A Graphical Query Language: VISUAL and Its Query Processing

Nevzat Hurkan Balkir, Gultekin Ozsoyoglu, *Member*, *IEEE Computer Society*, and Z. Meral Ozsoyoglu, *Member*, *IEEE Computer Society*

Abstract—This paper describes VISUAL, a graphical icon-based query language with a user-friendly graphical user interface for scientific databases and its query processing techniques. VISUAL is suitable for domains where visualization of the relationships is important for the domain scientist to express queries. In VISUAL, graphical objects are not tied to the underlying formalism; instead, they represent the relationships of the application domain. VISUAL supports relational, nested, and object-oriented models naturally and has formal basis. For ease of understanding and for efficiency reasons, two VISUAL semantics are introduced, namely, the interpretation and execution semantics. Translations from VISUAL to the Object Query Language (for portability considerations) and to an object algebra (for query processing purposes) are presented. Concepts of external and internal queries are developed as modularization tools.

Index Terms—Database query languages, query processing, scientific databases, object algebra.

1 Introduction

This paper discusses VISUAL, an object-oriented graphical database query language designed for scientific databases where the data has spatial properties, includes complex objects, and queries are of exploratory in nature. Although many query languages have been proposed for the object-oriented model (e.g., IQL [2], Hilog [12], F-logic [29], [30], ORION [27], O_2 [4], LLO [32], XSQL [28], NOODLE [37], spatio-temporal models [26], [5], [18]), there are very few visual query languages that handle multimedia data with spatial and sequence properties in a seamless manner.

VISUAL employs the example-element concept of Query-by-Example (QBE) [51], [43] to formulate query objects. VISUAL has such concepts as hierarchically arranged subqueries,[1] internal and external queries, etc., borrowed from Summary-Table-by-Example (STBE) [40], [42]. It has formal semantics and derives its power from a small number of concepts and constructs. VISUAL graphical objects are independent of the underlying formalism (unlike, say, DOODLE [14] and F-logic), and users can define icons as well as place them on the screen to express spatial or composition relationships. For querying, new constructs are defined in VISUAL to allow users to express relationships between application domain classes. These constructs are graphical representations of the constructs in the underlying formal language.

For ease of use and friendliness, graphical query objects and their placements imitate closely what domain scientists use in their applications to represent experiment data-related semantic relationships [48] (e.g., composition hierarchies, containment, class-subclass hierarchies, spatial relationships). This is because domain-specific methods and functions of the data model are represented as graphical icons and icon shapes are created by domain scientists (to recreate the environment that they are familiar with) and added to an icon class. In fact, for user friendliness, domain scientists (i.e., materials engineers) helped to design the graphical icon shapes for domain specific (i.e., materials engineering) functions and methods (as well as all other graphical icons) illustrated in the examples of this paper.

VISUAL can handle scientific experiment data-related data types:

1. It can retrieve and display electron microscopy pictures or their revised versions obtained by image processing software.
2. Data model objects of type collection, i.e., sets, bags, lists, sequences, etc., can be manipulated and generated by VISUAL query objects.
3. Data model object components can be sets.
4. Aggregate functions over collection object types (i.e., sets, bags, etc.) can be specified.
5. Set operations are directly specified.
6. Aggregate functions and set comparison operators are used to restrict the free use of negation and universal quantifier [42].
7. Recursion, in a limited form, is available.

VISUAL also has the following features:

1. Please note that we refer to the query object hierarchy that involves a single query (i.e., subqueries of a given query). Each query object is maintained in the system separately. We do not deal with the issue of organizing query objects themselves into a schema hierarchy.

- *N.H. Balkir is with Experian North America. E-mail: balkir@home.com.*
- *G. Ozsoyoglu and Z.M. Ozsoyoglu are with the Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH 44106. E-mail: {tekin, ozsoy}@eecs.cwru.edu.*

Manuscript received 17 Sept. 1997; revised 12 Jan. 2001; accepted 20 Mar. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 105701.

1. **Object-Oriented Query Specification Model.** Query language constructs of VISUAL are also objects. A query is implemented as an object and query objects interact with each other for query processing. We distinguish between query specification objects and data model objects by separating the *object-oriented query specification model* and the *object-oriented data model*, respectively. Advantages of having object-oriented query specification include

   a. *Uniformity.* Query objects are created, deleted, and accessed by object handles.
   b. *Query sharing.* The notion of query objects provides a paradigm for sharing queries among query objects.
   c. *A newparallel/distributed query processing paradigm.* The capability to request services of objects independently leads to a natural, user-specified multiprocessing environment.
   d. *Time-constrained query processing.* Services of a query object can be requested with time deadlines or with a rate of delivery guarantees (not covered in this paper).
   e. *Synchronized query processing.* Services of a query object can be requested in a synchronized manner (not covered in this paper).
   f. *Security.* Services of a query object can be made available only when an authorization is placed (not covered in this paper).

   In addition, the object-oriented query specification model also utilizes the usual advantages of object-oriented data models: query objects have class-subclass hierarchies, inheritance, service overloading and overriding, etc.
2. **Client-Server Query Object Model.** When the services of a query object is requested, the query object acts as a server and the requesting query object becomes a client. The client-server approach to query modeling and processing is modular and self-contained: Each query object requests the services of a series of query objects that interact with each other in a modular fashion. And, by adapting the client-server query object model, we benefit from the techniques introduced in the client-server model of processes in operating systems.
3. **Single Interpretation and Multiple Execution Semantics.** VISUAL query objects are specified and interpreted by what we call the *interpretation semantics* and executed by the *execution semantics.* There may be multiple execution semantics within a given query where some (sub)queries are merged and translated into a (complex) algebra expression, some others into an Object Query Language (OQL) [38] expression, and yet others may communicate with each other using directly the interpretation semantics or as clients and servers. In [3], we provide two VISUAL translations, one to OQL and another to a nested algebra [17]. These translations can be done both within a single query object and among several query objects.
4. **Evaluating Methods, Aggregate Functions, and Set Operations Uniformly.** For the bottom-up complex algebra and OQL execution semantics, we identify the common properties of methods, aggregate functions, and set operators, and introduce a new operator, called the Method Applier, which is simple and provides uniformity. This leads to the use of a single (complex algebra or OQL) query optimization technique for evaluating methods, aggregate functions, and set operators.

   The main contributions of this paper are:

   a. The language VISUAL is described in detail through extensive examples (Section 3).
   b. Direct VISUAL to OQL translation is described (Section 5). This is when VISUAL is to be used as a front-end to a DBMS that supports OQL.
   c. Novel parts of VISUAL to O-algebra query processing are discussed (Section 6), which involve three issues:

      - *Merging query objects.* If one executes VISUAL subqueries independently (i.e., without merging them into a single query), for queries with multiple subqueries, the query evaluation becomes extremely slow as each subquery (object) gets evaluated independently for each and every input parameter object (which can be viewed as an input "tuple" instance) passed to it from another query object. The issue of merging query objects is similar to evaluating nested SQL queries by first unnesting (flattening) them.
      - *The use of a single operator (method Applier) for uniform treatment of merging query objects that perform method executions, aggregate operations, and set operations.* This makes the implementation very simple and uniform.
      - *Composition (Hierarchy) Flatten operator.* This operator flattens the composition object hierarchy for query processing needs (i.e., to allow access to encapsulated object member variables and functions).

   d. We briefly describe the implementation and the architecture of the system that translates VISUAL into an object-oriented algebra, called O-algebra, and executes the O-Algebra expressions.

We elaborate on the above-listed and other features of VISUAL through examples in the rest of the paper. Section 2 describes a scientific application domain and its data model that is used as a running example in the rest of the paper. However, VISUAL can be used in other domains and applications by simply changing the methods and icon shapes in the query object specification model. Section 3 describes the query specification in VISUAL. Section 4 discusses the advantages of treating queries as objects in the query specification model. In Section 5, we introduce a mapping from VISUAL to OQL. Section 6 summarizes the techniques we use for handling method executions, aggregate function evaluations, and set operations in VISUAL to algebra translation. In Section 7, we briefly

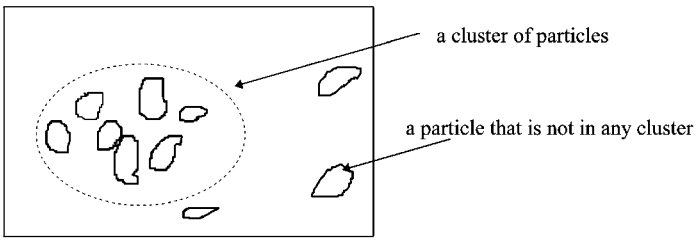


Fig. 1. A possible frame.

summarize the formal properties of VISUAL, in terms of the underlying formal language and safety. Section 8 summarizes the implementation effort. Section 9 reviews the related recent work. Section 10 is the discussion.

2 The Application Domain and the Data Model

The basic modeling primitive for VISUAL is object, which is represented by its object identifier. It can be constructed by a composition of atomic objects (of type integer, etc.), complex objects, and collections (set, bag, sequence) of objects.

We model the domain of materials engineers. Metal composites formed from aluminum are reinforced with Silicon Carbide (SiC) (rubber) particles. The addition of SiC particles provides elasticity improvement to the material, but the inhomogenieties in the distribution of particles may also initiate fractures. Thus, experiment data is collected about the behavior of fractures, particle movements, particle splitting, etc., under varying amounts of reinforcement and deformation processes. Queries involving the experiment data are ad hoc and exploratory in nature.

Before, during, and after an experiment, micrographs of a material surface are taken. The surface is formed into a grid and each micrograph represents a "frame" inside the grid. The micrographs are digitized and image data in pixel form is obtained. Queries on this data involve the geometric aspects of objects (i.e., particles, clusters of particles, frames, etc.) and, thus, in addition to the pixel-form image data, geometric shapes of particles in image data are also stored as frames. An experiment data consists of images, relations (tables about an experiment), summary tables (summary information about the experiment), graphs, histograms (analyzing the experiment data), video data (capturing the experiment itself), audio narration (of the scientists explaining experiment-related information), annotation (text, footnotes, notes, etc., of experiments), material composition and preparation information, experiment parameters (such as stress levels, etc.), a sequence of grids, and the start and the end times of the experiment.

Each frame contains SiC particles. Some of these particles form clusters. A frame has an explicit time (inherited from the grid time that contains the frame) referring to the image data creation time, from which the frame is derived. Through manual or automated procedures, a frame can be revised and a new frame can be obtained. Such revisions include splitting a particle (shape) in a frame into multiple particles (shapes) or merging different particles (shapes) into a single particle (shape). Revisions create versions of a frame and are captured by *child frames* and a *parent frame* of a given frame.

A particle in a frame is *evolved from* a particle in a given frame (although both particles are distinct) and *evolves into* (possibly, a set of) particle(s) in the next frame. A particle *splitEvolves* into multiple particles if it splits into multiple particles over a sequence of frames. A particle *singleEvolves* if it does not split over a sequence of frames. Since the set of particles evolved from a particle stay very close to each other in the materials science domain, they usually stay in the same frame. Each cluster consists of particles and is described by its size, density, minimum bounding rectangle, etc. A cluster may have different definitions according to its application. Roughly speaking, a cluster is a certain area inside a frame satisfying a certain criterion such as density, large particle counts, etc. Specification of a cluster is chosen by the user from a number of alternatives:

a. a grid cell: each cell is an equal-sized rectangle,
b. any number of contiguous grid cells,
c. a fixed-sized rectangle or a predefined shape (the boundary of a cluster is not limited to the grid lines), or
d. any shape at any location.

Fig. 1 contains a possible frame with particles (shapes). To simplify the presentation, the database schema in Fig. 2 shows only a part of the sample data domain that we modeled.

3 Object-Oriented Query Specification Model

3.1 Basic Features

A VISUAL query is represented by a window and refers to zero or more *external* and *internal* query objects, also represented by windows.[2] The query whose execution is to be requested is called the *main* query object. Each query object is composed of a *query head* object and a *query body* object (Fig. 3). The concepts of (query) head and (query) body are similar to rule head and rule body, respectively, in a Datalog rule [49].

Head object contains the name of the query (a unique name that distinguishes the query), query parameters (a list of input and output objects), and the output type specification. Although query name is distinct for every query object, more than one window may have the same query

2. A query object B is internal to another query object A if B's windows appears within the window of A either fully or with the annotation "internal;" otherwise, B is external to A. Section 3.4 covers internal and external query objects.

```
class Experiment [gridSequence: < grid > ;
                  annotations: {Annotation};
                  timeInterval: [begin:Time; end:Time];
                  ...];
class Frame [ particleSet: {Particle};
              clusterSet: {Cluster};
              annotations: {Annotation};
              location: [upperLeft,  lowerRight];
              ...];
class Particle [ frameIn: Frame;
                 evolvedFrom: [ frame:Frame; particle:Particle];
                 evolvedInto: [frame:Frame; particle:{Particle}];
                 centroid: [xdim:Integer; ydim:Integer];
                 ...]
class Grid[ time: Time;
            frames: {Frame};
            spatialRange: [upperLeft, lowerRight];
            ...];
class Cluster [ particleSet: {Particle};
                location:[upperLeft, lowerRight];
                ...];
class VersionFrame: base Frame
            [ childFrames: {VersionFrame}
              parent: VersionFrame;];
```

({}, [ ], and < > denotes sets, tuples, and sequences, respectively)

Fig. 2. The database schema.

