A VISUAL QUERY LANGUAGE FOR PART21 FILE
BASED ON EXPRESS DATA MODEL

A Thesis Presented to

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

by

Chunsheng Nie

March, 2000

OHIO UNIVERSITY
LIBRARY
THIS THESIS ENTITLED

"A VISUAL QUERY LANGUAGE FOR PART21 FILE
BASED ON EXPRESS DATA MODEL"

by Chunsheng Nie

has been approved

for the Department of Industrial and Manufacturing Systems Engineering

and the Russ College of Engineering and Technology

Robert P. Judd, Cooper Industries Professor
School of Electrical Engineering and Computer Science

Warren K. Wray, Dean
Fritz J. and Dolores H. Russ
College of Engineering and Technology
Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Robert P. Judd for his constant guidance, assistance, and patience in helping me complete this work. Without him, this thesis would never have been possible.

I would also like to thank Dr. David Koonce, who gave me much guidance in VQL design. I am thankful to the other members of my thesis committee, Dr. Charles Parks and Dr. David Juedes, for their co-operation and helpful comments.

Thank also goes to Mr. Lizhong Huang for providing the software resource EQL and TNR, Mr. Dilip Viswanath for helping me using Tcl/Tk.

My wife, Xuemei Li, my parents and my sisters gave me much encouragement and love. Thank you all!

This thesis is dedicated in memory of my grandmother, Meixiang Zen (1910 – 1996), who always recognized the importance of education.
Contents

1 Introduction ........................................... 1
  1.1 Introduction ........................................ 1
  1.2 Current Visual Query Languages ..................... 3
       1.2.1 QBE ........................................... 4
       1.2.2 ACCESS ........................................ 7
       1.2.3 SUPER ......................................... 8
       1.2.4 HQL ........................................... 10
       1.2.5 GQL ........................................... 12
       1.2.6 Others ......................................... 16
       1.2.7 Summary ....................................... 17
  1.3 Features for Proposed VQL ........................... 19

2 VQL: Graphical Interface and Syntax .................. 21
  2.1 Introduction ........................................ 21
2.2 EXPRESS Model and Its Visual Representation

2.2.1 EXPRESS Model .............................................. 22

2.2.2 Graphical Interface for EXPRESS Schema ................. 23

2.3 Visual Query Editor ........................................ 24

2.4 Entity Box .................................................... 27

2.5 Key Link ..................................................... 29

2.6 Condition Link .............................................. 31

2.6.1 Comparison Condition Link ................................ 33

2.6.2 Null Condition Link ...................................... 34

2.6.3 Empty Condition Link ..................................... 35

2.6.4 ISA Condition ............................................ 36

2.6.5 In Condition Link ......................................... 37

2.6.6 Contains Condition Link .................................. 38

2.6.7 Match Condition Link ..................................... 39

2.6.8 For Condition Link ....................................... 40

2.6.9 And/or Condition Link .................................... 42

2.6.10 Negative Condition Link ................................. 44

2.7 Order of Evaluation ......................................... 45

2.8 Query Examples ............................................. 46

3 Query Translation ............................................. 52
3.1 Introduction ................................................. 52
3.2 Data Structure for Translation .............................. 54
3.3 Variable Binding ............................................. 56
3.4 Grouped-by Expression Mapping ............................. 57
3.5 Conditional Logic Tree ....................................... 59
3.6 Mapping VQL to EQL ......................................... 63
  3.6.1 Select Expression ........................................ 64
  3.6.2 Variable Declaration ..................................... 66
  3.6.3 Condition Expression .................................... 67
3.7 Translation Example .......................................... 71

4 VQL Usage .................................................. 72
  4.1 Introduction ............................................... 72
  4.2 Overview of the Graphical User Interface .................... 73
  4.3 Entity Box .................................................. 76
  4.4 Key Link ..................................................... 78
  4.5 Condition Link .............................................. 79

5 VQL Implementation ........................................ 82
  5.1 Introduction ............................................... 82
  5.2 Overview of the Implementation ............................. 84
  5.3 Main Program ................................................ 86
5.4 Incr Tcl Classes .................................. 89

5.4.1 Level Classes ................................... 91
  5.4.1.1 Level Class .................................. 91
  5.4.1.2 SubLevel Class ............................... 93

5.4.2 Entity Class ...................................... 93

5.4.3 Key Link Class ................................... 96

5.4.4 Condition Link Classes ............................ 98
  5.4.4.1 ConditionLink Class ............................ 99
  5.4.4.2 BinaryLink .................................... 101
  5.4.4.3 UnaryLink Class ............................... 103
  5.4.4.4 ForLink Class .................................. 103
  5.4.4.5 LogicLink Class ............................... 104

5.4.5 Constant Class ................................... 104

5.4.6 Operator Classes .................................. 105
  5.4.6.1 Operator Class ................................ 105
  5.4.6.2 Anyall Class .................................. 107

5.4.7 Splitor Class ..................................... 108

5.5 Incr Tk Class ...................................... 109
  5.5.1 EntityBox Class .................................. 109
  5.5.2 ListboxDialog ................................... 110
  5.5.3 KeyDialog Class .................................. 110
5.6 TNR Programs ................................. 110
  5.6.1 eList.tnr .................................... 111
  5.6.2 aList.tnr .................................... 111
  5.6.3 output.tnr ................................... 111

6 Conclusions and Future Work .................. 112
  6.1 Introduction .................................... 112
  6.2 Conclusions .................................... 112
  6.3 Future Work .................................... 113

A An EXPRESS Sample .............................. 117

B A STEP File Sample .............................. 119
List of Tables

5.1 VQL Program Structure .................................................. 85
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Simple Relational Schema in QBE</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Illustrating Join Operation in QBE</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Specifying Complex Conditions in QBE</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>ACCESS Interface and a Query Example</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>A Simple Schema Diagram in SUPER</td>
<td>9</td>
</tr>
<tr>
<td>1.6</td>
<td>A Query Example in SUPER</td>
<td>10</td>
</tr>
<tr>
<td>1.7</td>
<td>A Simple Schema Diagram in HQL</td>
<td>11</td>
</tr>
<tr>
<td>1.8</td>
<td>A Query Example in HQL (Hybrid Query language)</td>
<td>12</td>
</tr>
<tr>
<td>1.9</td>
<td>A Simple Schema Diagram in GQL</td>
<td>13</td>
</tr>
<tr>
<td>1.10</td>
<td>Query 1 in GQL</td>
<td>15</td>
</tr>
<tr>
<td>1.11</td>
<td>Query 2 in GQL</td>
<td>16</td>
</tr>
<tr>
<td>1.12</td>
<td>Query 3 in GQL</td>
<td>17</td>
</tr>
<tr>
<td>2.1</td>
<td>A simple EXPRESS schema</td>
<td>25</td>
</tr>
<tr>
<td>2.2</td>
<td>The Hierarchy of Visual Types in VQL</td>
<td>26</td>
</tr>
<tr>
<td>2.3</td>
<td>The Syntax for Entity Box</td>
<td>27</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.4</td>
<td>The Syntax for Key Link</td>
<td>30</td>
</tr>
<tr>
<td>2.5</td>
<td>The Syntax for Comparison Condition Link</td>
<td>33</td>
</tr>
<tr>
<td>2.6</td>
<td>The Syntax for Null Condition Link</td>
<td>34</td>
</tr>
<tr>
<td>2.7</td>
<td>A Query Example Using Null Condition Link</td>
<td>35</td>
</tr>
<tr>
<td>2.8</td>
<td>The Syntax for Empty Condition Link</td>
<td>36</td>
</tr>
<tr>
<td>2.9</td>
<td>A Query Example Using Empty Condition Link</td>
<td>36</td>
</tr>
<tr>
<td>2.10</td>
<td>The Syntax for ISA Condition Link</td>
<td>37</td>
</tr>
<tr>
<td>2.11</td>
<td>The Syntax for In Condition Link</td>
<td>38</td>
</tr>
<tr>
<td>2.12</td>
<td>The Syntax for Contains Condition Link</td>
<td>39</td>
</tr>
<tr>
<td>2.13</td>
<td>The Syntax for Match Condition Link</td>
<td>40</td>
</tr>
<tr>
<td>2.14</td>
<td>The Syntax for For Condition Link</td>
<td>41</td>
</tr>
<tr>
<td>2.15</td>
<td>A Query Example Using For Condition Link</td>
<td>42</td>
</tr>
<tr>
<td>2.16</td>
<td>The Syntax for Logic Condition Link</td>
<td>43</td>
</tr>
<tr>
<td>2.17</td>
<td>Nested Logic Conditions</td>
<td>44</td>
</tr>
<tr>
<td>2.18</td>
<td>The Syntax for Negative Condition Link</td>
<td>44</td>
</tr>
<tr>
<td>2.19</td>
<td>Query 1 in VQL</td>
<td>46</td>
</tr>
<tr>
<td>2.20</td>
<td>Query 2 in VQL</td>
<td>47</td>
</tr>
<tr>
<td>2.21</td>
<td>Query 3 in VQL</td>
<td>48</td>
</tr>
<tr>
<td>2.22</td>
<td>Query 4 in VQL</td>
<td>48</td>
</tr>
<tr>
<td>2.23</td>
<td>Query 5 in VQL</td>
<td>49</td>
</tr>
<tr>
<td>2.24</td>
<td>Query 6 in VQL</td>
<td>49</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction

There is a wide range of database models and systems as well as data query languages that are used to extract information. Most current languages are textual; this causes a problem for users, especially for novices or casual users. A textual query language lacks a friendly interface, and it requires the user to understand the language syntax and be familiar with the inner structure for the database of interest.

To provide a better user interface, researchers in the database community have been developing various visual query languages, which make use of the two-dimensional nature of the computer screen and use different graphical representations, such as tables, diagrams, and icons, to construct a query. Many kinds of graphical query languages have been evaluated [2]. In this paper, graphical query languages are referred
to as Visual Query Systems (VQSs), which are based on the use of visual representations to depict the domain of interest and to formulate requests to the database by means of visual expressions. VQSs provide both a language to express the queries in a visual format and a variety of functionalities to facilitate user-system interaction. Therefore, they are oriented towards a wider range of users, especially those novice users who have limited database knowledge and who are not interested in the inner structure of the accessed database.

EQL (Express Query Language)[7], designed as a tool for software system integration, is an SQL-like query language which is used to access and manipulate data in part21 files. A part21 file is an object oriented data file using EXPRESS modeling format. Like all other kinds of linear textual query languages, EQL has a set of strict syntax rules and requires the user to understand the schema of the stored data. This is difficult for the casual user, especially when the data is stored in complicated structures. Therefore, to reduce the cognitive load on the database user and to help the user formulate error-free queries, it is desirable to build a VQL (Visual Query Language) that provides an intuitive graphical interface between user and EXPRESS formatted data using EQL as an internal query language. This is the primary objective of this research.
1.2 Current Visual Query Languages

Most visual query languages consist of two visual tools: database schema browsers and visual query editors. The former are applied to navigate in a database to examine the data schema—i.e., entities, relations, attributes, relationships etc.—while the latter are needed to query the data stored in a database using a certain data schema format. The overview in this section focuses on various visual query editors.

A large number of visual query languages for databases have been in the literature over the years, each language being designed for a specific data model and employing particular visual representation forms.

One fundamental classification of VQSs can be made based on the database model used. One group of VQSs is based on relational models, such as QBE [5] and ACCESS [11]. Another group is based on E-R (Entity-Relation) or variants of E-R models; for example, SUPER [4] uses ERC model, GRAQULA [10] uses E-R model, and HQL [1] uses extended E-R model. There are some visual query languages based on other database models; for instance, GQL [8] is based on a functional data model, and QUIVER [3] is based on an object-oriented data model proposed by Object Data Management Group (ODMG). HYPERLOG [6] is based on a deductive data model.

According to the visual representation form for the query, VQSs can be mainly classified as either table-based query languages or graph-based query languages. QBE, ACCESS, and ODEVIEW [9] are table-based while SUPER, GRAQULA, HQL, QUIVER, GQL and HYPERLOG are graph-based.
To introduce current visual query languages, several VQSs mentioned above are discussed with examples in this section.

1.2.1 QBE

QBE (Query By Example) is the first user-friendly visual query language that was developed at IBM Research. It is available as an IBM commercial product as a part of the QMF (Query Management Facility) interface option to DB2. It differs from SQL in that it displays table skeletons that are the column headings of relations. When using QBE, the user does not have to remember the names of attributes or relations. The query is formulated by filling in templates of relations that are displayed on a terminal screen. The user does not have to follow any rigid syntax rules for query specification; rather, constants and variables are entered in the columns of the table to construct an example related to the retrieval or update request. For a complicated query, the user inputs the condition in a condition box using textual expression; however, this defeats the graphical nature of the language.

In QBE, the user interface first allows the user to choose the relations needed to formulate a query by displaying a list of all relation names. The templates for the chosen relations are then displayed. Figure 1.1 shows the templates for the relations of “EMPLOYEE” and “WORKS_ON”. The user moves to the appropriate columns in the templates and specifies the query. When entering constant values into a template, we type them as they are; when entering a variable—called an example element in
QBE— we type the "-" character before the variable. The prefix "P.", standing for "print", is used to indicate that the values of a particular column are to be retrieved. The comparison operators other than = (such as > or ≥) must be explicitly entered before typing a constant value. The negation symbol ¬ is used in a manner similar to the NOT EXISTS function of SQL. QBE also supports grouping or common aggregate functions, such as AVG., SUM., CNT., MAX., and MIN. All conditions specified on the same row of a relation template are connected by the "and" logic connective, whereas conditions specified on distinct rows are connected by "or". A join operation is specified in QBE by using the same variables.

Figure 1.2 shows a query: list the names and addresses of all employees who work for the "Research" Department. In this query, the relations of "EMPLOYEE"
and "DEPARTMENT" are joined by variable .DX. Figure 1.3 shows a query: list the social security numbers of employees who work on both project 1 and project 2.

