downwards*, resembling a shopping list. Because an array is an indexed collection, a subscript is associated with each element – the number 1 thus appears as part of the array representation.

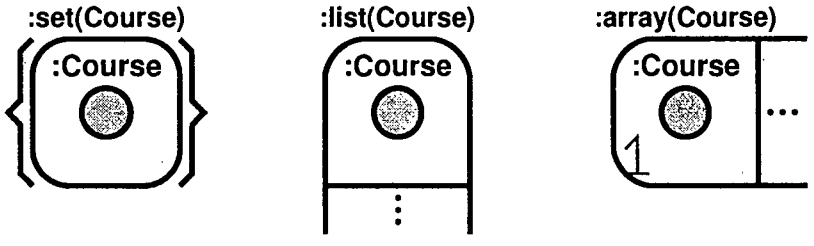


Figure 5.6: (a) A set, (b) a list and (c) an array

Figure 5.6 was not produced by QUIVER.

5.2.4 Properties

Properties are defined over objects and structures, and may have objects, literals or structures of objects and literals as their values. We see that both the *source* (the type over which it is defined) and the *target* (its value) of a property are represented as nodes. It is thus natural to represent a property as a *directed edge* between the two nodes. The tail of the property edge is the node representing

the value over which it is defined, and the head of the property edge is the node which represents its value.

Property edges are labeled by their property names. Property names do not contain any prefix or suffix parts, and remain constant regardless of whether full-labels are turned on or off.

Figure 5.7 shows property edges `birthdate` and `takes`, both defined over the `Student` object. The value of `birthdate` is a structure consisting of three integers, and the value of the `takes` property is a bag of `Module` objects.

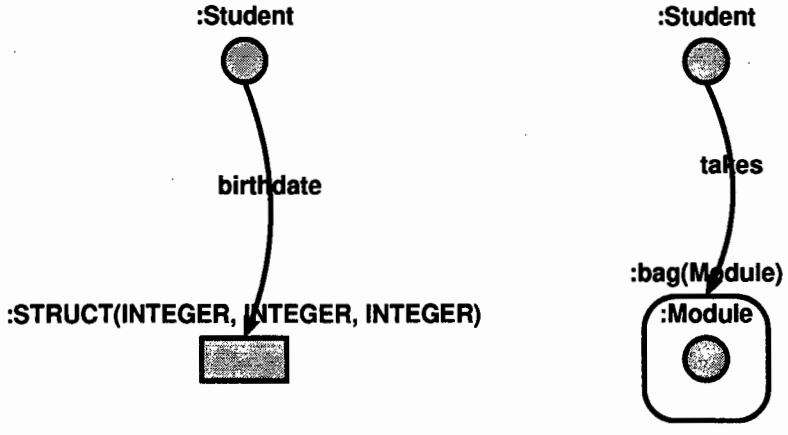


Figure 5.7: Property edges

5.3 Queries as Constraints

The query processing system of QUIVER tries to match the query *patterns* that are constructed by the user to data patterns in the database. Thus a visual query is interpreted as a constraint that is checked against persistent data instances. If the constraint pattern matches at least one persistent pattern, the query *succeeds* and `TRUE` is returned; otherwise the query *fails* and `FALSE` is returned.

Figure 5.1(b) is an example of the simplest type of query – it queries whether the `Jones` name references an object or not. If `Jones` does not refer to an object, it will have the value `nil`. The query will thus return `TRUE` if the person named `Jones` exists.

Definition 5.5 *An object is* `nil` *if its value is nil; it is* non-`nil` *if it refers to an object. A collection is* non-`nil` *if at least one of its elements is* non-`nil`. □

Whenever an object appears in the construction of a query, QUIVER checks that it is *non*-`nil`.

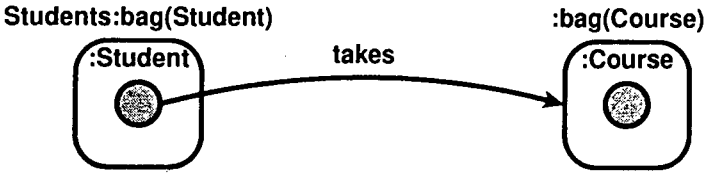


Figure 5.8: A query with one named value

The query in Figure 5.8 will succeed if the Students bag is *non*-nil and at least one of its elements has a *non*-nil takes property value. If there are no elements in the Students named value, or if there are no *non*-nil Student objects in Students, or if none of the students take any courses, the query will fail and return FALSE.

As is the case in Figure 5.8, there has to be at least one named value appearing in a query if the query is to return information about the database. This may be thought of as the *root* of the query as it is the source of the object instances that are traversed. In this case the Students collection is retrieved as it is the only named value, and each element in the collection is checked. When a *non*-nil object is found, its takes value is then examined.

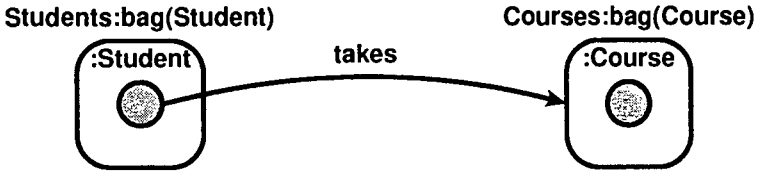


Figure 5.9: A query with more than one named value

If many collections in a query are named, such as in Figure 5.9, QUIVER arbitrarily decides which named value to retrieve first and use as the primary root of the query. This makes no difference to the query answers generated (query processing is further discussed in Chapter 7). The query in Figure 5.9 succeeds if there exists an element of Students such that the student's takes value is equal to the Courses named collection.

5.4 Equality and Subset Edges

The property edge has already been introduced. In this section, we describe two other edge types: the *equality* edge and the *subset* edge. An equality edge succeeds if the two items it joins are equal to each other. The equality operation is symmetric, and is thus represented by an undirected edge. The operation is not defined over any *particular* type – it is defined over all types by the query system.

The edge is thus unlabeled as well. Equality can only be checked between two items that have compatible types, such as between two integers, or between two bags of Student objects. Query translation cannot take place if, for example, a bag of characters and a bag of integers are checked for equality. Compatibility between types is further discussed in Chapter 6.

Figure 5.10: A query with an equality edge

The query in Figure 5.10 will succeed if some student has a *non*-`nil` `takes` value, which is also equal to the `Courses` collection. This query is equivalent to the query in Figure 5.9.

The subset operation is defined over collections by the query system but unlike the equality operation, the subset operation is not symmetric. It is thus represented by an unlabeled directed edge. The subset operation succeeds if the *source collection* (i.e. the collection represented by the node at the tail of the edge) is a subset of the *target collection* (i.e. the collection represented by the node at the head of the edge). Just as with the equality operator, the operands of the subset operator have to be of compatible types.

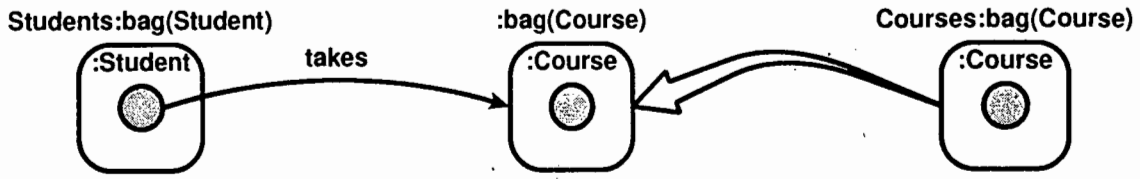


Figure 5.11: A query with a subset edge

The query in Figure 5.11 resembles the query in Figure 5.10, except that this query succeeds if some student takes *at least* all the courses found in the `Courses` bag – that is, `Courses` is a subset of the courses taken by the student. Note that since `Courses` is the extent of objects of type `Course`, this query is also equivalent to the one in Figure 5.9.

Formally, if being an object of type a implies the object is of type b, then the set A of objects of type a is a subset of the set B of objects of type b. The profile of the subset edge was thus chosen to resemble the implication ("$\Longrightarrow$") sign. We decided, however, to make the source of the edge a single point as in Figure 5.11, as it would then form a neat intersection with any node on screen.

5.5 Methods

In order to use a method, the user does not need to know the code that defines the method, and this code is always hidden by QUIVER. Method execution is thus represented by a *closed* square.

Methods are associated with objects, and a labeled directed edge is drawn from the object node to the method node, just as a property edge (properties were discussed in Section 5.2.4) is drawn from the object node to the value of the property. The edge label indicates the method name.

Unlike objects and literals, methods cannot be named, thus the prefix of a method node label is always blank. The suffix of a method node label indicates that it is a mapping from a *domain* to a *range* (from the *input* of the method to the *output* of the method) – the domain and the range are separated by a -> symbol.

Figure 5.12 represents the `courses_by_marks` method, which is defined over the `Student` class. The method has two integer parameters, which represent two marks *min* and *max*, and the method returns all the courses for which the student scored between *min* and *max*. The method can thus be represented as a mapping, where the domain is a `(INTEGER,INTEGER)` structure and the range is of type `bag(Course)`.

If full-labels are turned off, a method node has a blank label.

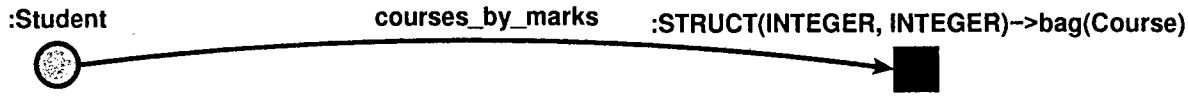


Figure 5.12: An object method

A method may have many parameters – in QUIVER this is modeled as the method having a single structured parameter (called a *method input node*) where each of the components of the structure corresponds to an original parameter of the method (structured types were discussed in Section 5.2.2).

All the parameters of a method are compulsory (the user has to supply values for *all* of them), and the user has to be able to specify the order of the parameters. These two requirements make the screened view of structure nodes unsuitable to represent all the information of a method input node. In a screened view, the user would not be able to determine the order of the structure properties, nor can the user easily determine if all the properties are displayed.

In order to solve both these problems, an open view variation of the node type used to represent structures is, by default, used. Each of the components of a structure is represented by a *cell*, and the cells are aligned horizontally. The node at the head of the *method expand* edge leaving the left-most cell represents the

value of the first parameter, the node at the head of the method expand edge leaving the second cell represents the second parameter, and so forth. The user is thus able to tell how many components the structure has, and is able to correctly sequence the parameters.

Only the open view variation of method input nodes has been implemented, and only the screened view variation of structure nodes (Section 5.2.2) has been implemented.

Figure 5.13 shows the `courses_by_marks` method with a method input node. The method input node has a label corresponding to the domain of the method mapping. The two parameters *min* and *max* have also been specified in the figure. Method expand edges are labeled with their parameter names and are directed away from the method input node to their values as they constrain the value that the input structure may have. The figure also shows the value type of the method, a bag of `Course` objects.

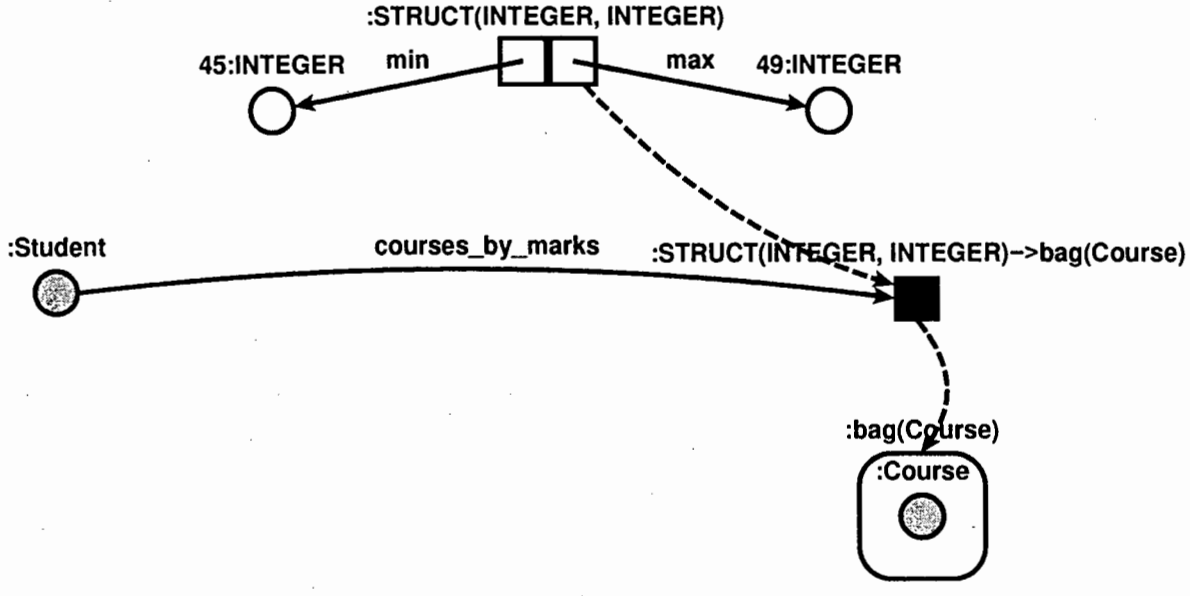


Figure 5.13: Further expansion of an object method

5.5.1 Data-flow edges

As displayed in Figure 5.13, a *data-flow* edge is used to connect the method input node to the method node. A data-flow edge is represented by an unlabeled, directed *dashed* edge, and represents the *transport* of data from the tail of the edge to the head of the edge.

Data-flow edges are used in three instances:

1. to represent the data flowing into and out of a method call,
2. to represent the data flowing into and out of a function call (Section 5.6), and
3. to represent the data flowing out of a subquery (Section 5.8).

A method thus has two data-flow edges associated with it – an incoming edge, representing the incoming flow of method parameters, and an outgoing edge, representing the outgoing flow of the method result. In Figure 5.13, the node at the tail of the incoming data-flow edge represents the method input node, and the node at the head of the outgoing data-flow edge represents the return value of the method.

5.5.2 Displaying methods

A method without any parameters is similar to an object property – both are associated with an object, and both produce a value without any additional inputs. The age of a student, for example, would usually be implemented as a method (computing the age from a date of birth) rather than as a property, although it may be easier to think of age as an object property. An alternative way of displaying parameter-less methods has thus been designed.

Figure 5.14 displays three alternatives for querying a student's age – Figure 5.14(a) is the default if age were a property, and Figure 5.14(b) is the default if age were a method. Figure 5.14(c) displays an extension to the implementation of QUIVER – this display would be allowed if age were a method requiring no parameters.

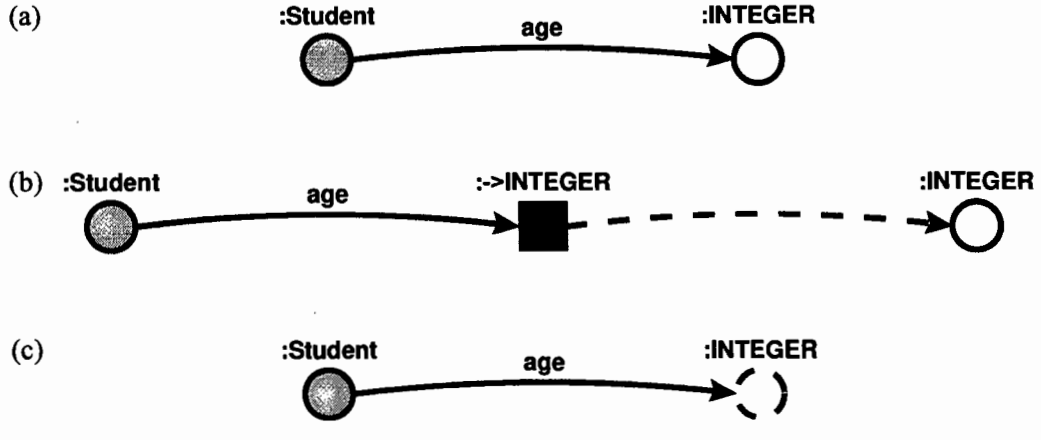


Figure 5.14: An alternative way of displaying a method with no parameters

In Figure 5.14(c), the method edge remains unchanged, but the method node is no longer displayed. Instead, the value of the method is displayed dashed (the

dashed circle node) to indicate the data flow from the method node to the value of the method. This shorthand is for parameter-less methods only, because of their functional resemblance to properties – it cannot be applied to a method that requires parameters.

Figure 5.14 was not produced by QUIVER.

5.6 Functions

Functions, sometimes called aggregate operators, include operations like `max`, `min`, `sum`, `count` and `average`. The `bag` function (explained later) could also be included, hence functions are not necessarily "aggregate".

Functions differ to methods in that they do not have any implicit inputs, as they are defined by the database system and are not associated with any particular class. A method and function, however, both involve the execution of code and may both accept inputs and both have outputs.

A function, therefore, is also represented by a closed square, and has a data-flow edge entering it (representing the input of its parameter) and a data-flow edge exiting it (representing the output of its value). Because a function is not defined over any particular class, there is no "function edge" (a solid directed edge) entering it.

Four functions have been implemented in QUIVER, namely `element`, `sum`, `count` and `bag`. The `element` function accepts a collection containing one element, and returns that element – the query fails if the input is a collection with more than or less than one element. The `bag` function does the converse, accepting a value x and returning a `bag` with one element x. The `sum` function can only accept a collection of integers or a collection of floating point numbers as input, and returns the sum of all the numbers in the collection. The `count` function accepts any collection and returns the number of elements in that collection – the return type is an integer.

As mentioned in Section 5.2.3, only the bag collection type has been implemented. In the current implementation, therefore, the `element` function accepts a bag containing one element, and returns that element. The input and output values of the other functions are similarly restricted.

The four functions each have one input; a shortcut to specifying the input has thus been implemented. A method input node is not used – a data-flow edge is drawn from the function input node to the function node. The outgoing data-flow edge remains unchanged in the shorthand form. Figure 5.15 shows the *bag* function being called.

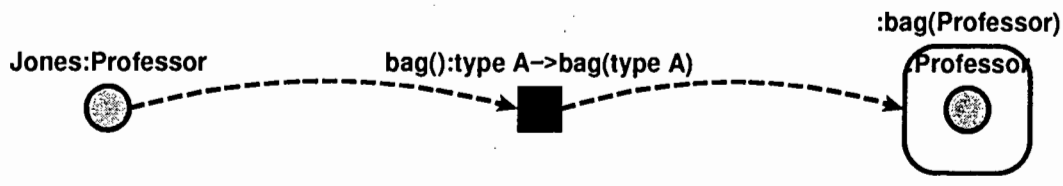


Figure 5.15: Calling the bag function

In the figure, the function's input is the Jones object, which is of type Professor. The output of the function is thus a bag containing one Professor object. Function names differ to method names in that function names are globally defined. The prefix of a function label thus contains the function name followed by the "()" symbol, indicating a function call. The suffix of the function label follows the same convention as the suffix of a method node label.

If full-labels are turned off, the function node label contains just the function name followed by the "()" symbol. In Figure 5.15, for example, the function node label would read "bag()".

5.7 Data Output

The queries presented thus far returned either TRUE or FALSE, depending on whether the query pattern matched at least one data instance or not. A more informative output, however, is the actual data instances that occur in the successful data patterns. In QUIVER, this output is specified by modifying the query graph; the query constraints and the query output are thus specified on the same graph. Modifying a visual representation to **bold** indicates output. If a successful path is found, the data instance or data instances represented by the bold node or nodes form the output of the query.

Collection nodes, object nodes, literal nodes and structure nodes may appear as output nodes. QUIVER allows data output of two types, namely output using *fully determined* nodes (Section 5.7.1) and output using *collecting blobs* (Section 5.7.2).

5.7.1 Fully determined nodes

In this case, a fully determined node is marked as the output of the query. Fully determined nodes are formally discussed in Chapter 7, but briefly, a fully determined node is a node which represents one data item only. An example is a named value – if the nodes representing Students (the extent of Student objects) are placed on the canvas, then the bag node (represented by a blob) is fully determined. The object node within the bag node is not fully determined, as it represents any Student object.

As another example, if the node of Jones (a named Professor object) is placed on the canvas, it is also a fully determined node. Professors teach many modules, and the collection node that represents the teaches property of Jones is fully determined as well. The object node within the collection node is not fully determined, and any property value attached to the object node is not fully determined either.

The output of these types of queries is specified by marking a single, fully determined node as bold. If the query returns TRUE, then the data item represented by the bold node is returned as the output of the query. If the query pattern does not exist in the database, the query fails.

An example is the query in Figure 5.16. Consider first the query if the bold object node were replaced by a non-bold object node. The query would return TRUE if Jones was *non*-nil. The query in Figure 5.16 thus returns the Jones object if it is *non*-nil. If Jones is nil, the query fails.

Figure 5.16: A query returning an object

QUIVER also allows the user to create a structure to construct the output of a query. Structures were first discussed in Section 5.2.2, where the screened view was used. The open view of structures was discussed in Section 5.5, when discussing method inputs. The open view of structures is again used here in order to allow the answer of a query to consist of parts of many other nodes. The open view is appropriate as it allows the user to specify the order of the components of the output structure.

The query in Figure 5.17 differs from the query in Figure 5.16 in that *output structure* nodes were created in order to specify output; an existing node was modified in order to specify output in Figure 5.16. The query in Figure 5.17 consists of two bold nodes; both bold nodes form the output of the query. The output node having no incoming output edges is called the *root* of the output, as output is calculated from this point.

Each of the cells in an output structure node indicates a component of the answer, and the order of the components corresponds to the order of the cells on screen, from left to right. Because the structure indicates output, it, as well as the edges of its components, are bold.