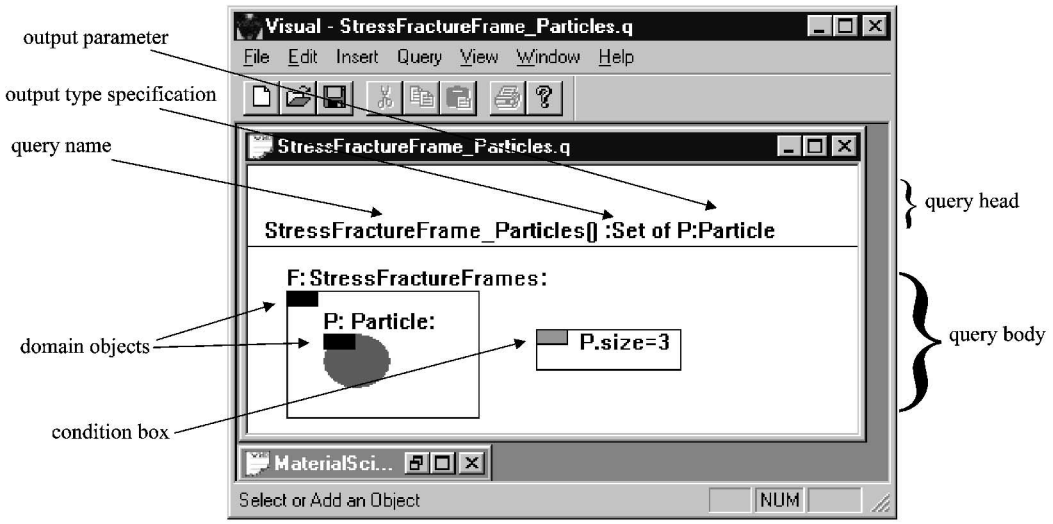


Fig. 3. StressFractureFrame_Particles.

name. In such cases, the query output is the union of all query outputs that have the same name. Input list is defined by query parameters, which function in different ways for different query types (internal, external, or main). In an internal query, query parameters define only output parameters and input parameters are identified indirectly. In external and main queries, input parameters are given in parentheses with their names and types. VISUAL is a strongly typed language and each output parameter is defined with a name and a type specification (after the output type of the query). The type specification can be an object type (the query returns a single object) or a collection type (the query returns a collection of object ids (oids)).

Example 3.1. In Fig. 3, query StressFractureFrame_Particles has no input parameters and one output parameter. The variable for the output parameter is of type Particle and named **P**. The output type is Set. As a result, query output is a set of Particle oids.

Query body may consist of *iconized objects*, condition boxes, and references to external or internal queries. The condition boxes may contain arithmetic or set expressions. An aggregate function that operates on the results of internal or external queries may be an operand in an arithmetic expression. A set expression may refer to internal and/or external query as operands. Set expressions may contain set membership $\{\in, \notin\}$ operators or set comparison $\{\supset, \supseteq, \equiv, \subset, \subseteq\}$ operators.

Example 3.2. In Fig. 3, the query body of StressFractureFrame_Particles contains two domain objects and a condition box. The first domain object is the rectangle-shaped icon **F** that is of type Frame and comes from the domain StressFractureFrames. The second domain object is the ellipse-shaped icon **P** that is of type Particle. The arithmetic condition box contains a restriction for the size of **P**.

3.2 Iconized Objects

Iconized objects represent data model objects, data model object collections, and spatial and domain-specific methods involving data model objects and/or data model object collections. There are four classes of iconized objects and each class has unique properties (e.g., color or shape) as defined in the query specification model:

a. *domain objects* represent base objects or base object collections in the database,

b. *method objects* represent domain-specific methods involving data model objects and/or data model object collections (i.e., domain icons),
c. *range object*, and
d. *spatial enforcement region object*.

A domain object contains a variable name, the corresponding type specification, a domain (optionally), and an iconic representation.

Example 3.3. The query object StressFractureFrame_Particles in Fig. 3 returns the particles each of which has a size equal to three microns and resides in a frame within the set of frames named StressFractureFrames.

3.3 Representing Semantic Relationships Among Data Model Objects

A domain object within another domain object represents one of the three semantic relationships: *spatial*, *composition*, or *collection membership.* The collection membership relationship occurs when 1) the outer iconized object represents a query call or 2) the outer iconized object has an attribute, which is a collection of the inner iconized object. When the outer iconized object represents a query call, the domain of the inner object is restricted by the output of the query object.

Example 3.4. In Fig. 4, the external query Particles_In_Window returns a set of Particle oids. The domain of icon **P** is restricted by the particles in the supplied window.

When the outer iconized object has an attribute, which is a collection of the inner iconized object, the domain of the inner object is restricted by this collection.

Example 3.5. In Fig. 3, the domain of P is restricted by the particleSet attribute of Frame class. Users specify the attribute name that corresponds to a collection relationship to avoid ambiguity.

We also use the same two-dimensional window space to represent both composition hierarchies and spatial relationships among data model objects. Composition hierarchies are represented when the outer domain object has an attribute that has the same type as the inner domain object. A *spatial attribute* is an attribute of a domain object that specifies the object's geometric coordinates (e.g., a particle's x and y coordinates on the grid). An object with spatial attributes is a *spatial object* (e.g., frame or particle); otherwise, it is a *nonspatial* object (e.g., experiment).

In VISUAL, a *selected set of* spatial relationships is visually represented among objects and classes, by using a region in the query body, called the *spatial-enforcement region*,[3] which represents a selected set of spatial relationships among spatial objects. Therefore, in a given query, if two spatial objects are in (or intersects with) the same spatial-enforcement region, then the query specifies the user-chosen set of spatial relationships between the two spatial objects.

3. Our choice for the spatio-enforcement region object is a shaded or slanted rectangle. Domain scientists can choose a different object with a different color or shape from the class of spatial-enforcement region objects of the query specification model. Also note that, for each spatial enforcement region, users are allowed to choose the spatial relationships that they want VISUAL to enforce. This is done to eliminate the exhaustive specification of all spatial relationships.

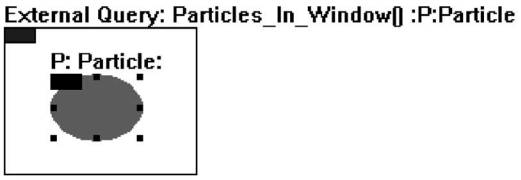


Fig. 4. Collection membership within a query object.

Example 3.6. Assume that the user wants to represent the spatial containment, spatial intersection, and spatial disjointness relationships among spatial objects frame **F**, particle **P**, and window **w**. Assume that annotations **A** and **A′** and experiment **E** are nonspatial objects. The graphical arrangement in Fig. 5 states that **P** is spatially contained in **F**; **w** spatially intersects with **F**, but not with **P**; **F** is composed of **P** and **A**. Fig. 5 states that **E** is composed of **A′**.

Class-subclass relationship between two data model object classes is represented by a labeled directed edge to the class from its subclass, with the label "is-a."

Example 3.7. Fig. 5 represents the class-subclass relationship between the Frame class and the VersionFrame class.

Example 3.8. The query in Fig. 6 is an example that elaborates on spatial and composition relationships. Particles that have passed through a window **W** (whose coordinates are specified in the condition box by its upper leftmost and lower rightmost points) in experiments **e** are located by this query.

The query of Fig. 6 asks for the "spatial containment" relationship between the particles and the window **W** and, hence, the ellipse icon object for **P** and the rectangle icon object for **W** are both enclosed within the spatial-enforcement region. Since the spatial relationship between frame **F** and range **W** is not explicitly specified in the query object, the spatial-enforcement region does not enclose or intersect with the iconized object for **F**; therefore, the spatial relationship between **F** and **W** is not considered in the query. The fact that the iconized object for **F** is inside the iconized object for **e** represents the composition hierarchy between **e** and **F**.

3.3.1 Incomplete Path Expressions and Nested Object Flattening

Fig. 7 shows the composition hierarchy of the application domain. Experiments contain Grids and Grids in turn contain Frames. Frames are composed of Clusters or Particles. Finally, Clusters contain Particles. Fig. 7 is an example for incomplete path expressions, "Frame **f** is contained in Experiment **e**," although, according to the composition hierarchy, Frames should be contained in Grids. Since there is no ambiguity about the path described in Fig. 7, specification of Grid between Experiment **e** and Frames **f** is omitted in the query to decrease visual

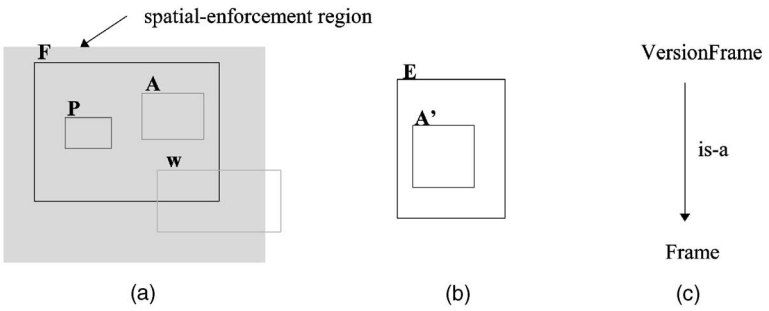


Fig. 5. Semantic relationships. (a) Spatial-enforcement region and composition hierarchy. (b) Composition hierarchy. (c) Class-subclass hierarchy.

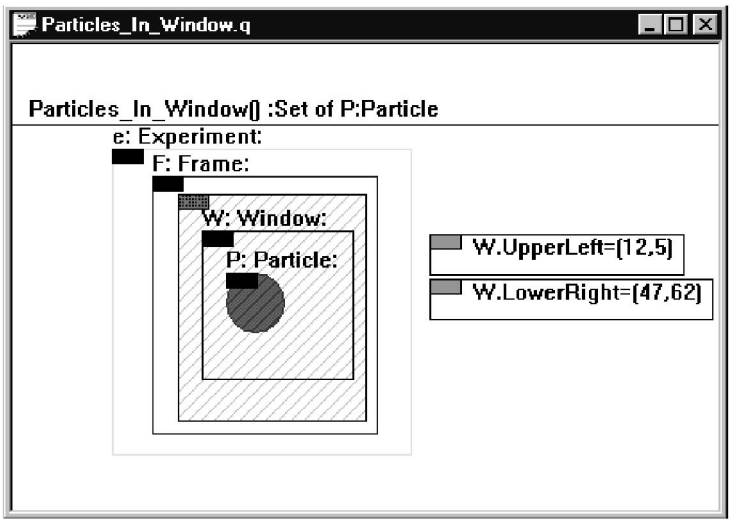


Fig. 6. Particles_In_Window.

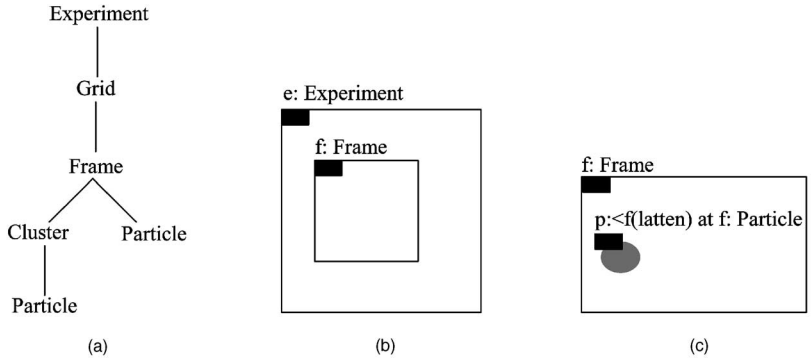


Fig. 7. Incomplete path expression and *nested object flatten* operator. (a) Composition hierarchy. (b) Incomplete path expression. (c) *Nested object flatten* operator.

complexity. Fig. 7 specifies, "Frame **f** is inside a Grid of Experiment **e**." Similarly, in Fig. 6, the semantic relation between Frame **F** and Experiment **e** contains an incomplete path expression.

Incomplete path expressions may not be used between domain objects when there is an ambiguity in the path that is described from the composition hierarchy.

Example 3.9. In the query in Fig. 6, only noncluster particles that are in frame **F** and also in range **W** are included into the output. If the user wanted particles that are in a Frame regardless of the fact that the Particles are in a Cluster of a Frame or not, he would have to add a *nested object flattening* operator.

Example 3.10. Fig. 7 shows an example of nested object flattening. **p** : < **f** (latten) at **f** > : Particle describes all Particles that are nested at different nesting levels of a frame **f**. In other words, **p** is instantiated by all Particles inside Frame **f** regardless of the fact that Particles reside in Clusters or not. Nested object flattening, a redundant operation, is provided to simplify the visual representation of the queries.

3.4 Internal and External Query Objects

Within the object-oriented query specification model, a VISUAL query object is composed of other query objects, which serve as modularization tools. Query objects are parameterized and, thus, serve a purpose similar to

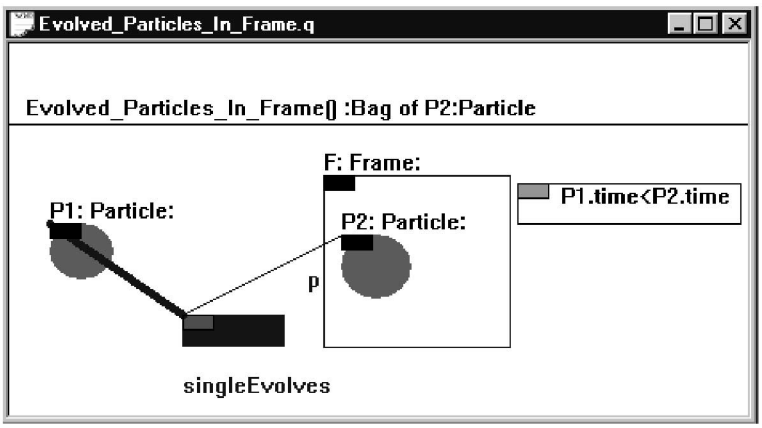


Fig. 8. Evolved_Particles_In_Frame.

procedures in programming languages and help to modularize query objects. Languages such as VQL [50], DOODLE [14], and GRAQULA [46] also allow parameterized queries, but not within an object-oriented query specification model environment. VISUAL query specification objects submitted to the system (such as queries, icons) can be saved and reused.

An external query is a VISUAL query that is defined independently (i.e., not defined within another query as an internal query). The uses of external queries are similar to procedures in programming languages: An external query is "called" from a query body through the specification of an iconized object labeled by the name of the external query. For the external query call, the query head of the external query must be specified along with its input and output parameters. Input parameters must be bounded by the variables used in the calling query.