In this case, the condition (.EX=.EY) in the condition box makes the .EX and .EY variables bind only to identical ESSN values.
1.2.2 ACCESS

ACCESS is a Microsoft software product. It also uses a table-based query language to retrieve information from a relational database. In ACCESS, the query window is split into two main sections. The top section is a relationship window. All relations and relationships being used in a query are shown in this window using diagrams and graphs. The bottom section is a table, called a QBE grid, which is used to formulate a query. Each column of the QBE grid is a field. For each field, a user defines query parameters, such as criteria and sorting, in the rows of the QBE grid. Figure 1.4 shows the ACCESS interface and a query: select the customer name and contract name handled by Speedy EXPRESS or United Package Company, and the shipdate is after 8/10/94.

Figure 1.4: ACCESS Interface and a Query Example
To construct a query, a user first chooses the interested relations from a relation dialog box and adds them into the relationship window. ACCESS will automatically find any relationships among multiple relations and show those relationships by drawing lines between the related attributes. Then the user double-clicks any needed attributes in the relationship window, and the names of the attributes and relations will be automatically filled in the QBE grid. Finally, the user has to fill in a selection condition in the QBE grid. Unlike IBM QBE mentioned above, ACCESS does not need to use variables in a QBE grid to represent join conditions; instead, the join relationship is graphically represented in the relationship window. In addition, ACCESS query uses one table to hold all of the interested relations and their attributes together, and all selection conditions can be input in the corresponding column. Although ACCESS has a diagrammatic representation for schema, the queries are built by filling in a table. Therefore, the queries are not graphical in nature.

1.2.3 SUPER

SUPER is graphical query language built on the ERC+ data model, which is an extension of the well-known E-R model. It consists of a set of basic objects called entities and relationships, and all these objects may have associated attributes. A relationship is characterized by its minimum and maximum cardinalities. An attribute may be either atomic or complex. Figure 1.5 shows a simple ERC+ diagram, where a rectangle represents an entity, a diamond represents a relationship, a single continuous
Figure 1.5: A Simple Schema Diagram in SUPER

line represents a 1:1 link, a single dotted line represents a 0:1 link, a double dotted line represents a 0:n link, and a double line with a single dotted line represents a 1:n link.

Construction of a query in SUPER consists of three steps:

- Defining the frame of the query: first, the user has to select the object and relationship types used in the query from the schema diagram, then the subset of the schema should be transformed to an acyclic tree with one root (represented by a heavy box), and all irrelevant attributes are removed.

- Formatting the output: the user selects which data items (attributes) are included in the structure of the result.

- Specifying predicates: in this step, the user fills all selection condition in so called “predicate box”.
Based on the data schema given by Figure 1.5, Figure 1.6 gives a query example: choose the names and addresses of persons who insure a 1984 Ford.

![Query Example Diagram]

**Figure 1.6: A Query Example in SUPER**

### 1.2.4 HQL

HQL is a hybrid query language, in which queries are expressed by means of a mixture of graphical and textual elements. This language is built on an extended E-R model called EER. Figure 1.7 is a graphical representation for EER schema that models the world of surfers. In this model, an entity is represented by a rectangle, a relationship is represented by a diamond, and an “IS A” relationship is represented by a triangle. Multivalued attributes and components are represented by an oval, including a square connected to the corresponding entity type or atomic value type.
via an arrow. A database based on the EER model has a formally defined and highly expressive SQL-like query language, called SQL/EER.

The advantage of such a hybrid language is to allow users the graphical formulation of those parts of a query that are more easily formulated graphically than textually and leave the other parts to be expressed in textual format. Therefore, users can construct queries in the easiest way. However, users are required to understand the syntax for SQL/EER. Figure 1.8 is an HQL query example that lists the names of the persons sharing an address with John. In this query, the "from" clause in SQL/EER is replaced by the graphical representation that is chosen directly from a
schema diagrammatic graph. The variables p1 and p2, used in the textual part, are declared in the graphical part.

1.2.5 GQL

GQL is a graphical query language based on a functional data model. The functional data model is similar to the E-R model except that the functional data model only supports binary relationships, called functions. In the graphical representation of a functional data model, entities are represented as nodes in a graph with the functions as the labeled directed arcs between them. There are two kinds of entities—abstract entities and lexical entities—which correspond to entities and attributes for the E-R model, respectively. Abstract entities are shown as circular nodes and lexical entities as ovals; single-valued functions are shown by open arrowheads, and multi-valued functions by closed arrowheads. Figure 1.9 shows a database schema using the functional data model.

A GQL query consists of copies of nodes and edges coming from some schema and a number of constructs that allow for the query specification. Unlike other visual query languages mentioned above, GQL not only directly inherits the graphical con-
structs for join operations from the schema, but other query operations are visualized by various intuitive graphical constructs, as well. For example, GQL uses ticks for representing the projection, relational operator for selection, union boxes for union, negated truth-value boxes for difference operator, etc. Therefore, users only need to enter minimum textual components. GQL has many constructs; some primary constructs are listed as follows:

- A tick is placed around some objects and denotes projection, i.e., inclusion of the object's value(s) in the result; the tick subscript determines the order that value(s) appears in the result.

- Constants are a particular type represented as nodes. Constants can be used with relational operators for expressing selection conditions.
• Comparison operators are edges labeled “=”, “(” “)”, etc. This kind of construct must connect two nodes of the same type and is used for comparing the value of two nodes. Different conditions are by default ANDed together.

• A box is represented by a rectangle and must contain one or more nodes and/or boxes. There are two kinds of boxes: collection boxes and true-value boxes. A box can be arbitrarily nested and also be ticked. A collection box results in an aggregate output (as in Figure 1.10) while a truth-value box returns true/false (as in Figure 1.11). There exists positive and negative types of true-value boxes. An unticked true-value box expresses existential or negated existential quantification. In addition, a box can represent a union, in which case it must contain at least two boxes (as in Figure 1.12).

• Clouds are used for representing disjunctions. A disjunction of the form “A or B or C...” can be represented by placing a cloud around each of A, B, C ... and then connecting these clouds together (as in Figure 1.11).

• Collection comparison operators are represented as edges between pairs of collection boxes and are labeled as “Includes”, “Equals”, etc. (as in Figure 1.12).

• Computed values are nodes (labeled as “.cv.”) used for holding the result of some computation (as in Figure 1.10).
Figure 1.10: Query 1 in GQL

- Aggregate functions are represented as edges going from a collection box to a computed value node and are labeled as "Count" (see Figure 1.10), "Sum", "Max", etc.

Figures 1.10-1.12 show three query example in GQL. Figure 1.10 shows the query: list the name and status of every supplier in Paris with a status greater than 10, together with the counts of orders from the supplier and the part name and quantity
for each of these orders (aggregate result). Figure 1.11 shows the query: list the name of every supplier with status greater than 10 or an order of a quantity exceeding 500 and not supplying any bolts. Figure 1.12 shows the query: list the name of every supplier who orders all parts that are blue or weigh more than 1 kilogram.

1.2.6 Others

ODEVIEW is based on an object-oriented database, and is a table-based query language. Like ACCESS, ODEVIEW formulates a query by selecting the objects of interest from a graphical schema and filling the condition expression in a query table.

GRAQULA is based on the E-R model, and is a graph-based query language. Like SUPER, GRAQULA needs a textual condition box to express some complicated conjunction/disjunction conditions.
HYPERLOG is based on a deductive data model, and is a graph-based query language. HYPERLOG designates a variable to each instance of an entity type.

Quiver is based on an object-oriented data model. It is also a graphical query language. Like GQL, Quiver provides a fairly rich set of graphical constructs and can express all kinds of query conditions in a graphical manner.

### 1.2.7 Summary

After reviewing current visual query languages, the following observations can be made:

1. Table-based query languages can represent structured data graphically, and users do not need to remember the database structure. However, they do not explicitly express the query predicates graphically and, therefore, the query operations, such as join and selection, must be typed in a textual format. On
the contrary, graph-based query languages can visualize object relationships in a natural way. Furthermore, by providing a set of graphical operation constructs (set, comparison, logic, etc.), they are able to formulate a query in a fully graphical manner.

2. Most current visual query languages are partial visual languages; users still need to input some plain text for the condition expression. Of all VQSs reviewed in this section, only GQL and QUIVER can construct query conditions graphically. For example, both SUPER and GRAQULA use textual condition boxes. Because of the lack of graphically expressive power for a complicated query, HQL allows a query to be a mixture of graphical and textual elements. In this way, the user still must understand the meaning of the syntax.

3. Most current graphical query languages work only for relatively simple queries. When the query becomes complicated—especially when involving a nested query, user specified aggregates, or existential qualifiers—existing graphical query languages do not have a good reasonable solution. For instance, ACCESS and QBE cannot support correlated subqueries; they must resort to SQL to construct this kind of query.

4. Some visual query languages, such as QBE and HYPERLOG, use variables to distinguish various objects. This is a feature of a textual query language and not a natural way for end users.
5. Although a few languages, such as GQL and QUIVER, have high graphical expressive power, a large number of visual constructs influence the simple and intuitive characteristics of visual query languages. Instead of understanding the textual language rules, the user must spend a significant amount of time to learn the graphical syntax.

6. There is no visual query language directly based on the EXPRESS model.

1.3 Features for Proposed VQL

EQL is a textual query language designed to perform ad hoc queries on part21 files in the EXPRESS data model. Both part21 and EXPRESS standards are included in ISO 10303. They are commonly referred as the Standard for the Exchange of Product Model Data (STEP). EXPRESS is an object-oriented data definition language, which is used to create a schema for a part21 file. It specifies both the definitions of data types and the constraints on their instances. It supports inheritance and non-atomic attributes (lists, sets, bags and arrays). Part21 is the clear text encoding of data in the format of the EXPRESS schema. It contains a data section holding all of the user’s data, each of which is an entity instance.

This work will build a graph-based visual query language (VQL) for part21 files. It acts as a graphical interface to the textual query language EQL. VQL should be characterized by:
1. A friendly, fully graphical interface. Each instruction of the textual language is expressed through a set of visual constructs. The user interacts with the system mainly with a mouse-like device, using the keyboard only when needed to input literals. Any user can easily query data without knowing the syntax for a text query language, even while expressing conditions and joins.

2. Unambiguity. VQL should provide precise interpretation of queries. The syntax and semantics for VQL should be defined.

3. Ease of learning. All graphical constructs should be built naturally. The set of visual constructs should be kept as minimal as possible. No variable will be used in the construct of queries. Any casual user should be able to get comfortable with this interface without much effort.

4. High expressive power. VQL should have expressive power to represent entities, relations, attributes, selections, projections, user defined aggregates, and logical operations. It should have at least the same expressive power as EQL.
Chapter 2

VQL: Graphical Interface and Syntax

2.1 Introduction

As mentioned in the previous chapter, the need of a friendly man-machine interaction is becoming crucial for users to extract the information of their interest from a database. VQL (an acronym for Visual Query Language) is designed as the graphical interface of the textual query language EQL to perform the queries on the data of a part21 file. The mechanism for VQL is as follows:

- Externally, VQL enables users to formulate the queries using some natural graphical notations provided by the VQL graphical interface.
- Internally, VQL maps graphical queries to EQL and obtains query results through the execution of EQL.

This chapter focuses on the VQL interface and syntax. The next chapter will discuss issues regarding language translation.

As the data structure of a part21 file is described by its EXPRESS schema and the construction of a query is directly based on this structure, the VQL graphical interface has two functions: visualization of a schema and visualization of queries. In this chapter, the EXPRESS data mode and its visual form are introduced first. Then a set of graphical query constructs and their syntaxes are described. Finally, some VQL query examples are given.

2.2 EXPRESS Model and Its Visual Representation

2.2.1 EXPRESS Model

VQL is built on an EXPRESS model, which is an object-oriented data model. Before the generation of a part21 file (also called STEP file), which is used to store data, a EXPRESS schema for it must be created. An EXPRESS Model views the world as consisting of entities, and each instance for an entity is distinguished by a unique object ID. Properties for an entity are described by its associated attributes. An EXPRESS model supports both atomic data types and aggregate data types.
Atomic data types include real, integer, number, string, boolean, logical, and binary, while aggregate data types include set, list, bag, and array. An EXPRESS model also supports user-defined data types such as enumeration data type and an entity data type. The attribute value for an entity data type is either an object ID or an aggregate of object IDs, which act as pointer(s) to maintain the relationship between entities. An EXPRESS model supports one-to-one, one-to-many, and many-to-many entity relationships. It also supports the inheritance relationships among entities on the basis of their common properties.

An EXPRESS schema called "class_record" and its associated STEP database file are attached in appendices A and B, respectively. All examples in this paper are based on these samples.

### 2.2.2 Graphical Interface for EXPRESS Schema

VQL is used to extract information from the data that has a data structure defined by an EXPRESS schema. To reduce the cognitive load on users, first, we should try to retrieve as much of the schema information as possible, such as entities, attributes, and link relationships, and visualize it.

When an EXPRESS schema file is compiled, an EXPRESS dictionary file called a ROSE file is generated, which keeps the data structure information in schema and can be accessed using EQL. Unfortunately, the ROSE file does not easily provide the
entity link relationships. Therefore, the VQL browser will only list entities and their associated attributes in the schema. The links will be based on the user's knowledge.

Figure 2.1 shows the visual representation for the "class_record" schema. The entity box displays all entities in the schema. Whenever users want to know the detailed information about a certain entity, they only need to click on the entity item in the entity box. Then an attribute box will list all associated attributes on the screen. To simplify the data types, all attributes are classified as either the scalar type, which are represented by the attribute name, or aggregate type, which are represented by the attribute name followed by a square bracket "[]". The object ID for each instance is represented by a special symbol "#". As the EXPRESS model supports the object inheritance relationship, the attribute box for an entity displays both the attributes explicitly specified in its entity definition and the ones inherited from its parent. The dotted lines are added in the figure to help the reader understand the link relationships between entities. Due to the reason mentioned above, VQL does not support the link line at present.