The query first determines if Jones is *non*-nil and if Jones has a *non*-nil bag of Person objects as its children property. If Jones does not, the query fails. On the other hand, if at least one persistent data pattern is found, the output of the query is a structure with two elements. The first element is named theProf and

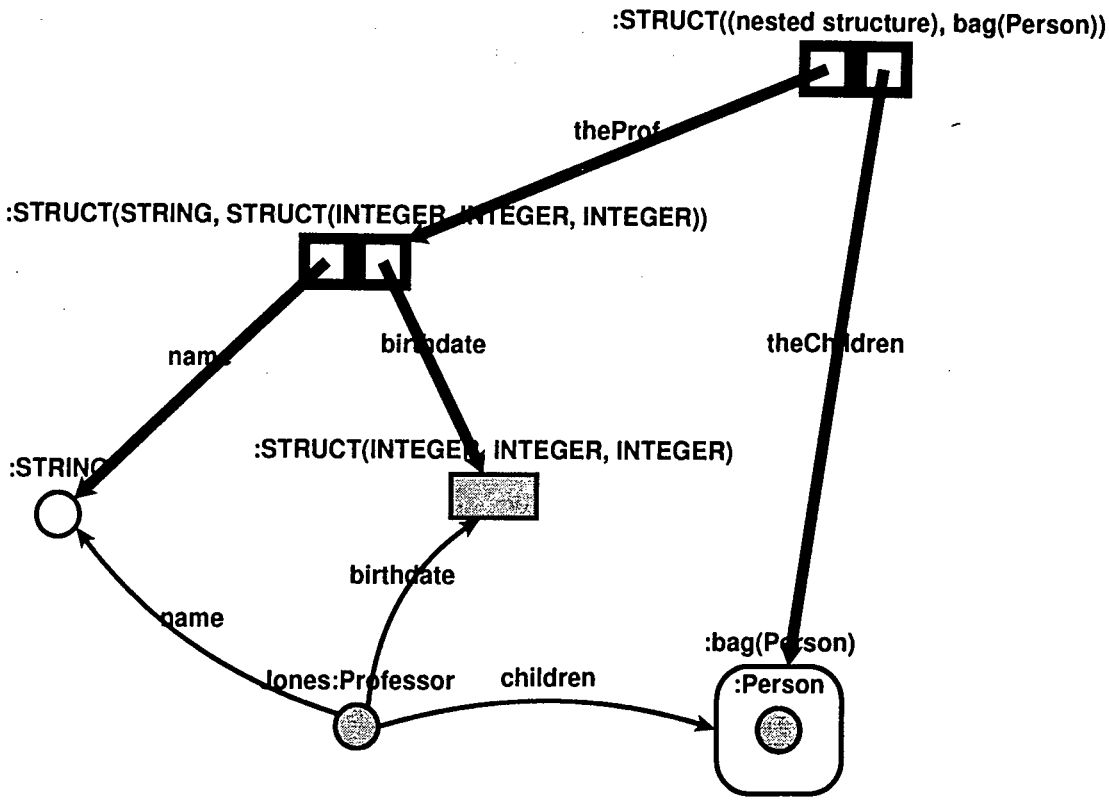


Figure 5.17: Constructing an output structure

the second is named `theChildren`. The value of `theChildren` is the children that `Jones` has, and the value of `theProf` is another structure with two components, named `name` and `birthdate`. The value of `name` is the name of `Jones`, and the value of `birthdate` is a structure containing `Jones`' birthdate.

The non-root output structure is fully determined as each of its edges points to a fully determined node; the structure can thus contain one value only. The root output structure is, similarly, also fully determined. If a query contains output structures, then one of these structures has to have no output edges entering it, and it becomes the root of the query.

The query in Figure 5.18(a) returns the `Students` extent if it is *non*-`nil`, and fails otherwise. Figure 5.18(b) is an invalid query as the bold node is not fully determined. The query in Figure 5.18(c) is also invalid as the inclusion group contains an object node which is not fully determined.

The output type of a query returning fully determined data items can be calculated by examining the type of the root output node – if a query contains only one output node, it automatically becomes the root.

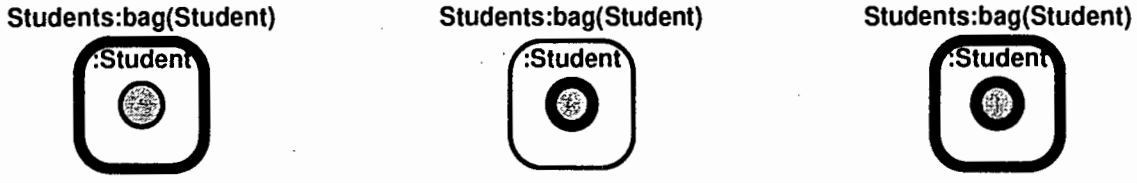


Figure 5.18: (a) Retrieving Students, (b) & (c) Invalid queries

5.7.2 Collecting blobs

Collecting blobs allow a query to return the data instances that occur in *every* matching data pattern. The query in Figure 5.19 illustrates the use of a collecting blob; the query finds all students that take at least one course. The query constructs a bag of Student objects (the output inclusion group) and returns this collection as the result of the query. The construction of the bag is indicated by both the equality edge connected to the output object node and by the output bag node placed around the object node.

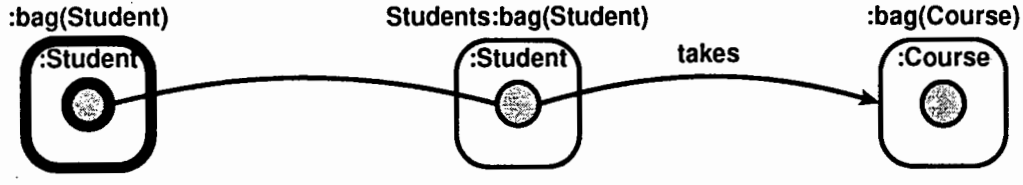


Figure 5.19: Output as a collection of objects

The Student object node is the only *reachable* node that is marked bold. Reachability is formally discussed in Chapter 7, but briefly, a node is reachable if it can be *traversed* (such as through property or equality edges) from other reachable nodes or if it is an element of a reachable collection. All named values are reachable. The output Student object is reached through the equality edge connected to it, as the non-output Student object is reachable due to it being an element of Students.

The inclusion group containing the bold node also contains a bold blob that is not reachable. This bold blob is called a *collecting blob*. Collecting blobs are always bold, and are always the outermost blobs of their inclusion groups. They are not reachable themselves, as they visually include the outermost reachable node.

Collecting blobs indicate that the data items within it are to be collected for every matching data instance pattern found. In this case, the Student object is added to the output collection for every matching data instance found. A collecting blob may only appear in an inclusion group that contains other output nodes, and a query may contain only one collecting blob. If a node that is not

fully determined is set as an output node, it has to appear within a collecting blob. An output node that is fully determined does not require a collecting blob around it.

Using the notation displayed in Figure 5.19, the inclusion group indicating output contains *only* output nodes. A shorthand for this notation, displayed in Figure 5.20, has been implemented. The output inclusion group and the inclusion group from which output is reached (the one containing the non-output Student object node in this case) are combined into a single inclusion group. The nodes connected to output nodes appear bold, and the collecting blob is drawn as a maximal node. The type of the output of the query can still be calculated by examining the bold nodes in the output inclusion group; the bold object node and the bold collecting blob indicate that the output is a bag of Student objects.

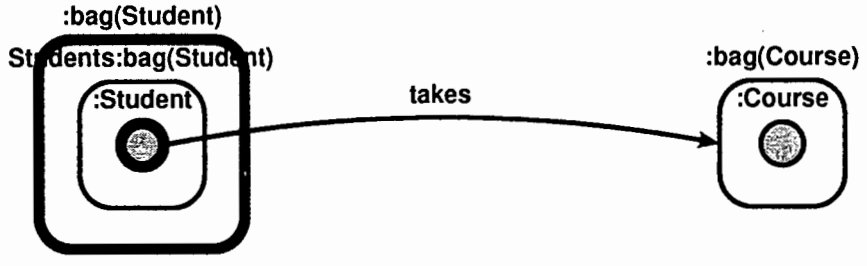


Figure 5.20: Output as a collection of objects, shorthand notation

The query in Figure 5.18(c) differs to the inclusion group containing the Student object in Figure 5.20. Both contain reachable output object nodes but Figure 5.18(c) contains another reachable output node, namely an output collection node. The output collection node in Figure 5.20 is not reachable. An inclusion group may indeed have up to two output nodes, but at most one of them may be reachable.

The query in Figure 5.21 resembles the query in Figure 5.20, except that it has a different output type. Once again, QUIVER searches for the appropriate Student objects. For every object found the entire Students collection is added to the output collection, as Students has been marked bold. Once all the paths have been considered the collection of these bags (the output type is a bag of Student bags) is returned as the answer of the query. The output consists of a bag with many identical elements. The Students blob is fully determined – both fully determined nodes and non-determined nodes can thus be surrounded by collecting blobs.

The suffix of a collecting blob's label always indicate the type of the output of the query. Because a collecting blob collects data instances, it adds a collection level to the output type. The prefix of an output blob is always empty, and if full-labels are turned off collecting blobs have a blank label.

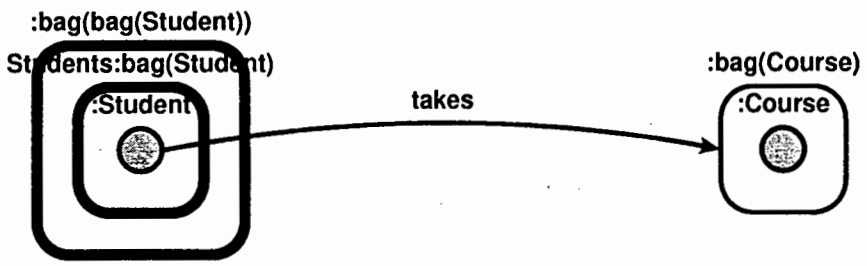


Figure 5.21: Output as a collection of bags

The query in Figure 5.22 shows the use of a collecting blob used in conjunction with an output structure node. The query finds each student that takes at least one course, and then constructs a structure containing the Student object and the courses taken by the student. This query would be syntactically incorrect if it did not contain the collecting blob, as the output structure node is not fully determined. If the query were to consist of more than one output structure node (such as the query in Figure 5.17), then only the root output structure node may be contained in a collecting blob.

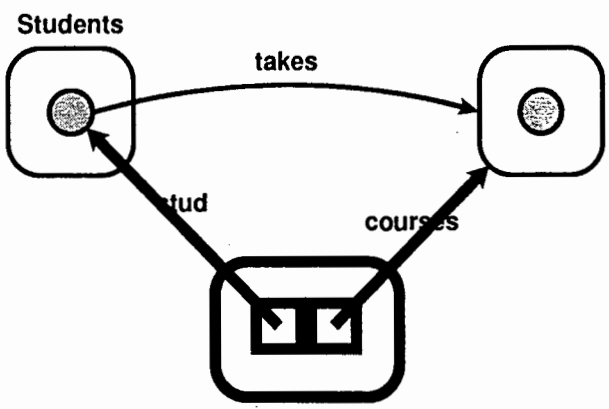


Figure 5.22: A collecting blob and an output structure node (full-labels turned off)

5.8 Disjunctions, Subqueries and Named Queries

The queries discussed thus far present their answers to the user once the query has been processed. *Subqueries* and *named queries* allow for the result of a query to be used in another query, rather than being presented to the user. Such a query can, for example, be used as a *disjoint component* in a query containing disjunctions of conditions. The term *disjoint component* is discussed in Section 5.8.1 but, briefly, a query with disjunctions is constructed by separating it into its disjoint

components. Using subqueries and named queries, queries can be constructed in a modular, piece-meal fashion.

The query fragments involved in disjunctions, subqueries and named queries are similar to each other in that they may be used long after their construction is complete. In order to define each of these, a *subquery node* (discussed in Section 5.8.2) is placed around the relevant nodes.

Boolean literals have not been implemented in QUIVER. In the current implementation (disjunctions of conditions have not been implemented either), therefore, a subquery or a named query may not have a boolean type as its output; this type is inferred if the subquery or named query contains no output nodes. In other words, an output inclusion group has to be present in a subquery or named query.

5.8.1 Disjunctions

Thus far, only conjunctions of conditions have been discussed. For example, if a circle node representing a Student object has two property edges leaving it, then the conditions associated with both edges have to be satisfied for a data instance to match the pattern.

Disjunctions of conditions are specified by separating a query into *disjoint components* and placing output nodes in each of the components. The outputs of the disjoint components have to be of identical types; queries containing disjunctions cannot be translated if their output types differ.

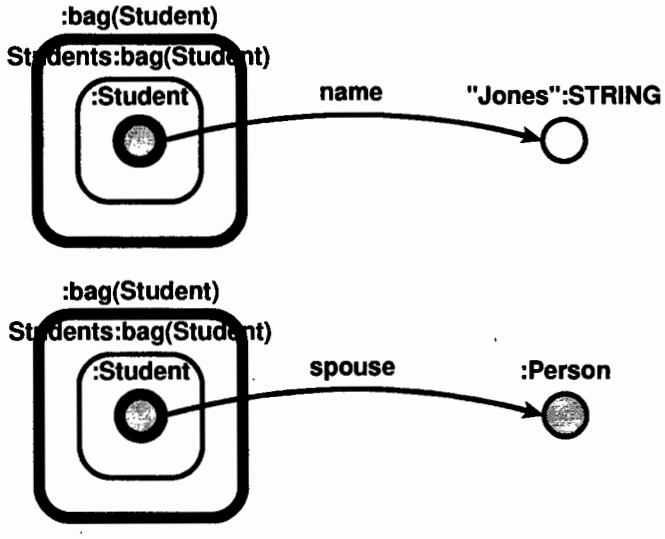


Figure 5.23: A query with disjunctions

The query in Figure 5.23 consists of two disjoint components. The output of each of the components is a bag of Student objects. The query finds all those students whose names are equal to Jones *or* who have spouses.

Note that the query in Figure 5.23 does not have subquery nodes around the disjoint components. In this case, disjoint components are implied as the query contains many root output inclusion groups; the subquery nodes are not necessary. If the disjoint components of a query are constraint queries, they have to be surrounded by subquery nodes.

As mentioned, the disjunction of conditions has not been implemented in QUIVER.

5.8.2 Subqueries

The execution of a subquery is similar to the execution of a function (functions were discussed in Section 5.6) – code is executed in both of these. In a function, the code is specified in some programming language; in a subquery, however, the *code* is a complete query, and can even consist of other subqueries. A subquery does not have any explicit input as a method does, thus it does not have a data-flow edge entering it. Just as with functions and methods, a data-flow edge leaving a subquery represents the data (the answer of the subquery) flowing out of it.

The user is able to construct and edit subqueries, hence the closed box notation used to represent functions and methods is unsuitable to represent subqueries. Subqueries, by default, are represented by an open box framing the inclusion groups that define it.

Figure 5.24 is an example of a query containing a subquery. The subquery is computationally identical to the query in Figure 5.20. The query finds all students who take at least one course. The output of this subquery is represented by the node at the head-end of the data-flow edge leaving the subquery frame – the output is a bag of Student objects. The outer query then finds the name values of these Student objects.

The subquery in Figure 5.24 is independent of the surrounding query. This is not the case in the query in Figure 5.25. Inside the subquery in Figure 5.25, all students who take at least one course are found. The bag of Course objects taken, however, has to contain a Course object that is fixed outside the subquery – the equality edge crossing the subquery frame asserts this. The output of the subquery is the bag of Student objects that take at least the single Course object that was fixed outside the subquery. The output of the subquery is asserted equal to the extent of Student objects (the Students named value). If the two bags are equal, the query succeeds.

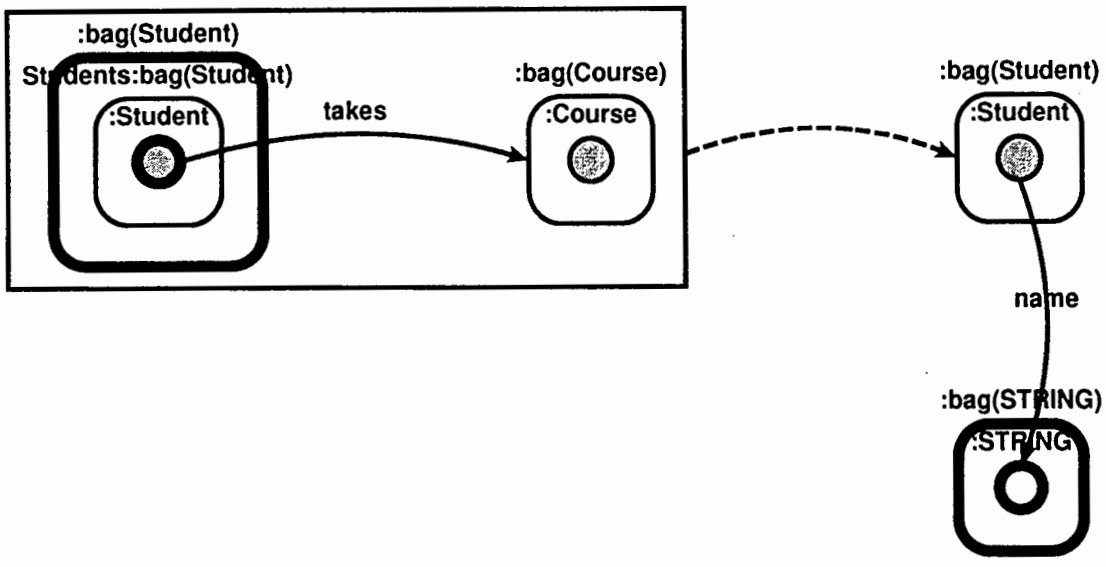


Figure 5.24: A query containing a subquery

The query thus finds the courses that are taken by *all* students – the output is a bag of `Course` objects.

A number of restrictions exist on edges that cross subquery frames. These restrictions are discussed in Chapter 7.

5.8.3 Collapsing subqueries

In order to simplify the display of subqueries, the query language design includes the ability for the user to *collapse* subqueries so that the "code" of the subquery is not visible. A similar notation is used to display a collapsed subquery as was used to display functions and methods, except that the square node is displayed screened rather than closed. Figure 5.26 displays the query in Figure 5.25 with the subquery collapsed. The equality edge that crossed the subquery frame now has its one end (the end that was previously attached to the `Course` node inside the subquery) in the middle of the screened subquery node, indicating that it was part of the subquery and that its position cannot be accurately shown with the subquery appearing in its collapsed form. The edge will appear in its correct position when the subquery is displayed in its *expanded* form again.

The suffix of the label of the collapsed subquery node indicates that it is a mapping with range of type `bag(Student)`. The domain of the subquery is, implicitly, the entire database and the domain of the label appears blank. If full-labels are turned off, a collapsed subquery will have a blank label. The label of a collapsed subquery node resembles the label of a parameterless method (Figure 5.14(b)) as neither has any explicit inputs.

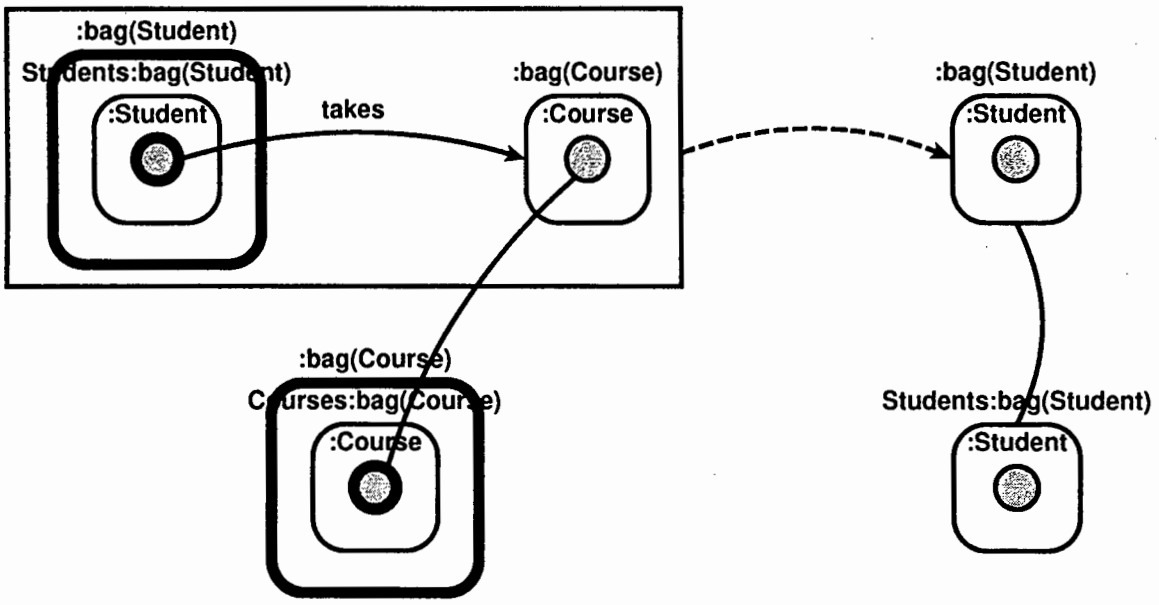


Figure 5.25: Another query containing a subquery

The collapsing of subqueries has not been implemented, and Figure 5.26 was not produced by QUIVER.

5.8.4 Named queries

Named queries are a variation of subqueries, and they facilitate, to a greater extent, the modular construction and the sharing of queries. Named queries allow the user to design a set of queries and store these in a *query library*. Whenever one of these queries is needed, it is retrieved from the library and inserted into the current query in a *cut-and-paste* action. Queries may thus be shared not only between queries, but between database users as well.

An alternative use for named queries is for the database administrator to design a set of queries for use by the database users. The queries may be sensitive in nature, containing property values that need to remain secure such as passwords and salaries. These named queries may then be used by others, but may not be edited or viewed by anyone other than the database administrator.

In its *expanded* view (applicable only where the user is permitted to view the expanded form of the query), a named query appears identical to a subquery except that it includes its name. A named query has its name appearing as the prefix of the inclusion group at the head of its outgoing data-flow edge. In its collapsed view, a named query appears as in Figure 5.27. The figure shows the subquery in Figure 5.24, appearing as a collapsed named query. The collapsed view is represented by a dashed circle inside a dashed blob. The nodes (as with

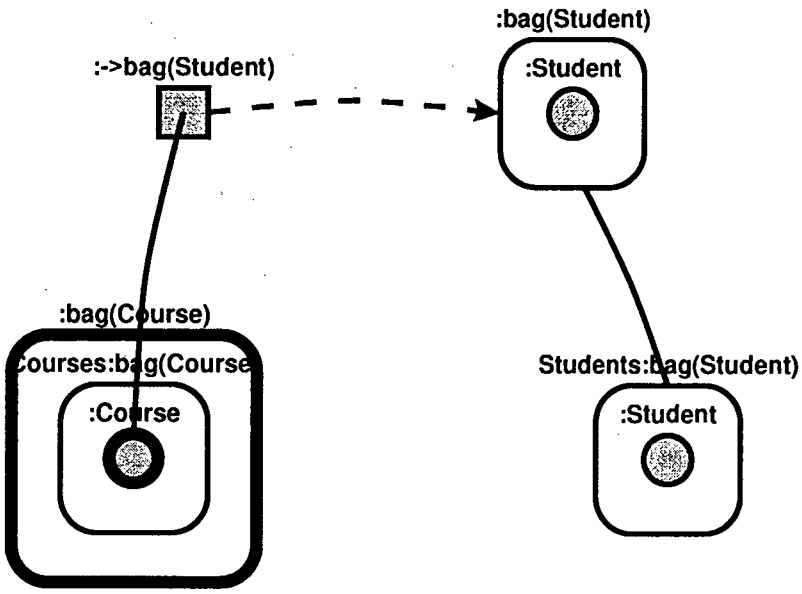


Figure 5.26: Collapsing a subquery

parameter-less methods in Section 5.5.2) are drawn dashed to indicate the data flow from the named query to the value of the named query. The prefix of the node contains the subquery name, as this is the name by which the named query is referenced.

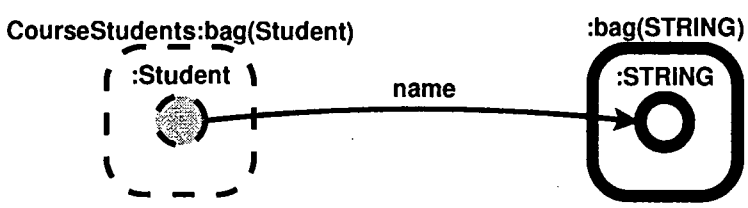


Figure 5.27: A collapsed named query

A named query may not contain an edge which connects any of its nodes to any node outside it. A named query can only be connected to parts of the query outside it through its output node, or if the named query is displayed collapsed, through the dashed node. Just as with subqueries, though, named queries may contain subqueries and other named queries. Functionally, named queries are thus more restrictive than nested queries.