Example 3.11. In Fig. 10, input parameter **E** of the external query Frames is defined within the main query Experiment_With_All_Frames_Having_Clusters. The output parameter of the external query Frames is the variable **F** of type Frame.

An internal query is a VISUAL query that is defined within another query (such as, say, the main query). The query parameters of an internal query only define the output parameters. Input parameters of an internal query are implicit and inherited from the query(ies), say Q, that contains the internal query: variables that exist both in query Q and in the internal query S constitute input parameters of the internal query S.

Example 3.12. In Fig. 12, there are no input parameters for the internal query Frame_Particles. The output parameter of the internal query Frame_Particles is specified by variable **P** of type Particle.

3.5 Methods

Icons and lines are used to express user-defined methods of VISUAL on screen. The base class of a method (the class that a method belongs to) is separated from the other parameters of the method by a thicker line.

Example 3.13. In Fig. 8, the use of method singleEvolves is shown. If the variable **P1** has evolved into the variable **P2**, then the method singleEvolves returns true. The query Evolved_Particles_In_Frame returns all the Frame Particles that have evolved without splitting.

Example 3.14. Query Split_Evolved_In_Cluster, in Fig. 9, returns all particles which has splitEvolved within the same cluster. Method splitEvolves takes two parameters, each of which are Particles.

3.6 A Tour in VISUAL

Users specify VISUAL queries using the interpretation semantics that implies a subquery evaluation for every different instantiation of query variables (due to the hierarchically arranged window structure of VISUAL query objects). This means, for every instantiation of query object variables with the corresponding data model objects (or object components) in the database, the methods and queries referred to within the query body are evaluated and the conditions in the condition box are checked (aggregate functions and query objects referred within the condition boxes are evaluated) with the current instantiation. The outputs are then retrieved if all the conditions in the query are satisfied.

We have already mentioned that set comparison operators can be specified in VISUAL. Other languages (such as STBE and VQL) can also specify set comparison operators, but, in VISUAL, set comparison operators are directly implemented. We discuss the implementation details of set comparison operators later in Section 5.

Example 3.15. The queries in Fig. 10 illustrate the use of set comparison operators and external queries for expressing the universal quantifier operator. The main query Experiment_With_All_Frames_Having_Clusters (Fig. 10) locates experiments that have at least one cluster in all of its frames. For a given instantiation **e** of experiment **E**, the query Frames (Fig. 10) returns a set of frames **f** such that **f** is in **e**. For a given instantiation **e** of experiment **E**, Frames_With_Clusters (Fig. 10) returns a set of frames **f** such that **f** is in **e** and **f** has at least one cluster. The condition with set equality on the output of external query objects in the condition box assures that, for each experiment **e** in the output of the main query, the set of all frames is the same set as the set of frames having at least one cluster.

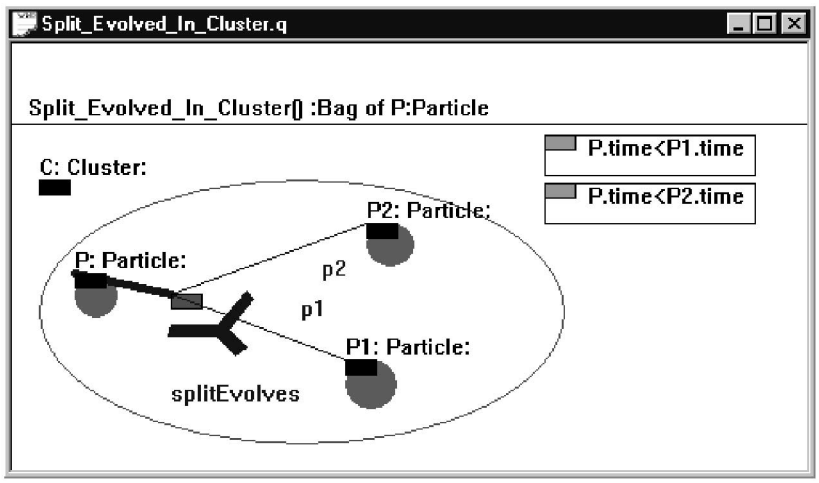


Fig. 9. Split_Evolved_In_Cluster.

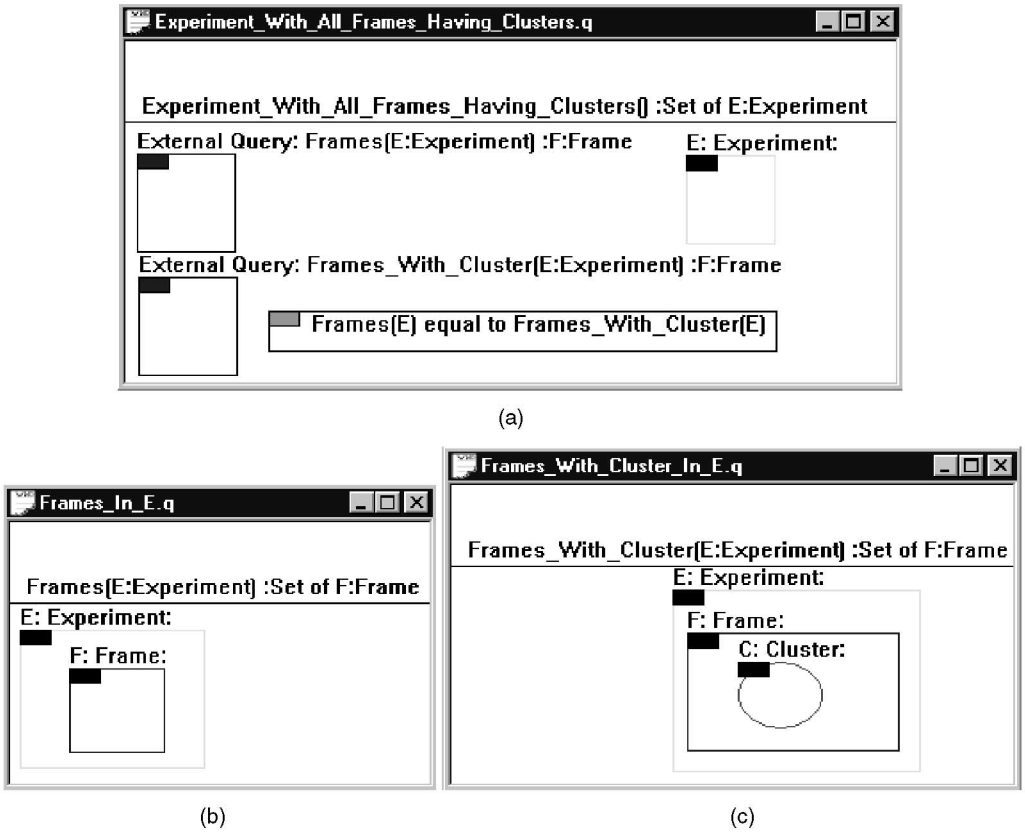


Fig. 10. A query with multiple subqueries. (a) Experiment_With_All_Frames_Having_Clusters. (b) Frames. (c) Frames_With_Cluster.

Example 3.16. The main query in Fig. 11 lists for a given experiment e the frames of e that do not have any clusters. VISUAL restricts the free use of negation by using set comparison operators and aggregate functions. The query in Fig. 11 expresses the negation operator via the set membership operator and the external query Frames_With_Cluster shown in Fig. 10.

In VISUAL, user-defined methods, aggregate functions and set comparison operators are treated uniformly during query processing. The following example illustrates the use of aggregate functions in VISUAL. Implementation of aggregate functions is discussed in Section 5.

Example 3.17. The query shown in Fig. 12 finds frames with more than five particles. This query also features an internal query called (Frame_Particles) inside the query body of Frames_With_More_Than_5_Particles. An internal query is defined with its own header and body. An internal query Q-IN called from a query named Q-OUT can refer to constants or variables of Q-OUT with the interpretation semantics that each variable of Q-OUT referred to from the body of Q-IN behaves as a constant inside Q-IN. For example, for a given instantiation, say f,

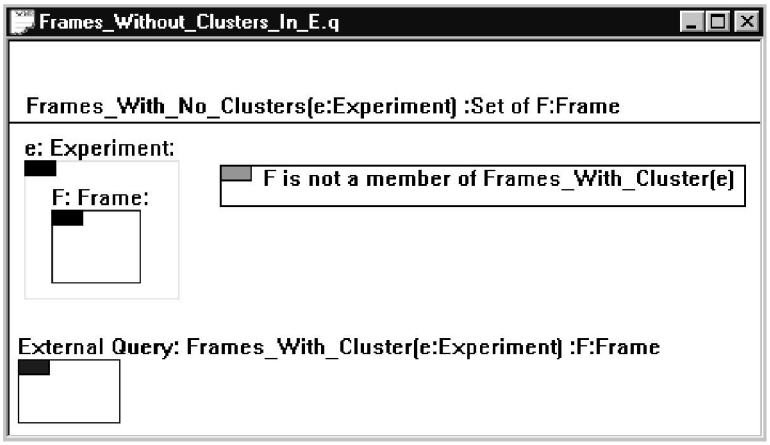
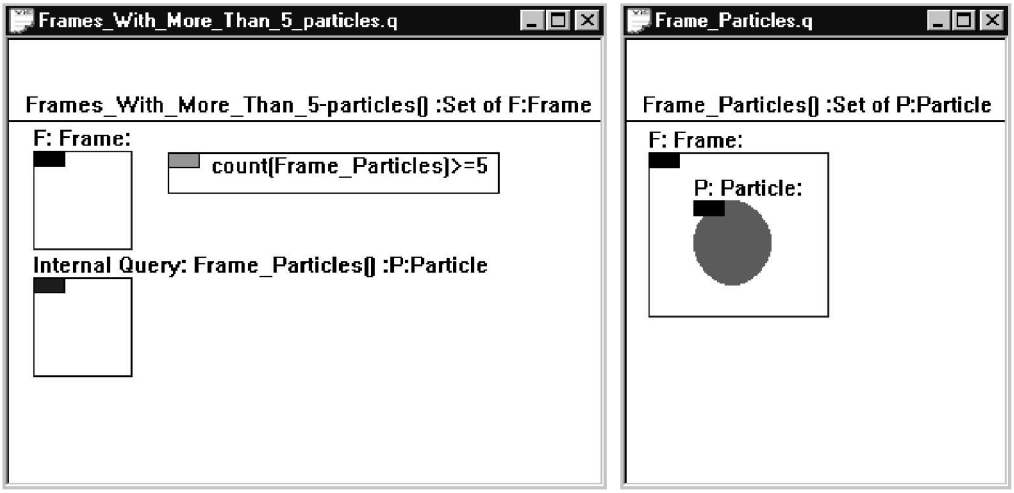


Fig. 11. Frames_With_No_Clusters.



(a) (b)

Fig. 12. (a) Frames_With_More_Than_5_Particles Figure. (b) Frame_Particle.

of frame variable **F**, the evaluation of Frame_Particles produces the particles of **f**. Please note that a call to an internal query (e.g., Frame_Particles) does not explicitly specify input parameters since the internal query object is "internal" to the main query. Instead, the definition of an internal query specifies its output parameters in its query header and the input parameters are those variables and constants that appear in the query that contain the internal query.

4 Advantages of the Object-Oriented Query Specification Model

Having a query specification model that is object-oriented allows

- The use of different icons and graphical objects for the same query specification construct or the same data model object, e.g., using a dotted circle for the spatial-enforcement region (Fig. 13) or using a circle for a particular object, and
- The use of VISUAL itself to query, add, delete icons, or graphical objects.

These capabilities bring flexibility to VISUAL at the user interface level. However, the notion of *query object* has far more reaching significance for VISUAL:

1. *Uniformity.* Query objects are accessed uniformly. Query objects are created, deleted, and referred to by using *object handles*, which is the same way an object-oriented operating system uses objects (such as a file object, server object, etc., in Windows NT[4]). Therefore, the object-oriented services in an ODBMS or operating system can be directly utilized.
2. *Query Sharing.* Query objects provide a paradigm for sharing queries between two or more query objects. Two query objects share another query object Q when they each open a handle to Q.
3. *A New Parallel/Distributed Query Processing Paradigm.* By requesting the services of two query objects independently, we provide a natural, user-specified multiprocessing paradigm for query processing. A computer with two processors can execute two query server objects simultaneously (if server objects

4. Our approach is a consistent extension of Windows NT Operating System [15], which is our prototype development platform for VISUAL.

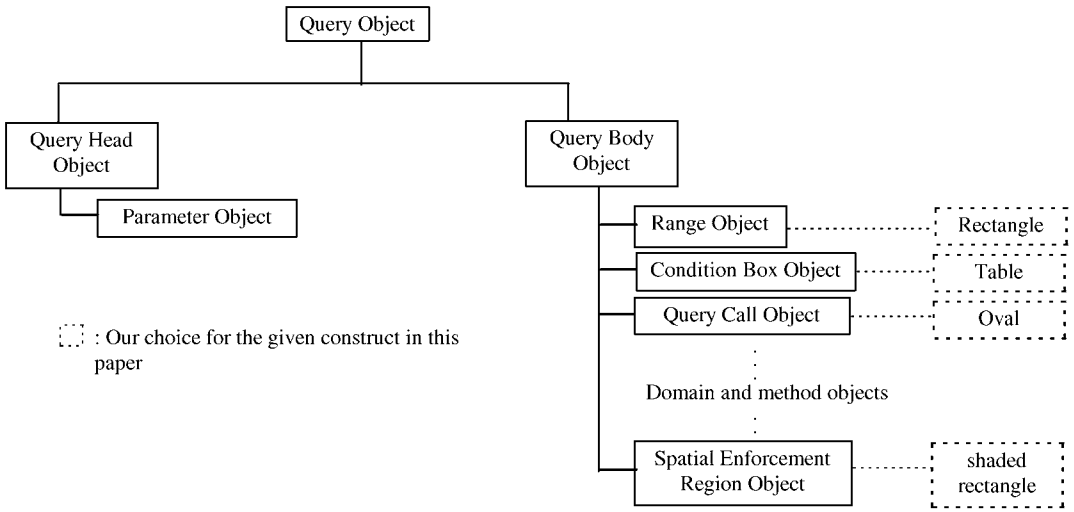


Fig. 13. Composition hierarchy for query object.

do not interact). This is a symmetric execution in the sense that query servers can run on any processor with no dedicated/static assignments of processors to query servers. In comparison, the only other parallel query processing paradigm in the literature, i.e., parallel query processing in the relational model, assumes that the user does not participate in the specification of the parallelism, that parallelism is hidden from users and that the DBMS makes parallel execution decisions.