2.3 Visual Query Editor

VQL is designed as a fully graphical query language. It is intended to provide users with an intuitive graphical interface to easily construct simple or complicated queries without having to learn the syntax of the textual query language.
Figure 2.1: A simple EXPRESS schema
VQL queries are involved in three activities: selecting the subset concepts such as entities and attributes from a schema, grouping entities with link relationships, and specifying query conditions. Corresponding to these activities, three graphical constructs are designed: **entity box**, **key link**, and **condition link**. Further, each type of these constructs consists of a variety of visual elements. Figure 2.2 illustrates the hierarchy of visual types in VQL. In the following sections, the syntax for these visual components will be discussed.
2.4 Entity Box

An entity box is a building construct used to define the query frame, i.e., elements of the schema that are relevant to the given query. It is used to construct two types of query operations. First, users must select the entities used in the query and then choose the associated attributes for each entity that will be output or used as an operand in the query. Figure 2.3 illustrates the syntax of the entity box.

![Image of Entity Box Syntax](image)

Figure 2.3: The Syntax for Entity Box

- **Entity**

An entity is represented by an entity box with the entity name placed at the top of the box. The entity name is selected directly from the schema entity list. It
stands for the set of all the instances of its kind in the database. All instances are distinguished by their object ID’s.

An entity may be of two types, which are represented by the border of an entity box. An entity is called a **grouped-by** entity if its entity box border is highlighted. Otherwise, it is called a **grouping** entity. Grouping entities and grouped-by entities are used to express the **groupby** operation defined by the key link construct that will be discussed later.

All instances of grouping entities in a query diagram are joined with Cartesian product. If a query uses an entity in several ways, users need only build different entity boxes with the same entity name.

- **Attribute**

  Attributes stand for properties associated with an entity. When an entity box is built, its associated attributes can be selected from the attribute list for that entity in the schema.

  An attribute can be either scalar or aggregate. It is referred to by an attribute name. If an attribute is an aggregate and each member of the aggregate needs to be considered individually in the query, a suffix “[i]” must be placed behind the attribute name. For an attribute of nested aggregates, its attribute name should be followed by additional square brackets to their terminal depth. In addition, an integer may be placed inside the square brackets as an index to specify the individual member of an aggregate. The index begins with zero.
For example, "assignments[1]" in entity "classOffering" represents the second element value of the aggregate attribute "assignments". One common attribute for every entity is an object ID. It is represented by the special symbol "#". All instances in a database are uniquely determined by the object ID.

In VQL, an attribute may be used to specify the output data in a query or be used as an operand to participate in a predicate specification (in a key link or condition link). These two types of attributes have different visual representations. An output attribute is represented by listing its attribute name at the bottom of the associated entity box, while an operand attribute is represented by an attribute label which is connected with the associated entity box by a line. If the operand is the object ID, the attribute label is omitted.

An entity box can be built on the top level or constructed in the nested level within a for condition link. However, an output attribute can only be specified in an entity box on the top level. Also, among all entity boxes in a query diagram, there must be at least one with an output attribute.

2.5 Key Link

Just as the relational model relies on primary/foreign keys to explicitly define the relationships between instances of entities, in an EXPRESS model the relationships are maintained by pointers, i.e., the attributes of entity data type. Depending on
whether it is a scalar or aggregate type, each pointer in an instance of an entity is associated with one or an aggregate of instances of another entity. For example, each classOffering has a group of assignments that are referred to by the attribute “assignments” in the entity “classOffering”, and each enrollment has a student that is referred to by the attribute “theStudent” in the entity “enrollment”. The key link construct is designed to specify the groupby operation on the entities that are associated by pointers. Figure 2.4 illustrates the key link syntax.

A key link connects two entity boxes by a solid line. It must have two operands: one is a pointer, which is represented as a label placed near its associated entity box on the line; the other is an object ID, which is directly represented by the other entity
box. These two operands must have the same entity data type. In VQL, each entity may be linked with multiple entities using different pointers. For any set of fully key linked entities, users are allowed to define one but only one entity as a grouping entity. The instances for all other entities (grouped-by entities) are grouped by the object ID for each instance of the grouping entity. In this way, objects involved in a groupby operation form a larger complex object with a hierarchical structure, where each attribute value in the grouping entity still keeps its original data type, but all attribute values in the grouped-by entities may be aggregated. For example, if the entity "assignment" is grouped by the entity "classOffering" through its pointer "assignments", then we construct a new object whose instances are uniquely determined by the object ID of "classOffering". It includes all attributes in both of the entities. As the pointer is the aggregate type, all attributes in the grouped-by entity "assignment" have been aggregated.

One important rule for key link construct is that any graph connected by key links cannot be cyclic; i.e., entities involved in a groupby operation form a tree with the grouping entity as the root.

2.6 Condition Link

As discussed above, entity boxes specify a query frame and key links define the groupby structure among the chosen entities. A query diagram comprising these graphical components will return all the corresponding data in the database according
to the specified data structure. However, in most cases users only need a subset of the data according to certain select criteria. In this section, a set of condition link constructs is introduced to limit a query’s result through user-defined conditions.

The conditions can be generally classified as atomic and compound conditions. Atomic conditions include comparison condition, null condition, empty condition, isa condition, in condition, contains condition, match condition, and for condition. They operate on one or two operands which may be attributes of entities or constants. Compound conditions include and condition, or condition, and negative condition. They are employed to execute logical operations on results of other conditions.

In VQL, all these kinds of conditions are represented by their corresponding condition links. A condition link is constructed with two types of visual elements: operator and operand.

- **Operators** stand for types of conditions. Except for the negative condition, whose operator is represented by a circle, a operator is represented by a polygon with the corresponding condition symbol in it. The input to a condition consists of operands which are connected to sides of the polygon, while the output of a condition operation is a boolean value represented by the bottom tip of the polygon.

- **Operands** may be attributes, outputs of conditions, or constants. The visual representations of the first two types of operands have been described. Constants are represented as labels; they are only data that must be manually typed
in by users and only used in condition links. Constants can be either scalar or aggregate. The notations for all different types of constants conform to the part 21 standard. For instance, a string should be surrounded by single quotes. An aggregate should be expressed as a list of elements separated by commas and surrounded by parentheses.

2.6.1 Comparison Condition Link

Comparison conditions are used for comparing the values of two operands. There are six types of comparison condition links (Figure 2.5). The compared operands

Figure 2.5: The Syntax for Comparison Condition Link
may be attributes or constants, and they must have the same data type with the exception of the case in which the data types can be converted meaningfully. For example, an integer can be compared with a real by converting the integer into a real.

Only equality and inequality can apply to aggregate and object IDs. If both aggregates have the same sizes and each element of them in sequence are equal, then they are equal; otherwise, they are not.

### 2.6.2 Null Condition Link

A null condition link (Figure 2.6) is represented by an operator (labeled "null") connected with an operand. A null condition checks if an operand is null (unknown) or not. It returns true if its operand is null. The operand must be an attribute of an entity, which can be either scalar or aggregate. In EXPRESS, if a value of an attribute or an element of an aggregate cannot be determined, an unknown value, which stands for absence of data for the value, is assigned. A null value is represented by a special symbol "$". Figure 2.7 shows an example: "list the titles for the classes in which the assignment information is unknown". This query uses a null condition to
select all instances of "classOffering" with the values of their attribute "assignments" being null.

A null value is different from an empty string, a zero integer, or an empty aggregate. The latter have specific values. Also if one element of an aggregate is null, it does not mean the aggregate itself is null.

2.6.3 Empty Condition Link

An empty condition link (Figure 2.8) is represented by an operator (labeled "empty") connected with an operand. The operand must be an attribute of an entity with an aggregate data type. An empty condition is used to test whether an aggregate attribute is empty (no elements exist) or not. It returns true if its operand is empty. An empty aggregate value is represented as "()". It is not equivalent to a null value of an aggregate ("$") . Thus, users should pay attention to the different
usages between a null condition and an empty condition. For instance, in contrast with the example of a null condition in Figure 2.7, the query in Figure 2.9 uses an empty condition to select all instances of "classOffering" whose "assignments" value is empty and returns all titles of the classes without assignments.

### 2.6.4 ISA Condition

An isa condition link (Figure 2.10) is represented by an operator (labeled "isa") connected with two operands, of which the left-hand side operand must be an object ID and the right-hand side operand must be one of the entity names that exist in the
EXPRESS schema. Every instance in a STEP file belongs to a certain class or entity. An isa condition is used to test this kind of instance-entity belongingness. If the instance is instantiated from the entity, it returns true. Otherwise, it returns false.

As the EXPRESS model supports object inheritance relationships, an instance of a child entity also belongs to its inherited entities. For example, an instance of "exam" belongs to both "exam" and its parent "gradeWork".

2.6.5 In Condition Link

An in condition link (Figure 2.11) is represented by an operator (labeled "<in" or "in>") connected with two operands, of which one must be an attribute of an entity and the other may be either an attribute or a constant. An in condition is used to check the membership of an aggregate. The right-hand side operand of an operator "<in" or the left-hand side operand of an operator "in>" is the unit operand, while the other side operand is the aggregate operand. The data type of the unit operand must be comparable with that of the members of the aggregate operand. If the unit
operand is equal to one of the members of the aggregate operand, then this condition returns true. If either operand is null, it returns false. For example, 1 is in (1,2,3), and (1,2,3) is in ((1,2,3),(4,5,6)). But $ is not in (1,2,3).

2.6.6 Contains Condition Link

A contains condition link (Figure 2.12) is represented by an operator (labeled "contain" or "contain") connected with two operands, of which one must be an attribute of an entity and the other may be either an attribute or a constant. A contains condition is used to check the subset relationship between two operands. Both of the operands must be of aggregate data, and the data types of the members of both operands should be comparable. If each member of the right-hand side operand
of an operator "<contain" is present in its left-hand side operand or each member of the left-hand side operand of an operator "contain>" is present in its right-hand side operand, then this condition returns true. If either operand is null or one of the members of either operand is null, it returns false.

2.6.7 Match Condition Link

A match condition link (Figure 2.13) is represented by an operator (labeled "<match" or "match>" ) connected with two operands, of which one is an attribute of string type and the other is a constant that defines a regular expression. The attribute is placed in the right-hand side of an operator "<match" or the left-hand side of an operator "match>"; the constant is placed in the other side. A match condition is used to test whether or not a string value satisfies a regular expression.
2.6.8 For Condition Link

A for condition is the most powerful condition operation to test members of an aggregate. The contains condition and in condition mentioned above can be expressed in for conditions. This condition splits an aggregate attribute and evaluates each member of the aggregate against a certain test condition. There are two kinds of for conditions: all condition and any condition. For an all condition, it returns true only if every member satisfies the test condition. For an any condition, it returns true if any one of the members satisfies the test conditions.

A for condition link (Figure 2.14) is represented by a for condition operator connected with an attribute operand. The operand may be an aggregate or nested aggregate. The operator is represented by a polygon which includes an operator label ("all" or "any"), a split operator, and a test condition area. The split operator is...
represented by a shaded rectangle. It outputs all elements of the aggregate operand that will participate in the test condition. The test condition area is used to specify the test condition. In fact, the test condition area works like a subquery in SQL. Entity boxes, key links, and condition links (including nested for condition) that are involved in the test condition can be built in this nested query level. However, no output attributes are allowed to be defined in entity boxes in this level. Figure 2.15 illustrates the usage of an any condition link. This query lists all the titles of the classes with one or more difficult assignments. The any condition is used to select the instances of “classOffering” where at least one member value of its attribute “description” is equal to “difficult.”
2.6.9 And/or Condition Link

An and/or condition link (Figure 2.16) is represented by an operator (labeled “and” or “or”) connected with one or more operands. Each operand may be the result of an atomic condition or another and/or condition. An and/or condition is employed to make a conjunctive/disconjunctive operation on results of other conditions. In the case of only one operand, the result of the and/or condition is equal to the operand itself.

If an and/or condition has more than one operand, it will first evaluate the condition operand whose graphical operator (polygon) lies in the leftmost position, then the second leftmost, etc. When two operands’ positions tie in the horizontal
direction, their evaluation orders are determined arbitrarily. If an and/or condition becomes false when it evaluates one of its operands, then the operands to the right of this operand will not be evaluated.

All the conditions that are not linked by and/or conditions are ANDed by default, and and/or conditions can be nested by inputting results of certain and/or conditions into other and/or conditions. In this way, condition links in a VQL query form a tree structure with each non-leaf as an and/or condition link and each leaf node as an atomic condition link. For example, Figure 2.17 shows the following operations: \(((A \text{ and } (B \text{ or } C)) \text{ or } E) \text{ and } F\).

Another rule for an and/or condition link is that any and/or condition links must take the operands on the same level. For example, an and/or condition link that is built in the test condition area of a for condition link can only use the results of conditions that are specified in that area as operands.
2.6.10 Negative Condition Link

A negative condition link is represented by placing a circle on the bottom tip of a polygon which stands for an output of a condition on which the negative condition operates (Figure 2.18). Any conditions can be negated. If an operand value is true,

this condition returns false. Otherwise, it returns true.
2.7 Order of Evaluation

According to users' requests, a variety of VQL queries can be formulated by the constructs introduced above. When these graphical components are combined to represent a query, the query operations denoted by them will be evaluated in different order during the query process.

**groupby** operations are executed first if key links exist. Objects involved in key links construct a grouped object. An instance of the grouped object maps directly to an instance of the grouping entity. The various attributes of the grouped-by entities are aggregated and inserted into an appropriate field of the grouped object.

**Cartesian product** operations are executed next if two or more grouping entities exist. The grouped objects built by groupby operations construct a larger object by joining all of their instances with Cartesian product. The desired query result is a subset of the data stored in this object.

**Select** operations are then executed if condition links are specified in the VQL query. In this stage, only the required instances that conform to the select criteria are chosen from the grouped object. As described earlier, condition links in a VQL query form a tree structure with each non-leaf node as an **and/or** condition link and each leaf node as an atomic condition link. Conditions that are operands of an **and/or** condition are evaluated in the left-to-right order. Then the **and/or** condition is executed. This process continues until the root node of the condition tree is reached.
Project operations are the final step. This step prints all the data specified in entity boxes.