Named queries have not been implemented, and the query in Figure 5.27 was not produced by QUIVER.

5.9 Inequalities

The query in Figure 5.28 presents a query with an inequality condition. An inequality edge (labeled with the inequality condition it represents) is placed from the age property of the Student object to a literal with value 25. The query succeeds (and returns TRUE) if any student is younger than 25 years of age.

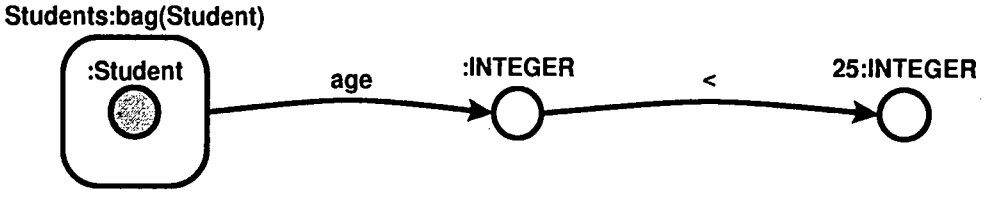


Figure 5.28: Query with an inequality

The inequality conditions "<", "≤", ">" and "≥" may be used with all defined literal types. Only compatible literal types, however, may be compared to each other. Inequalities have not been implemented, and the query in Figure 5.28 was not produced by QUIVER.

5.10 Not checking for `nil` data

As mentioned in Section 5.3, QUIVER checks that an object is *non*-`nil` if it appears in the construction of a query. The query in Figure 5.29 contains no object node; it just contains a collection blob representing the `Students` named value. The assertion that at least one element of `Students` is *non*-`nil` is not performed.

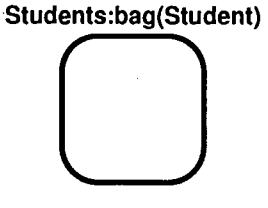


Figure 5.29: Not checking for `nil` objects

This query returns TRUE if `Students` is defined, regardless of how many elements it has or if its elements are `nil` or *non*-`nil`. This feature has not been implemented, and the query in Figure 5.29 was not produced by QUIVER.

5.11 Hierarchy of QUIVER Constructs

The diagram in Figure 5.30 displays a hierarchy of all QUIVER visual query constructs. Just like the hierarchy diagrams in Sections 4.2, instantiable types are shown upright and abstract types are shown *italicised*.

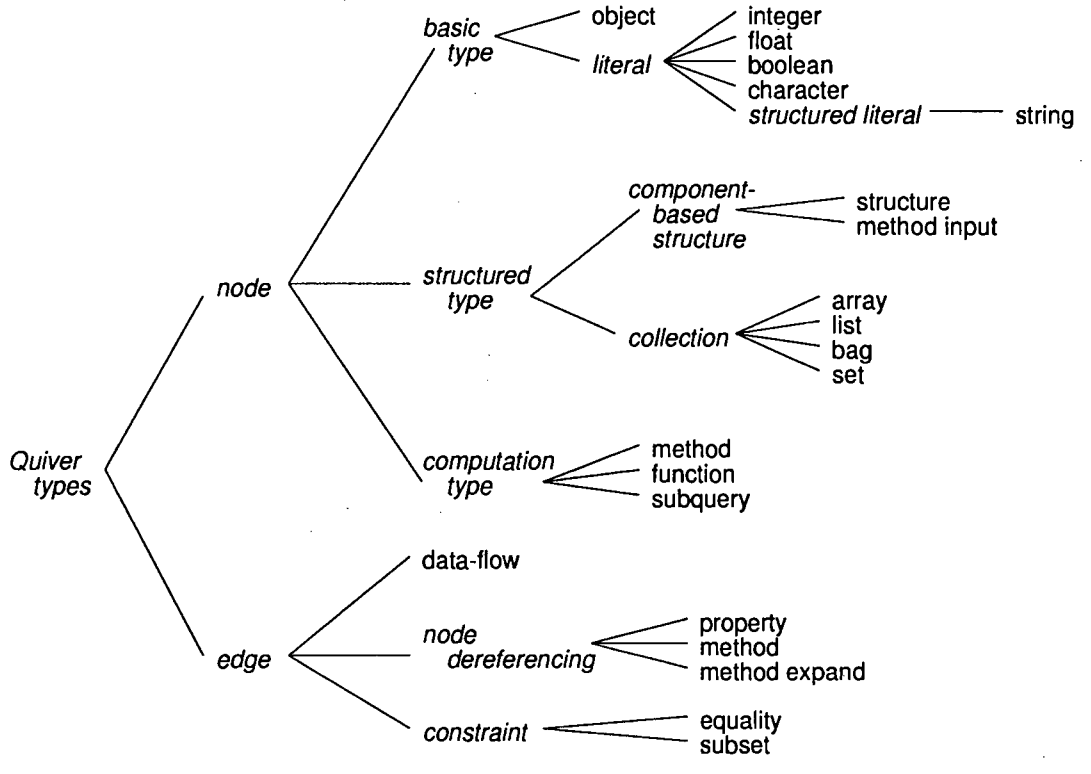


Figure 5.30: Type hierarchy in QUIVER

As mentioned, one of the goals of the QUIVER language is that it should be consistent; similar query concepts should be represented by similar looking and acting query constructs. The types in Figure 5.30 are organised by their functional similarity; the grouped types also have similar visual constructs representing them. There are two main types in the hierarchy, namely node types and edge types.

All basic data types are represented by circle nodes. Objects are represented by screened circles and literals by open circles. A component-based structure is represented by a rectangle; it appears screened if its details are hidden or open to display the cells that represent its components. Collection types all contain blobs or modified blobs in their visual representations and computation types are all represented by rectangles; methods and functions appear as closed squares and subqueries, when not collapsed, appear as open rectangles.

Although structures and computation types both appear as rectangles, visual ambiguity between these symbols can never arise. If a rectangle appears screened,

or appears open and contains cells, it has to be a component-based structure. If a rectangle appears closed, it is either a method node or a function node and if it appears open and does not contain cells, it represents the open view of a subquery or named query.

Data-flow edges are represented by dashed, unlabeled directed edges. Node dereferencing arcs, unlike constraint edges, are defined in the schema and are all represented by solid labeled edges. Constraint edges are represented by solid unlabeled arcs.

The visual constructs in QUIVER make minimal use of colour. As mentioned, this aids in making QUIVER usable across a broad range of hardware. Alternatively, this allows colour to be used in domain-specific contexts. Objects and literals are represented by circles; the database administrator could use colour icons to represent them instead, thus increasing the usability of QUIVER. The use of icons to represent objects and literals is further discussed in Chapter 6.

The hierarchy in Figure 5.30 serves both as a summary of QUIVER constructs as well as an indication of the commitment to visual consistency and a minimal use of visual constructs in QUIVER's design.

Chapter 6

Implementation of QUIVER

6.1 Introduction

This chapter describes the different processes that make up the QUIVER application. Figure 6.1 presents a diagrammatic view of the flow among these processes. The arrows between the components represent the flow of data. The data presenter, for example, requires input from O_2. The shaded boxes represent components that were not developed as part of this thesis, but already existed and were sourced.

As mentioned, O_2 (from O_2 Technology [34]) is used as the persistent store of QUIVER, and the associated O_2Look application is used to display data instances returned from a query. DOT [27] is used for graph layout.

Schema information from O_2 is retrieved using the schema parser which is discussed in Section 6.2. The Unix tools `lex` and `yacc` are used to parse the grammar of an O_2 ODL file.

The GUI was first mentioned in Chapter 2 and is discussed in Section 6.3. All of the graphical user interface is implemented using the University of Berkeley tools `Tcl` and `Tk` [38], and consists of approximately 15,000 lines of `Tcl/Tk` code. The query translation engine translates the graphical query constructed by the user to OQL; the OQL string is then transferred to O_2 for execution. The translation engine is explained in Chapter 7. The interface to the O_2 database is discussed in Section 6.4.

Graphs can be saved to and retrieved from a "query library"; each query is saved in a disk file. This file format, as well as the file formats used to communicate with the DOT graph placement engine are discussed in Section 6.5.

`C++` is used as the "glue" for these components, calling and managing the interfaces between each of them. The query translation, being data structure and

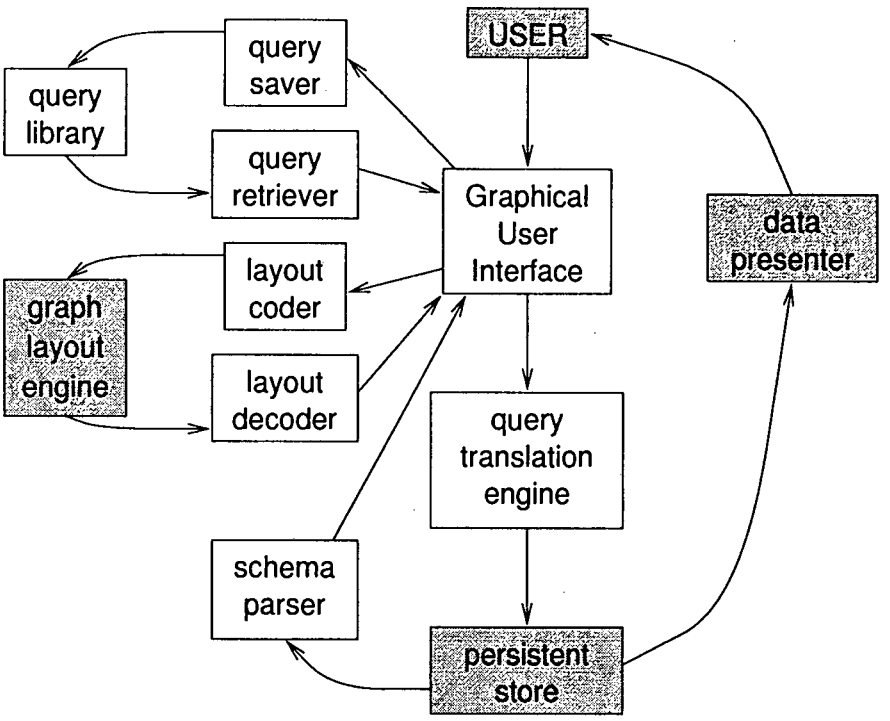


Figure 6.1: Process diagram representing QUIVER

computation intensive, is also implemented in C++; QUIVER consists of about 10,000 lines of C++ code.

6.2 Parsing an ODL Schema

QUIVER does not support the full grammar of an O_2 ODL file. Certain syntactic elements that are valid in O_2, therefore, are invalid in QUIVER. The following lists the features of O_2 that are not implemented:

- O_2 permits the importing of classes and types from other schemas; this is not permitted in QUIVER. All classes and types used in a schema have to be defined within that schema.
- Application-defined functions are not supported and the definition of these is ignored. Only the built-in aggregate functions `element`, `sum`, `count` and `bag` are currently supported.
- *Integers*, *characters* and *strings* are supported. *Boolean* literal types, *real numbers* and *bit strings* (bit strings are similar to strings, but may contain `NULL` characters) are not supported.

- O_2 supports the definition of "literal objects" and "collection objects". This permits the definition of, for example, mutable literals. QUIVER does not support these definitions. The following example defines a type `mutinteger`, which is a mutable integer: "`class mutinteger type integer end;`". QUIVER supports classes that are tuple-typed only.
- QUIVER supports only the *bag* collection type. *Lists*, *arrays* and *sets* are not supported.

QUIVER does support method and property renaming, as well as allowing a class to have multiple superclasses. When querying an object, all its properties and methods are always available, regardless of their visibility types (`private`, `public` or `read`). The visibility of methods and properties has no impact on querying, and is ignored while parsing.

The BNF grammar of the parser is listed in Appendix A. As mentioned in Section 4.3, an array is interpreted as a list in O_2, thus the ODL BNF does not contain the keyword `array`. In addition, enumerated types are implemented as integers in the O_2 ODL.

6.3 Graphical User Interface

The query language consists of ten types of nodes and eight types of edges. Two of the node types are *reachable blobs* and *collecting blobs*. Figure 6.2 lists each of the remaining eight types of nodes and the representation type (*open*, *screened* or *closed*) implemented for the node. Only one representation type has been implemented for each node type. The table also lists whether the node can appear within reachable blobs (the RB? column) and within a collecting blob (the CB? column); QUIVER allows only syntactically correct blobs to be placed around nodes.

Syntactical correctness is also aided by ensuring that unnamed literals, when placed using the *literal* button (○), *always* appear bold. If a literal-typed property of an object or structure needs to be set, that literal is drawn as a property value, and it does not appear bold by default. If a query consists solely of an unnamed literal, however, QUIVER infers that the output of the query is the literal value itself. Whenever the user places an unnamed literal that is not a property value, therefore, it appears bold. This can subsequently be modified by the user.

Figure 6.3 lists the types of edges and their functions.

Node name	Representation type	RB?	CB?
object (represents an object)	screened	√	√
literal (represents a literal value)	open	√	√
structure (represents a property consisting of a structure value)	screened	√	√
method (represents a method call)	closed	×	×
function (represents a function call)	closed	×	×
method input (represents the input elements of a method parameter)	open	√	×
output structure (denotes a query output structure)	open	×	√
subquery (for marking a subquery)	open	×	×

Figure 6.2: Types of query nodes

6.3.1 Type-correctness of queries

Edges are placed on the QUIVER canvas in one of two ways: either the user places the edge between two nodes that were constructed previously (between two *existing nodes*), or the user selects an edge type and a node and QUIVER generates the edge and the inclusion group to represent the other end of the edge. In some cases, the user selects the target of the edge and QUIVER generates the edge and the *source* inclusion group, and in other cases QUIVER generates the edge and the *target* inclusion group. When placing an edge between two nodes, QUIVER ensures that the two nodes are of compatible node types. When placing an equality edge, for example, QUIVER does not allow the user to place the edge between a literal-*cored* node and an object-*cored* node:

Definition 6.1 *The* core *of a node refers to the type of the innermost node that it contains (or the type of the node itself if it contains no other nodes). The node is said to be* cored *of that type. Inclusion groups, naturally, also have core types.*

When creating edges and inclusion groups simultaneously, nodes at both the source and target ends of the edge are always correctly typed.

The table in Figure 6.4 describes the behaviour of QUIVER when placing edges between two existing nodes. For each edge type that can be placed this way, the core of the node that could be selected first is listed in column `node 1` and the cores of the nodes that have to be selected second, in column `node 2`. The `direction` column states whether the edge has node 1 as its source node ($1 \rightarrow 2$)

Edge name	Edge representation
property edge	represents an object or structure property
output edge	defines the components of an output structure node
subset edge	asserts that one collection is to be the subset of another
equality edge	asserts that one node is to be equal to another
data flow in edge	represents the flow of data into a method or function
data flow out edge	represents the flow of data out of a method, function or subquery
method edge	connects an object to any method associated with it
method expand edge	connects a method node to a method input node and a method input node to its inputs

Figure 6.3: Types of edges

or node 2 as its source node (2 $\rightarrow$ 1). Equality edges do not have `direction` values as they are not directed.

The table in Figure 6.5 lists the behaviour when edges and inclusion groups are created simultaneously. For each edge type, the `first node` column gives the node type that can be selected first. The `direction` column states whether the edge has the first node as its source and the new created node as its target (`first` $\rightarrow$ `new`) or the first node as its target (`new` $\rightarrow$ `first`).

The data flow edges in Figures 6.4 and 6.5 have been described relative to subquery, method and function nodes. A *data flow in* edge is the data flow edge that has its head connected to a method node or a function node, and a *data flow out* edge has its tail connected to a method, function or subquery node. From Figures 6.4 and 6.5 we see that if a data flow in edge is placed, the first node that is clicked on is either a method node or a function node. If it is a method node, the method input type is generated by the system; if it is a function node, however, the node representing the input has to exist on the canvas. This behaviour exists (as mentioned in Chapter 5) because a shorthand means of specifying the input to a function has been implemented.

The rules described in the tables are checked during query construction; the user thus receives real-time feedback while placing edges. The following checks, however, are performed just before the query is translated:

- If an equality edge or a subset edge has been placed between two object-cored nodes, the objects have to be of the same type or one has to be a subtype of the other. If one is placed between two literal-cored nodes, the literals have to be of exactly the same type and if placed between structure-cored and method input-cored nodes, the nodes need to have identically

Edge type	Core of node 1	Core of node 2	Direction
output edge	output structure	object, literal, structure or output structure	$1 \rightarrow 2$
subset edge	object	object	$1 \rightarrow 2$
subset edge	literal	literal	$1 \rightarrow 2$
subset edge	structure	structure or method input	$1 \rightarrow 2$
subset edge	method input	method input or structure	$1 \rightarrow 2$
equality edge	object	object	-
equality edge	literal	literal	-
equality edge	structure	structure or method input	-
equality edge	method input	method input or structure	-
data flow in	function	object, literal, structure or method input	$2 \rightarrow 1$

Figure 6.4: Placing edges between two existing nodes

named and typed components. The ranks of the two nodes also have to be equal. The rank of a node is formally defined in Chapter 7 but, briefly, the rank of a node refers to its position within its inclusion group. In addition, a subset edge can only be placed between two collections and no edges may be attached to a collecting blob.

- When placing the *data flow in* edge of a function, the type of the node that was selected as input has to match the type of the function parameter.

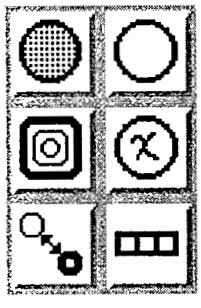
6.3.2 Icon buttons

The icon buttons in the interface are used to activate certain actions, such as placing a node or deleting an edge. The icons are arranged in five logical groups in the icon bar on the left of the query window. These groups are the *node*, *text modification*, *edge*, *function–method–subquery* and *deleting–node-placement* groups.

Each of these groups is now described:

Edge type	First node	Direction	New core type
property edge	object	first → new	object, literal or structure
property edge	structure	first → new	object, literal or structure
data flow out	method, function or subquery	first → new	object, literal or structure
data flow in	method	new → first	method input
method edge	object	first → new	method
method expand edge	method input	first → new	method input, object, literal or structure

Figure 6.5: Creating inclusion groups while placing edges

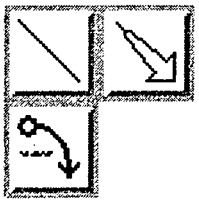


Node icons are used to place or modify nodes on screen. The icons (left to right, top to bottom) are

1. place an unnamed object
2. place an unnamed literal
3. place a collecting blob around a node
4. place a named value (object or literal value)
5. change the *bold* state of an item
6. place an output structure

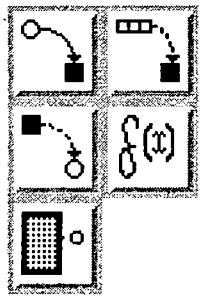
Text modification icons are used to alter the appearance of text labels.

1. change the text label of a literal
2. turn *full-labels* on or off (toggle switch)



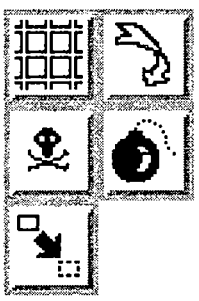
Edge icons are used to place and alter the appearance of edges.

1. place an equality edge between two nodes
2. place a subset edge between two blobs
3. place a property edge



Function, method and subquery icons are used to place functions, methods and subqueries, and to set the inputs and output of these nodes.

1. place a method of an object
2. place the input of a method/function
3. place the output of a method/subquery/function
4. place a function node
5. mark a subquery



Deleting and node-placement icons are used to move and delete nodes.

1. call the layout engine to layout all inclusion groups
2. undo the last layout
3. delete a node/edge
4. delete all items on canvas
5. move an inclusion group

Most of the buttons remain active until another button is activated; this action automatically deactivates the first button. The *toggle full-labels* button ( ) is an exception, as it may be activated or deactivated regardless of whether other buttons are active or inactive. Full-labels are turned on as long as this button is active. In addition, certain buttons such as the *layout* button ( ) and *undo layout* button ( ) automatically deactivate themselves after their action is complete.

When the user places a method node, the next action is usually to place the

inputs of the method. After the user places a method using the *add method* button ( ), therefore, the *method input* button ( ) is automatically activated and the user may immediately place the method inputs.

6.3.3 Drawing edges

When drawing edges, QUIVER draws some edges curved and some edges straight. Constraint edges are drawn curved, so that more than one edge may be placed between two inclusion groups without the edges overlapping. If one imagines the two inclusion groups (between the edge being placed) vertically aligned with the inclusion group created first below the one created second, then the first edge is placed to the *left* of an imaginary line joining the centers of the inclusion groups. The second edge placed is placed to the right, the third edge is also placed to the left, but to the *outside* of the first edge, and so forth. Alternatively, if the nodes are horizontally aligned with the inclusion group created first to the left, the first edge is placed above the imaginary line joining their centers.

A query containing multiple edges appears in Figure 6.6. The query contains a property edge (labeled `children`) and a subset edge between the two `Person`-cored blobs. Output edges are always placed straight as an output structure cell always has a single edge leaving it. Method expand edges (not shown in Figure 6.6) are also drawn straight for the same reason.

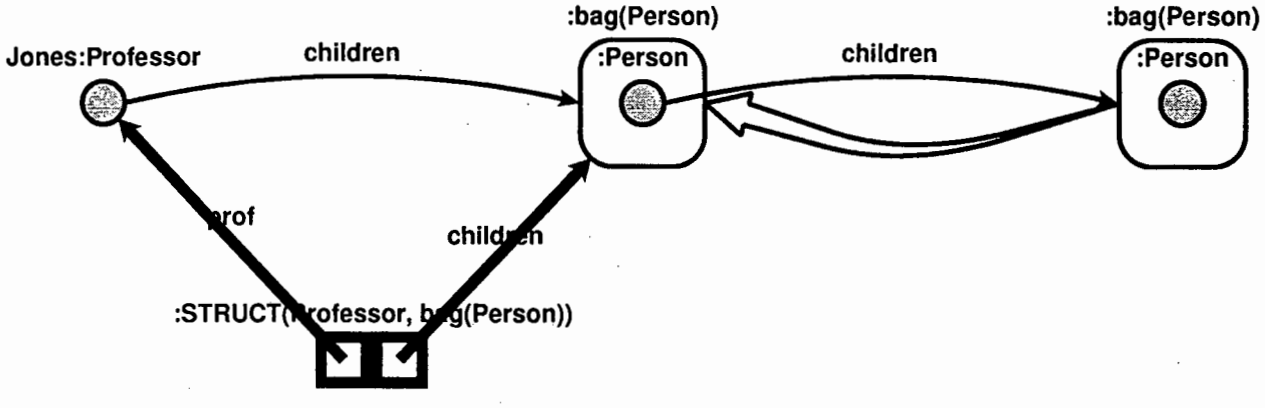


Figure 6.6: A query with multiple edges

In order to place output edges, the user activates the *add property* button ( ), and then clicks on the output structure node or another output edge. If the user clicks on the output structure node, a new cell is created as the *first* component of the output structure node and the tail of the new output edge is attached to that cell. If an output edge is clicked on, a new cell is created *after* the cell attached to the tail of the edge, and the tail of the new output edge is attached to the new cell. Deleting an output edge automatically adjusts the number of

cells in an output structure node. In this way, the order of the components of an output structure can be specified by the user.