4. *Security.* A natural use of query objects is in security. Services of a query object are available to a user (or another query object) if the user is authorized to use the query object. We can also use the "secure agent" concept, successfully used in operating systems (such as Unix[5]): Protected data in the database can be manipulated by a *trustworthy* query object (written and debugged by trustworthy users) and other nontrustworthy queries can only request the services of the trustworthy query object, i.e., they receive the output, which is aggregated and, thus, not protected.
5. *Synchronization Opportunities.* Services of two or more query objects can be requested in a synchronized manner. For example, in a multimedia database, two query objects, one with a video output and another with an audio, may return their output to another query object in a synchronized manner for the final output. This synchronization is possible because we provide different services from different computation units. Synchronization *within* a service by a single query object can also be specified.
6. *Time-Constrained Query Processing.* Services of a query object can be requested with time deadlines [39], [25] or with a *rate of delivery guarantees* (as in TIOGA [47]). The techniques we developed in [39], [25] are directly implementable in VISUAL.
7. *Schema Querying.* Query objects and query parts are under the same object hierarchy that makes schema querying equivalent to application domain object querying (Fig. 14).
8. *Easy Change of Application Domain.* Application domain dependent classes are at the lowest levels of the object hierarchy (dotted classes in Fig. 14); a new application domain can be defined for VISUAL by simply replacing these classes by new ones.

5. UNIX does this by the *effective userid* concept and the setuid bit.

5 VISUAL TO OQL

OQL [38] is a declarative language for querying and updating objects in ODBMS. OQL is based on SQL and is a standard for ODBMSs. The mapping from VISUAL to OQL provided below enables VISUAL to be used as a front-end to any ODBMS that supports OQL. This approach provides portability of VISUAL between different platforms.

OQL provides declarative access to entry points (objects) in the database. Methods can be invoked inside OQL statements and these methods themselves may contain OQL statements. OQL uses high-level primitives to deal with sets and other collection constructs (i.e., lists, arrays, and bags). Path traversal is supported through the use of the operators "." and "->." Any level of nesting of these expressions is supported except recursive queries. However, a query may call a programming language function which may in turn recursively make query or function calls. For the mapping below, we have used OQL release 1.1. The complete algorithm for mapping VISUAL to OQL is implemented and functional in the current VISUAL prototype.

5.1 Mappings for the Building Blocks of VISUAL

5.1.1 Queries and Query Parameters

There are three types of queries in VISUAL:

- main query object,
- internal query object, and
- external query object.

The differences between these three types of queries mainly lie within the different parameter passing mechanisms.

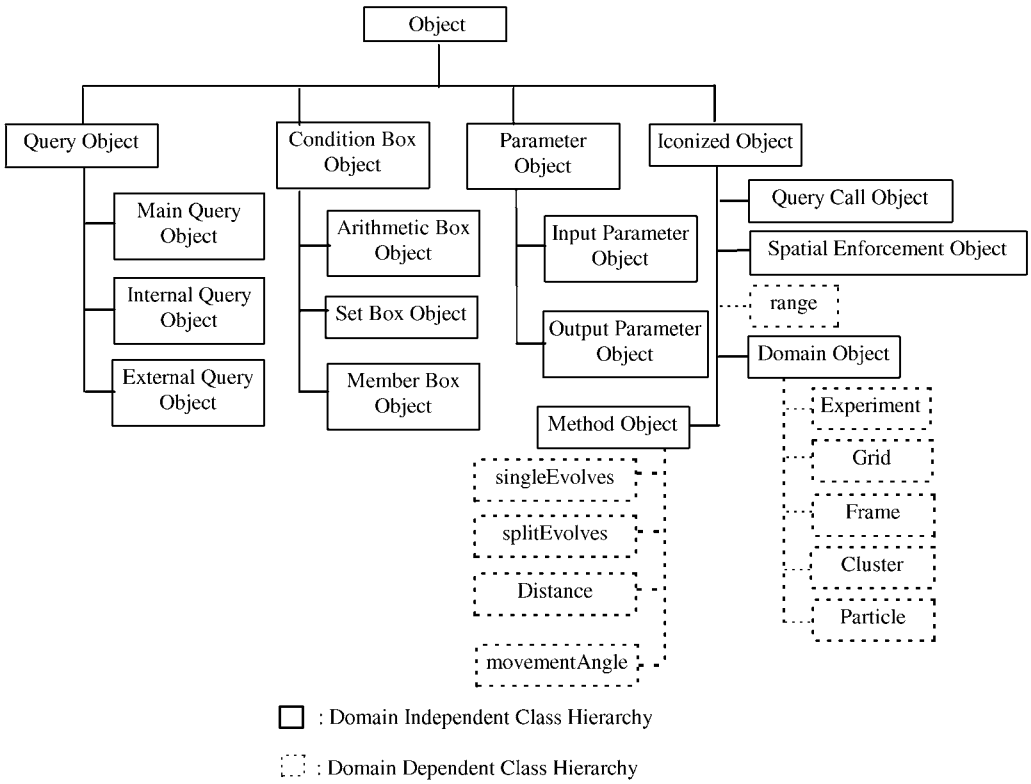


Fig. 14. Class hierarchies.

A. **Main Query Object.** Main query object is the entry point to the user query. Output parameters of the main query object define the output of the query. Domains of input parameters of internal and external queries referred from the main query are defined inside this query. Since there are no explicit parameter-passing techniques in OQL, input parameters for the referred external queries in a VISUAL query must be constructed as newly created objects (mutable objects in OQL terminology) inside the main query. This is not required for internal queries since a variable used in the main query will be a bound variable inside the internal query because of the scoping rules for variables in OQL. The scope of a variable in OQL is the immediate expression that encloses the variable and all of the enclosed expressions inside it.

Similar to input parameters of external queries, input parameters of the main query must also be constructed as newly created of objects. An OQL query consists of a set of query definition expressions followed by an expression. Since the main query must map to the expression at the end, input parameters for the main query must be defined in the query definition expressions.

B. **External and Internal Queries.** As explained before, input parameters of external queries must be constructed as object creations inside the main query. This is not allowed in OQL's syntax (BNF). Therefore, all external queries must be converted into internal queries before a mapping from a VISUAL query to an OQL query is performed. The concept of external query is important for VISUAL, as it allows queries to be stored either as definitions or as results (storing the query output for faster access for the next time the query is requested). Storing as a definition helps users to construct complex queries easily on top of previously defined queries. Storing as a result enables more efficient query processing. Since both of these considerations are not important for our mapping from VISUAL to OQL, we see no harm in converting external queries into internal queries.

Input parameters of internal queries are simulated in OQL by variables that are bound at the query that encapsulates the internal query. Output variables, on the other hand, are simulated by members of the object created by the encapsulated query in OQL.

C. **Return Types.** VISUAL allows queries with different return types. A query can return a set, a bag, an array, or a list of objects. The same structures are allowed in OQL (actually OQL allows arrays as well). The semantics of these return types and the operations between these types such as union, difference, etc., are exactly the same in both languages. For simplicity, for the rest of the mapping, we will only use the return type "set."

D. **Condition Boxes.** Condition boxes are crucial to expressiveness of VISUAL. Every propositional calculus formula can be specified in a condition box. Different kinds of condition boxes are mapped to different structures of OQL.

- **Arithmetic Expressions.** Arithmetic expressions in condition boxes are mapped into the **where** portion of the **select from where** clause. These conditions may contain arithmetic comparison operators as well as arithmetic operations. There is no unary arithmetic operator in VISUAL, so the mapping is valid for only binary arithmetic operators.
- **Member Expressions.** Member expressions in VISUAL are defined for both positive membership (i.e., $\in$) and negative membership (i.e., $\notin$). The negative membership expression allows VISUAL to use negation in queries. Positive membership operator in VISUAL is mapped to the membership-testing operator (i.e., **in**) in OQL. Negative membership operator in VISUAL is mapped to a combination of the unary operator **not** and membership testing operator in OQL.
- **Set Expression.** Set expressions in VISUAL can be mapped in more than one way to OQL's constructs. A set expression in VISUAL contains one of the operators $\supset$, $\supseteq$, $\subset$, $\subseteq$, and $\equiv$ (for simplicity, we will show only the mappings for $\subset$, $\subseteq$, and $\equiv$).

One possible way is to map the VISUAL set operators using the binary set operators of OQL (i.e., **intersect**, **union**, and **except**). In this case, $\mathbf{A} \subset \mathbf{B}$ (where $\mathbf{A}$ and $\mathbf{B}$ are sets) can be mapped to $(((\mathbf{A} - \mathbf{B}) = \{\})$ **and** $((\mathbf{B} - \mathbf{A})$!= {})); $\mathbf{A} \subseteq \mathbf{B}$ (where $\mathbf{A}$ and $\mathbf{B}$ are sets) can be mapped to $((\mathbf{A} - \mathbf{B}) = \{\})$; and $\mathbf{A} \equiv \mathbf{B}$ (where $\mathbf{A}$ and $\mathbf{B}$ are sets) can be mapped to $(((\mathbf{A} - \mathbf{B}) = \{\})$ **and** $((\mathbf{B} - \mathbf{A}) = \{\}))$.

An alternative way to map VISUAL set operators to OQL constructs is using the universal and existential quantification. Since universal and existential quantification in VISUAL are expressed through set operators, we prefer this mapping. In this case, $\mathbf{A} \subset \mathbf{B}$ (where $\mathbf{A}$ and $\mathbf{B}$ are sets and $\mathbf{x}$ a variable) can be mapped to ((**for all** $\mathbf{x}$ **in** $\mathbf{A}$: ($\mathbf{x}$ **in** $\mathbf{B}$)) **and** (**exist** $\mathbf{x}$ **in** $\mathbf{B}$: (**not**($\mathbf{x}$ **in** $\mathbf{A}$)))); $\mathbf{A} \subseteq \mathbf{B}$ (where $\mathbf{A}$ and $\mathbf{B}$ are sets and $\mathbf{x}$ a variable) can be mapped to (**for all** $\mathbf{x}$ **in** $\mathbf{A}$: ($\mathbf{x}$ **in** $\mathbf{B}$)); and $\mathbf{A} \equiv \mathbf{B}$ (where $\mathbf{A}$ and $\mathbf{B}$ are sets and $\mathbf{x}$ a variable) can be mapped to ((**for all** $\mathbf{x}$ **in** $\mathbf{A}$: ($\mathbf{x}$ **in** $\mathbf{B}$)) **and** (**for all** $\mathbf{x}$ **in** $\mathbf{B}$: ($\mathbf{x}$ **in** $\mathbf{A}$))).

E. **Graphical Objects.** Graphical objects in VISUAL are used to express a set of relationships between objects and/or to determine the domains of variables (i.e., possible object bindings for variables). The relationships between graphical objects may be domain dependent. In our application domain, the relationships that are represented are the spatial relationships between objects. These spatial relationships are already stored in attributes of objects themselves. The spatial relationships can be obtained by method applications, as well. If method applications are used, then these relationships should be handled as methods, which are explained in the next section.

There are two types of graphical objects. One is an object from our application domain (*domain graphical object*) and the second one is a reference to another VISUAL query (*query graphical object*). Below, we discuss various nestings of graphical objects in VISUAL and their OQL translations.

- **Domain Graphical Object Inside a Domain Graphical Object** (Composition Hierarchy). This means that one of the attributes of the outer domain graphical object holds a pointer to the inside domain graphical object, meaning that the inner object spatially lies inside the outer object in the real world. A pointer in the inner object to the outer object could have supplied this relationship as well. Or, the designer of the domain might decide to hold both of the pointers. In any case, this relationship is mapped to one of the conjuncts of the **where** portion of the **select from where** clause of the query and domains of the variables in the graphical objects must be defined in the **from** portion of the same clause (for example, if **F** is inside **e** then **select...from...**e:Experiment, **F**:Frame... where **F**.experimentIn( ) = **e**).
- **Domain Graphical Object Inside a Query Graphical Object.** Results of a query (internal or external) are accessed through spatial placement of icons on the screen. The type of the domain graphical objects inside the query graphical object must match the output type of the query. Mapping of this structure in VISUAL requires a **select from where** clause in OQL. The **from** and **where** portions implement the query body, while the **select** portion contains the variable names in the domain graphical icons (allowing the query output to be mapped to domain graphical objects). Notice that the number of domain graphical object groups determine the number of **select from where** clauses.
- **Method Icons, Spatial Windows, Spatial Enforcement Icons.** Method icons represent relationships that are not stored directly in the database. These relationships must be mapped to Boolean functions in the **where** portion of the **select from where** clause of the main query. Since signatures of methods are diverse, signatures of functions representing the methods are diverse as well. Domain graphical objects and other user-entered structures constitute input for methods in VISUAL. These inputs are mapped to the parameters of functions that represent methods in OQL.