2.8 Query Examples

All of the graphical constructs in VQL have been discussed above separately. In this section, a set of query examples based on schema shown in Figure 2.1 is formulated to further illustrate VQL syntax.

1. List the name and ID of all the students taking EE690 (Figure 2.19).

   Figure 2.19: Query 1 in VQL
2. List the name and ID of all the students taking EE690 or EE590 (Figure 2.20).

- Method 1: query outputs are uniquely determined by student IDs
- Method 2: query outputs may be duplicate

Figure 2.20: Query 2 in VQL
3. List the name and ID of all the students who take both EE590 and EE690 (Figure 2.21).

4. List the name and ID of all the students who take neither EE590 nor EE690 and all courses that those students take (Figure 2.22).
5. List the name of all the exams given in EE590 as an aggregate (Figure 2.23).

Figure 2.23: Query 5 in VQL

6. List the name of all the exams given in EE590 separately (Figure 2.24).

Figure 2.24: Query 6 in VQL
7. List the name and ID of the students who took EE590 or EE690 and whose final grades were all above 80 (Figure 2.25).

![Figure 2.25: Query 7 in VQL](image)

8. For each student who took EE590 and whose first exam score of that course was above 90, list his/her name, ID, and the first exam score (Figure 2.26).
a) Error

b) Correct

Figure 2.26: Query 8 in VQL
Chapter 3

Query Translation

3.1 Introduction

Having explored the VQL syntax and how to formulate a syntactically legal VQL query, it is time to define the semantics of VQL queries in terms of a translation to EQL.

The translation of a VQL query to EQL involves mapping sets of entity boxes, key links, and condition links to a textual string of the form “SELECT ... FROM ... WHERE ...”. As we discussed in the previous chapter, each type of VQL graphical construct and its internal visual elements are designed to represent one substructure or component of a query, and they are highly interrelated. A VQL query consisting of these components expresses the same query structure as its textual query counterpart. We can easily recognize the following mapping relationships between the VQL graphical components and the EQL syntactic textual constructs.
• The output items of the SELECT clause in EQL correspond to the output attributes defined in VQL entity boxes.

• The source entities of the FROM clause in EQL correspond to the entities represented by VQL grouping entity boxes.

• The conditions of the WHERE clause in EQL correspond to the conditions represented by VQL condition links.

The fundamental textual elements in EQL include constants, entities, attributes, attribute expressions, and conditional operators. They are directly or indirectly represented in VQL queries. Before construction of an EQL query string, some mapping work must be prepared.

First, since EQL uses variables to represent source entities which are represented by VQL grouping entity boxes, a unique variable name is assigned to each of these source entities. All other entities and attributes are expressed by path expressions that begin with these variables.

Next, since all atomic condition links are connected with and/or condition links and these logic condition links may be nested to form a more complicated conditional logic structure, it is necessary to build a logic tree with atomic conditions as leaves so that the WHERE clause in EQL can be simply constructed by traversing the logic tree.

Having finished all this preparatory work, a VQL query can be easily translated to an EQL query string according to the mapping rules mentioned above. In this
\[ Q = \{ \text{ebs, kls, cls, lr} \} \]
where:
\[ \text{EB} = \{ \text{eName, var, outputs, type} \} \]
where: \( \text{OUTPUT} = \{ \text{aName} \} \)
\[ \text{KL} = \{ \text{att1, att2} \} \]
where: \( \text{ATT} = \{ \text{eb, aName} \} \)
\[ \text{CL} = \text{bcl|ucl|fcl|lcl} \]
where:
\[ \text{BCL} = \{ \text{opd1, opd2, opr, neg, coordX} \} \]
\[ \text{UCL} = \{ \text{opd, opr, neg, coordX} \} \]
\[ \text{FCL} = \{ \text{opd, opr, neg, coordX, subQ, var} \} \]
\[ \text{LCL} = \{ \text{cls, opr, neg, coordX} \} \]
where: \( \text{OPD = att|const|var (in fcl)} \)

Figure 3.1: A Data Structure for VQL Translation

In this chapter, a data structure used for VQL translation will be discussed first, then the mapping algorithm will be described, and, finally, some translation examples will be given.

### 3.2 Data Structure for Translation

In order to translate VQL to EQL, a textual data structure shown in Figure 3.1 is needed to hold all information represented by the graphical constructs in a VQL query as well as to keep intermediate translation results. In Figure 3.1, a VQL query \( Q \) consists of entity boxes (\( \text{ebs} \)), key links (\( \text{kls} \)), condition links (\( \text{cls} \)), and a root of a conditional logic tree (\( \text{lr} \)), which is built on the condition links during translation. It should be noted that the query structure \( Q \) only keeps the entity boxes, key links, and condition links that are specified in the top level. Those constructs specified within
for condition links are stored in their corresponding for condition links as sublevel query structures.

Each entity box includes an entity name (eName), a variable expression (var), a set of output attributes (outputs), and an entity type (type). The variable expression is a binding variable of a grouping entity or a path expression of a grouped-by entity that is generated during translation. Each output attribute in outputs is represented by its attribute name (aName), which may include a suffix. The value of an entity type can be either “grouping” or “grouped-by”.

Each key link operates on two attribute operands (att1 and att2), which are represented by their associated entity box (eb) and attribute name.

Condition links are classified into four categories: binary condition links (bcl), unary condition links (ucl), for condition links (fcl), and logic condition links (lcl). Binary condition links include comparison, in, contains, match, and isa condition links; unary condition links include null and empty condition links; logic condition links include and and or condition links; and for condition links include any and all condition links. All condition link structures include an operator name (opr), a negative flag (neg)—which indicates a positive or negative condition—and an x coordinate (coordX) of its operator symbol, i.e., polygon, which is used to determine the evaluating order when it participates in a logic operation. In a for condition, a subQ structure is used to hold all test condition data; it bears the same data structure as the top level query structure Q. Also, a variable expression (var) is
added in a \textbf{for} condition link to store the binding variable of its aggregate operand, which is used to represent the members of the aggregate value and acts as an operand in its sublevel condition link. Except for a logic condition link, which takes a set of condition links as operands (\textit{cls}), all condition links operate on operands of attributes, constants, or variables that are assigned to \textbf{for} condition links. A binary condition link has two operands (\textit{opd1} and \textit{opd2}) while a unary or \textbf{for} condition link has only one operand (\textit{opd}).

Since this data structure is used to record all information of a VQL query diagram, the translation of VQL to EQL is equivalent to translation of data stored in this structure to an EQL string. In the following sections, the translation algorithms are described.

### 3.3 Variable Binding

In EQL, every grouping entity (source entity) is given a unique variable name, which is used to identify instances of the entity during a query execution. Grouping entities may be specified in more than one level if \textbf{for} condition links exist. All grouping entities specified in the top level are assigned variables in the FROM clause, and all grouping entities specified in a \textbf{for} condition link are assigned variables within the \textbf{for} condition expression. These variables have unique variable names. In addition, in a \textbf{for} condition link, a variable name should be assigned to the aggregate operand
variableBinding \((Q)\)

input: a VQL query \(Q\)

output: every grouping entity and for condition link have been assigned a variable

1. for each grouping entity box \(eb\) in \(Q.ebs\) do
2. begin
3. assign a unique variable name to \(eb.var\)
4. end
5. for each “for” condition link \(cl\) in \(Q.cls\) do
6. begin
7. assign a unique a variable name to \(cl.var\)
8. call variableBinding \((cl.subQ)\)
9. end

Figure 3.2: Variable Binding

upon which the condition operates. As described in the previous section, this variable is stored in the for condition link structure.

Figure 3.2 shows the variable binding algorithm. The function variableBinding first assigns a unique variable name to each grouping entity specified in the top level. Then if for condition links exist in that level, each of them will be assigned a unique variable name, and the function will be recursively called by passing the sublevel query structure \((subQ)\) specified within that for condition.

3.4 Grouped-by Expression Mapping

In EQL, a grouping entity is referred to by its corresponding binding variable and a grouped-by entity, which is directly or indirectly associated with its grouping
entity by key links, is represented by a path expression that starts from its grouping entity variable and is followed by the attributes (pointers) specified in the key links with "." or "->" as delimiters.

In a path expression, "." delimiter is placed between attributes and "->" delimiter is placed between a variable and an attribute. There are two kinds of pointers in key links. If the identification of one entity is based on the other linked entity and the key link pointer is an attribute of the latter, then that pointer is called a normal pointer. A normal pointer is represented by its attribute name in a path expression. In contrast, if the identification of one entity is based on the other linked entity but the key link pointer is an attribute of the former, then that pointer is called an inverse pointer. An inverse pointer is represented by its attribute name followed by the inverse sign ("<<") and its entity name. For example, if entity A has been assigned a variable x and it is linked with entity B with A's pointer a, and entity B is linked with entity C with C's pointer c, then B's expression is translated as "x->a" and C's expression is translated as "x->a.c<<C". In this case, as x is a variable, it is followed by "->"; as a is a normal pointer, it is represented by itself; as c is an inverse pointer, it is represented as "c<<C".

Since no cyclic key links are allowed to be specified in VQL, a VQL query diagram contains a set of grouping trees. Each tree consists of one grouping entity which acts as a root and a set of grouped-by entities that depend upon the root. In a grouping tree, the grouping entity is allowed to be connected with multiple
grouped-by entities by key links, and each grouped-by entity may also associate with other grouped-by entities by key links. The identification of an grouped-by entity in the lower level of a tree is dependent upon the entity node in the higher level until reaching the root entity. Therefore, it is easy to determine the path expressions of grouped-by entities in the top-down order.

Figure 3.3 shows the algorithm for mapping key links to grouped-by entity expression. The function `grouped_byEntityExpr` repeatedly calls the subfunction `translateGroupingTree` to determine grouped-by entity expressions in each grouping tree. It also recursively calls itself to assign path expressions to the grouped-by entities that are specified within the `for` condition links.

Given the root entity of a grouping tree and the set of key links involved in the tree, the function `translateGroupingTree`, following the translating rules described above, recursively assigns path expressions to grouped-by entities from top-to-bottom until meeting a leaf node.

### 3.5 Conditional Logic Tree

In this section, conditional logic trees are constructed for each set of logic links specified in the top level or in the nested levels (within `for` condition links). A conditional logic tree is used to map condition links in VQL to condition expressions of `WHERE` clause in EQL, which will be described later.
grouped_byEntityExpr ($Q$)
input: a VQL query $Q$
output: every grouped-by entity has been given a path expression

1. for each grouped-by entity box $eb$ in $Q.ebs$ do
2. begin
3. translateGroupingTree ($eb$, $Q.kls$)
4. end
5. for each "for" condition link $cl$ in $Q.cls$ do
6. begin
7. grouped_byEntityExpr ($cl.subQ$)
8. end

translateGroupingTree ($root$, $kls$)
input: 1. root of a grouping tree
2. all possible key links in the tree
output: every grouped-by entity in the tree has been given a path expression

1. for each key link $kl$ in $kls$ and $root$ is connected by $kl$ do
2. begin
3. if $root == kl.att2.eb$
4. then swap ($kl.att1$, $kl.att2$)
5. if $kl.att2.eb.var$ is processed
6. then stop
7. if $root$ is a grouping entity
8. then
9. delimiter = ‘->’
10. else
11. delimiter = ‘.’
12. if $kl.att1.aName$ = ‘#’
13. then
14. $kl.att2.eb.var = root.var + delimiter + kl.att2.aName$
15. + ‘<<’ + $kl.att2.eb.eName$
16. else
17. $kl.att2.eb.var = root.var + delimiter + kl.att1.aName$
18. call translateGroupingTree ($kl.att2.eb$, $kls$)
19. end

Figure 3.3: Grouped-by Entity Expression Mapping
If only one atomic condition link exists, a conditional logic tree takes that condition link as one and only one node (root); otherwise, it has a hierarchical structure with each non-leaf node as a logic condition link and each leaf node as an atomic condition link. Since logic condition links operate on other condition links, if the root of a conditional logic tree is located, then the tree is determined.

When the default and condition exists in a VQL query, a new and condition link is added into the set of condition links to explicitly represent that default condition. The created and condition link is the root of the tree. For example, if there exist four condition links in a VQL query, such as A, B, C are atomic condition links and D is an or condition link which links A and B, then according to the VQL condition link rule, C and D should be ANDed by default. So an and condition link E that operates on D and C must be explicitly constructed, and it acts as the root of the logic tree. This tree expresses the following logic relationship: (A or B) and C. If no default and condition exists in a VQL query—i.e., there is only one condition link whose output does not participate in further logic operation—then this condition link is the root of the tree.

In Figure 3.4, the function condLogicTree is designed to build conditional logic trees. First, it uses a variable s to hold all condition links specified in the top level. Next, it iteratively checks each condition link in s. If the link is a logic condition link, it calls the function remove to take away that logic condition link as well as all condition links that belong to the subtree using that logic condition link as the root.
condLogicTree(Q)
input: a VQL query Q
output: a conditional logic tree is built for each set of condition links specified in different levels.