6.3.4 Graphical user interface options

If the user hits the CONTROL-O key while running QUIVER a *dialog box* allowing the user to configure run-time options is displayed. The dialog box is displayed in Figure 6.7.

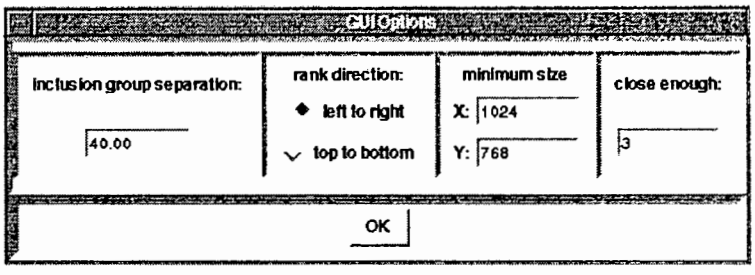


Figure 6.7: GUI Options

The user is allowed to set four options, namely the *inclusion group separation* factor, *rank direction*, *minimum size* of the canvas and a *close enough* factor. The inclusion group separation factor and the rank direction affect the automatic layout of inclusion groups (when hitting the *layout* button –). The separation factor allows the user to set the minimum distance between adjacent inclusion groups, and the rank direction lets the user choose between a *left to right* graph orientation and a *top to bottom* orientation.

As mentioned in Chapter 2, the scroll bars of the canvas allow it to represent a larger area than the canvas occupies on screen. When laying out a large graph, the size after layout may be much larger than the size before layout and the canvas size is adjusted automatically. When laying out a "small" graph, the size of the canvas does not fall below the *minimum size* values.

The *close enough* factor is used to aid the "clicking" of nodes and edges. It indicates the space (in pixels) that may lie between the mouse cursor and a node or edge for QUIVER to consider that query item to be under the cursor. A high number allows the user to move the cursor near the item rather than exactly on it for it to be highlighted, but may result in an incorrect node or edge being highlighted.

6.4 The O_2 Interface

If the user hits CONTROL-C while in QUIVER, a dialog box pops up allowing the user to choose a schema and base to query. The names of the schemas and bases are retrieved from the O_2 database by executing the command "o2 display stat". The output is redirected to a file and parsed for all schema names and the bases associated with each schema.

Saving a schema to disk is done by executing the "save schema" command. To save the ue schema to a file o2_schema_ue, the following command is executed: "save schema ue o2_schema_ue". This file is then parsed to obtain all class and type information.

Calling O_2 to retrieve the names of the schemas, bases and the schema definition each time QUIVER is executed represents a significant time overhead. The names of the schemas, bases and all the definitions are therefore stored on disk, and the user may *reget* the information (using CONTROL-R) only when necessary.

O_2Look is invoked when displaying the instances of a query answer, and the O_2 code used to do this is displayed in Figure 6.8.

```
set base base name;
query;
EOF
OQL query
EOF
quit
EOF
quit;
```

Figure 6.8: Using O_2Look to display query answer instances

Base names are unique across schemas, thus there is no need to set a schema first and then identify a base within that schema.

6.5 File Formats

This section describes the file formats used by QUIVER and its related applications. Section 6.5.1 describes the file format used by QUIVER to save and retrieve queries to and from disk. Sections 6.5.2 and 6.5.3 describe the file formats used to communicate with the DOT graph placement engine. Section 6.5.2 describes the format used when defining the graph in DOT and Section 6.5.3 describes the format of the output from DOT.

6.5.1 QUIVER graph file format

In the QUIVER graph file format, the information about each inclusion group appears in sequence according to the internal numbering of the inclusion groups; internal numbering is discussed in Chapter 7. Information about *all* the inclusion groups appears before the edge-related information appears. Each inclusion group is saved according to the format displayed in Figure 6.9.

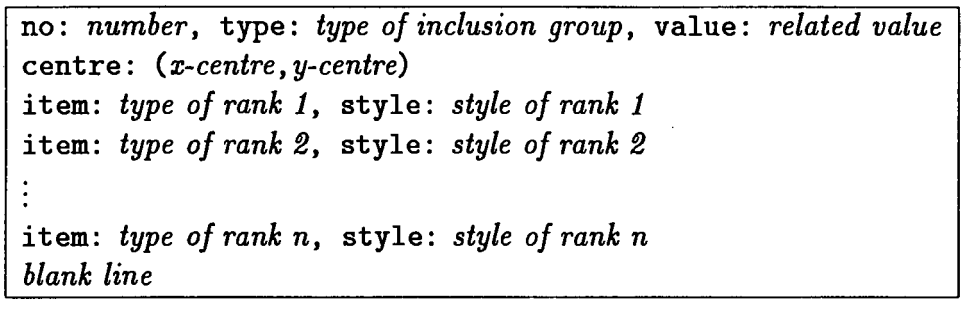


Figure 6.9: File format for an inclusion group

As displayed in the figure, an inclusion group may have many nodes, and information about an inclusion group is terminated by a blank line. Each inclusion group has a unique number (appearing in the *number* field), and the *type of inclusion group* field gives the type of the core node (such as `object`, or `integerliteral`). The *related value* supplies addition information relating to the state of the inclusion group, such as an object name or an integer value. The *x-centre* and *y-centre* fields give the coordinates of the centre of the inclusion group, relative to the top left corner of the canvas.

Each node of the inclusion group has an associated style; the style may either be `1` to indicate that the node appears bold, or `0` for non-bold. The type of the core node is related to the type of the inclusion group; an `object` inclusion group, for example, always has a `circle` as its core node.

The first line after the centre of the inclusion group corresponds to the core node; the next line corresponds to the node that has rank two (i.e. the node that visually includes the core and no other node), and so forth. The non-core nodes may either be reachable blobs or collecting blobs; the style of a reachable blob may either be `0` or `1`, but the style of a collecting blob is always `1`.

Edges appear after the inclusion groups, and each edge follows the format as displayed in Figure 6.10. Each edge also has a unique number associated with it. In most cases, the numbers are not used by QUIVER, but the identification numbers of *output edges* are used in the *related value* fields of output structure nodes – the sequence of the edges is stored with the output structure information.

The type of the edge is stated next, and the *related value* field again supplies additional information. A property-typed edge, for example, contains the name of the property in its *related value* field.

The `si` and `ti` fields contain the identification numbers of the source and target inclusion groups. The `sr` and `tr` fields contain the rank numbers within those inclusion groups to which the edge is attached. All edges are saved as having *sources* and *targets*, even equality edges. When saving an equality edge, the `si` and `ti` fields merely indicate the two inclusion group to which an equality edge is attached.

`no:` *number*, `type:` *type of edge*, `value:` *related value*
`si:` *source inclusion group*, `ti:` *target inclusion group*,
`sr:` *source rank*, `tr:` *target rank*
blank line

Figure 6.10: File format for an edge

The figure displayed in Figure 6.12 lists the saved graph corresponding to the query in Figure 2.10, page 13. The query is displayed with its internal numbers in Figure 6.11. Some edges do not require any related values, such as the data flow edge (edge `Methodoutputedge`, number 10), the equality edge (edge `EEedge`, number 11) and the subset edge (edge `FEedge`, number 12), and this field is left blank in each case.

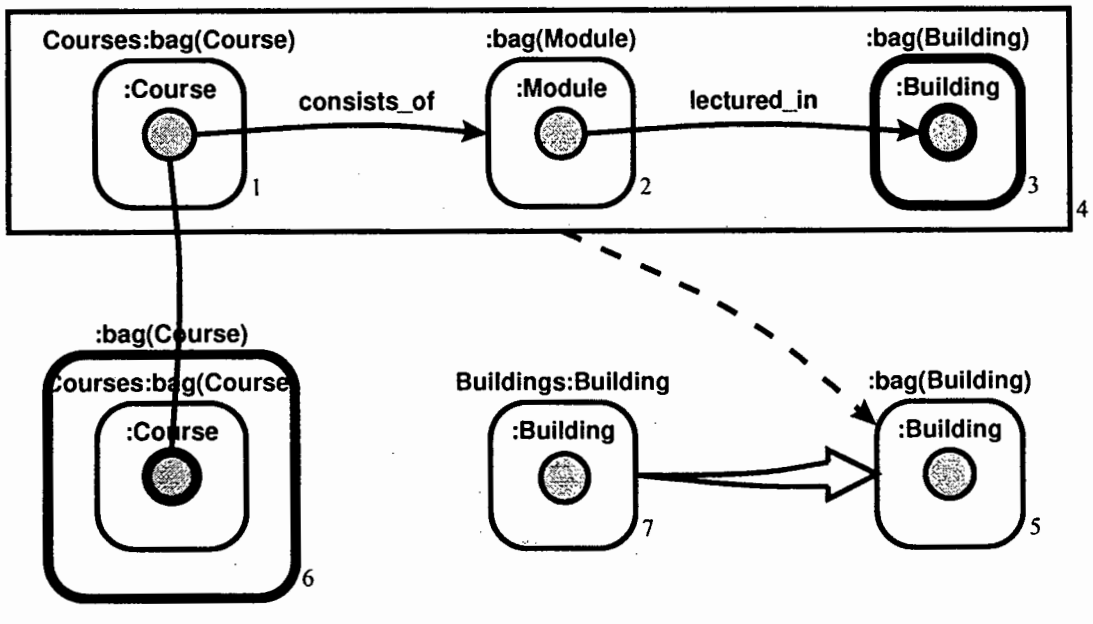


Figure 6.11: A query with a subquery

```
no: 1, type: Object , value: Courses
centre: (100.0,100.0)
item: circle , style: 0
item: blob , style: 0

no: 2, type: Object , value: Module
centre: (300.0,100.0)
item: circle , style: 0
item: blob , style: 0

no: 3, type: Object , value: Building
centre: (500.0,100.0)
item: circle , style: 1
item: extrablob , style: 1

no: 4, type: Subquery , value: 1 2 3
item: rect , style: 0

no: 5, type: Object , value: Building
centre: (500.0,300.0)
item: circle , style: 0
item: blob , style: 0

no: 6, type: Object , value: Courses
centre: (100.0,300.0)
item: circle , style: 1
item: blob , style: 0
item: extrablob , style: 1

no: 7, type: Object , value: Buildings
centre: (300.0,300.0)
item: circle , style: 0
item: blob , style: 0

no: 8, type: Attribute , value: consists_of
si: 1, ti: 2,  sr: 1, tr: 2, style: 0

no: 9, type: Attribute , value: lectured_in
si: 2, ti: 3,  sr: 1, tr: 1, style: 0

no: 10, type: Methodoutputedge , value:
si: 4, ti: 5,  sr: 1, tr: 2, style: 0

no: 11, type: EEedge , value:
si: 1, ti: 6,  sr: 1, tr: 1, style: 0

no: 12, type: FEedge , value:
si: 7, ti: 5,  sr: 2, tr: 2, style: 0
```

Figure 6.12: File format for a query graph

A graph file is intended to be created and used through QUIVER, not directly by the user. If the user does create or edit a file, however, the user is not required to calculate any coordinates in order to place edges or blobs; the only coordinates that need to be specified are the centres of the inclusion groups. This makes a graph file relatively easy to create and maintain.

6.5.2 Saving a graph to DOT

A DOT graph contains nodes and edges. DOT nodes may not contain other nodes (except DOT subgraph nodes, which are explained later) as QUIVER nodes visually include other nodes; a DOT node thus represents a QUIVER inclusion group. If an edge in QUIVER is drawn from any node in inclusion group x to another node in inclusion group y, it is represented by a DOT edge drawn from the border of the node representing x to the border of the node representing y. In addition, QUIVER determines from an edge's types if it is to be drawn staight or curved; to simplify translation, therefore, DOT is also instructed to compute the layout as if all edges are drawn straight.

Figure 6.13 displays the DOT input graph of the query in Figure 6.11. The `weight` attribute is always 100, and indicates that all edges are to be computed as straight edges by DOT. The `nodesep` and `ranksep` attributes are equal to

the *inclusion group separation* value mentioned in Section 6.3.4. The `rankdir` attribute is either `LR` (to indicate a left-right graph orientation) or `TB` (to indicate a top-bottom orientation).

```
digraph quiver {
weight=100.00
nodesep=40.00
ranksep=40.00
rankdir=LR
subgraph sub4 {
node1
node2
node3
}
node1 [shape=box, width=127, height=78];
node2 [shape=box, width=77, height=78];
node3 [shape=box, width=82, height=81];
node5 [shape=box, width=82, height=78];
node6 [shape=box, width=127, height=119];
node7 [shape=box, width=135, height=78];
node1 -> node2
node2 -> node3
node1 -> node5
node2 -> node5
node3 -> node5
node1 -> node6
node6 -> node1
node7 -> node5
}
```

Figure 6.13: Graph input to DOT for the query in Figure 6.11

If the query contains a subquery, as the query in Figure 6.11 does, the DOT nodes representing the subquery inclusion groups are defined to exist within a *subgraph*; a subgraph groups nodes within a rectangular region, as required by a subquery. After these attributes are stated, the names and sizes of the nodes are stated, and the edges between them are defined. The sizes of the nodes are calculated in screen pixels. DOT does not permit edges to be connected to a subgraph border – we decided therefore to represent an edge that is drawn from a subquery node to an inclusion group x by edges going from *each* inclusion group within the subquery to x. In the example, we cannot place an edge "`node4 -> node5`", and so we place edges "`node1 -> node5`", "`node2 -> node5`" and "`node3 -> node5`". An equality edge between nodes x and y is defined as an edge from x to y, and another edge from y to x; the graph input thus contains the lines "`node1 -> node6`" and "`node6 -> node1`".

6.5.3 Reading a graph from DOT

The output of DOT is redirected to a file, which is parsed to read the new coordinates of the inclusion groups. Figure 6.14 displays the DOT output graph of the query in Figure 6.11.

The definitions of the nodes appear first in the file, followed by the definitions of the edges; each line defining a node starts with "`node`" and each line defining an edge starts with "`edge`". The edges in QUIVER are computed automatically, and this section of the DOT output is ignored. The first line of the output gives a scaling factor, and the bounding box of the graph. The next few lines thereafter return information about each node; the coordinates of the nodes are relative to

```
graph 1.000 591.000 400.056
node node1  63.500 119.250 127.000 78.000 node1 solid box black
node node2  230.500 278.028 77.000 78.000 node2 solid box black
node node3  401.500 359.556 82.000 81.000 node3 solid box black
node node5  550.000 219.028 82.000 78.000 node5 solid box black
node node6  230.500 59.500 127.000 119.000 node6 solid box black
node node7  401.500 80.000 135.000 78.000 node7 solid box black
edge node1 node2 4 104.514 158.250 131.208 183.625 165.764 216.472 191.944 241.361 solid black
edge node1 node5 4 127.000 132.278 228.361 153.056 423.931 193.167 508.917 210.597 solid black
edge node1 node6 4 127.000 81.917 140.069 76.722 153.833 71.806 166.903 67.639 solid black
edge node2 node3 4 269.000 296.389 296.208 309.361 332.639 326.722 360.403 339.972 solid black
edge node2 node5 4 269.000 270.917 329.611 259.722 446.722 238.097 508.903 226.611 solid black
edge node3 node5 4 442.500 320.750 463.194 301.181 488.222 277.486 508.931 257.903 solid black
edge node6 node1 4 127.097 111.111 140.167 106.944 153.931 102.028 167.000 96.833 solid black
edge node7 node5 4 443.167 119.000 463.722 138.250 488.444 161.403 508.931 180.569 solid black
stop
```

Figure 6.14: Graph output from DOT for the query in Figure 6.11

the bottom left corner of the bounding box. Each line contains the node name, its *horizontal centre*, its *vertical centre*, its *horizontal size*, *vertical size* and other attributes. Only the *horizontal centre* and *vertical centre* coordinates of each node are used by QUIVER.

Chapter 7

Translating QUIVER Queries to OQL

7.1 Introduction

QUIVER translates a visual query into the standard object-oriented database query language OQL. This chapter describes the mechanisms of the query translation first by means of examples and then by a more general discussion. QUIVER is able to generate two types of queries:

1. A query that returns information *about* the database. Such a query returns a boolean value, depending on whether or not the query pattern exists at least once in the database. If the pattern does exist, `TRUE` is returned; `FALSE` is returned otherwise. Such a query is structured around the `exists` clause of OQL, and will be referred to as an `exists`-type query. This translation is performed if the user constructs a *constraint* query, as discussed in Section 5.3.

2. The second type is a query that returns data of the database. This translation is performed if the user constructs a query with output items, either with output as a fully determined node (Section 5.7.1) or output using a collecting blob (Section 5.7.2). This type of query is structured around the `select...from...where` clause of OQL, and will be referred to as `SFW`-type queries.

`exists`-type queries are discussed in Section 7.2, and `SFW`-type queries in Sections 7.3 and 7.4. As mentioned in Chapter 5 the full design has not been implemented; the translation computations presented here are correct for the current implementation only. More specifically, the query is assumed to consist

of one disjunctive component only. In addition, query translation will initially be discussed without consideration of subqueries. The translation of subqueries is discussed in Section 7.5.

A query is translated by processing all its inclusion groups and all its edges. Query subclauses are generated for each inclusion group, and these subclauses are assembled to form the final the query.

7.2 exists-type Queries

An exists-type query consists solely of query pattern constraints; it contains no output items. Assume that the query in Figure 7.1 was constructed by the user. The query is based on the schema in Figure 4.5 and is similar to the query in Figure 2.6, except that the inclusion group containing the Students named value has been replaced by the named Student object John; John has been added to the schema for this example. The query returns TRUE if John has scored a first for any course containing the Graphics1 module.

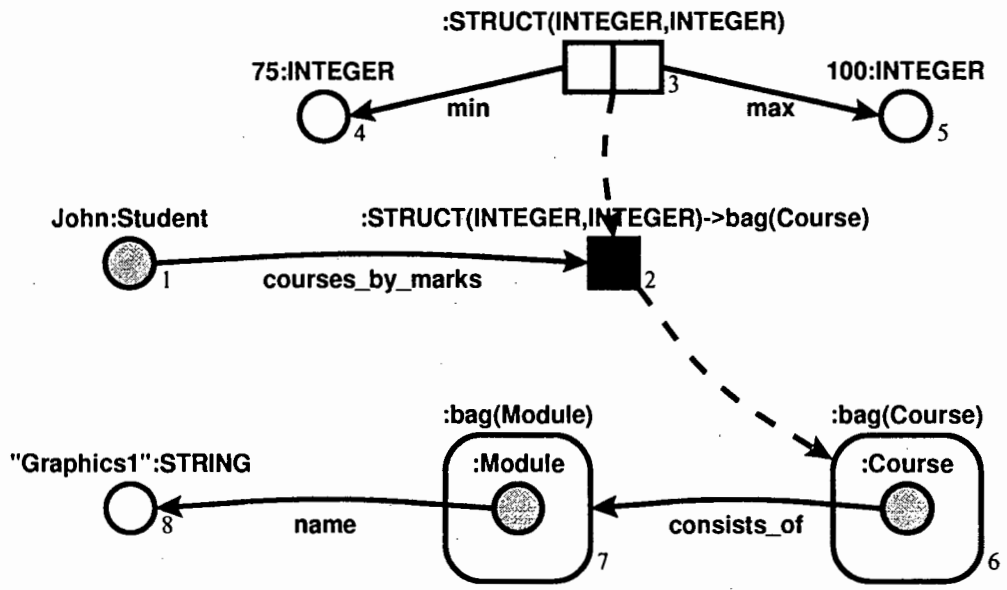


Figure 7.1: A boolean query in QUIVER

In the query, numbers have been placed at the bottom right corners of the inclusion groups in order to aid the discussion of the query generation. The OQL translation (or the *query string*) generated is shown in Figure 7.2.

Internally, QUIVER assigns numbers to the inclusion groups (the numbers that are displayed in Figure 7.1). The first inclusion group created by the user is assigned the number 1, the second 2, and so forth. A node is uniquely determined by the

```
exists VAR1_1 in bag(John): (John != nil and
exists VAR4_1 in bag(75): (
exists VAR5_1 in bag(100): (
exists VAR6_2 in bag(John.courses_by_marks(75, 100)): (
exists VAR6_1 in VAR6_2: (VAR6_1 != nil and
exists VAR7_2 in bag(VAR6_1.consists_of): (
exists VAR7_1 in VAR7_2: (VAR7_1 != nil and
exists VAR8_1 in bag("Graphics1"): (
VAR7_1.name = "Graphics1"))))))))
```

Figure 7.2: exists-type OQL query generated from the QUIVER query of Figure 7.1

inclusion group in which it appears and its position within the inclusion group. The position of a node within an inclusion group is referred to as the *rank* of the node.

Definition 7.1 *The* rank *of a node is its position within the inclusion group in which it appears. The innermost node of an inclusion group is assigned rank 1, the node that visually contains it and no other node is assigned rank 2, and so forth.*

A node will now be referred to by using the number of its inclusion group and its rank; these two numbers will be surrounded by angle brackets. The collection node of inclusion group 7, for example, is node <7,2>. Most nodes are assigned variable names, which are derived from this numbering. For example, node <7,2> is assigned variable name VAR7_2.

For each node, QUIVER generates an *alias*. Certain nodes also produce *collection*-subclauses and some of these produce, in addition, *qualification*-subclauses. The function of each of these is briefly explained: node aliases are used when asserting the conditions between nodes, such as when processing certain equality edges. The *collection*-subclause of a node is the bag of which the node is an element. The *qualification*-subclause is the condition associated with the node for it to be an element of the bag, such as that the node cannot represent a `nil` value.

Inclusion groups are considered in numeric order and the nodes of an inclusion group are processed from the highest ranked one to the lowest. If an inclusion group satisfies sufficient conditions for its aliases and, where appropriate, query subclauses to be generated, the strings are generated and the inclusion group is termed *processed*; edges are processed as well and each inclusion group and each edge is processed once. If an inclusion group can be processed, then *all* the

nodes of that inclusion group are processed before any other inclusion group is considered. The aliases and subclauses of all the nodes of one inclusion group are thus generated before the aliases and (where appropriate) subclauses of any other inclusion group are generated. When the *collection-* and *qualification-*subclause of a node are generated, they are immediately appended to the query string. The alias of a node, however, is not immediately appended; it is stored in a table for subsequent use.

An inclusion group is only processed if its maximal node is reachable; reachability is formally discussed in Section 7.6.1. If an inclusion group is considered but cannot be processed, no strings are generated and the next inclusion group in the sequence is considered. Just after an inclusion group is processed, the edges entering or leaving it (that are not yet processed) that have their other ends connected to processed nodes as well, are processed. Thereafter, the inclusion groups from number 1 are considered again. Query translation continues until an entire iteration executes with no new subclauses being generated. When this fixed-point is reached, QUIVER checks that all nodes and edges were processed. If this is not the case, query translation fails and it is assumed that the user constructed an incorrect query.