Spatial windows have meaning only when they are inside spatial enforcement icons. Any spatial relationship (i.e., inclusion, intersect, or disjunction) between a domain graphical object and a spatial window (which is in a spatial enforcement icon) in VISUAL can be mapped to a function in OQL. We prefer to map these relationships to a function rather than comparing the coordinates of the objects in

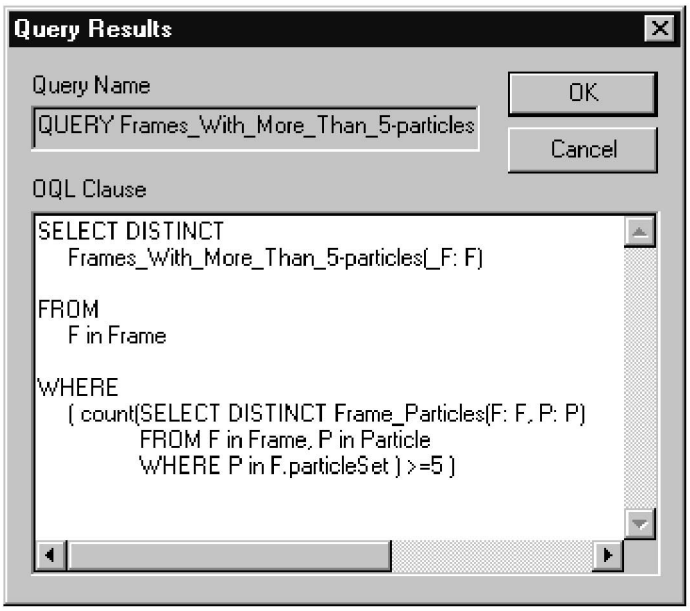


Fig. 15. Frames_With_More_Than_5_Particles (OQI translation by VISUAL).

question in the **where** portion of a **select from where** clause mainly because, in an arbitrary application, spatial objects can be more complex than just spatial windows (in which case, the relationship might require more than simply comparing the coordinates—we leave this detail to be handled inside the function).

Example 5.1. Fig. 15 gives the OQL translation of the VISUAL query Frames_With _More_Than_5_Particles given in Fig. 12.

6 VISUAL TO COMPLEX-ALGEBRA

In this section, we explain parse tree creation for VISUAL queries in Complex-Algebra. First, we discuss how the differences between internal and external queries are handled. Next, we discuss an operator that allows us to flatten composition hierarchies and another operator that uniformly manages methods, aggregate, and set operations. Balkir [3] contains the complete algorithm for creating parse trees from VISUAL queries and its illustration with examples.

Interpretation semantics of VISUAL implies subquery evaluation for every different instantiation of query variables (similar to tuple-by-tuple subquery evaluation strategy in STBE) due to the hierarchically arranged window structure of VISUAL queries. This means, for every instantiation of query variables with the corresponding objects (or object components) in the database, the methods and the queries referred within the query body are evaluated and the conditions in the condition box are checked (aggregate functions and queries referred within the condition box are evaluated). The output objects are then retrieved if all the conditions in the query are satisfied.

Query evaluation with respect to interpretation semantics of VISUAL employs a top-down query evaluation and may not be cost (both time and memory) efficient. To overcome this problem, we translate the query into a complex algebra that uses a bottom-up approach as defined below. Naturally, the query object output of the bottom-up approach and the interpretation semantics for a query should be equivalent.

Below, we discuss the translation of any VISUAL query to a complex-algebra expression. The complete algorithm that creates a complex-algebra parse tree for a given VISUAL query (the query must be in nonrecursive VISUAL) is in [3]. The algorithm consists of three phases: 1) translating the query head, 2) translating the query body, and 3) attaching subqueries. The first two phases of the algorithm must be applied to each query before the third phase is applied (i.e., attachment of subqueries).

Before a VISUAL main query is submitted to the algorithm, recursion in the main query object and other query objects that it refers to are checked for safe query evaluation. A query (directed) graph is constructed using the rules below:

- For every query object Q (Q can be an external, an internal, or the main query object), create a node.
- For every reference of Q′ in Q, add the edge (Q, Q′) into the graph.

Topological sort is applied on this graph to determine the existence of a cycle. The reverse order of the result of the topological sort will determine the evaluation order of the query objects (attachment order of the parse trees for queries into a single tree).

The query evaluation technique we use is a bottom-up approach. For that reason, each subquery output includes the input parameter values as well as the output parameter values. In other words, each output object has members corresponding to input and output parameters. This is consistent with the interpretation semantics of VISUAL.

In VISUAL, the parameter list of an internal query object does not contain any of the input parameters inherited from

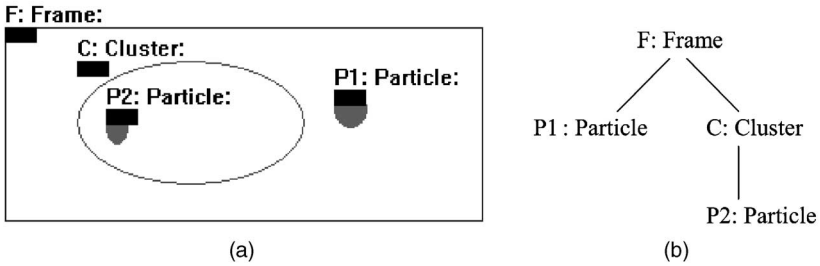


Fig. 16. Composition hierarchy in VISUAL. (a) Iconic composition hierarchy. (b) Composition hierarchy tree.

the calling query. However, we modify this during the parse tree creation so that all the input parameters of an internal query object Q are also in its parameter list. Input parameters of Q inherited from the query objects that contain Q are named as *implicit input parameters*.

6.1 Composition Hierarchy and Composition Flatten Operator

For every iconic composition hierarchy representation in the query body, we create the corresponding composition hierarchy. The nodes of the hierarchy are the graphical icons in the iconic representation. Composition hierarchy is constructed iteratively by attaching the iconized objects in the iconic representation to the hierarchy. Hierarchy construction starts at the outermost icon object, which becomes the root of the hierarchy. Next, the nodes for icons that are immediately contained in the root icon are attached as children. This process continues iteratively for every iconized object in the iconic representation. Leaves of the hierarchy are the graphical icons at the innermost level of the iconic representation.

Once the composition hierarchy is obtained, we create a parse tree node $\mathbf{p}$ from each branch $\mathbf{B}$ (a root-to-leaf path) by a composition flatten operator. The composition flatten operator basically obtains a collection of objects by flattening the object hierarchy recursively. The composition flatten operator starts with a root object $\mathbf{r_i}$ instance (where $\mathbf{r_i} \in \{\mathbf{r_1}, \mathbf{r_2}, \ldots, \mathbf{r_m}\}$) and dereferences (allows access to the encapsulated object member variables and functions) $\mathbf{r_i}$. As a result of the composition flatten operator application, we obtain a collection of objects $\{\mathbf{o_1}, \mathbf{o_2}, \ldots, \mathbf{o_n}\}$ represented by the node $\mathbf{p}$ for branch $\mathbf{B}$. A set of objects of the form $(\mathbf{r_i}, \mathbf{o_j})$ is obtained by coupling each $\mathbf{o_j} \in \{\mathbf{o_1}, \mathbf{o_2}, \ldots, \mathbf{o_n}\}$ with $\mathbf{r_i}$. This process is repeated recursively until the composition hierarchy is flattened completely (i.e., each node on the branch is traversed) for each instance of the root. If constant objects appear at any node in the path, only the oids of the constant objects are used in the coupling.

The following example illustrates the construction of a composition hierarchy and shows the output of the composition flatten operator on this hierarchy.

Example 6.1. Assume that query Q contains the definitions in Fig. 16 in its body. The composition hierarchy of the iconic composition hierarchy representation is also shown in the figure.

Now let us apply the composition flatten operator to each path of the composition hierarchy in Fig. 16. Let $\mathbf{f}$ be the only instantiation of frame $\mathbf{F}$ with a cluster set $\{\mathbf{c_1}, \mathbf{c_2}\}$, and the set $\{\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}\}$ contain the instantiations of $\mathbf{P1}$ and $\mathbf{c_1}$ and $\mathbf{c_2}$ are clusters with particle sets $\{\mathbf{p_4}, \mathbf{p_5}\}$ and $\{\mathbf{p_6}, \mathbf{p_7}\}$, respectively. Then,

$$\text{composition-flatten}(\text{path } \mathbf{F-C-P2}) = \{(\mathbf{f}, \mathbf{c_1}, \mathbf{p_4}), (\mathbf{f}, \mathbf{c_1}, \mathbf{p_5}), (\mathbf{f}, \mathbf{c_2}, \mathbf{p_6}), (\mathbf{f}, \mathbf{c_2}, \mathbf{p_7})\}$$

and

$$\text{composition-flatten}(\text{path}\mathbf{F-P1}) = \{(\mathbf{f}, \mathbf{p_1}), (\mathbf{f}, \mathbf{p_2}), (\mathbf{f}, \mathbf{p_3})\}.$$

Note that we have specified the nested object flattening in order to provide a simplified interface to all objects of the same type in a composition hierarchy that are nested at different levels with a simple interface. Composition hierarchy creation in the existence of the nested object flattening operator is similar to the previous general case, in that the nested object flattening operator is applied first and then the composition-flatten operator is used. A composition hierarchy in this specific case consists of all possible paths in the iconic composition hierarchy representation between the specified graphical icons. Note that new variable names have to be created for the intermediate nodes (nodes that were not referred in the query, but are necessary for unambiguous processing) of the composition hierarchy which can be projected out (since we will not be using these newly created variable names) once the result of the composition flatten operator is obtained.

Example 6.2. Fig. 17 shows a query body with nested object flattening operator and the corresponding composition hierarchy. Intermediate nodes G:Grid, F:Frame, and C:Cluster of the composition hierarchy are not in the query. The root E:Experiment and all the leaves P:Particle are the two objects specified by the nested object flattening operator.

6.2 Merging Query Objects for Complex-Algebra Execution Semantics

In VISUAL, a query, say $\mathbf{A}$, uses the services of another query, say $\mathbf{B}$, in only three different computations, namely, method invocations (this includes direct query calls using icons), aggregate function computations, and set operator evaluations. We treat all three computations uniformly, and, if a bottom-up query processing approach is desirable, merging of $\mathbf{B}$ into $\mathbf{A}$ is done through a single technique. This uniformity is directly due to the object-oriented query specification of VISUAL. In the rest of the section, we discuss query object merging in VISUAL. We will discuss query object merging with respect to a complex algebra [17]. However, for the sake of comprehension, the reader can

Fig. 17. Nested object flattening.

simply view the operators in a way similar to relational algebra operators—except objects replace tuples.

All operations that involve methods, aggregate functions, and set operators have the following common properties: They all

- have an input collection,
- are an application of a method (function) to members of the input collection, and
- produce an output collection containing the results of the method application to the members of the input collection.

This common behavior motivates us to device an operator, called Method Applier, which handles methods, aggregate functions, and set operations by performing the common actions defined above. The strength of this approach lies in its simplicity and uniformity. Optimization strategies for VISUAL queries are greatly simplified by this uniformity. Without Method Applier, structures of the complex algebra expression representing methods, aggregate functions, and set operations are different and, thus, different optimization strategies need to be devised to consider these behavioral differences separately. On the other hand, when Method Applier is used, the structures of the complex algebra expression for the three types of operations are similar which, in turn, enables the use of a single optimization strategy for methods, aggregate operations, and set operations.

In this section, we discuss the translation of VISUAL queries to complex algebra expressions.

6.2.1 Method Applier

For the sake of simplicity, we restrict ourselves to queries that do not include sequence or bag operations. Also, in VISUAL, the parameter list of an internal query does not contain any of the input parameters inherited from the calling query. However, we modify this during the complex algebra expression creation so that all the input parameters of an internal query **Q** are also in its parameter list. Input parameters of **Q** inherited from the queries that contain **Q** are called *implicit input parameters*.

One can think of the method applier as a black box with some inputs and an output. Inputs are 1) a specific method, an aggregate function, or a set operation (we refer to this input as *InputFunction*), 2) the domain of the related function (we refer to this input as *InputSet*), and 3) generalized projection parameters (we refer to this input as *GeneralizedProjection*). Generalized projection (defined as in [49]) allows creating duplicates of the input attributes while regular projection does not. This is required in VISUAL since we might decide to use the same object for different inputs of a function (such as, distance between a particle and itself is 0). Output is a collection of objects that contain an object from the input set and the output of the InputFunction for this object.

InputSet is a set that contains the domain of the InputFunction. InputSet is created by a natural join among the sets, each of which contain a domain for a parameter of InputFunction. We prefer a natural join (to equi-join) since it eliminates unnecessary duplicates. Parameters of the natural join are determined by the variable names used in the query. Elements of the InputSet may contain attributes that are not parameters of the InputFunction. A method or an aggregate function may be operating on only a portion of the input of the method applier. Thus, we identify the domain of the input function by GeneralizedProjection (list for the generalized projection). The *i*th element in GeneralizedProjection and the *i*th parameter of the input function are related and used together.

Method applier finds the output of InputFunction by calling the specified function execution for each element in InputSet. Note that, by using generalized projection, only those parts of each element that are needed for function evaluation are passed to the function. Parameters of the generalized projection are obtained from the parameter list of InputFunction. The output of the method applier has the form **new_element**$^\circ$**function_output**, where $^\circ$ denotes concatenation. new_element is the element created in the natural join and function_output is the output of InputFunction.

The rest of the section illustrates the use of method applier for methods, aggregate operations, and set operations.

6.2.2 Use of Method Applier for Methods

Method applier is used in conjunction with a natural join, which is denoted by $\otimes$ and a selection for methods as in Fig. 18. Method Applier returns a concatenation of each element in InputSet and the output of the method. The selection predicate **SP** is defined by the method signature.

Example 6.3. This example illustrates how the method applier is used for methods by using the query Evolved_Particles_In_Frame defined in Fig. 8. The method singleEvolves+ is used to find the evaluation sequence of a particle. Fig. 19 shows the method applier with its inputs and output for the method singleEvolves+(p,P). Each evaluation of singleEvolves+ gets two particles (one is always the same particle **p**, a constant, and the other one is the current instantiation of variable **P**) as input parameters and returns true if

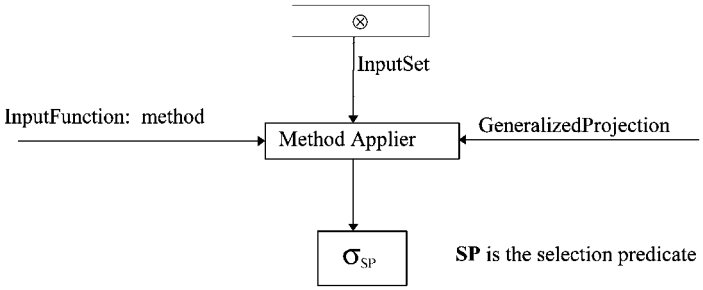


Fig. 18. Use of method applier for methods.