1. \( s = Q\.cls \)
2. for each condition link \( cl \) in \( s \) do
3. begin
4. if \( cl \) is a logic condition link
5. then remove\((cl, s)\)
6. end
7. if only one condition link in \( s \)
8. then
9. \( Q\.lr = s \)
10. else
11. new.\( cls = s \)
12. new.\( opr = 'and' \)
13. add new to \( Q\.cls \)
14. \( Q\.lr = new \)
15. for each “for” condition link \( cl \) in \( Q\.cls \) do
16. begin
17. condLogicTree\((cl\.subQ)\)
18. end

remove\((lcl, s)\)
input: root of a subtree \( lcl \) and set of condition links \( s \)
output: all nodes in the subtree are removed from \( s \)
1. remove \( lcl \) from \( s \)
2. for each condition link \( cl \) in \( lcl\.cls \) do
3. begin
4. if \( cl \) is a logic condition link
5. then
6. remove\((cl, s)\)
7. else
8. remove \( cl \) from \( s \)
9. end

Figure 3.4: Build Conditional Logic Tree
translateQuery (Q)
Input: VQL query Q
Output: EQL string text

1. variableBinding(Q)
2. grouped_byEntityExpr(Q)
3. condLogicTree(Q)
4. Let S be the set of entities with output attributes in Q.ebs
5. Let F be the set of grouping entities in Q.ebs
6. Let W be the logic root of conditional logic tree, i.e. Q.Ir
7. Let s be EQL string
8. if S is empty
9. then
10. return fail
11. else
12. s = 'SELECT ' + selectExpr(S)
13. s += ' FROM ' + variableDecl(F)
14. if W is not empty
15. then s += ' WHERE ' + condExpr(W)

Figure 3.5: Mapping VQL to EQL

Third, if only one condition link remains in s, this condition link is the root of the final conditional logic tree; otherwise, an and condition link will be created as the root. Finally, the function condLogicTree will be recursively called to construct a conditional logic tree for the set of condition links specified within each for condition link.

3.6 Mapping VQL to EQL

Figure 3.5 shows an algorithm for translation of VQL to EQL. Given a VQL query Q, the EQL string is obtained by calling the function translateQuery. After
the preparatory translation work of binding variables to grouping entities, determining path expressions for grouped-by entities and building conditional logic tree for condition links, it constructs the query string in the following sequence: first, if no output attributes are specified in Q, an error message will be returned; otherwise, a SELECT clause will be generated from S—the set of grouping entities with output attributes—by calling the function `selectExpr`. Next, a FROM clause is generated from F—the set of grouping entities specified in the top level—by calling function `variableDecl`. Finally, if condition links exist in Q, a WHERE clause is generated from W—the root of conditional logic tree in the top level—by calling function `condExpr`.

In the following sections, the algorithms to generate the fragments of the EQL expression corresponding to S, F, and W, are described in detail.

### 3.6.1 Select Expression

In EQL, a select expression is used to determine which kind of information in the collected data must be retrieved. It consists of a list of attribute expressions separated by commas. These attributes correspond to the output attributes specified in VQL entity boxes. In Figure 3.6, the function `selectExpr` is used to generate the select expression by repeatedly appending the expression of every output attribute specified in each entity box and using "," as a delimiter.

The function `attrExpr` is employed to construct an attribute expression. Given an attribute name and its associated entity (the entity expression is known), if the
selectExpr(S)
input: a set of entities with output attributes
output: a select expression string

1. for each entity box eb In S do
2. begin
3. for each output attribute a in eb do
4. begin
5. s += attrExpr(eb, a) + ','
6. end
7. end
8. remove last ',' from s
9. return s

attrExpr(eb, aName)
input: an attribute name and its associated entity box
output: an attribute expression string

1. if aName == "#" then
2. return eb.var
3. else
4. if eb is a grouping entity box then
5. return eb.var + '->' + aName
6. else
7. return eb.var + '.' + aName

Figure 3.6: Construction of a Select Expression
variableDecl(F)
input: a set of grouping entity boxes
output: a variable declaration string

1. for each entity box eb in F do
2. begin
3.  $s += eb.var + ' in ' + eb.eName + ',$$
4. end
5. remove last ',', from $s$
6. return $s$

Figure 3.7: Variable Declaration

attribute is object ID, then its expression is the same as its associated entity expression; otherwise, the attribute expression is directly constructed by combining the entity expression and the attribute name with an appropriate delimiter ("." or ">").

3.6.2 Variable Declaration

As introduced earlier, in EQL, each of the source entities (corresponding to grouping entities in VQL) must be assigned an identifying variable in the FROM clause or in the WHERE clause if it is specified within a for condition. All other objects (entities or attributes) must be referred to by a path expression from these source entities. The text format for a variable declaration is “var in eName”. If more than one entity must be declared, the declaration list is separated by commas.

Figure 3.7 shows the algorithm to generate an EQL variable declaration string.
3.6.3 Condition Expression

In EQL, condition expressions in WHERE clause are used as select criteria. If more than one atomic condition exists, they are connected with logic operators (and and or). Logic conditions can also be nested by using parentheses to represent the operational priority. Starting from the root condition link of the conditional logic tree in a VQL query Q, an EQL condition expression can be constructed using the algorithm shown in Figures 3.8 and 3.9.

Given the root of a conditional logic tree W, the function condExpr works as follows:

1. If the given condition link is operated by a negative condition, then a not operator is prefixed to its condition expression.

2. If the given condition links is a logic condition link, then all condition link in its child nodes (W.cls), which act as its operands, need to be sorted in the ascending order based on their x coordinate values so that each child condition is placed from left-to-right according to their positions in the VQL query. Next, we connect each of these child condition expressions, which should be parenthesized, with the given logic condition's operator (and or or). Each of the child condition expressions is generated by recursively calling the function condExpr and passing the corresponding condition link as argument. In this way, the hierarchical logic structure in a VQL logic tree and the placing order of operands in a logic condition are mapped to EQL format.
condExpr (W)
input: the root of a conditional logic tree
output: an EQL condition expression string

1. if W.neg == true
2. then
3. s += 'NOT(' 
4. if W is a logic condition link
5. then
6. sort W.cls in ascending order based on coordX values
7. for each condition link cl in W.cls do
8. begin
9. s += '(' + condExpr(cl) + ')' + W.opr
10. end
11. remove last W.opr from s
12. if W is a binary condition link
13. then
14. if W is a "comparison" or "isa" condition link
15. then
16. s += opdExpr(W.opd1) + W.opr + opdExpr(W.opd2)
17. if W is a "in", "contains" or "match" condition link
18. then
19. if W.opr == 'in>'
20. then
21. s += opdExpr(W.opd1) + 'in' + opdExpr(W.opd2)
22. if W.opr == '<in'
23. then
24. s += opdExpr(W.opd2) + 'in' + opdExpr(W.opd1)
25. if W.opr == 'contains>'
26. then
27. s += opdExpr(W.opd1) + 'contains' + opdExpr(W.opd2)
28. if W.opr == '<contains'
29. then
30. s += opdExpr(W.opd2) + 'contains' + opdExpr(W.opd1)
31. if W.opr == 'match>'
32. then
33. s += opdExpr(opd1) + 'match' + opdExpr(W.opd2)
34. if W.opr == '<match'
35. then
36. s += opdExpr(opd2) + 'match' + opdExpr(W.opd1)

Figure 3.8: Condition Expression (to be continued)
37. if $W$ is a unary condition link \\
38.    then \\
39.    $s \leftarrow opdExpr(W.opd) + ' is ' + W.opr$ \\
40. if $W$ is a "for" condition link \\
41.    then \\
42.        Let $f$ be the set of grouping entities in $W.subQ$ \\
43.        Let $w$ be root of conditional logic tree in $W.subQ$, i.e, $W.subQ.lr$ \\
44.        $s \leftarrow 'for' + opdExpr(W.var) + ' in ' + opdExpr(W.opd)$ \\
45.        if $f$ is not empty \\
46.            then \\
47.                $s \leftarrow ',' + variableDecl(f)$ \\
48.            $s \leftarrow ',' + W.opr + '(' + condExpr(w) + ')$ \\
49. if $W.neg == true$ \\
50.    then \\
51.    $s \leftarrow ')'$ \\
52. return $s$

\[\text{opdExpr}(opd)\]
input: an operand  
output: an operand expression

1. if $opd$ is constant or variable in a "for" condition link \\
2.    then \\
3.    return $opd$ \\
4. else \\
5.    return $\text{attrExpr}(opd.eb,opd.aName)$

Figure 3.9: Condition Expression (continued)
3. If the given condition link is a binary condition link, then the condition expression is generated by connecting its two operand expressions with its operator. The placing order of the operands in some condition expressions is dependent on the direction sign '<' or '>' of their operators.

4. If the given condition link is a unary condition link (null or empty), then its target translating format is “opdExpr is opr”.

5. If the given condition link is a for (any/all) condition link, its target translating format is “for varDeclList any/all (conditionExpr)”. The variable declaration list is constructed by binding the variable stored in the for condition link structure to its aggregate operand and, if there exist sublevel grouping entities, calling the function variableDecl (see Section 3.6.2) by passing these entities. The nested condition expression is constructed by calling condExpr, which accepts the root of the logic tree in the for condition link as its argument.

During translation of condition links, the function opdExpr is used to build an operand expression. Its argument may be an operand of a constant, an attribute, or a binding variable stored in a for condition link structure. If it is a constant or a variable, its expression is represented by itself; otherwise, its expression is constructed by calling the function attrExpr (see Section 3.6.1).
3.7 Translation Example

Based on the translation algorithm mentioned above, some VQL queries given in Chapter 2 can be translated as follows:

Query 1 (see Figure 2.19a):

SELECT v1->id, v1->name
FROM v1 in student
WHERE 'EE690' in v1->theStudent<~enrollment.enrollments[]<<classOffering.id

Query 2 (see Figure 2.20b):

SELECT v3->id, v3->name
FROM v1 in classOffering, v2 in enrollment, v3 in student
WHERE ((v1->id == 'EE590') or (v1->id == 'EE690')) and (v2 in v1->enrollments[]) and (v3 == v2->student)

Query 3 (see Figure 2.22):

SELECT v1->id, v1->name
FROM v1 in student
WHERE (not('EE590' in v1->theStudent<~enrollment.enrollments[]<<classOffering.id))
and (not('EE690' in v1->theStudent<~enrollment.enrollments[]<<classOffering.id))
Chapter 4

VQL Usage

4.1 Introduction

Based on the VQL syntax and mapping rules introduced in the previous chapters, a VQL prototype has been implemented in the Silicon Graphics Workstations in Stocker 292, Ohio University. This prototype should work well on other platforms if relevant developing resources (EQL and TNR) are available.

This chapter provides a high level description of the GUI as seen by end users; it constitutes an abbreviated user's manual, which describes how users interact with the GUI to carry out query activities.
4.2 Overview of the Graphical User Interface

The VQL main window consists of three pages (subwindows), namely, the visual query editor page (see Figure 4.1), EQL text page (see Figure 4.2), and output page (see Figure 4.3). They enable users to formulate graphical queries, check translated textual EQL statements, and obtain query results in a spreadsheet. Users can easily switch among these pages by pressing the page button “Query constructor”, “EQL”, or “Query Result”.

One pull-down menu for file appears at the top right of the window. It is used to specify which STEP file and ROSE file (data dictionary file) is to be used. If users choose the “Open” option in the file menu, a file dialog box will be activated to enable them to select the interested file. Users only need to select the STEP file. The corresponding ROSE file name can be found from the STEP file.
SELECT v1->theStudent.<enrollment.examGrades[], v1->id, v1->name

FROM v1 in student

WHERE (("EE690" in v1->theStudent.<enrollment.enrollments[].classOfferingId) Or
  ("EE590" in v1->theStudent.<enrollment.enrollments[].classOfferingId)) AND (FOR v2 in
  v1->theStudent.<enrollment.finalGradeArray(v2 >= 80))

Figure 4.2: EQL Page

Figure 4.3: Output Page
The visual query editor is the most important part of VQL. It consists of two main components: tool bar and work area. The tool bar is at the left of the page and consists of five action buttons; each button represents a specific query activity. The work area is a canvas widget. It is the large area to the right of the window and is the area in which a query is formulated. The scroll bars at the bottom and the right of the canvas allow it to be larger than the window.

The VQL query editor is a syntax-directed editor. The constructed queries must conform to all the syntax rules specified in Chapter 2. Any action of creating a VQL construct that leads to violation of the VQL syntax, such as an attempt to create a cyclic key link, will fail.

All VQL constructs are manipulated using the mouse except for constants, which are typed in from the keyboard. There are three buttons in the mouse. The left button (B-1) is used to activate action buttons in the tool bar or acts as a pointer to locate the position of the construct being created in the work area. The middle button (B-2) is used to move the created constructs in the work area. The right button (B-3) is used to manipulate each created construct and its internal elements by activating the corresponding pop-up action menu.

The first step to edit a VQL query is to activate an action button in the tool bar. An action button is activated by moving the mouse cursor over it and clicking B-1; it is deactivated by activating another button in the tool bar. A VQL query consists of three types of graphical objects, entity boxes (including the output attributes), key
links, and condition links. They can be created separately by activating the “Entity”, “Key Link” or “Condition Link” buttons. The other two buttons in the tool bar are used to delete a graphical object or clear the work area for constructing new queries.

The following sections describe the details of how to create and manipulate these query objects.

4.3 Entity Box

When the “Entity” button is activated, VQL first enters a roaming mode where it waits for a user to position the entity box in the work area. Then a schema entity dialog box (Figure 4.4a) appears at that place to help a user select the entity name of interest. Depending on the position, an entity box can be either created in the top level or nested within an any/all operator. The initial type for an entity box is grouping type (its border is highlighted).
When a user moves the cursor over the entity name of an entity box and clicks B-3, an entity menu (Figure 4.4c) pops up. It enables a user to specify the output attributes, change the entity type (grouping or grouped-by), or delete the entity box. Because the VQL query editor is syntax-directed, the “Add Attribute” option in the entity menu can be activated only for an entity box created in the top level. Also the “Groupby” option can be activated only for the entities involved in key links.

If the “Add Attribute” option is activated, an attribute dialog box (Figure 4.4b) for that entity is displayed, and a user can choose single or multiple attributes within it. All these chosen attributes will be listed in the output attribute area of the entity box. If the “Delete Entity” option is clicked, the entity box and all its involved key links and condition links are removed from the editor screen.

If a user wants to manipulate the output attribute, he/she needs to move the cursor over the attribute and click B-3. Then an attribute menu (Figure 4.4d) is displayed. By choosing an appropriate menu option, a user can append/delete a suffix of “[i]” or “[index]” to the attribute or delete the specific attribute. This attribute menu is also applied to the pointers in key links, as well.