In the next four subsections, the mechanism of query translation is discussed in detail using the example in Figure 7.1. Section 7.2.1 discusses the sequence in which inclusion groups are processed and Section 7.2.2 discusses aliases. *Collection-* and *qualification-*subclauses are discussed in Sections 7.2.3 and 7.2.4 respectively.

7.2.1 Order of translation

When translating the query in Figure 7.1, inclusion group 1 is considered first. Its maximal node is reachable, as it represents the named value John (as mentioned in Section 5.7.2 all named values are reachable). Thus it can be processed.

Inclusion group 2 is now considered. A method node is reachable only if the source nodes of its incoming method edge and incoming method expand edge are both processed. This is not the case, and the next inclusion group (number 3) in the sequence is considered. A method input node is reachable if the nodes at the target ends of *all* its method expand edges are processed. This inclusion group also cannot be processed, and inclusion group 4 is considered.

Inclusion groups 4 and 5 are both reachable, as they represent literal values. They are marked as processed and the unprocessed inclusion groups after inclusion group 1 are again considered. Inclusion group 3 is now processed as inclusion groups 4 and 5 are reachable. The method expand edges were traversed to "reach" the method input node, and they are marked as processed as well.

Inclusion group 2 is now processed and the method edge and its incoming data flow edge are both marked as processed. Inclusion group 6 is processed next as

its outermost node is reachable through the traversal of its incoming data flow edge, and this data flow edge is also marked as processed. Inclusion group 7 is processed next as its outermost node is reached through the traversal of the `consists_of` property edge (the edge is also marked as processed).

Inclusion group 8 is similar to inclusion groups 4 and 5 in that it represents a literal value, and it is processed next. The **name** edge is now processed as its one end is connected to the node just processed and its other end is connected to a node that was processed previously. Note that the **name** property edge was *not* traversed to reach node <8,1>. If node <8,1> had not been reached previously, however, it could have been reached through the traversal of this edge.

7.2.2 Aliases

The alias of a node is that value by which the node is known after its inclusion group is processed; every node is assigned an alias value. The table in Figure 7.3 lists all the aliases generated, in the order that they were generated. The alias of node <1,1> is equal to the named value that it represents, namely **John**.

The aliases of nodes <4,1>, <5,1> and <8,1> are equal to their literal values, as displayed in Figure 7.3. Method input nodes, such as node <3,1>, differ to all other nodes in that they produce two aliases, namely a *structure*-alias and a *parameter*-alias. The *parameter*-alias is used when setting the parameters of its associated method, while the *structure*-alias is used when asserting conditions on the node, such as comparing it to other structures. Both aliases are derived from the aliases of the nodes at the head ends of its method expand edges; the *structure*-alias also uses the names of the method input edges.

The aliases of a method node (node <2,1>) are derived from the aliases of its associated class node (the class node is determined using the incoming method edge) and the *parameter*-alias of its method input node (determined using the incoming data flow edge). The alias of node <6,2> is copied from the alias of node <2,1> as it is reached by traversing the data flow edge from node <2,1>. The alias of node <6,1> is now generated – in this example, the alias of any non-maximal node is equal to the variable associated with that node. The alias of node <6,1> is thus `VAR6_1`.

Node <7,2> is reachable through the traversal of the `consists_of` property edge. Its alias is derived from the alias of the source node of that edge and the property name. Node <7,1> is a non-maximal node and its alias is thus equal to `VAR7_1`.

7.2.3 Generating *collection*-subclauses

If *collection*-subclauses are generated for an inclusion group, they are generated after the aliases are generated. In this example (Figure 7.1), all but the method

Node	Alias
<1,1>	John
<4,1>	75
<5,1>	100
<3,1>	"STRUCT(min:75, max:100)" (*structure*-alias) and "75, 100" (*parameter*-alias)
<2,1>	John.courses_by_marks(75,100)
<6,2>	John.courses_by_marks(75,100)
<6,1>	VAR6_1
<7,2>	VAR6_1.consists_of
<7,1>	VAR7_1
<8,1>	"Graphics1"

Figure 7.3: Alias generated for Figure 7.1, displayed in the order generated

node and the method input node produce *collection*-subclauses.

As mentioned, this subclause indicates the bag of which the node is an element. If the node does not appear within any blob (nodes <1,1>, <4,1>, <5,1>, <6,2>, <7,2> and <8,1>), a bag is "constructed" in the query string, as in Figure 7.4. Recall that the form of an exists-type query is: "exists *variable* in *collection*: (*conditions*)"; if a node does not represent an element of a collection (such as a maximal node) a collection is constructed by surrounding the node's value with the symbols "bag(" and ")". This sets the variable representing the node equal to the one element within the bag, namely that value. A bag has to be constructed as OQL does not permit exists-type queries of the form "exists *variable* = *value*: (*conditions*)".

Node	*Collection*-subclause
<1,1>	bag(John)
<4,1>	bag(75)
<5,1>	bag(100)
<6,2>	bag(John.courses_by_marks(75,100))
<6,1>	VAR6_2
<7,2>	bag(VAR6_1.consists_of)
<7,1>	VAR7_2
<8,1>	bag("Graphics1")

Figure 7.4: *Collection*-subclauses generated for Figure 7.1, in the order generated

The *collection*-subclause of a maximal node is derived from its alias. For example, the alias of node <1,1> is John, and the *collection*-subclause is `bag(John)`. The *collection*-subclauses of inclusion groups 4 and 5 are similarly derived from their aliases. The *collection*-subclause of a non-maximal node is equal to the variable name associated with the node ranked one higher than it (nodes <6,1> and <7,1>).

The table in Figure 7.4 lists all the *collection*-subclauses generated. From the table it is clear how a large part of the OQL query in Figure 7.2 is constructed.

7.2.4 Generating *qualification*-subclauses

In these subclauses, two types of conditions are asserted: certain objects cannot have `nil` values, and the conditions generated while processing certain edges are stated. Whenever an object node appears in a query pattern, an assertion is made that the object has to be *non*-`nil`; this clause is constructed using the alias of the node. These conditions are generated for objects only, namely nodes <1,1>, <6,1> and <7,1>.

As stated in Section 7.2, after an inclusion group is processed, unprocessed edges connected to it and to any other previously processed node are processed. For example, after inclusion group 8 is processed, the `name` edge is processed and the condition represented by it is added to the *qualification*-subclause of node <8,1>. The condition is generated using the aliases of the source node of the edge and the property name. This subclause is asserted to be equal to the alias of the target node; the resulting subclause is "`VAR7_1.name = "Graphics1"`". The table in Figure 7.5 lists all the *qualification*-subclauses for the OQL query of Figure 7.2.

Node	*Qualification*-subclause
<1,1>	`John != nil`
<6,1>	`VAR6_1 != nil`
<7,1>	`VAR7_1 != nil`
<8,1>	`VAR7_1.name = "Graphics1"`

Figure 7.5: *Qualification*-subclauses for the query in Figure 7.1

7.3 Output of a Fully Determined Node

This is the first of two `SFW`-type queries that can be generated; the other type (using collecting blobs) is discussed in Section 7.4. Queries here are similar to

exists-type queries, except that the query returns a *fully determined* data instance if the query pattern is found; if the query pattern is not found, the query fails.

Assume that the query in Figure 7.6 was constructed by the user. The query is similar to the one in Figure 7.1, except that node <6,2> is bold. This query returns the courses for which John scored a first, provided that he scored a first in at least one course containing the Graphics1 module.

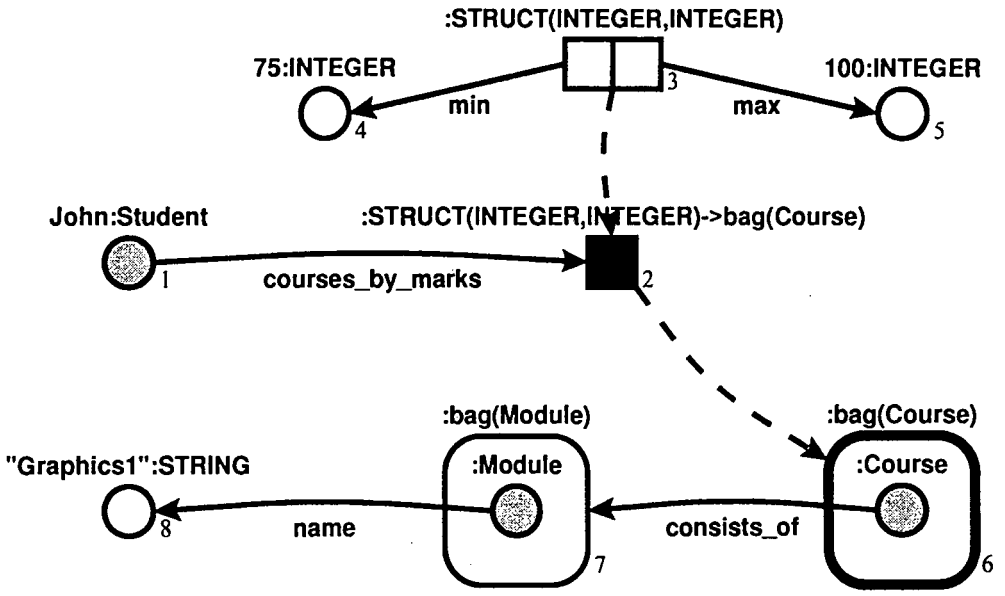


Figure 7.6: Retrieving the courses taken by student John

The OQL produced by QUIVER is displayed in Figure 7.7. The where clause of the query is identical to the query in Figure 7.2; it was produced in exactly the same way and this query produces the same aliases and *collection-* and *qualification-*subclauses as the query in Figure 7.1. After producing the where clause, QUIVER calculates the fully-determined name of the bold node.

Determined values are calculated by an iterative algorithm, similar to the query translation algorithm described in Section 7.2. When the *determined value* of the bold node is found, the algorithm halts and the from clause of the query string is generated. If a fixed point is reached without finding the determined value, the user is informed that an incorrect query was constructed and query translation fails.

Nodes representing named values and literal values are fully determined; the determined value of a node representing a named value is equal to that named value, of a literal is that literal value. Inclusion group 1 is first considered, and node <1,1> is assigned the determined value John, as displayed in Figure 7.8. Inclusion groups 4 and 5 then have their determined values generated. The determined value of inclusion group 3 is identical to its *parameter*-alias; the determined

```
element(
select DETVAR
from DETVAR in bag(John.courses_by_marks(75, 100))
where exists VAR1_1 in bag(John): (John != nil and
exists VAR4_1 in bag(75): (
exists VAR5_1 in bag(100): (
exists VAR6_2 in bag(John.courses_by_marks(75, 100)): (
exists VAR6_1 in VAR6_2: (VAR6_1 != nil and
exists VAR7_2 in bag(VAR6_1.consists_of): (
exists VAR7_1 in VAR7_2: (VAR7_1 != nil and
exists VAR8_1 in bag("Graphics1"): (
VAR7_1.name = "Graphics1")))))))
)
```

Figure 7.7: SFW-type OQL query generated from the QUIVER query of Figure 7.6

values of inclusion group 2 and 3 are both equal to "`John.courses_by_marks(75, 100)`", and these are calculated in much the same way as their aliases were calculated. After the determined value of node <6,2> is found, the algorithm terminates, as node <6,2> is marked bold. The only other node for which a determined value could have been calculated is node <8,1> (with determined value equal to "`"Graphics1"`").

If node <7,1> were marked bold, the query could not be translated as node <7,1> is not fully determined. The determined values generated are listed in Figure 7.8.

Node	Determined value
<1,1>	John
<4,1>	75
<5,1>	100
<3,1>	75, 100
<2,1>	John.courses_by_marks(75, 100)
<6,2>	John.courses_by_marks(75, 100)

Figure 7.8: Determined values generated for the Figure 7.6

Due to the structure of SFW-type queries, the `from` clause has to be a collection. The `from` clause is constructed from the determined value of the bold node, and in this case is equal to "`from DETVAR in bag(John.courses_by_marks(75, 100))`". The `select` clause is always equal to "`select DETVAR`"; `DETVAR` is an arbitrary

variable name. The query, without the `element` clause, returns (if the query succeeds) a collection containing one element x. The `element` operation is used to transform the answer to return just x.

7.4 Output using a Collecting Blob

This query translation is performed if the query contains a collecting blob. Assume that the query in Figure 7.9 was constructed by the user; the query returns the modules contained in the courses for which a student scored firsts. The answer type is a structure containing the student object and the associated modules; the components of the output structure are named `student` and `modules`. The OQL produced by QUIVER is displayed in Figure 7.10.

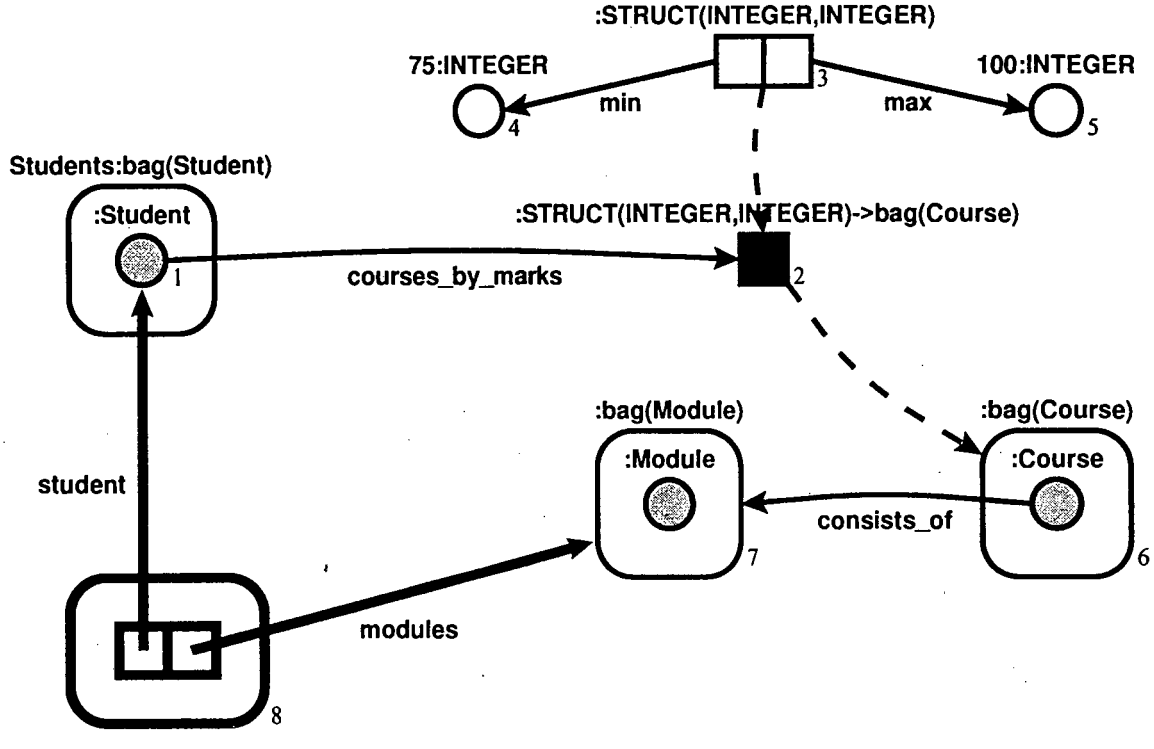


Figure 7.9: Retrieving the modules taken by students

Because queries here return *all* data instances that satisfy the query pattern, the `from` clause of the query is used to generate permutations of all data patterns, and the conditions constraining the data are placed in the `where` clause. The queries in Sections 7.2 and 7.3 were generated by appending all subqueries to the query string as the subqueries were generated. Queries here differ in that the `from` and `where` clauses of the query are constructed as two different strings; all *collection*-subclauses are appended to a `from` string and all *qualification*-subclauses are

```
select STRUCT(student:VAR1_1, modules:VAR6_1.consists_of)
from VAR1_2 in bag(Students),
   VAR1_1 in VAR1_2,
   VAR4_1 in bag(75),
   VAR5_1 in bag(100),
   VAR6_2 in bag(VAR1_1.courses_by_marks(75, 100)),
   VAR6_1 in VAR6_2,
   VAR7_2 in bag(VAR6_1.consists_of)
where VAR1_1 != nil and
   VAR6_1 != nil and
   exists VAR7_1 in VAR7_2: (VAR7_1 != nil)
```

Figure 7.10: SFW-type OQL query generated from the QUIVER query of Figure 7.9

appended to a where string. Similar to the queries in Figures 7.1 and 7.6, the method node and method input node do not generate *collection-* and *qualification-*subclauses. The nodes in the output structure inclusion group (inclusion group 8 in Figure 7.9) do not generate these subclauses nor do they generate aliases either. In addition, node <7,1> does not generate a *collection-*subclause because it is not *significant*, a term explained in Section 7.6.3.

The aliases are generated using the same computations as described in Sections 7.2 and 7.3; the aliases produced are listed in Figure 7.11.

Node	Alias
<1,2>	`Students`
<1,1>	`VAR1_1`
<4,1>	`75`
<5,1>	`100`
<3,1>	"`STRUCT(min:75, max:100)`" (*structure*-alias) and "`75, 100`" (*parameter*-alias)
<2,1>	`VAR1_1.courses_by_marks(75, 100)`
<6,2>	`VAR1_1.courses_by_marks(75, 100)`
<6,1>	`VAR6_1`
<7,2>	`VAR6_1.consists_of`
<7,1>	`VAR7_1`

Figure 7.11: Aliases generated for the query in Figure 7.9

Each insignificant node does not generate a *collection-*subclause but generates a

modified *qualification*-subclause. The *qualification*-subclause asserts that a collection contains at least one element and, where appropriate, that the element is *non*-`nil`. The *collection*- and *qualification*-subclauses generated are shown in Figure 7.12.

Node	*Collection*-subclause	*Qualification*-subclause
<1,2>	`bag(Students)`	
<1,1>	`VAR1_2`	`VAR1_1 != nil`
<4,1>	`bag(75)`	
<5,1>	`bag(100)`	
<6,2>	`bag(VAR1_1.courses_by_marks(75, 100)`	
<6,1>	`VAR6_2`	`VAR6_1 != nil`
<7,2>	`bag(VAR6_1.consists_of)`	
<7,1>		`exists VAR7_1 in VAR7_2: (VAR7_1 != nil)`

Figure 7.12: Subclauses generated from Figure 7.9

If any edge has to be processed after the inclusion groups that it is attached to are processed (such as the `name` property edge in Figure 7.6), the subclause generated by this condition is appended to the `where` string. No such edges exist in this query.

The `select` string is constructed using the alias of the bold node, or if the query contains an output structure node, the aliases of the nodes pointed to by the output edges. The alias of node <1,1> is `VAR1_1` and the alias of node <7,2> is `VAR6_1.consists_of`; the `select` clause "`STRUCT(student:VAR1_1, modules:VAR6_1.consists_of)`" is generated. The `select`, `from` and `where` clauses are concatenated to produce the query string as in Figure 7.10.

7.5 Translation of Subqueries

A subquery is represented by a *subquery node*, which appears as an open rectangle framing the nodes that form the subquery. Subquery nodes, just like inclusion groups, have numbers associated with them and are also considered in numeric order along with all inclusion groups. A *subquery inclusion group* consists of a subquery node together with all the inclusion groups that form the subquery.

Subqueries are processed using a recursive algorithm, thus subqueries may appear within other subqueries. The nodes within a subquery are only considered when

that subquery is considered; these nodes remain computationally "hidden" within the subquery node at all other times. The aliases of the nodes within a subquery may thus only be used when considering the subquery for translation; the nodes are considered unprocessed otherwise and may not be used. If a subquery *b* appears within a subquery *a*, *b* may only be considered while *a* is being considered. The only node of a subquery that is available after the subquery is processed is the subquery node itself.

For a subquery to be processed, all the nodes and edges of the subquery have to be processed; this includes edges that have only one end connected to a node within a subquery. For a subquery to be processed, therefore, all nodes outside the subquery connected to any node within the subquery have to be processed first. As mentioned, if an inclusion group cannot be reached, it is again considered during a subsequent iteration of the algorithm; if a subquery cannot be processed, the next inclusion group is considered and the subquery will again be considered during a subsequent iteration.

Consider the query in Figure 7.13. The query is the same as that displayed in Figure 2.10, and finds the courses whose modules are lectured across *all* buildings on campus.

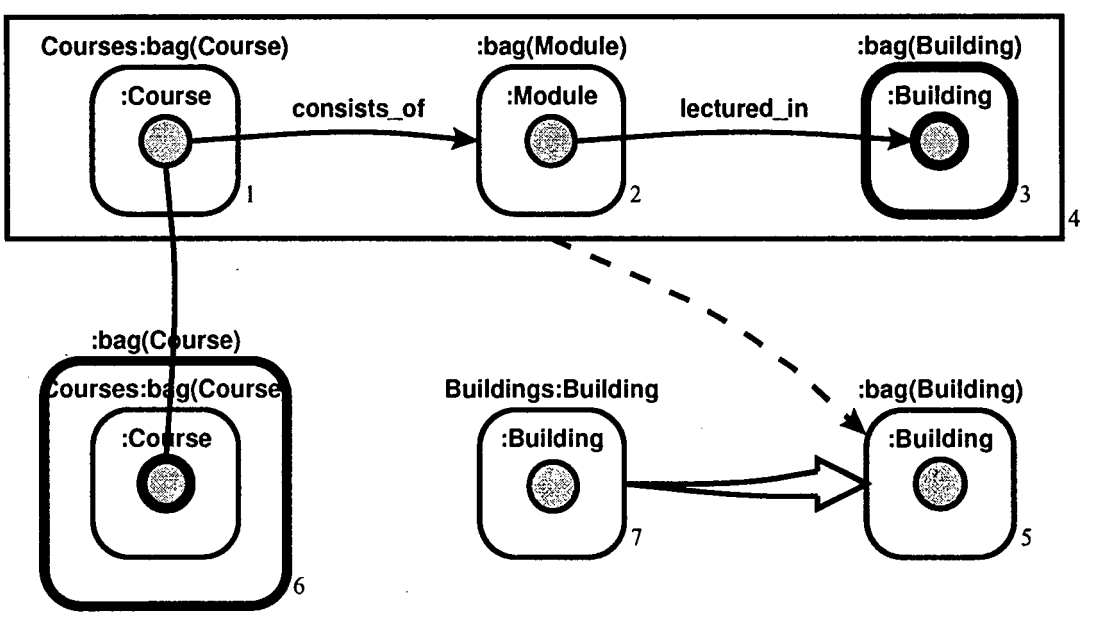


Figure 7.13: A query with a subquery

The subquery node is first considered and all the inclusion groups within it are reachable. Before the subquery is marked processed, however, a check is made that all its nodes and edges are processed as well. Since the equality edge from node <1,1> to node <6,1> is not processed, the subquery fails and it is not marked as processed.

Inclusion group 6 is then processed, after which the subquery may be processed. The aliases of the subquery inclusion group are displayed in Figure 7.14; the alias of the subquery node is equal to the OQL string generated for the subquery.