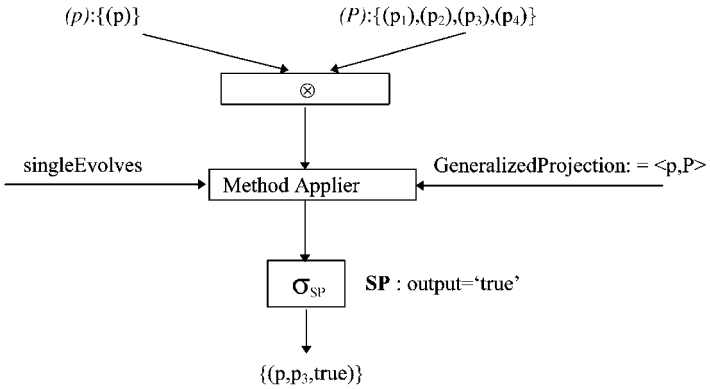


Fig. 19. Parse tree generation for the single-evolves method.

the first particle ($\mathbf{p}$) evolves into the second particle (the current instance of $\mathbf{P}$) in one or more steps, and false otherwise. Assume that ($\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$, $\mathbf{p_4}$) are instances of $\mathbf{P}$. Thus, InputSet is obtained by a natural join on the sets $\{(\mathbf{p})\}$ (since $\mathbf{p}$ is a constant) and $\{(\mathbf{p_1}), (\mathbf{p_2}), (\mathbf{p_3}), (\mathbf{p_4})\}$. These two sets contain the "domains" for the method arguments. Please note that the first and the second argument of the method get their values from the first attribute of the elements in $\{(\mathbf{p})\}$ and $\{(\mathbf{p_1}), (\mathbf{p_2}), (\mathbf{p_3}), (\mathbf{p_4})\}$, respectively; this is determined by

$$\text{GeneralizedProjection} =< \mathbf{p}, \mathbf{P} > .$$

Assume that $\mathbf{p}$ isevolved only into $\mathbf{p_3}$ in the next frame. Then, the output of the method applier is

$$\{(\mathbf{p}, \mathbf{p_1}, \text{false}), (\mathbf{p}, \mathbf{p_2}, \text{false}), (\mathbf{p}, \mathbf{p_3}, \text{true}), (\mathbf{p}, \mathbf{p_4}, \text{false})\}.$$

The selection predicate is defined as *method-output-attribute = true*. As a result of the selection, final output is $\{(\mathbf{p}, \mathbf{p_3}, \text{true})\}$.

Note that the method signature determines the selection predicate. For example, if we have a method call with a constant argument of the form *distance(10)*, the selection predicate will be set to *method-output-attribute = 10*. On the other hand, if we have a method call with an unbound argument of the form *distance(X)*, then the selection predicate will be defined as *select all* (which in effect means that selection is *not* applied).

6.2.3 Use of Method Applier for Aggregate Operations

For aggregate functions, a grouping construct, a projection, and a selection are used with the method applier, as shown in Fig. 20.

The reason behind the grouping construct is the bottom-up creation of the complex algebra expression. The subquery being referred to in the aggregate function call is evaluated for every instantiation of the subquery input parameters. The final result, hence, requires a grouping with respect to the input parameters of the subquery (e.g., $\text{count}(\text{Q}(\text{X}, \text{Y}))$, where Q is an external query with input parameters represented by the array X and output parameters represented by the array Y. With respect to the interpretation semantics of VISUAL, count should be called for every instantiation of X. This means that a grouping of the output of Q with respect to X is needed).

The grouping construct is in fact the Group-by-template construct [40], which ensures that all input groups are used as a template for output and all input groups are returned. This construct avoids the empty partitioning problem [40]. In VISUAL, queries which are referred from an aggregate function cause an empty partitioning problem, i.e., if the output of a subquery is empty then the output of, say, COUNT(subquery) is lost (rather than returning 0 which should be the case). Group-by-template takes the set to be grouped, the grouping arguments, and a template as input. In VISUAL, a template for a query contains all possible instantiations of the input parameters of the query. First, the query output is grouped with the grouping arguments. If an instance of the template does not exist in any of the elements in the grouped set (note that the grouping arguments and the template attributes are the same), this instance coupled with an empty set is added to the grouped set.

Note that the parameter the aggregate function is applied to (which is explicitly specified in VISUAL queries) may be

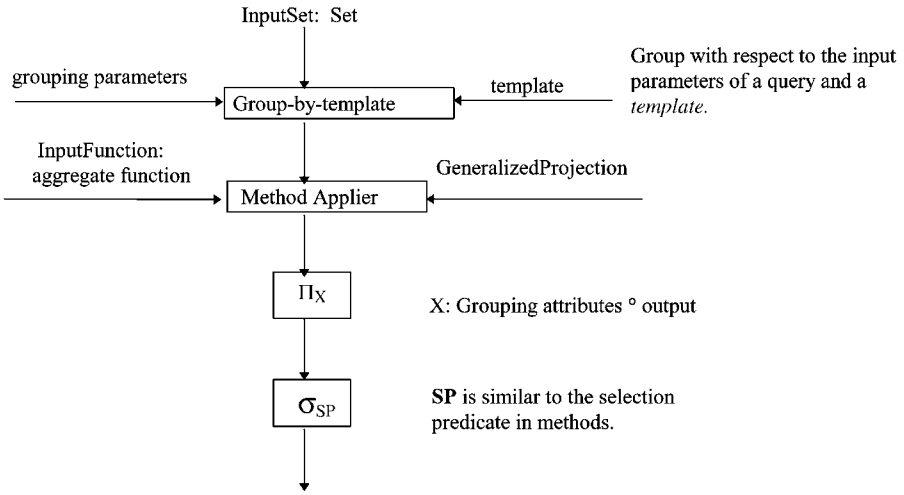


Fig. 20. Use of method applier for aggregate functions.

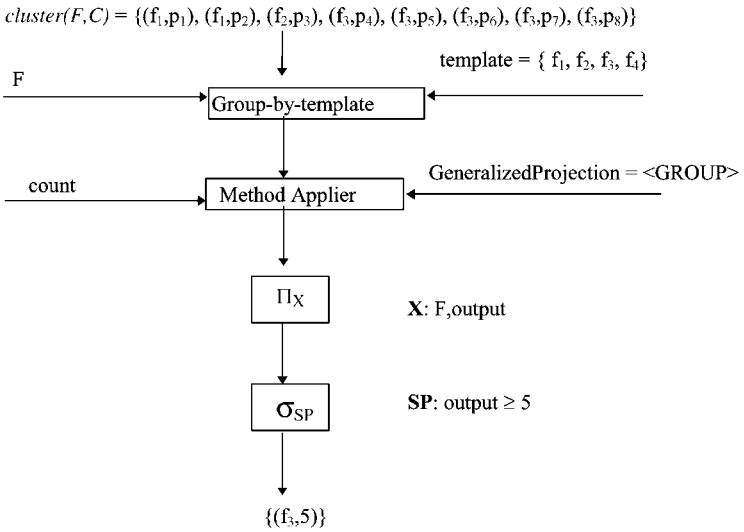


Fig. 21. Complex algebra expression for count.

a part of the group, but not necessarily the whole group (obtained by grouping the input set with respect to the input parameters). Consequently, the explicitly specified parameter should be reported to the method applier, so that the method applier passes this information to the aggregate function. This information is provided by a constant element in GeneralizedProjection. The constant shows the location of the explicitly specified parameter within the group. The method applier needs to be projected on the grouping attributes and the output of the aggregate function.

Example 6.4. Consider the aggregate function *count(*Frame_Particles)* in Fig. 12 $^{*}Frame\_Particles$ is an internal query object where F is an input parameter and P is an output parameter. Assuming the output of

$$^{*}Frame\_Particles\ is\ cluster(F, C) = \{(\mathrm{f}_1, \mathrm{p}_1), (\mathrm{f}_1, \mathrm{p}_2), (\mathrm{f}_2, \mathrm{p}_3), (\mathrm{f}_3, \mathrm{p}_4), (\mathrm{f}_3, \mathrm{p}_5), (\mathrm{f}_3, \mathrm{p}_6), (\mathrm{f}_3, \mathrm{p}_7), (\mathrm{f}_3, \mathrm{p}_8)\}$$

and the template is $\{\mathrm{f}_1, \mathrm{f}_2, \mathrm{f}_3, \mathrm{f}_4\}$. Fig. 21 shows the use of the method applier for this aggregate function call. GROUP refers to the attribute name created by the group-by-template operator for the grouped clusters.

6.2.4 Use of Method Applier for Set Operations

For set operations, method applier is used in combination with grouping and a natural join operation. The complex algebra expression creation for the set comparison operators is very similar to that of aggregate functions. Grouping arguments for set operations change with the definition of the input sets. Possible combinations that are allowed in VISUAL are shown in Table 1. Input parameters of external queries in a set operation must have the same variable names in the same order.

Since the query evaluation is bottom-up, query output will include the input parameters as well as the output parameters. In other words, each output object has members corresponding to input and output parameters. This is consistent with the interpretation semantics of VISUAL.

TABLE 1
Grouping Arguments

Input Set 1	Input Set 2	Grouping Parameters
constant set	internal query	Output of the internal query is grouped with respect to its input parameters;
internal query	internal query	Each query output is grouped with respect to its own input parameters
external query	external query	Output of each query is grouped with respect to the input parameters that are assumed to be of the same type in both queries

Natural join operation combines the grouped sets and creates InputSet for Method Applier. The inputs and the related operations needed for set operations are shown in Fig. 22.

The following example illustrates using the method applier and parse tree generation for a VISUAL query with set operation (i.e., set equality).

Example 6.5. Consider the query Experiments_With_All_Frames_Having_Clusters in Fig. 10 and the set equality Frames(E,F_1) $\equiv$ Frames_With_Cluster(E,F_2) in the query. Frames and Frames_With_Cluster are two external query calls both having the same input (E) and output (F_1, F_2), respectively. Frames returns all the frames in a given experiment, whereas Frames_-With_Cluster returns the frames that have at least one cluster in a given experiment. Fig. 23 shows the parse tree generation for the "$\equiv$" operation with the assumptions that Frames(E,F_1) = {(e_1,f_1), (e_1,f_2), (e_1,f_3), (e_2,f_4), (e_2,f_5), (e_3,f_6)}, Frames_ With_Cluster(E,F_2) = {(e_1,f_1), (e_1,f_2), (e_1,f_3), (e_2,f_4)} and the template for E from the calling query is {e_1, e_2, e_3}. The outputs of grouping Frames(E,F) and Frames_With_Cluster(E,F_1) with respect to E are (E,$GROUP_1$) = {(e_1,{f_1,f_2,f_3}), (e_2,{f_4,f_5}), (e_3,{f_6})} and (E,$GROUP_2$) = {(e_1,{f_1,f_2,f_3}), (e_2,{f_4})}, respectively. ($GROUP_1$ and $GROUP_2$ are the new attribute names created by the Group-by-template operator for the grouped attributes). The projection list is {*E,output*}, where E is the grouping argument of Frames and Frames_With_Cluster, and output is the function-output attribute. The selection predicate is (*output=true*).

7 Formal Properties for Nonrecursive Relational VISUAL

The underlying language for (nonrecursive) VISUAL is very similar to the relational calculus with sets (RC/S) [42]. Since we express it using rules (to add recursion), we call it D(atalog)-VISUAL. However, in this paper, which describes the current implementation, we have not dealt with recursion.

Please note that D-VISUAL does not have universal quantification, but provides the same expressive power using set comparison operators, as in RC/S [42]. It is shown that [42] one can replace the universal quantifier of relational calculus with set comparison operators, limit the use of the negation operator, and still have the same expressive power of relational calculus. This language is referred to as "relational calculus with sets" (RC/S) in [42] and its safety was also investigated in [42].

A D-VISUAL program uses built-in predicates of type

1. $X\theta_1 Y$ where $\theta_1 \in \{=, \neq, <, \leq, >, \geq\}$ and X and Y are either constants or variables,
   - X θ_2 S, where $\theta_2 \in \{\in, \notin\}$, X is a variable or a constant, and S is a set,
   - S_1 θ_3 S_2, where $\theta_3 \in \{\subset, \subseteq, \supset, \supseteq, \equiv\}$ and S_1 and S_2 are sets,

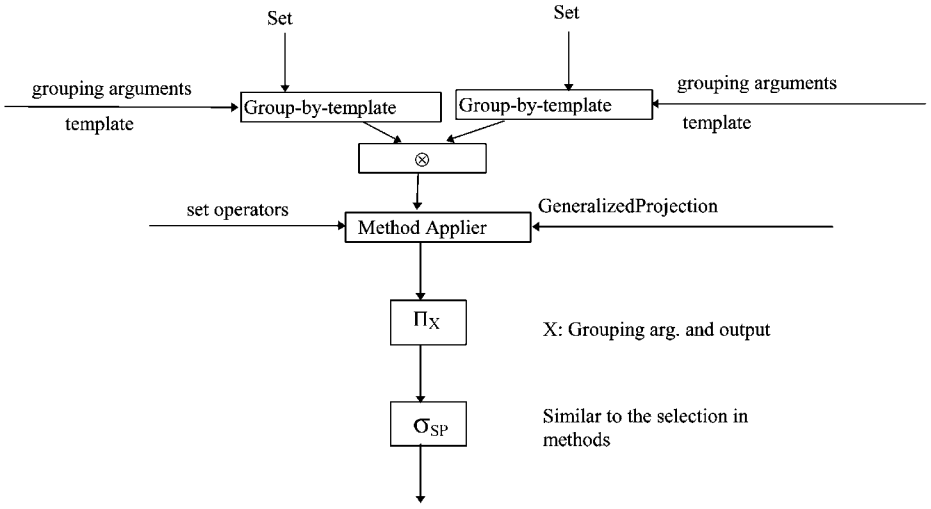


Fig. 22. Use of method applier for set operations.