A user can also move an entity box by pressing B-2 on the entity name and moving the mouse.
4.4 Key Link

When the "Key Link" button is activated, VQL waits for the user to select the two entities to which a new key link connects. Then a dialog box for the key link (Figure 4.5) is activated, which consists of two attribute list boxes for the chosen entities. After the pointer of the key link is selected, a red line will connect the two involved entity boxes by placing the pointer label near its associated entity box on the line. Since among all the entities involved in a groupby operation, only one entity can be a grouping entity, whenever an entity box is modified from a grouped-by type to a grouping type, the original grouping entity box will be automatically switched to a grouped-by type.
When a user clicks B-3 on the pointer label, an attribute menu (Figure 4.4d) is activated. It allows the user to append/delete a suffix to the pointer or delete the existing key link.

4.5 Condition Link

When the "Condition Link" button is activated, VQL first enters a roaming mode where it waits for a user to locate the position where a new operator will be placed. Then an operator dialog box (Figure 4.6a) is activated to enable the user to select the operator name. After the condition operator is constructed, the user must complete the condition link by adding its operands. If the condition link is an atomic condition type, it must operate on attributes or constants. An attribute operand is
created as follows: a user first moves the cursor over the entity name of an entity box and clicks B-1. Then its attribute dialog box is activated. When the user chooses the attribute name, a link line is drawn to connect the entity and the operator by placing the attribute label near its associated entity box on the line. To construct a constant, a user first clicks B-1 on the canvas, then a prompt dialog box (Figure 4.6f) is displayed to let him/her input the constant value and, finally, a link line will connect the constant to the operator. If the condition link is a logic condition type, then its operands are created simply by clicking B-1 on the operators of the conditions on which it operates.

In a for condition link, all the VQL constructs involved in its test condition must be created within its operator. The size of a for condition operator cannot be predefined. It will dynamically change its size to make sure all the inside graphical objects are visible.

An element (operator or operand) in a condition link can be manipulated through the corresponding pop-up menus (Figure 4.7). For instance, if a user clicks
B-3 on an operator, an operator menu is activated. It allows a user to negate or modify the operator or delete the whole condition link (for a logic condition link, it also enables a user to add another operand). If the user chooses the “Negate” option, then a positive operator will change to negative and a negative operator will reverse to positive. If a user chooses the “Modify” option, depending on which type (binary, unary, logic, or any/all) of operator it is, an operator dialog box of the specific type (Figure 4.6b-e) will be activated to help the user choose the replaced operator name.

Like entity boxes, operators and constants in condition links can also be moved in the work area by pressing B-2 and moving the mouse.
Chapter 5

VQL Implementation

5.1 Introduction

VQL is implemented using Incr Tcl/Tk, an extension for Tcl/Tk. Tcl is a simple scripting language for controlling and extending applications, which provides generic programming facilities, such as variables, loops, and procedures that are useful for a variety of applications, while Tk is a toolkit for the X Window System. Tk extends the core Tcl facilities with commands for building user interfaces. So Tcl/Tk together provide a programming system for developing and using graphical user interface (GUI) application. As a script language, Tcl/Tk enables a programmer to write a program at a higher level than a compiled language like C or C++. Incr Tcl/Tk is to Tcl/Tk what C++ is to C. It is an object-oriented language which introduces the notion of objects as building blocks for an application. Objects are organized into
"classes" with identical characteristics, and classes can inherit functionality from one another. This object-oriented paradigm adds another level of organization on top of the basic variable/procedure elements, and the resulting code is easier to understand and maintain. Incr Tcl/Tk also provides a variety of new widgets called mega-widgets and allows users to construct new widgets easily.

TNR is employed to communicate the VQL application with the Part21 database. It acts as a script command in Tcl and returns the query result by executing an EQL embedded TNR file.

Figure 5.1 shows the VQL architecture, where the arrows between components represent the flow of data and the shaded boxes represent external components that
are not part of the VQL development. For a database in EXPRESS modeling format, its schema structure that can be accessed by EQL is stored in a data dictionary file (a ROSE formatting file) while its data is stored in a STEP file. Users formulate a graphical query based on the interested data structure information obtained from the ROSE file, and the VQL query is mapped to the EQL textual string through the translator. By execution of this translated EQL, the query result is retrieved from the STEP file and is sent back to users through the query presenter. It should be noted that, due to a single directional data flow, both data stored in the ROSE file and the STEP file will not be modified during a query process.

In this chapter, the program organization is overviewed first; then the implementation details—especially with respect to internal representations—are presented.

### 5.2 Overview of the Implementation

As mentioned earlier, VQL is written using the object-oriented scripting language Incr Tcl/Tk, and the data in STEP or ROSE file are retrieved by EQL embedded TNR programs. By providing a lot of high-level graphical component constructs called Tk widgets and Incr Tk mega-widgets, Incr Tcl/Tk helps overcome many of the low-level implementation hurdles commonly encountered when developing a GUI.

According to the VQL system architecture and user interface design, the VQL implementation programs can be classified into the following four parts (see Table 5.1):
Table 5.1: VQL Program Structure

<table>
<thead>
<tr>
<th>File name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Program</td>
<td>vql</td>
<td>Creates the VQL main window and binds action buttons with command procedures</td>
</tr>
<tr>
<td></td>
<td>proc.tcl</td>
<td>Defines event handlers called by vql</td>
</tr>
<tr>
<td></td>
<td>level.itcl</td>
<td>Level Sub: SubLevel</td>
</tr>
<tr>
<td></td>
<td>entity.itcl</td>
<td>Entity</td>
</tr>
<tr>
<td></td>
<td>kLink.itcl</td>
<td>KeyLink</td>
</tr>
<tr>
<td>Incr Tcl Class</td>
<td>cLink.itcl</td>
<td>ConditionLink Sub: BinaryLink UnaryLink Sub: ForLink LogicLink</td>
</tr>
<tr>
<td></td>
<td>Constant.itcl</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td>operate.itcl</td>
<td>Operator Sub: Anyall</td>
</tr>
<tr>
<td></td>
<td>splitter.itcl</td>
<td>Splitter</td>
</tr>
<tr>
<td>Incr Tk Class</td>
<td>entity.itk</td>
<td>EntityBox</td>
</tr>
<tr>
<td></td>
<td>tBoxDialog.itk</td>
<td>ListboxDialog</td>
</tr>
<tr>
<td></td>
<td>keyDialog.itk</td>
<td>KeyDialog</td>
</tr>
<tr>
<td>TWR Program</td>
<td>sList.tnr</td>
<td>Used to retrieve the attribute list of a given entity from a ROSE file</td>
</tr>
<tr>
<td></td>
<td>eList.tnr</td>
<td>Used to retrieve the entity list from a ROSE file</td>
</tr>
<tr>
<td></td>
<td>output.tnr</td>
<td>Used to obtain the query result from a STEP file using the translated EQL statement</td>
</tr>
</tbody>
</table>
• **Main program.** It creates the VQL main window and enables users to construct a variety of VQL graphical objects or execute the query by calling the corresponding procedures defined in “proc.tcl”.

• **Incr Tcl Classes.** A set of Incr Tcl classes is stored in “.itcl” files. They are used as templates for the objects constructed in VQL.

• **Incr Tk Classes.** Three Incr Tk classes are stored in “.itk” files. Each of them is used to build a mega-widget.

• **TNR Programs.** Three TNR programs are stored in “.tnr” files. They act as VQL interfaces to access the data stored in the database.

In the following sections, the high-level description for each part of these programs will be provided.

### 5.3 Main Program

The VQL main program consists of two kinds of scripts: an initialization script and event handlers. The initialization script (stored in file `vql`) is executed when VQL starts. It is responsible for building the VQL main window and providing users with an interface to carry out query activities. First it builds every visible interface component in the VQL main window. These components include a pull-down menu for file, three VQL pages labeled “Query Constructor”, “EQL”, and “Query Result”, a toolbar (with five action buttons: “Entity”, “Key Link”, “Condition Link”, “Delete”,

...
and "Clear"), and a work area canvas in the visual query editor page. Then it creates a Level type object top, which keeps all created VQL constructs specified in the top level and is used to translate VQL to EQL and obtain the query result. Finally, it constructs the following dialog boxes and pop-up menus that will be activated to assist users to make choices during formulation of a VQL query.

- **FileDialog** is used to select the STEP file.

- **EntityDialog** lists all entity names in an EXPRESS schema and enables users to choose the entity name of interest during construction of an entity box.

- **OperatorDialog** lists all condition operators and enables users to choose one during construction of an operator.

- **BinaryDialog, UnaryDialog, LogicDialog, and AnyallDialog** list all binary, unary, logic or for condition operators, respectively, and enable users to choose one during modification of the operator of that specific type.

- **ConstDialog** is used to prompt users for the literal values during construction of a constant operand in a condition link.

- **EntityMenu** enables users to choose the manipulating method for an entity box.

- **AttrMenu** enables users to choose the manipulating method for an output attribute or the pointer of a key link.
- **OperatorMenu** enables users to choose the manipulating method for the operator in a condition link.

- **LogicOperandMenu** enables users to choose the manipulating method for an operand in a logic condition link.

- **AtomicOperandMenu** enables users to choose the manipulating method for an operand in an atomic condition link.

Once initialization is complete, VQL enters an event loop to wait for user interactions. Whenever an interesting event occurs, such as a user invoking a file menu entry or pressing a page button, a script procedure (stored in file proc.tcl) is invoked to process that event. Following are listed the event handlers that are invoked by interface components in the main window. Other event handlers invoked in the created VQL constructs will be introduced as class methods later.

- **openFile.** This procedure is invoked when users select the “Open” option in the file menu, thus activating the **FileDialog** box to enable users to select the name of the STEP file from which the corresponding ROSE file name is found. Then it calls **elist.trn** to retrieve all entity names from the ROSE file and inserts them into the **EntityDialog** box.

- **canv.1.** This procedure is invoked when clicking B-1 on the work area canvas. It is employed to create an entity box or condition link or modify a constant operand in a condition link. The entity box object is instantiated from the
Entity class, the condition link object is instantiated from the BinaryLink, UnaryLink, LogicLink, or ForLink class, and the constant object is instantiated from the Constant class.

- EQLText. This procedure is invoked when the “EQL” page button is activated. It translates the VQL query diagram to an EQL query string by calling the buildQuery method of the object top and sends the results to the window.

- result. This procedure is invoked when the “Query Result” page button is pressed. It obtains the query result by calling the output method of the object top and outputs it in a spreadsheet that is constructed by the table widget.

5.4 Incr Tcl Classes

By using object-oriented techniques, such as encapsulation, inheritance, and composition, a set of Incr Tcl classes is defined as the template for the creation of a variety of VQL constructs and their elements. Each class maintains a specific data structure and has a set of procedures or methods to manipulate its objects. The class diagram in Figure 5.2 shows the relationships among these classes. In this diagram, a box shows a class name and its main methods, an arrow line represents an inheritance relationship, and a line ending with a circle represents a compositional relationship, where a solid line is used to define a class variable and a dot line is used to reference a variable to its associated class. For example, the BinaryLink,
UnaryLink, and LogicLink class are derived from the ConditionLink class and the ForLink class is derived from the UnaryLink class. The Level class uses the variable keyLinks, entities and cLinks to specify a set of KeyLink, Entity, and ConditionLink objects, respectively, and the Constant class references to its associated ConditionLink class by the variable cLink.

This section contains an introduction to each class. All derived classes are explained together with their base class. For brevity, only main member variables and methods are described here.

5.4.1 Level Classes

The Level class and its derived class SubLevel are VQL construct containers. They keep all VQL constructs that are specified in the top level or within for condition links, respectively, and are used to translate VQL and obtain the query result.

5.4.1.1 Level Class

• Variables

- entities, keyLinks, and cLinks store the list of the Entity, KeyLink or ConditionLink objects, respectively.

- indptEntities stores the list of the grouping Entity objects.

- selectedEntities stores the list of the Entity objects with output attributes.
- **andList** stores the list of the **ConditionLink** objects that are ANDeD by default or the root object of the conditional Logic tree.

- **levelList** keeps the list of all **Level** objects. This is a common variable (like a static data member in C++).

**Methods**

- The **constructor** creates a **Level** object and adds its name to the variable **levelList**.

- **addEntit**, **addKeyLink**, and **addCLink** insert the name of a created **Entity**, **KeyLink**, or **ConditionLink** object into the variable **entities**, **keyLinks**, or **cLinks**, respectively.

- **removeEntity**, **removeKeyLink**, and **removeCLink** remove the name of a destroyed **Entity**, **KeyLink**, or **ConditionLink** object from the variable **entities**, **keyLinks**, or **cLinks**, respectively.

- **buildQuery** translates the current VQL query to an EQL textual string by calling the method **buildSelectedEntities**, **buildIndptEntities**, **buildAndList**, **assignVariable**, and **removeLogicLinkNodes**. The mapping algorithm has been discussed in Chapter 3.

- **output** returns the query result by calling the TNR program **output.tnr**.
5.4.1.2 SubLevel Class

The SubLevel class is derived from Level class. It acts as a member class of the ForLink class, which will be described later. In addition to the variables and methods inherited from Level class, it also adds one variable constants to store all Constant objects created in the corresponding any/all operator.

5.4.2 Entity Class

The Entity class is used to create and manipulate entity boxes (including its internal elements—output attributes). Because of the high relationships among VQL constructs, it also carries out some operations dealing with key links or condition links.

- Variables
  
  - context specifies which level this Entity object belongs to. If an entity is created in the top level, then the context value is top; otherwise, its value is the name of the ForLink object to which this entity belongs.
  
  - keyList and condList store all the KeyLink objects or ConditionLink objects that connect with this Entity object, respectively.

  - name stores the entity name.

  - entityWin stores the entity box widget.

  - select stores the specified output attributes.
- variable stores the binding variable (grouping entity) or grouped-by entity expression during VQL translation.

- first specifies the first selected entity during creation of a key link. It is a common variable.