Node	Alias
<1,2>	`Courses`
<1,1>	`VAR1_1`
<2,2>	`VAR1_1.consists_of`
<2,1>	`VAR2_1`
<3,1>	`VAR2_1.lectured_in`

<4,1>	`(select VAR2_1.lectured_in`
	`from VAR1_2 in bag(Courses),`
	`VAR1_1 in VAR1_2,`
	`VAR2_2 in bag(VAR1_1.consists_of),`
	`VAR2_1 in VAR2_2,`
	`VAR3_1 in bag(VAR2_1.lectured_in)`
	`where VAR1_1 != nil and`
	`VAR1_1 = VAR6_1 and`
	`VAR2_1 != nil and`
	`VAR2_1.lectured_in != nil)`

Figure 7.14: Aliases generated for the subquery in Figure 7.13

Inclusion group 5 is processed next; the aliases of the remaining nodes are displayed in Figure 7.15 and the OQL generated from the query is displayed in Figure 7.16.

Node	Alias
<6,2>	`Courses`
<6,1>	`VAR6_1`
<5,2>	*same as alias of <4,1>*
<5,1>	`VAR5_1`
<7,2>	`Buildings`
<7,1>	`VAR7_1`

Figure 7.15: Aliases generated for the outer query in Figure 7.13

After inclusion group 7 is processed, the condition represented by the subset edge is added to the *qualification*-subclause of node <7,2>. The subset edge is translated using the `for all ... exists` clause, and the aliases of the source

```
select VAR6_1
from VAR6_2 in bag(Courses),
   VAR6_1 in VAR6_2,
   VAR7_2 in bag(Buildings),
   VAR5_2 in bag(alias of <5,2>)
where VAR6_1 != nil and
   exists VAR7_1 in VAR7_2: (VAR7_1 != nil) and
   exists VAR5_1 in VAR5_2: (VAR5_1 != nil) and
   (for all subsa in Buildings: exists subsb in alias of <5,2>:
      subsa = subsb))
```

Figure 7.16: The OQL generated from the query in Figure 7.13

and target nodes are used within the clause. The variables used in the clause are arbitrarily named subsa and subsb.

The calculation of the *collection*-subclauses and remaining *qualification*-subclauses of this query proceeds similarly to the calculation of these clauses presented earlier.

7.6 General Rules

Query translation, thus far, has been explained by means of examples. This section discusses general rules pertaining to query translation. Section 7.6.1 discusses the computation of aliases. Section 7.6.2 discusses the translation of edges and Section 7.6.3 discusses the terms *significant* and *insignificant*.

7.6.1 Aliases and reachability

If an inclusion group contains a collecting blob, that blob is the maximal node of its inclusion group but it does not have an alias associated with it, as it is used solely to determine output. The *maximal reachable* node (the *m-r node*) is thus the maximal node to have an alias associated with it; in the case where the inclusion group does not contain a collecting blob, the *m-r* node is also the maximal node. In the context below, sub-*m-r* nodes refer to the nodes within the *m-r* node. The alias of a *m-r* node is equal to its *reachability value*; reachability values are generated solely for the computation of aliases. The aliases of sub-*m-r* nodes are derived from the alias of their surrounding *m-r* node. The computation of reachability values is now discussed.

The reachable value of a node representing a named value is equal to the named value itself, and due to the design of QUIVER, the node will always be the *m-r* node of its inclusion group. Reachability is also transferred by traversing certain edges; reachability is transferred from a *reachability source* (a node that already has an alias) to the *reachability target* (the node that does not not have an alias). For an edge to transfer reachability, the reachability target node has to be the *m-r* node of its inclusion group. Note that the reachability source is not necessarily at the tail of the edge and the reachability target is not necessarily at the head of the edge transferring the reachability. Before a query is translated, it is first checked to be *type correct* (type-correctness was discussed in Section 6.3.1 and the rules appeared in Figures 6.4 and 6.5). This ensures, for example, that an equality edge appears between compatible object-cored nodes, or between compatible literal-cored nodes. When referring to the alias of a method input-cored node in the context below, the *structure*-alias of the node is implied.

Reachability can be transferred to object-cored, literal-cored, structure-cored and method input-cored inclusion groups through the traversal of an equality edge. In the case of a object-, literal- or structure-cored inclusion group, the nodes of the reachability target are assigned aliases equal to the aliases of the corresponding nodes in the reachability source. If the reachability target and reachability source are both method input-cored, the *parameter*-aliases and *structure*-aliases of the source are copied to the *parameter*-aliases and *structure*-aliases of the target. If the reachability source is not method input-cored (it then *has* to be structure-cored), its alias is copied to the target's *structure*-alias. The target's *parameter*-alias is derived by expanding each of the source alias' components, and separating them by comma symbols.

No *collection*- and *qualification*-subclauses are generated if reachability is transferred using an equality edge. Therefore, a node does not have a variable associated with it if it was reached through the traversal of an equality edge; a variable is introduced for a node only if it generates a *collection*-subclause. Reachability through the traversal of an equality edge is the *only* case in which a class-, literal- or structure-cored node will not have a variable associated with it. Method input nodes never have variables associated with them, as they are reached either through the traversal of an equality edge or through "construction"; construction of a method input node is discussed later.

Class-cored, literal-cored and structure-cored inclusion groups may also be reached through the traversal of a property edge with its head end connected to the reachability target. The reachability value is derived from the source alias and the property name.

A function node can only be reached through the traversal of a data flow in edge; its reachability value is derived from the function name and the source alias. A method node can be reached if it has a reachability source connected to the tail

Core type of node	Edge(s) that may transfer reachability
class, literal or structure	equality edge
class, literal or structure	property edge *in*
class, literal or structure	data flow edge *in*
method node	data flow edge *in* and method edge *in*
function node	data flow edge *in*
method input	equality edge
method input	all method expand edges *out*
output structure	all output edges *out*
subquery node	-

Figure 7.17: Transferring reachability

of an incoming method edge and a reachability source connected to the tail of a data flow in edge. The reachability value of the method node is derived from the alias of the method edge source, the method name and the *parameter*-alias of the data flow in edge source.

A method input-cored inclusion group may also be reached (through *construction*) if it contains just one node and all its outgoing method expand edges are connected to nodes with aliases. The aliases of reachability sources are used to derive the *parameter*-alias of the target. The edge names and the reachability sources are used to derive the *structure*-alias. Reaching a method input node through the construction of its components, or reaching a function node or a method node does not generate any *collection*- and *qualification*-subclauses. QUIVER does not permit the free construction of collections, thus a method input-cored inclusion group can only be reached through construction if it does not appear within a blob. A method input-cored inclusion group (appearing with or without blobs) can be reached through the traversal of an equality edge.

The table in Figure 7.17 contains a summary of the rules pertaining to reachability. The table lists the core types of the inclusion groups and the edges that may be considered in order to transfer reachability to the inclusion groups.

Output structure-cored inclusion groups are only considered when no other inclusion groups can be processed. They are used to construct the `select` clause of a query, and the `select` clause is always constructed last. An output structure-cored inclusion group is reachable if all the nodes connected to the head ends of its outgoing output edges were reached; if this is not the case the query fails. A subquery node is only reachable if the query that it represents is processed. The alias of a subquery node is equal to the OQL string of that subquery.

	equality	property	data flow	subset	method expand
class	√	both	in	both	in
literal	√	in	in	both	in
structure	√	both	in	both	in
method			out		
function			out		
method input	√			both	both

Figure 7.18: Processing edges

7.6.2 Conditions represented by edges

As mentioned, when a node is processed all edges connected to it that have their other ends connected to processed nodes, are processed. For example, when a class-cored, literal-cored, structure-cored or method input-cored node is processed, all subset edges leaving or entering it are examined for possible processing. The table in Figure 7.18 lists these rules.

The columns represent the edge types and the rows represent the core types of the inclusion groups. For each inclusion group core type, many edges may be considered, and each of these edges is considered either if it has its head connected to the inclusion group under consideration ("in" in the table) or its tail connected to the inclusion group ("out"). If the edge may have its head *or* tail connected to the inclusion group "both" appears in the table. Equality edges are undirected, and the appropriate cells contain a tick mark ("√") if equality edges are considered. Blank cells indicate that the particular edge is not considered for that inclusion group type.

When testing for equality between two objects, and the objects are of different types, O_2's implementation requires that the subclass type appears to the left of the equal-to sign, and the superclass appears to the right. In O_2, therefore, the query "`Employees=Persons`" is invalid, but the query "`Persons=Employees`" is valid.

7.6.3 Significance of nodes

Consider the query in Figure 7.19; the `Student`-cored inclusion group is inclusion group 1 and the other is inclusion group 2. The query finds those students who take at least one course. The desired output of the query is thus to return each `Student` object provided that the object has at least one *non*-`nil` `Course` object in its `takes` property. A (non-QUIVER) query translation returning duplicate answers is displayed in Figure 7.20.

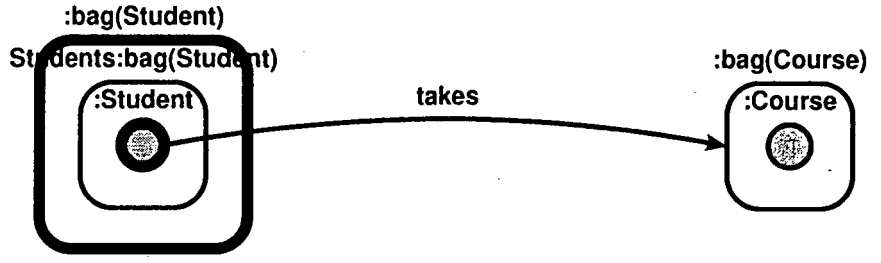


Figure 7.19: Finding students that take courses

```
select VAR1_1
from VAR1_2 in bag(Students),
   VAR1_1 in VAR1_2,
   VAR2_2 in bag(VAR1_1.takes),
   VAR2_1 in VAR2_2
where VAR1_1 != nil and
   VAR2_1 != nil
```

Figure 7.20: Query translation of the query in Figure 7.19 returning duplicate answers

For each subclause that appears in the `from` clause, a join is performed and these tuples are checked against the conditions in the `where` clause. Because "`VAR1_1`" and "`VAR2_1`" appear in the `from` clause, a tuple is generated for each course that a student takes. The student object will thus be returned for each course that the student takes. If a student takes five courses, for example, that student object will be returned five times.

Removing the subclause "`VAR2_1 in VAR2_2`" from the `from` clause and the associated "`VAR2_1 != nil`" subclause from the `where` clause (and adding a subclause to the `where` clause) prevents redundant answers being produced. In this query, the `Course` object node is *insignificant*, as it has no edges attached to it and it is not marked bold, and may thus be removed from the `from` clause. The terms *significant* and *insignificant* apply to those nodes that could generate *collection*-subclauses, namely to object-, literal- and structure-cored nodes only.

The rank of the *lowest significant node* of an inclusion group is the lower of two values: the lowest-ranked node having any edge attached to it and the rank of the inclusion group's reachable bold node. If an inclusion group does not contain a bold node, the *lowest significant node* is equal to the lowest-ranked node having any edge attached to it. *Collection-* and *qualification*-subclauses are generated, as discussed previously, for all nodes from the *m-r* node to the to the lowest significant node. For the nodes ranked lower than the lowest significant one

(the insignificant nodes), *qualification*-subclauses are generated to assert that a collection contains at least one element and, where appropriate, that the element is *non*-nil.

The QUIVER translation of the query in Figure 7.19 appears in Figure 7.21.

```
select VAR1_1
from VAR1_2 in bag(Students),
   VAR1_1 in VAR1_2, VAR2_2 in bag(VAR1_1.takes)
where VAR1_1 != nil and
   exists VAR2_1 in VAR2_2: (VAR2_1 != nil)
```

Figure 7.21: QUIVER translation of the query in Figure 7.19

7.7 Alternative Translations

As mentioned in Section 7.4, *all* significant nodes (not considering nodes of method-, method input- and output structure-cored inclusion groups) generate *collection*-subclauses in order to simplify query translation; this could lead to a variable being introduced in the *collection*-subclause that is not used anywhere else. This inefficiency appears in the translation of the inner query of Figure 7.13 (the translation appears as the alias of node <4,1> in Figure 7.14); node <3,1> generates a *collection*-subclause which is not used. The translation could be made more efficient if the following subclause was removed from the `from` string: "`VAR3_1 in bag(VAR2_1.lectured_in)`". Alternatively, the `select` clause could be altered to read "`select VAR3_1`" instead of "`select VAR2_1.lectured_in`", as the variable `VAR3_1` is instantiated in the `from` clause.

The query in Figure 7.16 is particularly inefficient as it requires the computation of the subquery twice (the alias of node <5,2> appears twice). An optimization to the query would be to replace the second occurrence of the alias by `VAR5_2`, as `VAR5_2` appears in the `from` clause. A more general rule would be to use variables names of nodes (derived from the inclusion group and rank of the node) instead of the alias of the node while constructing *qualification*-subclauses. This "optimization", however, does not always produce correct results, as not all nodes have variables associated with them. Such cases arise when reachability is transferred through the traversal of an equality edge, as discussed in detail in Section 7.6.1.

The translation mechanism implemented by QUIVER does not always produce efficient queries, but does produce correct queries. The O_2 database includes

a query optimizer [36], and many of the inefficiencies in the OQL produced by QUIVER can be eliminated by a powerful query optimizer.

Chapter 8

A User Evaluation of QUIVER and OQL

8.1 Introduction

In order to evaluate the usability of QUIVER as a querying application, a user evaluation was conducted. The speed with which one could construct queries in the QUIVER query language (using the QUIVER system) was compared to the speed with which one could construct identical queries in textual OQL (using paper and pen), and the ease of producing correct queries in each system was evaluated. The procedure and the results of this evaluation are discussed in this chapter.

8.2 Experimental Setup

The evaluation consisted of the input of two groups of participants. The one group (GROUP A) consisted of seven participants and the second group (GROUP B) consisted of six participants. All thirteen participants were Computer Science Honours (fourth year) students taking a course in *Visualisation*.

The participants were thus not chosen voluntarily. Participants, however, did reach consensus amongst themselves as to who would join GROUP A and who would join GROUP B.

Each participant was given a list of eight queries to construct. The seven participants in GROUP A were each to construct the queries in the QUIVER query language, and the six participants in GROUP B were each to construct them in OQL. A day before the user evaluation, each participant was given a graphical representation, as well as a textual representation of the query schema; they

were encouraged to familiarize themselves with the schema. The actual queries, though, were only presented to each participant at the evaluation. The queries are presented in Appendix B.

Some participants had used SQL before. All prior SQL experience was gained from a Computer Science 3rd year Databases course. Three participants in each of GROUP A and GROUP B had no prior experience in using SQL. None of the participants had any prior experience with QUIVER or with OQL. Two introductory lectures (one on QUIVER and one on OQL) were prepared and presented to the participants. Participants in GROUP A were naturally compelled to attend the QUIVER introductory lecture (and vice-versa) but all participants were encouraged to attend both lectures.

QUIVER has been designed to aid the user in constructing queries. It, for instance, automatically constructs data types used in queries and allows the user to spatially arrange the query. It was thus decided that GROUP A participants would use QUIVER to construct queries, necessitating an *on-line* test. GROUP B participants did not need electronic assistance, and constructed their queries using paper and pen.

At the time of the user evaluation, QUIVER's query processing engine was not yet operational, so GROUP A participants had no way of checking the data instances produced by their queries. This placed GROUP A participants equal with GROUP B participants, as GROUP B participants were also unable to electronically check the correctness of their OQL queries.

All user evaluations were completed in the morning of the test day. During the afternoon of the test day, participants were asked to write down the comments they had on their experience. In order to encourage a critical appraisal, participants were marked on their comments; these marks contributed to their Visualisation Course marks. GROUP A participants were asked to write their comments about QUIVER interface and the QUIVER query language, and the GROUP B participants were asked to give their comments on OQL. The number of correct QUIVER or OQL queries produced would not have any effect on their course marks.

8.2.1 Introductory lectures

The introductory lectures consisted of overhead slides and moved from simple concepts through to more advanced concepts. There exists a fairly close correspondence between OQL query concepts and the QUIVER language query concepts, and so the same examples and the same sequence of examples were used in each lecture. The QUIVER lecture was conducted by myself, and the OQL lecture was conducted by Professor PT Wood.

Each lecture was forty-five minutes long and the lectures were presented on the same day. The lecture style was informal, and participant were encouraged to ask

questions during the lecture. Each lecture consisted of ten sections. The table in Figure 8.1 gives the title of each section and a brief description of each section's contents.

Section name	Section contents
classes and literals	Classes and literals, named objects and named literals, queries consisting solely of named values.
structures	Modeling behavior of structures and suitability of structures to model a birthdate or a student's address
collections	Its was emphasized that visual inclusion represents an `element` operation, and not a `subset` operation. The properties of the `bag` collection type, in particular, was discussed.
properties	Modeling behaviors with respect to objects and structures.
simple queries	Query processing, roots of persistence.
other edges	Equality edges and subset edges.
methods	Object methods, method inputs, method output and data flow edges.
functions	The difference between methods and functions.
output	The use of **bold** to indicate output, the difference between reachable blobs and collecting blobs.
subqueries	Query nesting, order of query processing, ways in which a subquery can be linked to other parts of a query.

Figure 8.1: Contents of introductory lectures

For the QUIVER introductory lecture, each section consisted of either one or two examples as constructed in QUIVER. For the lecture sections after `simple queries`, a query stated in English as well as in OQL was also presented.

For the OQL introductory lecture, each section consisted only of an English query and the OQL translation.

The introductory lectures were given three days before the user evaluation. The examples used in the lectures were carefully chosen to be from a schema different to the schema that was to be used in the user evaluation.

8.2.2 Hypotheses

Two hypotheses were to be tested. The first hypothesis was that OQL queries can be constructed just as easily as QUIVER queries. This means that GROUP A

participants would attempt the same number of queries as GROUP B participants. The second hypothesis being tested was that GROUP A participants constructed a similar number of correct queries to GROUP B participants.

8.2.3 Experimental setup for using QUIVER

As mentioned, the participants using QUIVER conducted the test on-line. The chosen venue enjoyed privacy, infrequent interruptions by other students and contained four identical workstations. Thus the seven participants were divided into two groups.

Before the participants started the evaluation, they were given a QUIVER INFORMATION SHEET to familiarize themselves with the QUIVER interface. They were given ten minutes to read this document, and were free to use QUIVER while referring the document.

Participants were then given a sheet of USER EVALUATION INSTRUCTIONS which contained one "test" query to construct. The test query was not counted as one of the evaluation queries, and was based on the same schema as used in the introductory lectures. After the participants successfully completed the test query, they were assumed able to construct QUIVER queries by themselves. During the evaluation, participants were free to consult the QUIVER INFORMATION SHEET.

I remained present during both sessions of the on-line test, and answered any questions that participants had on the QUIVER interface. Participants were told that the evaluation was to be conducted under test conditions, and that they could ask questions about the interface only.

Participants were given forty-five minutes to complete the construction of the queries. All participants did not attempt all of the queries.

8.2.4 Experimental setup for using OQL

Participants formulating queries in OQL used a different venue also chosen due to its privacy and infrequent interruptions by other students.

The participants were given the same USER EVALUATION INSTRUCTIONS sheet as mentioned in Section 8.2.3, and were told to ignore the QUIVER specific instructions. All six participants conducted the test simultaneously, with similar test conditions applying as for GROUP A participants.

The participant were also given forty-five minutes to construct the eight queries. Professor PT Wood remained present with the six participants to ensure that test conditions were met.

8.2.5 Gathering results

In order to prevent discussion amongst the groups of participants, none of them were allowed to remove any material from the test venue. They were also instructed not to discuss the evaluation until after everyone had been tested.

Each of the queries of each of the participants was marked into one of the categories displayed in Figure 8.2. All the queries were marked by myself.

Symbol	Condition
$\surd$	Query was completely correct.
$\surd\times$	Minor errors – the query contained minor syntactic or semantic errors.
$\times$	Wrong - query contained major syntactic or semantic errors.
	Query was not attempted

Figure 8.2: Classification of Answers

8.3 Results

The results that were produced from participants in GROUP A are displayed in Figure 8.3, and the results produced from GROUP B are displayed in Figure 8.4. The tables show five counts for each participant:

1. the number of queries attempted,
2. the number of queries marked correct,
3. the number of queries marked essentially correct,
4. the number of queries marked wrong,
5. a ratio r, giving the sum of the number of queries marked correct or with minor errors, as a ratio of the total number of queries attempted.

The first two categories in Figure 8.2 are grouped together for the computation of r as they represent queries that are *essentially correct.* The participants in GROUP A were numbered P1 through to P7, and the participants in GROUP B were numbered P8 through to P13.

Category	P1	P2	P3	P4	P5	P6	P7
number attempted	6	6	8	8	6	5	8
number $\surd$	5	2	7	2	3	3	5
number $\surd\times$	1	2	1	5	1	1	1
number $\times$	0	2	0	1	2	1	2
r	1	0.667	1	0.875	0.667	0.8	0.75

Figure 8.3: Summary of results of GROUP A (QUIVER)

Category	P8	P9	P10	P11	P12	P13
number attempted	8	8	8	6	7	6
number $\surd$	1	0	0	0	0	0
number $\surd\times$	5	5	4	1	3	2
number $\times$	2	3	4	5	4	4
r	0.75	0.625	0.5	0.167	0.429	0.333

Figure 8.4: Summary of results of GROUP B (OQL)

The statistical test used to quantify the data for both hypotheses was the *Mann-Whitney-Wilcoxon* procedure [19]. This test was chosen because the data arises from two *independent* samples and, because of the small sample size, we cannot assume that the data follows a normal distribution.

8.3.1 Hypothesis 1

For the first test we wish to determine if participants in GROUP A attempted more queries than participants in GROUP B. In order to determine this, the absolute count of the number of queries attempted by each participant was taken (the first rows of the tables in Figures 8.3 and 8.4).

Let M_a represent the mean of the number of queries attempted in GROUP A, and let M_b represent the mean of the number of queries attempted in GROUP B. The hypothesis H is that $M_b = M_a$, and the alternative A is that $M_b > M_a$. We obtain a p[1] value of 0.29. This p value is large, as we would only be able to conclude A with 71% confidence. This offers no convincing evidence that H is false; we thus conclude that QUIVER queries can be constructed just as easily as OQL queries.

[1]The p value is the probability of a sample result as extreme (in the appropriate direction) as that observed when an associated H is true.

8.3.2 Hypothesis 2

For the second test, we wish to determine if participants in GROUP A obtained more correct queries than participants in GROUP B. This was determined by using r, as explained above (the fourth rows of the tables in Figures 8.3 and 8.4).

Let M_a represent the mean score of GROUP A, and let M_b represent the mean score of GROUP B. The hypothesis H is that $M_a = M_b$, and the alternative A is that $M_a > M_b$. We obtain a p value of 0.003. The data does not support H and the p value is small enough to be significant. H is rejected, and we may, with 99.7% confidence, conclude A. We thus conclude that it is easier to construct correct QUIVER queries than it is to construct correct OQL queries.

8.4 Sources of Error

One source of error could be the way in which participants were taught either QUIVER or OQL. Not much familiarity can be gained from one forty-five minute lecture, and if such an experiment were to be repeated, participants would be given more time and more assistance on learning the appropriate language.