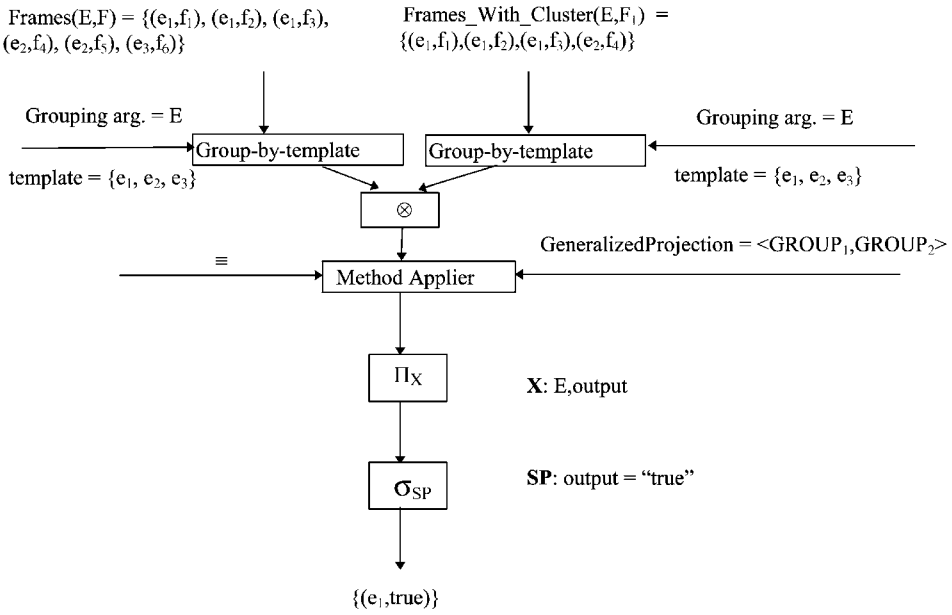


Fig. 23. Complex algebra expression for set equality.

2. $S_1\ \theta_4\ S_2 \equiv \Phi$, where $\theta_4 \in \{\cup, \cap, -\}$ and S_1 and S_2 are sets, and Φ denotes the empty set,
3. $S \equiv \Phi$, where S is a set,
4. $S \equiv I^k$, where S is a set with degree k, I is the set of integers, $I^i = I \times I^{i-1}$, $I > 1$, and $\times$ denotes Cartesian product.

D-VISUAL uses *positive* or *negative* (non-built-in) *predicates* of the form $R(\mathbf{X})$ and $\neg R(\mathbf{X})$, respectively, where R is a base relation and $\mathbf{X}$ is a vector of variables and constants. A D-VISUAL program is a set of *rules* of the form "head :- body" where *head* is a positive predicate and *body* is a conjunction of predicates.

A D-VISUAL set is either a set of constants defined using set constructors {,} or a positive predicate or a set of rules of the form $< \mathrm{rule}_1, \ldots, \mathrm{rule}_k >$.

We permit three types of set operators: 1) set manipulation operators $(-, \cup, \cap)$, 2) set comparison operators $(\subset, \subseteq, \supset, \supseteq, \equiv)$, and 3) set membership operators $(\in, \notin)$. Among these operators, set manipulation operators and $S_1 \equiv S_2$ are redundant [42].

It is straightforward to show that (nonrecursive) D-VISUAL has identical constructs with RC/S and RC/S is shown to be equal to RC [42], we have RC/S, RC, and nonrecursive D-VISUAL are equivalent in power.

Remark 6.1. RC/S, RC, and nonrecursive D-VISUAL are equivalent in power.

However, the definition of D-VISUAL in Section 3 is not safe, i.e., may produce infinite output or may take infinite time to evaluate. For safety, in D-VISUAL, we do not allow built-in predicates of type (6) and use the following restrictions:

1. A variable appearing in a rule head also appears in the rule body.
2. All variables in a rule body, excluding those that appear only in a set, are limited. A variable X is limited if
   a. X appears in a positive predicate, outside a set, in the body.
   b. X appears in $X \in S$, where S is a set without X in any of the rules in S.
   c. X appears in $X = Y$ and Y is limited.
3. Each rule defining a set satisfies (a) and (b).

Remark 6.2. Nonrecursive D-VISUAL with restrictions (a), (b), and (c) is safe.

The example below illustrates some D-VISUAL rules.

Example 6.1. Assume p, q, s, and t are base predicates. The following are valid D-VISUAL rules:

1. $$w(x,y) :- p(x), q(y), < r(z,v) : -s(z,v,x,y) > \subseteq < u(z,v) : -t(z,v,x,y) > .$$

2. $$w_1(x,y) : -p(x), q(y), s \in < r(z,v) : -s(z,v,x,y) > .$$

3. $$w_2(x) : -p(x), \{1,3\} \subseteq < r(z,v) : -t(z,v,x,y) > .$$

Please note that <rule> is specified through external/internal queries in relational VISUAL. Examples below specify the D-VISUAL rules for the relational version of our materials database. The following relational schemes represent only a part of the database schema in Fig. 2:

$$\textit{experiment}(\underline{\text{eid}}, \text{first-time}, \text{last-time}),$$
$$\textit{frame}(\underline{\text{fid}}, \text{ftime}, \text{parent}, \text{eid}),$$
$$\textit{cluster}(\underline{\text{cid}}, \text{fid}, \text{centroid\_X}, \text{centroid\_Y}),$$
$$\textit{particle}(\underline{\text{pid}}, \text{fid}, \text{centroid\_X}, \text{centroid\_Y}),$$
$$\textit{particle-in-cluster}(\underline{\text{pid}}, \text{cid}), \textit{splitEvolves}(\underline{\text{pid}}, \text{pid}),$$

where the primary key attributes in each relation are underlined. Please note that *eid* attribute in the *frame* relation is used to represent the one-to-many relationship "experiment is-composed-of frames." Similarly, *fid* attributes in *cluster* and *particle* relations represent the relationships "frame is-composed-of cluster" and "frame is-composed-of particle," respectively. The relationship "cluster is-composed-of particle" is represented by the relation *particle-in-cluster*; and, the tuple (p_1, p_2) in *splitEvolves* represents the fact that p_1 is split and one of the newly formed particles is p_2.

To represent the time dimension (for the sake of illustration), we use the relation scheme *next-time*(before, after), where a tuple (t, t_1) in *next-time* indicates that t_1 is the next time point after t.

Example 6.2. VISUAL query in Fig. 6 is rewritten in D-VISUAL in this example.

$$\text{Particles\_In\_Window(P)}: -\text{experiment(e,,)}, \text{frame(F,,,e)},$$
$$\text{particle(P, F, X, Y)}, \text{window\_X}_{\text{low}} < \text{X}, \text{X} < \text{window\_X}_{\text{high}},$$
$$\text{window\_Y}_{\text{low}} < \text{Y}, \text{Y} < \text{window\_Y}_{\text{high}}.$$

Note that, in this query, the range window W is specified in X dimension by $\text{window\_X}_{\text{low}}$ and $\text{window\_X}_{\text{high}}$ (constants), and in Y dimension by $\text{window\_Y}_{\text{low}}$ and $\text{window\_Y}_{\text{high}}$.

Example 6.3. VISUAL query in Fig. 10 is rewritten in D-VISUAL in this example.

$$\text{Experiment\_With\_All\_Frames\_Having\_Clusters(E)} : -$$
$$\text{experiment(E,,)}, < \text{Frames(E, F}_1) : -\text{experiment(E,,)},$$
$$\text{frame(F}_1\text{,,,E)} > < \text{Frames\_With\_Cluster(E, F}_2) : -$$
$$\text{experiment(E,,)}, \text{frame(F}_2\text{,,,E)}, \text{cluster(C, F}_2\text{,,)} > .$$

Example 6.4. The following query is the D-VISUAL version of the VISUAL query in Fig. 11.

$$\text{Frames\_With\_No\_Clusters(F)} : -\text{experiment(e,,)},$$
$$\text{frame(F,,,e)}, \text{F} \notin < \text{Frames\_With\_Cluster(F}') : -$$
$$\text{frame(F}'\text{,,,)}, \text{cluster(C, F}'\text{,,)} > .$$

8 Implementation

User interface of VISUAL is designed with efficiency and ease of use in mind. Consistency between user-interface object definitions is aimed as much as possible since this cuts down the training time of users in using the system.

Graphical user interface design of VISUAL was influenced by the Microsoft Foundation Classes Library and the Document-View user-interface design of Microsoft Visual C++. In our design of the graphical user interface, we have used the following rules:

- Clarity: User-interface objects are designed to be unambiguous (each user interface object clearly identifies (only) one application domain object).
- Feedback: Every action of the user is feedback by changes reflected or by changes not reflected on the objects on screen.
- Flexibility: Users are allowed to change entries they have made or reverse their actions.
- Aesthetics: System control on the VISUAL queries have certain standards on appeal and clarity.
- Consistency: Query entry checks consistency between key strokes.

The user interface of VISUAL is implemented. The interface consists of three parts: the *application*, *documents* (objects corresponding to queries and query templates (VISUAL schemas)), and *graphical views*. The application enables users to specify queries, manage documents, and display the views.

The query template documents are schemas for VISUAL queries. Query templates act as a middle layer between the actual database schemas and the iconic representations on the screen. This increases flexibility of VISUAL since different views can be supplied to different users of the same database schema (similar to views in conventional databases). Users define classes and methods along with their corresponding graphical icons in query templates. Spatial relationships and composition relationships are attached to the underlying database schema using query templates. Spatial relationships can be defined in three dimensions; left-right, up-down, and depth of screen are the dimensions available. Every query document is based on one query template.

Two different views (one in ASCII and another one that uses graphical objects on the screen) are implemented. Query object merging in complex algebra and in OQL are implemented.

VStore is the object storage manager for VISUAL. It is the entry point for database schemas and object storage functionality. Fig. 24 shows the VISUAL—VStore architecture. Queries that are created in VISUAL and converted to OQL can be submitted third party ODBMSs. We have implemented a link between VISUAL and O_2 [4] using a socket interface. In this scenario, VISUAL is used as the front-end and O_2 is used as the back-end of the ODBMS.

9 Recent Related Work

Now, we briefly summarize the recent work in graphical query languages, also referred to as visual query interfaces or visual query systems. Most of the recent work on graphical query languages is on spatial databases or on formalisms to describe visual query interfaces.

Catarci et al. [7] is a comprehensive survey to date about visual query systems for databases. It classifies visual query systems in terms of their representations as form-based, diagram-based, icon-based, or hybrid approaches. VISUAL would categorize as a hybrid approach as it uses diagrams and icons in query formulation. Catarci et al. also compares visual query systems based on their interaction strategies (namely, in terms of understanding the reality of interest

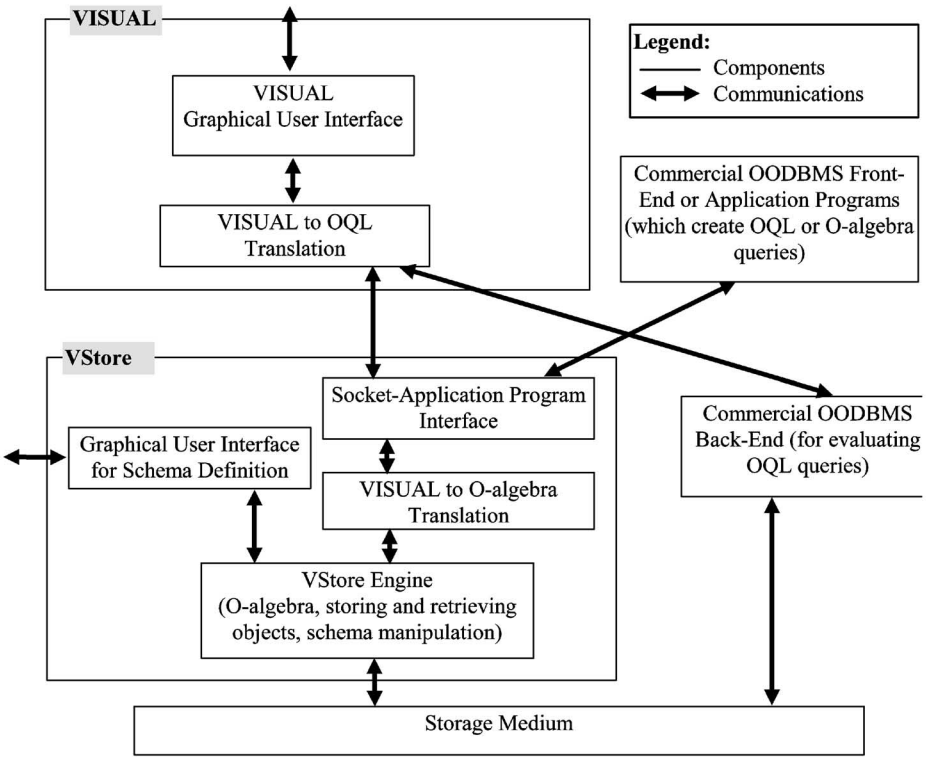


Fig. 24. VISUAL—VStore architecture.

and in formulating the query), summarizes the usability studies in the literature, discusses the formal aspects of visual query systems such as syntax and semantics, and expressive power.

OUIVER [8] is a graph-based visual query language for object databases. OUIVER queries are translated into OQL queries and executed. OUIVER does not have most of the concepts that VISUAL has, such as the notion of subqueries, user-specified icons, etc. OUIVER also does not deal with query processing, as it is a frontend to O2 database management system. Cruz and James [10], Cruz and Leveille [11] present Delaunay which is a database visualization system for object-oriented databases. Presented in Cruz et al. [9], Cruz and James [10], Delaunay^{MM} is a multimedia extension to Delaunay that allows users to visually represent the spatial layout of the data to be retrieved from distributed multimedia repositories. Delaunay^{MM} query language interface supports standard SQL clauses. Delaunay^{MM} does not deal with query processing, but translates the SQL queries specified by users into the syntax of the underlying data repositories.