- Methods

  - The constructor constructs an entity box by calling the method mkEntity and creates an AttrDialog box of this entity, which enables users to choose attribute names when they specify output attributes in this entity or construct attribute operands in condition links.

  - The destructor destroys the Entity instance as well as all key links and condition links that connect with this entity.

  - command is invoked to produce a key link between this object and the entity whose name is stored in a variable first by calling the method mkKLink, create an attribute operand in a condition link by calling the method mkCLink, or delete this Entity object.

  - entityMenuCommand activates the EntityMenu to enable users to choose the manipulating method for the entity. Users can switch the entity type, add an output attribute, or delete this Entity object by calling the corresponding method.

  - attrMenuCommand activates the AttrMenu to enable users to choose the
manipulating method to the attribute. Users can append/delete a suffix or delete the attribute by calling the corresponding method.

- mkEntity creates an entity box using the user-defined mega-widget EntityBox.

- mkKLink receives an Entity object as argument. If the variable first is not empty, this method will build a key link object that is instantiated from the KeyLink class between the given entity and the entity whose object name is stored in the variable first. Otherwise, first is updated with the given object name.

- mkCLlnk creates an attribute operand in a condition link. It first activates the AttrDialog to enable users to choose an attribute name. Then the attribute operand is constructed by calling the method selectOperand of the associated ConditionLink object.

- setGroupBy reverses a grouped-by entity to a grouping entity.

- addAttribute adds an output attribute into the entity box.

- deleteAttribute deletes an output attribute from the entity box.

- appendSuffix and deleteSuffix add or delete a suffix ("[]" or "[index]") from an attribute.

- assignVariable assigns a variable or path expression to the entity during VQL translation.
moveEntity is invoked whenever users press B-2 on the entity name label and move the mouse. It is used to move the entity box. When an entity box is moved, all its connecting condition links and key links are also moved with it.

5.4.3 Key Link Class

The KeyLink class is used to create and manipulate VQL key link constructs. It maintains a data structure to represent the linked attributes and their associated entity boxes.

- Variables

  - context specifies which level this KeyLink object belongs to.
  
  - node1 and node2 are used to hold the two key-linked Entity objects, respectively.
  
  - attributes1 and attributes2 store the attribute name of node1 or node2, respectively. One of them is a pointer and the other is object ID ("#”).
  
  - attrLabel specifies the label widget that is used to represent the pointer attribute.
  
  - ok acts as a flag to signal whether the current building KeyLink object is legal or not. During construction of a KeyLink object, if users change their mind and press the “cancel” button when the KeyLinkDialog box is
activated, the ok is set as false. Therefore, the current building key link will be deleted.

- nodes is a common array variable with a pair of the key-linked Entity objects as index. By checking whether an array element exists, we can make sure only one key link is built between two entities (see method exists).

- **Methods**

  - The constructor is invoked whenever the “Key Link” button is activated and users consecutively click B-1 on two entity boxes. The constructor first activates a KeyLinkDialog box to enable the user to choose the pointer. Then a link line is drawn to connect the two entity boxes and a label widget of the chosen pointer is created and placed near its associated entity box on the line.

  - The destructor destroys the key link line and the pointer label. It also will check to see whether the remains of the VQL diagram is syntactically legal. If either of the original linked entities and all other entities directly or indirectly linked with it are all grouped-by entities, then that entity will be modified as a grouping entity.

  - attrMenuCommand is invoked when users click B-3 on the pointer label. It activates the AttrMenu to enable users to choose the manipulating action
on the \textbf{KeyLink} object. Users can add/delete a suffix to the pointer or delete the \textbf{KeyLink} object.

- \textit{moveKeyLink} receives as argument an \textbf{Entity} object involved in the \textbf{KeyLink} object. Whenever the \textbf{Entity} object is moved, the key link line and the pointer label will then be moved as well.

- \textit{exists} returns a boolean value to specify whether two entities have been key-linked. If two entities have been linked, no additional key links can be built between them.

\section*{5.4.4 Condition Link Classes}

As mentioned in Chapter 3, VQL condition links can be divided into four groups: binary, unary, logic, and for condition links. The \textbf{ConditionLink} class acts as a base class upon which the \textbf{BinaryLink}, \textbf{UnaryLink}, and \textbf{LogicLink} classes are directly derived. The \textbf{ForLink} class is derived from the \textbf{UnaryLink} class. The objects instantiated from these derived classes contain the data structures and methods to create and manipulate all kinds of condition links. Some elements in these classes, such as operators and constants are instantiated from the subcomponent classes that will be discussed in the following subsections.
5.4.4.1 ConditionLink Class

- Variables

  - context specifies to which level this ConditionLink object belongs.

  - operatorOb specifies the Operator object.

  - operator stores the operator name, such as ">", "==", "in", etc.

  - output stores the name of the LogicLink object if this condition participates in a logic expression; otherwise, it is empty.

  - negate indicates whether the condition link is a positive type or negative type. Its initial value is false (positive).

  - cord.x stores the horizontal position of the operator symbol (polygon) in the work area. It is used to decide the evaluation order of this condition when it participates in a logic operation.

  - mode, type, lastObject, ok, and modify are important common variables to control the constructing sequence for a ConditionLink object. The variable mode has value “operator” or “operand”. It specifies to which phase the current construction belongs. If it has the value “operator”, then an Operator class will be created when users click B-1 on the work area canvas; otherwise, it is time to create an operand. The variable type is used to indicate whether the current building condition is of atomic type or logic type. If it is atomic type, then only constants and attributes can be operands; otherwise, the operands are other ConditionLink objects.
The variable `lastObject` keeps track of the `ConditionLink` object that is being created or modified. The variable `ok` is a boolean value. It indicates whether a `ConditionLink` object is constructed completely. Whenever the construction of a `ConditionLink` object is interrupted by another exclusive process, `ok` is set as `false` and the incomplete object will be deleted (see method `clear`). The variable `modify` is a boolean value. If it is `true`, it means the `lastObject` is being modified (such as change operator, add a suffix to an attribute operand, etc.) All other construction activities must be postponed until the modifying action is complete.

- **Methods**

  - The `constructor` is responsible for creating the operator in the `ConditionLink` object. If the operator is not “any” or “all”, the created operator object is instantiated from the `Operator` class; otherwise, it is instantiated from the `Anyall` class.

  - `negate` negates the current operator by calling the `negate` method of the `Operator` object in the condition link.

  - `moveOperator` moves the operator in the condition link by calling the `moveOperator` method of the `Operator` object in the condition link.

  - `operandMenuCommand` is invoked when users click B-3 on a operand in an atomic condition link. It activates the `AtomicOperandMenu` and enables
users to append/delete a suffix to the attribute operand or change the operand by calling the corresponding method. This is a class procedure.

- `clear` deletes current the `ConditionLink` object if the flag variable `ok` is `false`. This method is a class procedure.

- `operandExpression` is used to determine the operand expression based on the given attribute operand and its associated `Entity` object. This is a class procedure.

### 5.4.4.2 BinaryLink

- Superclass: `ConditionLink`

- Variables

  - `node1` and `node2` hold the two nodes with which the condition link connects. An operand of the condition link may be an attribute, constant, or the split operator within an `any/all` operator. If the operand is an attribute, the corresponding node is its associated `Entity` object; if the operand is a constant, the corresponding node is the `Constant` object. Otherwise, the node is a `Splitor` object.

  - `operand1` and `operand2` hold the contents of the two operands. The content may be an attribute name, constant value, or empty if the operand is a split operator.
- operandWin1 and operandWin2 store the label widgets of attribute operands. For other type operands, these variables are empty.

• Methods

- selectOperand creates operand elements in the BinaryLink object. The first created operand is linked with the left side of the operator, and the second created operand is linked with the right side. When both operands have been constructed, the inherited variable ok is set as true. If the operand is an attribute, an attribute label is created and is placed on the line that connects its associated entity box and the operator. If the operand is a constant, a constant label is created. The label is directly connected with the operator by a line. If the operand is the split operator within an any/all operator, then the split operator is connected with the operator by a line.

- moveCondLink is invoked whenever users press B-2 on either of the operand nodes (entity boxes or constants) in the condition link and move the mouse. It is used to move the condition link line so that it always connects with the moving node.

- expression formulates the binary condition expression during VQL translation.
5.4.4.3 UnaryLink Class

The **UnaryLink** class is derived from **ConditionLink** class. It is similar to the **BinaryLink** class except that it only has one operand. Therefore, it uses the variable `node`, `operand`, `operandWin` to specify the node connected by the operator, the operand content, and the attribute label widget (if the operand is an attribute). The method `expression` is used to formulate the unary condition expression during VQL translation.

5.4.4.4 ForLink Class

- **Superclass:** **UnaryLink**

- **Variables**
  
  - `subLevel` stores the name of a **SubLevel** object, which contains all the VQL constructs specified within the `any/all` operator of this **for** condition link.
  
  - `variable` stores the binding variable of the operand of the **for** condition link.

- **Methods**
  
  - `moveLinkLine` is invoked when the `any/all` operator of this **for** condition link is resized. It redraws all the link lines that connect with this operator.
  
  - `expression` formulates the **for** condition expression during VQL translation.
5.4.4.5 LogicLink Class

- Superclass: `ConditionLink`

- Variable

  - `nodes` stores all of the `ConditionLink` objects that act as operands in this logic condition link.

- Methods

  - `logicMenuCommand` is invoked whenever users press B-3 on the line that connects the logic operator with an operand (i.e. condition operator). It activates the `LogicOperandMenu` to enable users to modify or delete the operand.

  - `selectOperand` creates an operand in the current logic condition link.

  - `expression` formulates the logic condition expression during VQL translate.

5.4.5 Constant Class

The `Constant` class is used to create and manipulate the constant operand in atomic condition links. Its main data members and methods are listed below:

- Variables

  - `cLink` specifies to which the `ConditionLink` object this constant belongs.
- constWin stores the name of the constant label created by this Constant object.

- Methods

  - The constructor creates a constant label on the canvas and connects this label to the condition operator by calling the selectOperand method of its associated ConditionLink object.

  - moveConst is invoked when users press B-2 on the constant label and move the mouse. It moves the constant label first. Then it calls the moveCondLink method of its associated ConditionLink object to move its link line.

5.4.6 Operator Classes

One important element in a condition link is operator. The Operator Class is used to create and manipulate all types of operators except for the any/all operator, which is supported by the Anyall class, a subclass of the Operator class.

5.4.6.1 Operator Class

- Variables

  - cLink specifies the ConditionLink object to which this operator belongs.
- *leftX* and *leftY* keep the x and y coordinates on the left top point of the polygon.

- *width* and *height* store the width and height value of the polygon.

- *ncoord, scoord, wcoord, and ecoord* store the coordinates of the north, south, west, and east connecting nodes of the polygon. The operator is linked to its operands from one of these nodes. The west node and east node is used for binary operators, north node is used for unary and logic operators, and the south node (the tip of the polygon) is used to connect the operator to its associated logic operator.

- **Methods**

  - The *constructor* draws a polygon and text item on the work area canvas. The text item displays the operator name. The constructed operator is positive type.

  - *negate* reverses the type of the current operator. If it is a positive operator, a small circle is drawn below the tip of the polygon. If it is a negate operator, the small circle is removed from the operator.

  - *moveOperator* is invoked when users press B-2 on the operator and move the mouse. It keeps moving the operator to the new pointer of the mouse.

  - *operatorMenuCommand* is invoked when users click B-3 on the operator. It activates the *OperatorMenu*. By choosing different options, users can
negate/modify the operator, or delete the whole condition link. For a logic condition, it also enables users to add another operand.

- *modifyOperator* is invoked when users click the “modify” option in *OperatorMenu*. Depending on the type of current operator, a *BinaryDialog*, *UnaryDialog*, *AnyallDialog*, or *LogicDialog* box will be activated. Users are allowed to choose any one of the operators listed in the dialog box to replace the current one.

### 5.4.6.2 Anyall Class

- **Superclass:** *Operator*

- **Variable**

  - *splitor* stores the name of the *Splitor* object in this *any/all* operator.

  - *splitorcoord* keeps track of the split operator position within the *any/all* operator.

- **Methods**

  - The *constructor* creates the operator by calling its *parent class constructor* first. Then it constructs another polygon, which acts as the test condition area, at the bottom of the *any/all* operator. It also draws a line between the operator name and the test condition area. A small button which is created by an *Splitor* object is placed at the center of the separate line.
This button acts as the split operator. The test condition area is used to hold all the VQL constructs involved in the `for` condition link.

- The `destructor` is used to destroy all elements of the `any/all` operator as well as all the VQL constructs that are created in its test condition area.

- `moveOperator` invoked when users press B-2 on the operator and move the mouse. It keeps moving the operator as well as all VQL constructs created within the operator to the new pointer of the mouse.

- `resize` is invoked whenever a new VQL construct is created in the `any/all` operator or an old one is deleted. It redraws the operator so that the size of the test condition area in the operator is fit for displaying all the nested VQL constructs.

- `command` is used to create entity boxes and condition links within the `any/all` operator.

### 5.4.7 Splitor Class

The `Splitor` class is employed to create and manipulate the small button (split operator) in an `any/all` operator. This small button acts as operands of the condition links specified within the `any/all` operator.

- **Variables**

  - `cLink` specifies to which condition link the `Splitor` object belongs.
- *cLinks* keeps all the condition links that take this split operator as operands.

- *splitWin* stores the name of the small button created by the *Splitor* object.

- **Methods**
  
  - The *constructor* creates a small button widget at the center of the separate line in its *any/all* operator.
  
  - *splitorLink* connects the split button and a condition operator created within the *any/all* operator during construction of a sublevel condition link.

### 5.5 Incr Tk Class

Incr Tk allows programmers to create new widgets, using the normal Tk widgets as component parts. These mega-widgets look and act like ordinary Tk widgets. This feature makes it easier to construct the user-defined widgets. In this application, three mega-widget classes have been built. Each of these classes is briefly described below.