Participants were not selected voluntarily. This may have introduced apathetic attitudes to the experiment, despite the understanding that their comments would be graded and would influence their course marks. In future, participants should be selected voluntarily.

Participants did not have similar computing experiences. As mentioned, six of the thirteen participants had no prior experience with database query languages, seven did. The six participants were distributed evenly between the two groups, but a more accurate test could have been achieved if all participants had similar computing backgrounds.

A larger sample size would have produced more accurate results. Samples of six and seven participants are small, and larger sample sizes would reduce the influence of any chance phenomena.

8.5 Discussion of Results

The experiment was a success, as we were able to conclude, beyond reasonable doubt, one of our hypotheses.

QUIVER is not intended to be an application that helps users produce queries quickly. It is intended to help users construct queries simply and correctly. The results of hypothesis 1 suggest that participants take longer to construct queries

in QUIVER than in OQL. Although a reason for this may be that participants were still learning how to use the interface, this produces little concern.

The results of hypothesis 2 are extremely encouraging, as it shows that the QUIVER query language, together with the QUIVER application, does aid the user in producing more correct queries. The test statistic ($p = 0.003$) is small enough to be conclusive.

8.6 Comments by Participants

Participants gave their comments on both QUIVER and OQL.

OQL is an extension of SQL, and this was considered positive. Being an extension of SQL, OQL immediately has a large user base and would thus be easily accepted amongst database users. Visual query languages, as it is a developing field, has no standards. Any prior experience with visual query languages is unlikely to be transferable to QUIVER (or vice-versa). QUIVER, being a visual query language, is difficult to embed in a programming language such as C++, even though its output OQL could be embedded.

GROUP A participants felt that compared to QUIVER, OQL queries would require more planning. OQL queries would require more thought whereas construction of a QUIVER query could begin early by choosing the roots of persistence.

GROUP A participants found the QUIVER interface to useful and intuitive. QUIVER automatically generates the types for attributes, properties and subquery output, and this was found useful in reducing errors and focusing on the semantics of a query rather than on the syntax. The icons and the button-driven interface was easy to understand and use. The alphabet of the visual query items was found to be a good balance between expressive power and simplicity.

The representation of subqueries was found to be especially useful as it clearly showed the order of evaluation and the logical nesting of a query. The visual display of attributes makes it simple to establish the type of a property – be it an object or a set of objects. This information was found useful in understanding the schema and constructing correct queries. One GROUP B participant (who attended the QUIVER introductory lecture) found it helpful, while working through the OQL constructions, to represent collections of objects as QUIVER does.

As mentioned, at the time of the user evaluation the QUIVER query processing engine was not operational. The participants were thus not able to view the results of the query construction – this included not being able to view the OQL query or being able to view the data instances that would be returned. Users found this distracting, as there was no (other than examining the OQL produced)

feedback on the correctness of their queries. The QUIVER concepts of output, reachable blobs and collecting blobs were also not understood by one participant.

Other than the lack of query feedback, other QUIVER implementation issues were also criticised. These included not being able to move an entire subquery across the screen (only individual inclusion groups may be moved) and the slow speed of the QUIVER interface.

The need for user feedback, such as rephrasing the query in a natural language, was also raised.

8.7 Conclusion

Participants were very positive about QUIVER. Its visual metaphors were useful, simple and powerful. The QUIVER application successfully simplified the construction of queries and significantly aided in the construction of correct queries.

It has been quantitatively shown that QUIVER eases the construction of OQL queries.

Chapter 9

Conclusion and Future Work

In this thesis, we have designed a visual query language for object-oriented databases and implemented a substantial part of it. The visual query system, QUIVER, is ODMG-compliant; it interacts with an ODMG-compliant database and produces OQL query statements. As far as we are aware, QUIVER is the *only* ODMG-compliant visual query system.

A brief overview of QUIVER, concentrating on the graphical user interface, was provided in Chapter 2. In order to place in context QUIVER and other systems, an explanation of terms and a survey of several visual query systems was presented in Chapter 3. Further background information was provided in Chapter 4; the ODMG and O_2 were introduced, and various implementation issues concerning the ODMG model and the O_2 model and the differences between them were discussed.

The query language design was explained fully in Chapter 5. Distinction was also drawn between features that were designed but not yet implemented. We have shown that QUIVER has a consistent query language. Similar query concepts are represented by similar looking and similar acting query constructs, thus reinforcing basic query notions and reducing the number of query symbols the user has to work with. For example, methods, functions and subqueries are all represented by rectangles, but with different shadings of the rectangle. The query language is also comprehensive; it is able to represent a large range of query concepts using its relatively small set of query constructs.

Implementation issues including those concerning saving and retrieving query graphs were discussed in Chapter 6, as well as issues dealing with the communication between QUIVER and related applications O_2, O_2Look, and DOT.

Chapter 7 discussed the query translation engine. The explanation was by means of examples; this was followed by a more general discussion on the interpretation

of nodes and edges. The translation engine is robust; it can, with one exception[1], translate all queries that can be constructed in the visual interface.

Finally, a user evaluation was conduction in order to evaluate the usability of QUIVER and establish its viability in terms of being a useful, ad-hoc querying utility. This produced extremely encouraging results, and the experimental setup and various other discussions concerning the evaluation are contained in Chapter 8.

No formal proof of the expressive power of QUIVER is provided in this thesis. Appendix C, however, provides a reasonable feel of the versatility of the query language by providing QUIVER translations to queries constructed in other visual query systems.

9.1 Future Work

Although an extensive implementation has been achieved, the full design of QUIVER, has not been implemented. Immediate future work would thus be to extend the implementation to incorporate all of the design. This would involve extending the GUI and the translation engine to include sets, arrays and lists, as well as named queries. The ODL parser would also be extended to support the *entire* syntax of O_2 ODL files; it currently does not.

As mentioned in Chapter 7, the OQL queries generated by QUIVER are sometimes inefficient. Optimizing these would involve maintaining more information about the variables and aliases generated and used during query translation, as well as the contexts in which they are used.

As a visual query system, the feasibility of QUIVER was amply demonstrated in Chapter 8. A significant part of the future work of QUIVER thus lies in extending the visual query language. Query constructs and translation techniques that, particularly, need to be investigated are the negation of conditions and the transitive closures of data.

The graphical user interface may, especially in the light of new query constructs, require revision. The GUI could be enhanced to, for example, allow the user to move *clusters* of inclusion groups, rather than individual inclusion groups as is currently supported. The GUI could also be enhanced to display all views of nodes. The user should, for example, be able to switch between the open view appearance and the screened view appearance of a structure. The GUI could also aid the user in constructing disjunctions of conditions by automatically copying and adjusting inclusion groups; disjunct components sometimes contain similar query fragments to each other.

[1]Boolean literals are not implemented, so subqueries *have* to contain output inclusion groups.

At present, QUIVER provides very limited query feedback; it permits the user to view only the completed OQL query. A more desirable operation is *incremental feedback*; QUIVER could provide query testing information as constructs are placed on the canvas. This involves, amongst others, translating to English equality and subset constraints and data membership conditions.

Various features, mentioned throughout the thesis, have been designed but not implemented such as displaying parameter-less methods, Figure 5.14(c). As mentioned in Chapter 5, full-labels may either be on or off. If used in conjunction with a graphical icon editor, this allows nodes to be replaced by icons representing their contexts. A graphical icon editor, though, has not been integrated with QUIVER. At present, QUIVER does not permit more than one core node in an inclusion group; such constructs would be necessary in order to construct arbitrary collections. Three object nodes within a blob, for example, would indicate the construction of a bag of three objects. The construction of arbitrary collections needs to be investigated.

The constructs of the query language are general enough to be used to graphically view the schema of an ODMG-compliant database. This is extremely desirable, as the user may then use similar interpretations of the query constructs to understand the schema of the database. For example, classes in the schema viewer could be represented by object nodes and class and structure properties could be represented by property edges. The full design of this, however, is still to be investigated.

The `group...by` query construct (Section 4.4) is partially supported, as QUIVER permits, for example, the courses that a student takes to be associated with that student. The full expressive power of `group...by` and `sort...by` queries, however, still need to be incorporated into QUIVER.

Appendix A

O_2 ODL Grammar

The grammar is given using an informal BNF notation. The grammar consists of alphanumeric characters, an underscore symbol and rounded parenthesis.

- { } enclose items which may be repeated zero or more times.
- [ ] enclose an optional item.
- **keyword** is a terminal of the grammar.
- ::= means "is defined as".
- | means "or".

```
rules ::= {rule}
rule ::=  class-definition
        | [create] name-definition;
        | [create] type-definition;

class-definition ::= class-name [class-inheritance] [class-renaming] class-type
                     [class-methods] end [class];
class-name ::=  [create] class class-name
class-inheritance ::=  inherit class-name
                       {, class-name}
class-renaming ::=  rename property-renaming
                    {, property-renaming}
property-renaming ::=  method-or-attribute property-name
                       [from superclass-name]
                       as new-property-name
method-or-attribute ::=  method
                         | attribute
```

```
access-method ::= public
                | private
                | read
class-type ::= [access-method] [type-spec]
class-methods ::= method method-declaration
                {, method-declaration}
method-declaration ::= [access-method] method-name signature
signature ::= [parameter-list] [return-type-spec]
parameter-list ::= (parameter-name : type-spec
                {, parameter-name : type-spec} )
return-type-spec ::= type-spec
O2-type ::= name-of-class
          | type-name
          | tuple(attribute-list)
          | (attribute-list)
          | list(type-spec)
          | set(type-spec)
          | unique(type-spec)
simple-type ::= char
              | integer
              | real
              | boolean
              | string
              | bits
type-spec ::= O2-type
            | simple-type
attribute-list ::= attribute
                 {, attribute}
attribute ::= access-method attribute-name : type-spec

name-definition ::= [constant] name type-name : type-spec

type-definition ::= type type-name : type-spec

type-name ::= string-literal
class-name ::= string-literal
property-name ::= string-literal
superclass-name ::= string-literal
new-property-name ::= string-literal
method-name ::= string-literal
parameter-name ::= string-literal
```

```
letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M
         | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
         | a | b | c | d | e | f | g | h | i | j | k | l | m
         | n | o | p | q | r | s | t | u | v | w | x | y | z | _
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter-or-digit ::= letter
                  | digit
string-literal ::= letter
                 {letter-or-digit}
```

Appendix B

User Evaluation Details

This appendix contains the eight queries that were presented to the participants in the user evaluation (Chapter 8). The queries were chosen so that their translations would require complicated traversals and would include all implemented features of QUIVER. The user evaluation was conducted with a schema similar to the university schema presented in Figure 4.5, page 47; the schema was modified to contain *bedrooms*. Buildings on campus each contain a bag of bedrooms, and some students each stay in a bedroom. If a student does not stay on campus, that student's `stays_in` property value is `nil`.

The modified schema (in O_2 ODL) is displayed in Figure B.1.

The queries follow:

1. Are there any students that have their birthdays today?
2. Find all the level 3 courses.
3. Find those professors who teach their own children.
4. Find the students that live in the same building as they are lectured in (in any of their courses).
5. Find the courses whose modules are lectured in *all* buildings, ie. if you sum all the buildings where a course's modules are lectured, you get the bag of all buildings on campus.
6. Find the students taking all level 3 courses.
7. Find all the students that qualify for a supplementary examination in at least 1 course (i.e. they scored between 45 and 49 inclusive).
8. What is the average enrollment in all the sections taught by professor Jones?

```
type Date : tuple(day: integer,
        month: integer,
        year: integer);

type Address : tuple(number: integer,
        street: string);

class Person public type
        tuple(name: string,
                        birthdate: Date,
                        spouse: Person,
                        lives_in: set(Address),
                        children: set(Person))
end;

class Professor inherit Person public type
        tuple(teaches: set(Module)) end;

class Student inherit Person public type
        tuple(takes: set(Course),
                        stays_in: Bedroom)
        method
                courses_by_marks (min: integer, max: integer): set(Course)
/* the method returns the Courses taken by this student, for which the  */
/* student has scored between min and max */
end;

class Employee inherit Person public type
        tuple(boss: Employee) end;

class TA inherit Student, Employee public type
        tuple(assists: set(Module)) end;

class Course public type
        tuple(name: string,
                        consists_of: set(Module),       /* course consists of many modules */
                        level: integer)                 /* level=3 means 3rd year, etc */
end;

class Module public type
        tuple(name: string,
                        lectured_in: Building,
                        enrollment: integer,
                        dept: string) end;

class Building public type
        tuple(name: string,
                        bedrooms: set(Bedroom))         /* building consists of many rooms, */
end;

/* named sets follow.  They are assumed to be the extents of their types */
name Persons: set(Person);
name Professors: set(Professor);
name Students: set(Student);
name TAs: set(TA);
name Courses: set(Course);
name Modules: set(Module);
name Employees: set(Employee);
name Buildings: set(Building);
name Bedrooms: set(Bedroom);

/* Jones is a reference to a particular Professor object */
name Jones: Professor;
```

Figure B.1: University database schema used in the user evaluation

Appendix C

Comparison with other Systems

In this appendix, queries constructed in the visual query systems discussed in Chapter 3 are compared to similar queries constructed in QUIVER. All QUIVER queries are displayed with full-labels turned off. This appendix is not a proof of the expressive power of QUIVER queries, but is meant to provide a reasonable feel of the versatility of the QUIVER query language.

C.1 VQL, Vadaparty *et al*

Figure C.1 is identical to the VQL query in Figure 3.3. It represents the query "Get the models of all the blue cars that are driven by the presidents of the company that manufactures them".

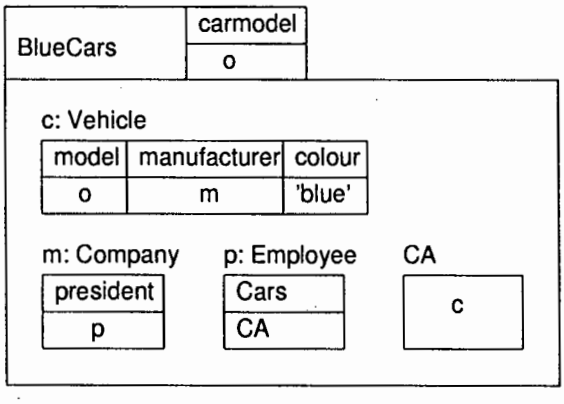


Figure C.1: Finding models of blue cars

The QUIVER query of the VQL query in Figure C.1 is displayed in Figure C.2. The classes `Vehicle`, `Company` and `Employee` and the extent `Employees` have

been defined. A vehicle has a model, a colour and is manufactured by a company. A company has a president and an employee owns many vehicles (the `Cars` property).

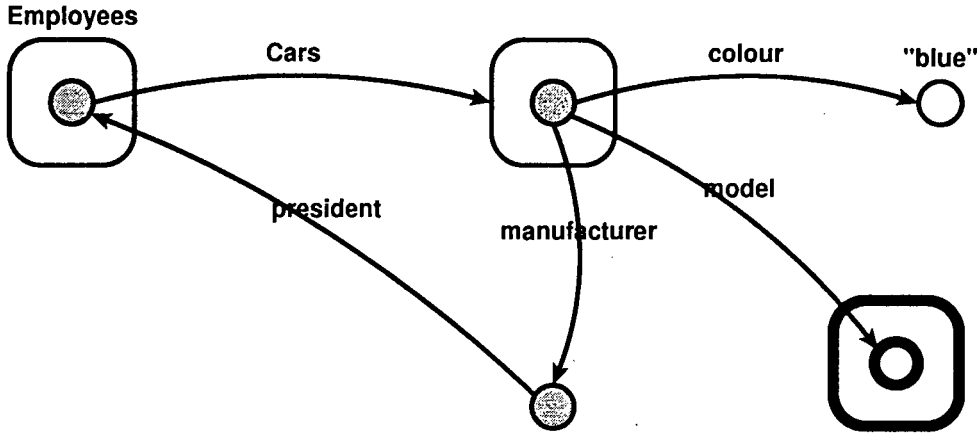


Figure C.2: QUIVER equivalent of the query in Figure C.1

Consider the query "Get those item names that are sold by *every* department on the second floor". The VQL query in shown in Figure C.3.

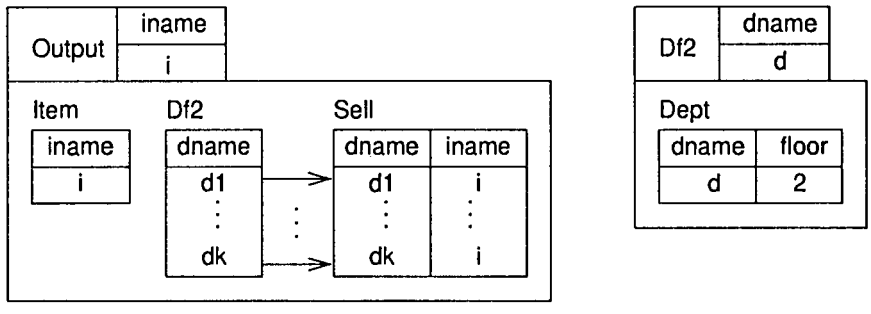


Figure C.3: Finding items on the second floor

The equivalent QUIVER query is displayed in Figure C.4. The classes `Dept` and `Sell` and extents `Depts` and `Sells` have been defined. `Dept` and `Sell` each have two components; instances of `Dept` link a department name to its floor and instances of `Sell` link a department name to the name of an item sold in it.

The subquery toward the top left of the query finds all department names that are on the second floor. The other subquery finds all department names that sell a particular item. If those departments are a superset of all departments on the second floor, the item name is added to the output.

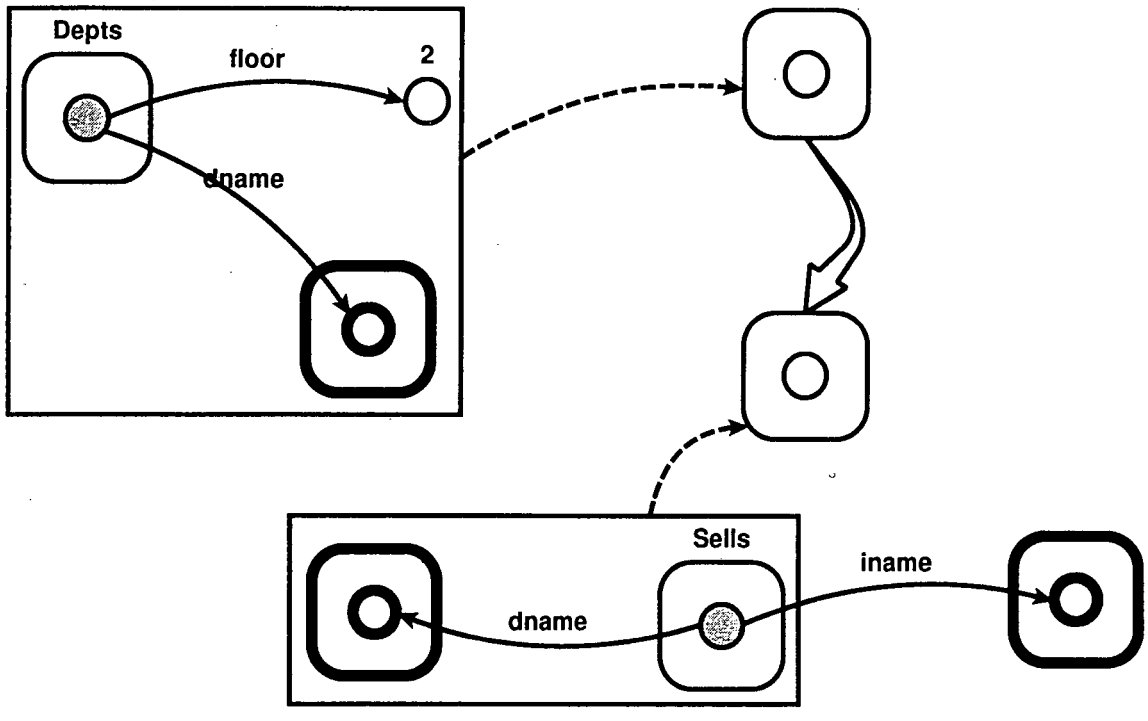


Figure C.4: QUIVER equivalent of the query in Figure C.3

C.2 VQL, Mohan *et al*

The query in Figure C.5 finds those department names that sell all the items that are manufactured by the manufacturer "Ping"; the query is identical to the query in Figure 3.5.

The QUIVER query is presented in Figure C.6. The schema consists of three classes, namely `Item`, `Manufacturer` and `Department`. An item is made by a manufacturer, a manufacturer has a name and a department sells a set of items; a department has a name as well. The extents `Items` and `Departments` are also defined.

C.3 GOOD

Figure C.7 is identical to the GOOD query that was presented in Figure 3.7. The query in Figure C.8 is its QUIVER equivalent; the query finds all *grandparent* relationships in the database. The answer consists of a bag of tuples, each consisting of components `grandparent` and `grandchild`.

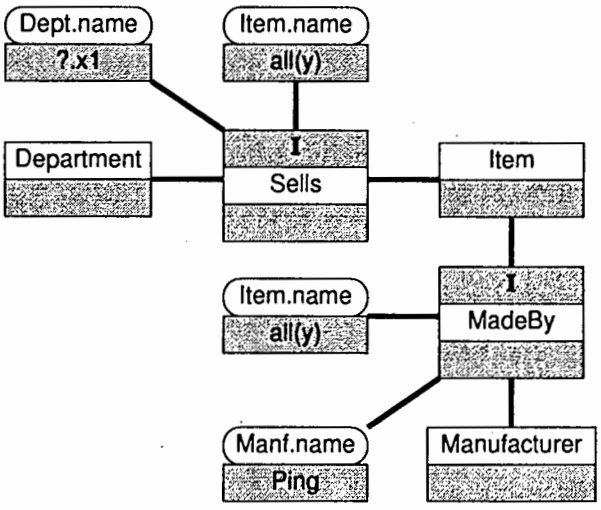


Figure C.5: Finding manufacturers in VQL

C.4 OdeView

Figure C.9 is identical to the OdeView query that was presented in Figure 3.10. The query retrieves all employees in department 11252 who either earn more than 5000 dollars, or whose ages are between 50 and 60.

The QUIVER translation is displayed in Figure C.10. The schema defines the classes `Employee` and `Department` and the extent `Employees`. An employee works in a department (the `dept` property) and has a salary. The method `age_between` returns `true` if an employee's age lies between a specified range of ages and `false` otherwise. A department has a department number.

Note that the query in Figure C.10 cannot be translated under the current implementation of QUIVER, as inequalities and the disjunction of conditions have not been implemented.

C.5 GraphLog

Consider the Hy$^+$ query in Figure C.11; this query appeared in [14] and was presented earlier in Figure 3.11. The `mem` edge in the pattern is thick, and is a visual way of distinguishing edges that the user wants to see after the match is found. The query finds subclasses of class `Object` that redefine some function `F` defined in `Object`. The subclasses identified are either direct subclasses or indirect subclasses, as they are found using a transitive closure operation. The relation `mem` between between the subclass `C` and the inherited function `F` is displayed to the user.