#### 5.5.1 EntityBox Class

This class is used to create the entity box widget. It is derived from the base class * itk::Widget*. Two components are added into the class: a frame and a list. The border of the frame acts as the border of the entity box, which is used to indicate
a grouped-by or grouping entity type. The list component, which is packed in the frame, is a "scrolledlistbox" widget. It has a label to display the entity name, and a listbox to display all output attributes in the entity box.

5.5.2 ListboxDialog

A set of dialog boxes, such as EntityDialog, AttrDialog, OperatorDialog, BinaryDialog, UnaryDialog, AnyallDialog and LogicDialog, are created from the mega-widget class ListboxDialog. This class first inherits the base class iwidgets::Dialog; then an “scrolledlistbox” widget is added into the dialog. It has a label to display the dialog name and a listbox to list the multiple entries, which are provided for users to choose.

5.5.3 KeyDialog Class

The KeyDialog class is used to create the KeyLinkDialog box, which allows users to choose the pointer during construction of a key link. This class is also derived from the base class iwidgets::Dialog. It adds two “scrolledlistbox” widgets into the dialog box (one at the left and the other at the right). Each of them is responsible to list the attributes of one of the key-linked entities.

5.6 TNR Programs

TNR programs may be used to access the data from the ROSE file or STEP file. They send EQL statements to the EQL query engine and receive query results
from it. Following are listed the three TNR programs in VQL.

### 5.6.1 eList.tnr

Given a ROSE file name, it searches all entities specified in the schema and returns all these entity names as a list, which is inserted into the \textit{EntityDialog} box.

### 5.6.2 aList.tnr

Given a ROSE file name and an entity name specified in the schema, this program searches all the attribute names in the entity. For an aggregate attribute, a square bracket "[]" is appended to its name. It returns these attribute names as a list, which is used to insert into the \textit{AttrDialog} box for that entity.

### 5.6.3 output.tnr

Given a STEP file, this program reads the translated EQL textual string from the standard input and executes it. The query result is sent back to the Incr Tcl application and displayed in the spreadsheet of the output page.
Chapter 6

Conclusions and Future Work

6.1 Introduction

The past chapters describe the design and implementation of a new visual query language called VQL. What follows in this chapter is a discussion of the conclusions that can be drawn from this effort, and areas for future work have been identified.

6.2 Conclusions

VQL provides a friendly graphical interface for users to formulate queries on the data stored in a Part21 file without having to know the syntax of a textual language. It has a well-defined syntax, and its semantics are defined by the translation of VQL to EQL. It possesses all the intended features that were discussed in Chapter 1.
VQL is a fully graphical query language. All query activities can be formulated through three kinds of graphical constructs: entity boxes, key links, and condition links. The entity boxes are used to select the entities and attributes of interest from a schema, the key links are used to specify the groupby operations, and the condition links are used to specify the various select criteria.

Also, each of VQL constructs and their internal elements have a distinct meaning. This unambiguity provides for precise interpretation of queries.

One important feature of VQL is the ease of use. It provides a graphical interface that uses popular, convenient techniques (e.g., windows, action bars, and menus); all query constructs are built by just clicking the mouse except for constants, which are typed in from the keyboard. The constructs have been designed to obey visual criteria (readability, intuitive understanding, ...) without any attempt to mimic operations of the underlying data manipulation language. No variables are used in VQL. All these features enable a casual user to learn the VQL usage and formulate queries very easily.

Finally, VQL has the same expressive power as its textual counterpart—EQL.

6.3 Future Work

There is no doubt that VQL can be both improved and extended. As pointed out earlier, current VQL version provides only the entity names and attribute names in the schema, and the link relationships is based on the users' knowledge. It still needs users to be familiar with the data schema. One area for improvement in the
current VQL version is building a VQL schema browser that enables users to obtain all the schema information that is needed to formulate a VQL query, including attribute types and link relationships.

Another extension to VQL is the introduction of the aggregate functions, such as maximum, minimum, average, count, etc. Although EQL does not support these mathematical functions, they can be implemented in TNR.

In addition, in order to format the query result according to a user’s requirements, it is necessary to add the functions of field ordering and sorting in the output spreadsheet.
Bibliography


Appendix A

An EXPRESS Sample

SCHEMA class_record;

TYPE QUARTER = ENUMERATION OF (fall, winter, spring, summer);
END_TYPE;

ENTITY classOffering;
  title: STRING;
  id: STRING;
  instructor: STRING;
  callNumber: INTEGER;
  quarterName: QUARTER;
  year: INTEGER;
  enrollments: SET OF enrollment;
  assignments: SET OF assignment;
  exams: SET OF exam;
  UNIQUE
    key: id;
    alternate: title;
END_ENTITY;

ENTITY student;
  name: STRING;
  id: INTEGER;
  INVERSE
    myEnrollments: SET OF enrollment FOR theStudent;
  UNIQUE
    key: id;
END_ENTITY;

ENTITY enrollment;
  theStudent: student;
  finalGrade: REAL;
  assignmentGrade: REAL;
  examGrade: REAL;
  assignmentWeight: REAL;
  examWeight: REAL;
  assignmentGrades: ARRAY OF REAL;

examGrades: ARRAY OF REAL;

ENTITY student

theClass: classOffering FOR enrollments;

UNIQUE key: theStudent, theClass;

END_ENTITY;

ENTITY gradeWork

ABSTRACT SUPERTYPE OF (ONEOF (assignment, exam));

name: STRING;

description: STRING;

possibleScore: REAL;

weight: REAL;

END_ENTITY;

ENTITY assignment

SUBTYPE OF (gradeWork);

INVEMBER

theClass: classOffering FOR assignments;

UNIQUE key: name, theClass;

END_ENTITY;

ENTITY exam

SUBTYPE OF (gradeWork);

INVEMBER

theClass: classOffering FOR exams;

UNIQUE key: name, theClass;

END_ENTITY;

END_SCHEMA;
Appendix B

A STEP File Sample

\singlespace
ISO-10303-21;
HEADER;
// Exchange file generated using ST-DEVELOPER v1.5 */

FILE_DESCRIPTION(
    /* description */ '\n',
    /* implementation_level */ '\2;1');

FILE_NAME(
    /* name */ '\hw9',
    /* time_stamp */ '\98-02-24T15:48:41-04:00',
    /* author */ '\n',
    /* organization */ '\n',
    /* preprocessor_version */ '\ST-DEVELOPER v1.5',
    /* originating_system */ '\n',
    /* authorisation */ '\n');

FILE_SCHEMA ('class_record');
DATA;

#10=CLASSOFFERING('Advanced CIM', 'EE690', 'JUDD', 12345, 'WINTER', 1998,
    (#2000, #2030, #2060, #2090),
    (#5000, #5010, #5020, #5030, #5040, #5050),
    (#4000, #4010, #4020));

#20=CLASSOFFERING('Programming With C++', 'EE590', 'JUDD', 67890, 'FALL', 1997,
    (#2070, #2100, #2120),
    (#5100, #5110, #5120, #5130, #5140, #5150, #5160, #5170, #5180, #5190),
    (#4030, #4040));

#30=CLASSOFFERING('Statistics', 'ISE490', 'KOOYCE', 23456, 'WINTER', 1998,
    (#2050, #2100, #2180),
    (#5200, #5210, #5220),
    (#4050, #4060));

#40=CLASSOFFERING('Data Modeling', 'ISE700', 'KOOYCE', 34567, 'SPRING', 1999,
    (#2000, #2080, #2170, #2190),
    (#5300, #5310, #5320, #5330, #5340),
    (#4070, #4080));

#50=CLASSOFFERING('Artificial Intelligence', 'ISE799', 'PARKS', 45678, 'SUMMER', 1998,
    (#2010, #2040, #2110, #2130, #2140, #2150),
    (#5400, #5410),
    (#4090));

#5000=ASSIGNMENT('hw1', 'easy', 10., 1.);
#5010=ASSIGNMENT('hw2', 'easy', 10., 1.);
#5020=ASSIGNMENT('hw3', 'moderate', 20., 1.);
#5030=ASSIGNMENT('hw4', 'moderate', 30., 1.);
#5040=ASSIGNMENT('hw5', 'difficult', 50., 1.);
#5050=ASSIGNMENT('hw6', 'difficult', 50., 1.);
#5100=ASSIGNMENT('hw1', 'easy', 10., 1.);
#5110=ASSIGNMENT('hw2', 'easy', 10., 1.);
#5120=ASSIGNMENT('hw3', 'moderate', 20., 1.);
#5130=ASSIGNMENT('hw4', 'moderate', 30., 1.);
#5140=ASSIGNMENT('hw5', 'difficult', 50., 1.);
#5150=ASSIGNMENT('hw6', 'difficult', 50., 1.);
#5160=ASSIGNMENT('hw7', 'easy', 10., 1.);
#5170=ASSIGNMENT('hw8', 'easy', 10., 1.);
#5180=ASSIGNMENT('hw9', 'difficult', 50., 1.);
#5190=ASSIGNMENT('hw10', 'difficult', 50., 1.);
#5200=ASSIGNMENT('hw1', 'easy', 10., 1.);
#5210=ASSIGNMENT('hw2', 'easy', 10., 1.);
#5220=ASSIGNMENT('hw3', 'moderate', 20., 1.);
#5300=ASSIGNMENT('hw1', 'easy', 10., 1.);
#5310=ASSIGNMENT('hw2', 'easy', 10., 1.);
#5320=ASSIGNMENT('hw3', 'moderate', 20., 1.);
#5330=ASSIGNMENT('hw4', 'moderate', 30., 1.);
#5340=ASSIGNMENT('hw5', 'difficult', 50., 1.);
#5400=ASSIGNMENT('hw1', 'difficult', 10., 1.);
#5410=ASSIGNMENT('hw2', 'difficult', 10., 1.);
#4000=EXAM('exam1', 'hard', 110., 1.0);
#4010=EXAM('exam2', 'moderate', 120., 1.0);
#4020=EXAM('final', 'moderate', 150., 1.0);
#4030=EXAM('exam1', 'easy', 100., 1.0);
#4040=EXAM('final', 'easy', 100., 1.0);
#4050=EXAM('exam1', 'hard', 120., 1.0);
#4060=EXAM('final', 'easy', 200., 1.0);
#4070=EXAM('exam1', 'hard', 100., 1.0);
#4080=EXAM('final', 'easy', 100., 1.0);
#4090=EXAM('final', 'hard', 100., 1.0);
#1000=STUDENT('kevin', 1000000000);
#1010=STUDENT('larry', 111111111);
#1020=STUDENT('frank', 2222222222);
#1030=STUDENT('gary', 3333333333);
#1040=STUDENT('julia', 4444444444);
#1050=STUDENT('ema', 5555555555);
#1060=STUDENT('wendy', 6666666666);
#1070=STUDENT('dorothy', 7777777777);
#1080=STUDENT('bob', 8888888888);
#1090=STUDENT('david', 9999999999);

#2000=ENROLLMENT(#1000, 97.0, 94., 100., 0.4, 0.6,
    (10.,10.,18.,29.,48.,50.),
    (110.,120.,180.));
#2030=ENROLLMENT(#1010, 83.0, 82., 84., 0.4, 0.6,
    (7.,10.,15.,30.,50.,35.),
    (80.,90.,125.));
#2060=ENROLLMENT(#1030, 66.0, 60., 75., 0.4, 0.6,
    (7.,6.,15.,20.,40.,35.),
    (70.,80.,110.));
#2090=ENROLLMENT(#1040, 0.0, 0.0, 0.4, 0.6,
    ()
    ());

#2070=ENROLLMENT(#1030, 83.0, 80., 89., 0.4, 0.6,
    (8.,10.,18.,27.,49.,45.,9.,10.,50.,44.),
    (91.,87.));
#2100=ENROLLMENT(#1040, 97.0, 99., 95., 0.4, 0.6,
    (10.,10.,20.,28.,50.,45.,10.,10.,50.,50.),
    (100.,90.));
#2120=ENROLLMENT(#1050, 90.0, 92., 88., 0.4, 0.6,
    (10.,8.,20.,25.,45.,45.,10.,10.,50.,40.),
    (90.,86.));
#2050=ENROLLMENT(#1010, 96.0, 93., 100., 0.4, 0.6,  
(10.,6.,20.),  
(120.,200.));  
#2160=ENROLLMENT(#1070, 94.0, 92., 97., 0.4, 0.6,  
(8.,10.,19.),  
(110.,195.));  
#2180=ENROLLMENT(#1080, 73.0, 75., 70., 0.4, 0.6,  
(5.,10.,15.),  
(80.,140.));  

#2020=ENROLLMENT(#1020, 95.0, 95., 95., 0.4, 0.6,  
(9.,10.,15.,20.,50.),  
(95.,95.));  
#2080=ENROLLMENT(#1030, 72.0, 68., 75., 0.4, 0.6,  
(5.,10.,20.,20.,45.),  
(80.,70.));  
#2170=ENROLLMENT(#1070, 90.0, 92., 88., 0.4, 0.6,  
(10.,10.,15.,25.,45.),  
(88.,88.));  
#2190=ENROLLMENT(#1080, 84.0, 88., 80., 0.4, 0.6,  
(8.,8.,15.,28.,45.),  
(90.,70.));  

#2010=ENROLLMENT(#1000, 95.0, 95., 95., 0.4, 0.6,  
(10.,9.),  
(95.));  
#2040=ENROLLMENT(#1010, 40.0, 35., 45., 0.4, 0.6,  
(3.,4.),  
(45.));  
#2110=ENROLLMENT(#1040, 62.0, 65., 60., 0.4, 0.6,  
(7.,6.),  
(60.));
#2130=ENROLLMENT(#1050, 96.0, 95., 97., 0.4, 0.6,
(10.,9.),
(97.));

#2140=ENROLLMENT(#1060, 0.0, 20., 0., 0.4, 0.6,
(2.),
());

#2150=ENROLLMENT(#1070, 87.0, 90., 84., 0.4, 0.6,
(8.,10.),
(84.));

ENDSEC;

EMD-ISO-10303-21;