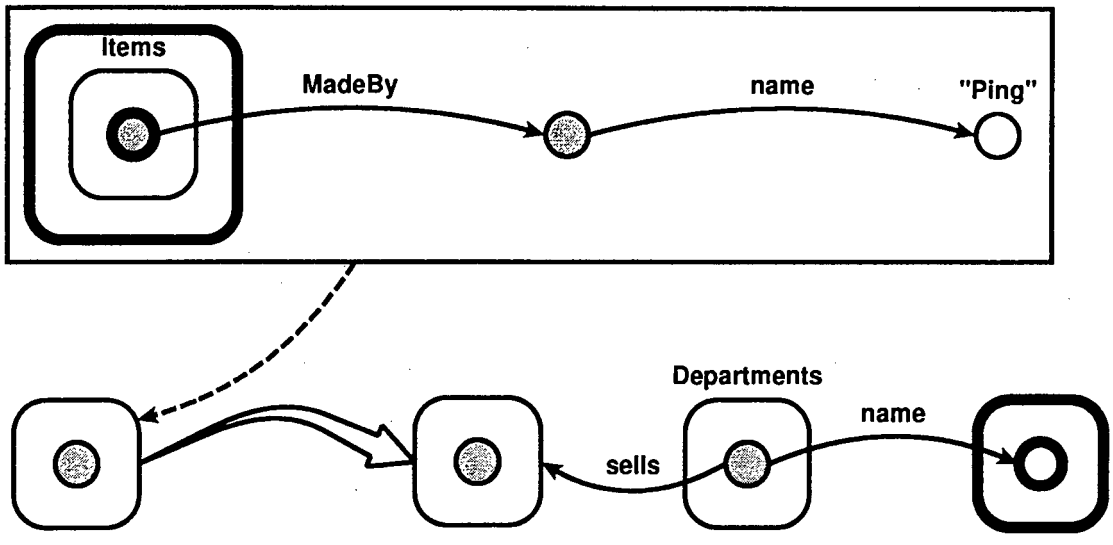


Figure C.6: QUIVER equivalent of the query in Figure C.5

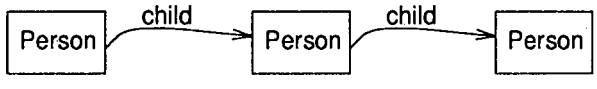


Figure C.7: Finding grandparent relationships

QUIVER does not support transitive closures, and so the O_2 schema has been defined accordingly. Classes `ObjectType` and `FunctionType` (with their extents) have been defined. The `subclasses` property of `ObjectType` returns the transitive closure of all subclasses of the object, and `mem` returns the member functions defined by a class. The QUIVER translation is displayed in Figure C.12

The query returns a bag of tuples, each consisting of components `class` and `function`.

C.6 QBD*

Consider the QBD* query in Figure C.13; the query appeared in [4]. The query finds all students whose names are `Brown` and whose credits are equal to 30.

The QUIVER equivalent is displayed in Figure C.14. The schema defines a class `Student` (with extent `Students`); each student has a name, credits and an age.

C.7 VDL

Figure C.15 is identical to the query in Figure 3.16; the query returns the names of students who take at least one course from each instructor.

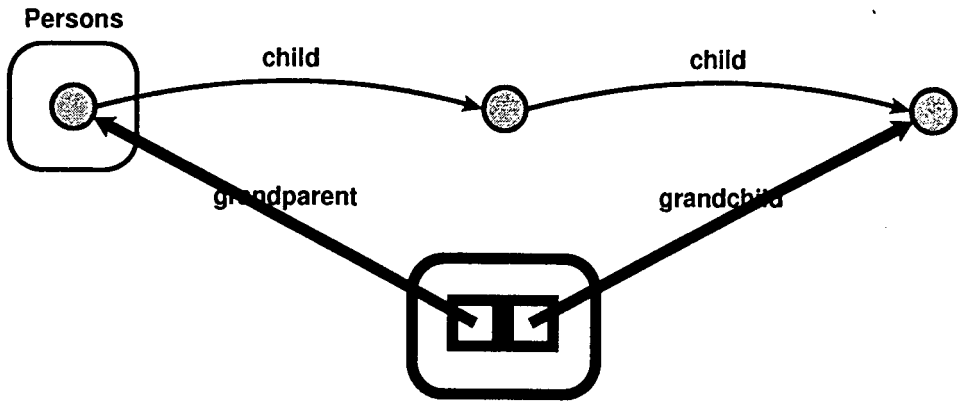


Figure C.8: QUIVER equivalent of the query in Figure C.7

Class employee Selection

Extend	OPERAND	OP	OPERAND	OP	OPERAND
Shrink	dept->dno	==	"11252"	&&	()
Clear					

Extend	OPERAND	OP	OPERAND	OP	OPERAND
Shrink	salary	>	5000	||	age_between
Clear					
Done					

Clear	ARG 0	ARG 1
Done	50	60

Generated Predicate
((strcmp(p->dept->dno,"11252"==0) && ())

Figure C.9: Retrieving employees

The QUIVER translation is displayed in Figure C.16.

The schema contains `Student`, `Course` and `Instructor` objects, and extents `Students` and `Instructors`. Each instructor has an `instruct` property, defined as a bag of two components: `course`, which indicates the course that the instructor instructs and `students`, indicating the students instructed in that course. The courses that a student takes can be retrieved through the `courses` property of a student.

The subquery finds all the instructors that instruct a particular student for a course that the student takes. If this bag of instructors contains at least all the instructors in the database, the student's name is added to the query output.

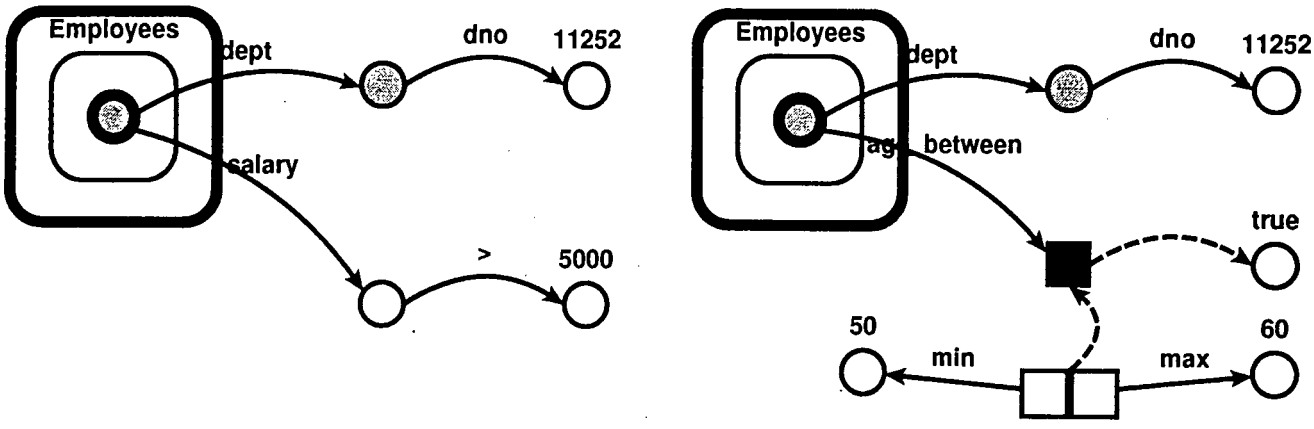


Figure C.10: QUIVER equivalent of the query in Figure C.9

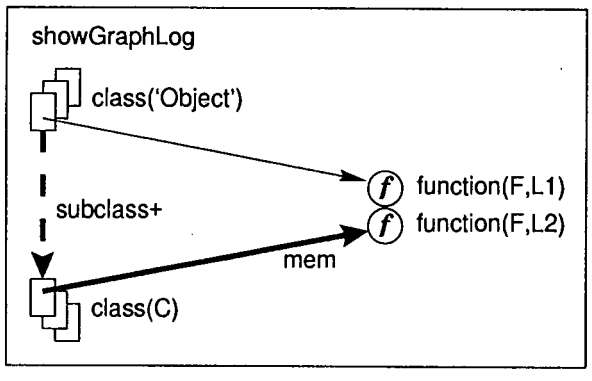


Figure C.11: A filter query in Hy$^+$

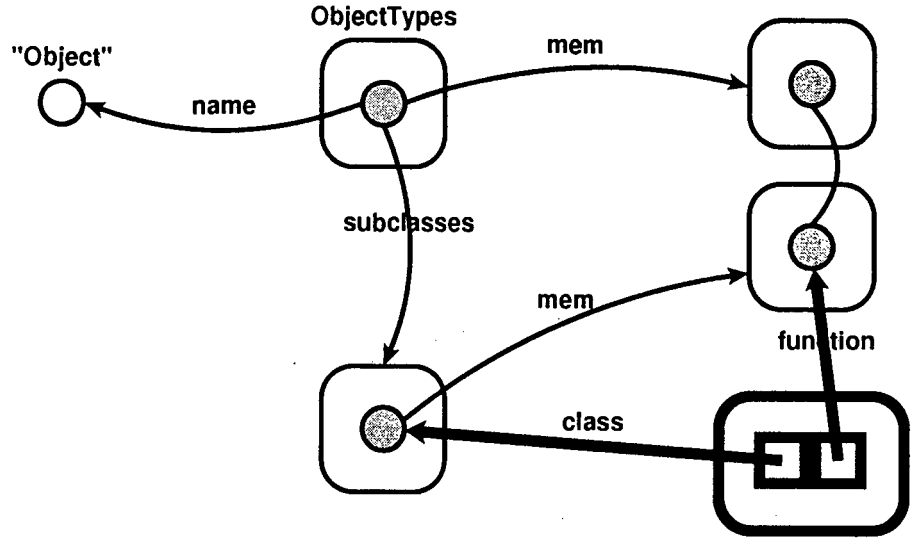


Figure C.12: QUIVER equivalent of the query in Figure C.11

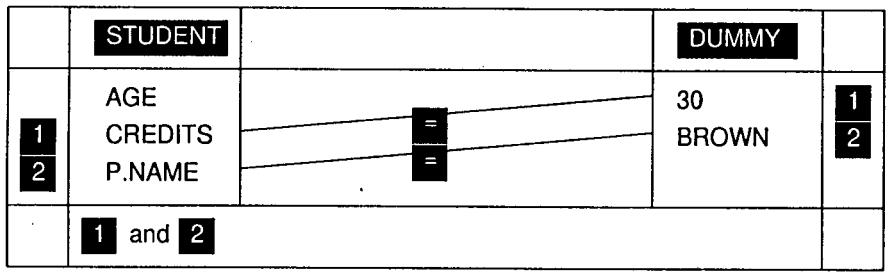


Figure C.13: Finding students

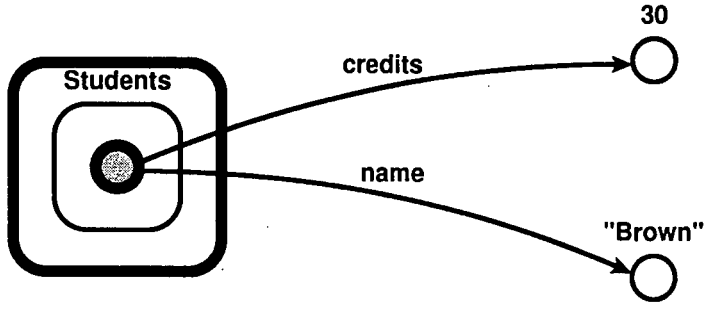


Figure C.14: QUIVER equivalent of the query in Figure C.13

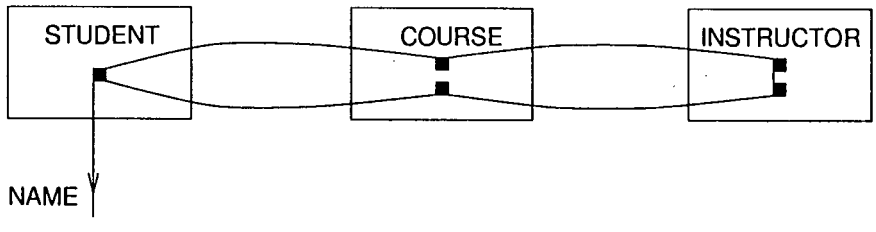


Figure C.15: Returning names of students

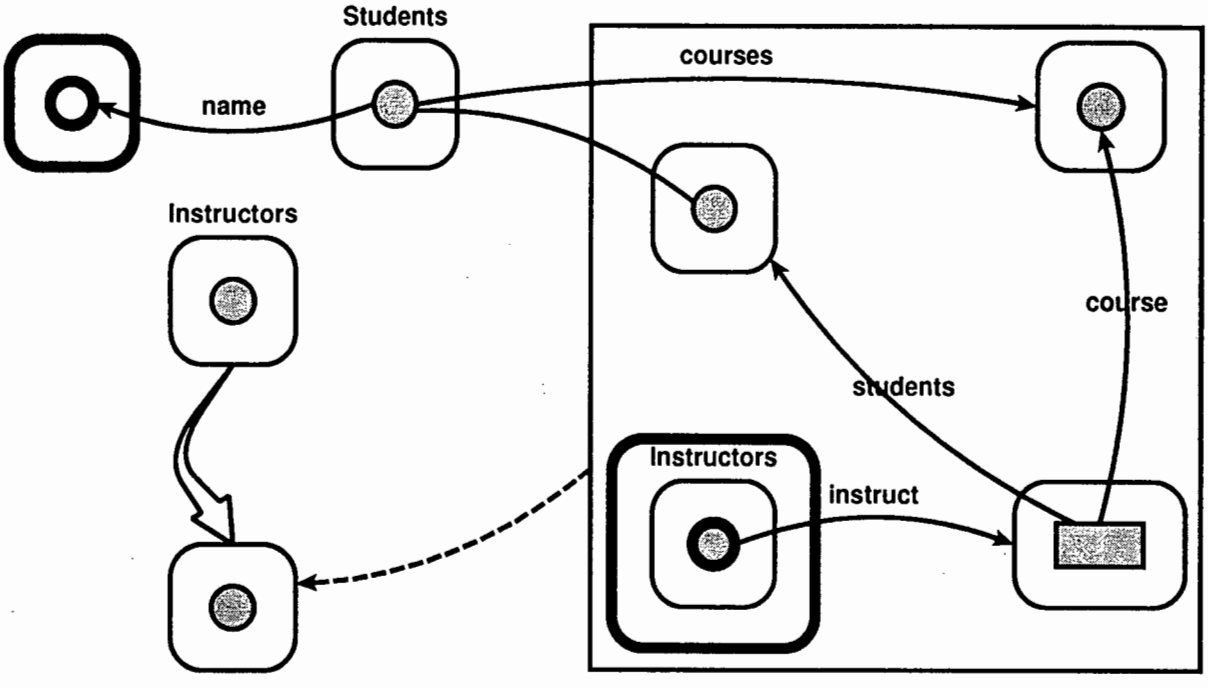


Figure C.16: QUIVER equivalent of the query in Figure C.15

Bibliography

[1] R. Agrawal and N. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *SIGMOD International Conference on Management of Data*, pages 36–45, Portland, Oregon, 1989.

[2] R. Agrawal, N. Gehani, and J. Srinivasan. OdeView: The Graphical Interface to Ode. In *SIGMOD International Conference on Management of Data*, pages 34–43, Atlantic City, NJ, 1990.

[3] A. Alashqur, S. Su, and H. Lam. OQL : A Query Language for manipulating Object-Oriented Databases. In *Proceedings of the 15th Very Large Data Bases Conference*, pages 434–442, Amsterdam, The Netherlands, August 1989.

[4] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: A graphical query language with recursion. *IEEE Transactions on Software Engineering*, 16(10):1150–1163, October 1990.

[5] I. Arpinar, A. Dogac, and C. Evrendilek. MoodView: An Advanced Graphical User Interface for OODBMSs. *SIGMOD Record*, 22(4):11–18, December 1993.

[6] T. Atwood, J. Duhl, G. Ferran, M. Loomis, and D. Wade. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1993.

[7] P. Atzeni and P. Chen. Completeness of Query Languages for the Entity Relationship Model. In *Second International Conference on the Entity Relationship Approach to Information Modelling and Analysis*, pages 111–124, Washington, D.C., 1981. North-Holland.

[8] C. Batini, T. Catarci, M. Costabile, and S. Levialdi. Visual query systems: analysis and comparison. *Journal of Visual Languages and Computing*, 8(2), June 1997.

[9] A. Chandra and D. Harel. Horn Clause Queries and Generalization. *Journal of Logic Programming*, 1(1):1–15, 1985.

[10] S. Chang. *Principles of Pictorial Information Systems Design.* Prentice-Hall, 1989.

[11] E. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[12] E. Codd. *Relational Completeness of Database Sub-Languages.* Prentice-Hall, 1972.

[13] M. Consens and A. Mendelzon. GraphLog: a Visual Formalism for Real Life Recursion. In *Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville, Tennessee*, pages 404–416, 1990.

[14] M. P. Consens, F. C. Eigler, M. Z. Hasan, A. O. Mendelzon, N. E. G., A. G. Ryman, and D. Vista. Architecture and applications of the Hy^+ visualization system. *IBM Systems Journal*, 33(3):458–476, 1994.

[15] I. Cruz. DOODLE: A Visual Language for Object-oriented Databases. In *SIGMOD International Conference on Management of Data*, pages 71–80, San Diego, California, June 1992.

[16] S. Dar, H. Gehani, V. Jagadish, and J. Srinivasan. Queries in an Object-Oriented Graphical Interface. *Journal of Visual Languages and Computing*, 6(1):27–52, March 1995.

[17] C. Date and H. Darwen. *A Guide to the SQL Standard.* Addison-Wesley, 3rd edition, 1993.

[18] M. Gemis, J. Paradeans, and I. Thyssens. A Visual Database Management Interface Based on GOOD. In *Interfaces to Database Systems (IDS) Conference*, pages 155–175, Glasgow, Scotland, 1992.

[19] J. Gibbons. *Nonparametric Methods for Quantitative Analysis.* Holt, Rinehart and Winston, 1976.

[20] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. Technical Report 327, Computer Science Department, Indiana University, 1991.

[21] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[22] D. Hipp (`drh@vnet.net`). Embedded Tk (ET). Hipp, Wyrick & Company, Inc., available via *ftp* from `ftp.std.com/pub/drh/et1_1c.tar.gz`.

[23] Y. Ioannidis, M. Livny, and E. Haber. Graphical User Interfaces for the Management of Scientific Experiments and Data. *SIGMOD Record*, 21(1):47–53, March 1992.

[24] T. Kato, T. Kurita, H. Shimogaki, T. Mizutori, and K. Fujimura. A Cognitive Approach to Visual Interaction. In *International Conference on Multimedia Information Systems*, pages 109–120, Singapore, 1991.

[25] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[26] R. King. A Database Management System Based on an Object-Oriented Model. In L. Kerschberg, editor, *Expert Database Systems.* The Benjamin/Cummings Publishing Inc, 1986.

[27] E. Koutsofios and S. C. North. DOT: a directed graph layout engine. AT&T Bell Laboratories, Murray Hill, NJ, October 1993. Available via *World Wide Web* from `http://seclab.cs.ucdavis.edu/~hoagland/Dot.html`.

[28] M. Levene, A. Poulovassilis, K. Benkerimi, S. Schwartz, and E. Tuv. Implementation of a Graph-Based Data Model for Complex Objects. *ACM SIGMOD RECORD*, 22(4):26–31, December 1993.

[29] D. Libes. A Debugger for Tcl Applications. In *Tcl/Tk Workshop*, Berkeley, California, June 1993. National Institute of Standards and Technology. Available via *World Wide Web* from `http://www.x.co.uk/of_interest/-tcl/busco.lanl.gov/pub/tk-debug.tar.gz`.

[30] L. Mohan and R. L. Kashyap. A Visual Query Language for Graphical Interaction with Schema-Intensive Databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):843–858, October 1993.

[31] S. Naqvi. A Logic for Negation in Database System. In *Workshop on Logic in Databases*, Washington, D.C., 1986.

[32] J. Nijtman (`nijtmans@nici.kun.nl`). Dashed and Stippled outlines in Tk4.2 and Tk4.1(p1), September 1996. Available via *World Wide Web* from `http://www.nici.kun.nl/~nijtmans/tcl/patch.html`.

[33] O_2 Technology. *O_2 OQL Manual*, Version 4.5, November 1994.

[34] O_2 Technology. *O_2Tools User Manual*, Version 4.5, November 1994.

[35] O_2 Technology. *O_2C Reference Manual*, February 1995.

[36] Object Database Management Group. Response to the ODMG-93 Commentary written by Dr. Won Kim of UniSQL, Inc. *SIGMOD Record*, 23(3):3–7, September 1994.

[37] L. Orman. A Visual Data Model. *Data and Knowledge Engineering*, 7:227–238, 1991.

[38] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994. The `Tcl/Tk` home page is at the *World Wide Web* address `http://www.sco.com/-Technology/tcl/Tcl.html`.

[39] J. Paredaens, J. Van den Bussche, M. Andries, M. Gemis, M. Gyssens, I. Thyssens, D. Van Gucht, V. Sarathy, and L. Saxton. An Overview of GOOD. *SIGMOD Record*, 21(1):25–31, March 1992.

[40] F. Paulisch. *The design of an extendible graph editor*, chapter 5, pages 73–99. Springer-Verlag, 1993.

[41] A. Poulovassilis and M. Levene. A Nested-Graph Model for the Representation and Manipulation of Complex Objects. *ACM Transactions on Information Systems*, 12(1):35–68, 1994.

[42] A. Poulovassilis and C. Small. A Functional Programming Approach to Deductive Databases. In *Proceedings of the 17th Very Large Data Bases Conference*, pages 491–500, Barcelona, 1991.

[43] E. Ramez and S. B. Navathe. *Fundamentals of Database Systems*, chapter 3, pages 39–68. The Benjamin/Cummings Publishing Company, Inc., 1994.

[44] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, General Electric Research and Development Center, Schenectady, New York, 1991.

[45] Y. Shirota, Y. Shirai, and T. Kunii. Sophisticated Form-Oriented Database Interface for Non-Programmers. In T. Kunii, editor, *Visual Database Systems*, Tokyo, Japan, April 1989. North-Holland.

[46] S. Sippu and E. Soisalon-Soininen. A Generalized Transitive Closure for Relational Queries. In *Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, Texas*, 1988.

[47] K. Vadaparty, Y. Aslandogan, and G. Özsoyoglu. Towards a Unified Visual Database Access. In *SIGMOD International Conference on Management of Data*, pages 357–366, Washington, D.C., May 1993.

[48] J. Wiener and Y. Ioannidis. A Moose and a Fox can Aid Scientists with Data Management Problems. In *Database Programming Languages*, pages 376–398, New York City, 1993. Springer-Verlag.

[49] A. Zeller and D. Lütkehaus. *DDD – A Free Graphical Front-End for UNIX Debuggers.* Abteilung Softwaretechnologie, Technische Universität Braunschweig, Germany. Available via *ftp* from `ftp.gre.ac.uk/pub/tools/-debuggers/ddd/`.

[50] M. Zloof. Query-By-Example: A Database Language. *IBM Systems Journal*, 16(4):324–343, 1977.