Rendering by Example (RBE) [31] is a declarative language to express rendering of data (with subsequent browsing and interaction semantics). ICBE language [19], [20] subsumes RBE, is extensible and based on domain calculus, Horn-clause logic-based language, called Logic++ [20]. ICBE does not deal with an object-oriented query specification model. The query processing techniques of ICBE are not available. Pictorial Query-By-Example (PQBE) [45] is designed to retrieve "direction relations" stored in symbolic images using a visual language and, thus, allows queries involving spatial inference. In [13], a hypergraph-based theoretical framework, called Structure Modeling Hypergraphs (SMH), is proposed for visual interaction with databases. SMH is not a data model, but a representation tool able to capture the features of data models and useful in the design of new visual interfaces. It is also shown in [13] how SMH can be queried by formal systems closed under queries. Query-by-Diagram (QBD*) [1] defines a recursive graphical query language based on the Entity-Relationship model. QBD* is a navigational language, interactive, relationally complete, and uses the E-R model symbols to express queries.

SUPER [16] is a visual interface for object+relationship data models and is designed as a front-end to relational and object-oriented DBMSs in all the phases of the database cycle, i.e., creation, manipulation, and evolution. SUPER is built on the ERC+ data model, an object-based extension of the E-R model. As it is a front-end, it does not deal with query processing.

Gql [44] is a formal graphical query language based on the functional data model. Query processing of Gql is not discussed. Haarslev et al. [23] presents a description logic-based formalism for specifying the semantics of visual spatial query languages. The logic also supports reasoning about query subsumption and applying default knowledge. In [6], the design and implementation of hierarchical visual query system (HVQS) is described where users visually formulate a query and watch the query results on a few hierarchically arranged higher layers (i.e., a hierarchical graph) of database schema.

Murray et al. [36] presents a framework for the systematic description of data model and its graphical user interface. The utility of the framework is then demonstrated

by describing two visual query interfaces, namely, Gql [44], and QBE [51] using the framework.

10 Conclusions and Future Work

We have presented VISUAL, an object-oriented graphical query language designed for scientific databases where the data has spatial properties and exploratory queries are common. The data model used is an object-oriented data model with complex objects that can be built by set, bag, and tuple constructors. VISUAL has an object-oriented query specification model. That is, a query object is composed of subquery objects and query specification objects, which can be saved and reused.

Features of the VISUAL query language are described and illustrated by examples. Novel parts of VISUAL query processing, namely, 1) merging query objects, 2) method applier, and 3) composition flatten are discussed. Other parts of VISUAL query processing involves a subset of the traditional and well-known query optimization techniques such as pushing selections and joins down the query plan, etc. Therefore, this paper does not discuss the rest of the query processing techniques used in VISUAL.

In this paper, we have described the current implementation of VISUAL (which is version 2.0) and we have not discussed recursion. However, when recursion is allowed in D-VISUAL, the interplay between recursion and negation needs to be controlled. Built-in predicates of type $x \notin S$, $S1 \cap S2 \equiv \Phi$, and $S1 \cup S2 \equiv \Phi$ introduce negation directly into D-VISUAL:

1. $x \notin S$ is equivalent to $\neg(x \in S)$.
2. Let S_1 be

$$< p_1() : -p() >$$

and S_2 be $< q_1(\mathbf{X}) : -q(\mathbf{X}) >$. Then,

- $S_1 \cap S_2 \equiv \Phi$ is equivalent to $(\forall)(p(\mathbf{X}) \rightarrow \neg q(\mathbf{X}))$.
- $S_1 \cup S_2 \equiv \Phi$ is equivalent to $(\forall)\neg(p(\mathbf{X}) \vee q(\mathbf{X}))$.
- $S \equiv \Phi$ is equivalent to $(\forall)\neg(p(\mathbf{X}))$.

In recursive VISUAL, we do not permit the three built-in predicates of item (b) and we permit $x \notin S$. In addition, the negative predicate $\neg(R(\mathbf{X}))$ has explicit negation and set comparison predicates introduce (implicit) negation since, for example,

$$< p_1(\mathbf{X}) : -p(\mathbf{Y}) > \subseteq < q_1(\mathbf{X}) : -q(\mathbf{Y}) >$$

is equivalent to $(\forall \mathbf{Y})(p(\mathbf{Y}) \rightarrow q(\mathbf{Y}))$ or $(\forall \mathbf{Y})(\neg p(\mathbf{Y}) \vee q(\mathbf{Y}))$. By restricting D-VISUAL programs to be stratified, we guarantee a unique minimal model (to be reported later).

Object-oriented features of VISUAL and the use of oids for large binary objects leads to optimization possibilities, which could be exploited by object (O-)algebras. One such algebra is defined in [33]. O-Algebra uses oids, referencing, and dereferencing operators extensively to improve efficiency (time complexity) of queries. Also, the use of oids decreases the amount of data moved within a database system, which in turn decreases the space complexity of queries and temporary spaces.

One request from our Materials engineering colleagues who used VISUAL was a "special" query that involved the evaluation of an "expensive predicate" [21]: locating automatically the split and moved particles (this particle location problem, solved manually by them, is extremely labor-intensive), tracing their movements across frames, *and* querying the electron microscopy images simultaneously. We ended up solving the particle location problem separately—which can be viewed as an "expensive" user-defined method. In our VISUAL implementation, we did not pursue research into other query processing and execution issues such as "expensive predicate placement in query plans" or "caching expensive methods during query execution," as covered in [21], [22], [24]. Applying/refining the techniques listed in Hellerstein et al.'s work for VISUAL query processing is an interesting research direction.

Our chosen application domain of Materials Engineering experiments can be modeled using a spatio-temporal data model with discretely moving objects (i.e., particles moving within frames). In our examples, we have chosen to represent the same particle in different frames as a distinct object with a separate oid (i.e., appearances of the same particle in different frames are distinct) and model the relationship between the instances of the same particle using *evolvedFrom and evolvedInto* relationships. This can be viewed as a model for "discretely moving objects." The work by Guting et al. [18] presents a data model and a query language for continuously moving objects in spatio-temporal databases. Extending VISUAL to represent and query continuously moving objects using graphical versions of the data types and operations of the query language presented in [18] is a possible research direction.

A graphical language, such as VISUAL, can greatly aid domain scientists in using ODBMSs in many application areas. With special constructs like spatial enforcement region and nested flattening operator, VISUAL is especially suited for scientific domains. Another domain of interest for VISUAL is multimedia presentation graphs. Multimedia presentation graphs require temporal operators for querying. An extension to VISUAL, called GVISUAL, with temporal operators (such as *next*, *connected*, and *until*) is given in [34], [35].

Acknowledgments

This research is partially supported by the US National Science Foundation grants IRI-92-24660 and IRI-90-24152. A preliminary version of this paper was published at the IEEE ICDE '96 conference.

References

[1] M. Angelaccio, T. Catarci, and G. Santucci, "QBD*: A Graphical Query Language with Recursion," *IEEE Trans. Software Eng.*, vol. 16, no. 10, Oct. 1990.
[2] S. Abiteboul and P.C. Kanellakis, "Object Identity as a Query Language Primitive," *Proc. ACM SIGMOD Conf.*, 1989.
[3] N.H. Balkir, "VISUAL," master's thesis, CES Dept., Case Western Reserve Univ., Cleveland, 1995.
[4] F. Bancilhon, S. Cluet, and C. Delobel, "The O2 Query Language Syntax and Semantics," technical report, INRIA, 1990.
[5] A.D. Bimbo, E. Vicario, and D. Zingoni, "Sequence Retrieval by Contents through Spatio Temporal Indexing," *Proc. IEEE Symp. Visual Languages*, 1993.
[6] P.-K. Chen, G.-D. Chen, and B.-J. Liu, "HVQS: The Hierarchical Visual Query System for Databases," *J. Visual Query Languages and Computing*, vol. 11, pp. 1-26, 2000.

[7] C. Catarci, M.F. Costabile, S. Levialdi, and C. Batini, "Visual Query Systems for Databases: A Survey," *J. Visual Languages and Computing,* vol. 8, no. 2, pp. 215-260, 1997.
[8] M. Chauda and P.T. Wood, "Towards an ODMG-Compliant Visual Object Query Language," *Proc. VLDB Conf.,* p. 456, 1997.
[9] I.F. Cruz et al., "Delaunay: A Database Visualization System," *Proc. ACM SIGMOD Conf.,* pp. 510-512, 1997.
[10] I.F. Cruz and K.M. James, "A User-Centered Interface for Querying Distributed Multimedia Databases," *Proc. ACM SIGMOD Conf.,* pp. 590-592, 1999.
[11] I.F. Cruz and P.S. Leveille, "Implementation of a Constraint-Based Visualization System," *Proc. IEEE Visual Languages Symp.,* pp. 13-20, 2000.
[12] W. Chen, M. Kifer, and D.S. Warren, "Hilog as a Platform for Database Languages," *Proc. Workshop Data Bases and Programming Languages,* 1989.
[13] T. Catarci and L. Tarantino, "A Hypergraph-Based Framework for Visual Interaction with Databases," *J. Visual Languages and Computing,* vol. 6, pp. 135-166, 1995.
[14] I. Cruz, "Doodle: A Visual Language for Object-Oriented Databases," *Proc. ACM SIGMOD Conf.,* 1992.
[15] H. Custer, *Inside Windows NT.* Microsoft Press, 1992.
[16] Y. Dennebouy et al., "SUPER: Visual Interfaces for Object+Relationship Data Models," *J. Visual Languages and Computing,* vol. 5, pp. 73-99, 1995.
[17] V. Deshpande and P.-A. Larson, "An Algebra for Nested Relations," technical report, Univ. of Waterloo, Ontario, Canada, 1988.
[18] R.H. Güting et al., "A Foundation for Representing and Querying Moving Objects," *ACM Trans. Database Systems,* vol. 25, no. 1, pp. 1-42, 2000.
[19] N. Goyal et al., "Is GUI Programming a Database Research Problem?" *Proc. ACM SIGMOD Conf.,* pp. 517-528, 1996.
[20] N. Goyal et al., "Picture Programming Project," *Proc. ACM SIGMOD Conf.,* pp. 514-516, 1997.
[21] J.M. Hellerstein, "Practical Predicate Placement," *Proc. ACM SIGMOD Conf.,* pp. 325-335, 1993.
[22] J.M. Hellerstein, "Predicate Migration: Optimizing Queries with Expensive Predicates," *Proc. ACM SIGMOD Conf.,* pp. 267-276, 1993.
[23] V. Haarslev, R. Moller, and M. Wessel, "On Specifying Semantics of Visual Spatial Query Languages," *Proc. IEEE Visual Languages Symp.,* pp. 4-11, 1999.
[24] J.M. Hellerstein and J.F. Naughton, "Query Execution Techniques for Caching Expensive Methods," *Proc. ACM SIGMOD Conf.,* 1996.
[25] W. Hou and G. Ozsoyoglu, "Processing Real-Time Aggregate Queries in CASE-DB," *ACM Trans. Database Systems,* June 1993.
[26] W.W. Chu et al., "A Temporal Evolutionary Object-Oriented Data Model and Its Query Language for Medical Image Management," *Proc. Very Large Data Base Conf.,* 1992.
[27] K. Kim, W. Kim, and A. Dale, "A Cyclic Query Processing in Object-Oriented Databases," *Proc. IEEE Int'l Conf. Data Eng.,* 1989.
[28] M. Kifer, W. Kim, and Y. Sagiv, "Querying Object-Oriented Databases," *Proc. ACM SIGMOD Conf.,* 1992.
[29] M. Kifer and G. Lausen, "F-Logic: A Higher-Order Language for Reasoning About Objects, Inheritance, and Scheme," *Proc. ACM SIGMOD Conf.,* 1989.
[30] M. Kifer, G. Lausen, and J. Wu, "Logical Foundations of Object-Oriented and Frame-Based Languages," *J. ACM,* vol. 42, no. 4, pp. 741-843, 1995.
[31] R. Krishnamurthy and M. Zloof, "RBE: Rendering by Example," *Proc. IEEE Int'l Conf. Data Eng.,* pp. 288-298, 1995.
[32] Y. Lou and Z.M. Ozsoyoglu, "LLO: An Object-Oriented Deductive Language with Methods and Method Inheritance," *Proc. ACM SIGMOD Conf.,* 1991.
[33] J. Lin and Z.M. Ozsoyoglu, "Processing OODB Queries by O-Algebra," *Proc. Eighth Int'l Conf. Information and Knowledge Management,* pp. 134-142, Nov. 1996.
[34] T. Lee, L. Sheng, T. Bozkaya, N.H. Balkir, Z.M. Ozsoyoglu, and G. Ozsoyoglu, "Querying Multimedia Presentations Based on Content," *IEEE Trans. Knowledge and Data Eng.,* vol. 11, no. 3, May/June 1999.
[35] T. Lee, L. Sheng, N.H. Balkir, A. Al-Hamdani, Z.M. Ozsoyoglu, and G. Ozsoyoglu, "Query Processing Techniques for Multimedia Presentations," *J. Multimedia Tools and Applications,* vol. 11, no. 1, May 2000.
[36] N. Murray, C. Goble, and N.W. Paton, "A Framework for Describing Visual Interfaces to Databases," *J. Visual Query Languages and Computing,* vol. 9, pp. 429-456, 1998.
[37] I.S. Mumick and K.A. Ross, "Noodle: A Language for Declarative Querying in an Object-Oriented Database," *Proc. Int'l Conf. Deductive and Object-Oriented Databases,* 1993.
[38] *The Object Database Standard: ODMG-93—Release 1.1.* R.G.G. Cattell, ed., Morgan Kaufmann, 1993.
[39] G. Ozsoyoglu, S. Guruswamy, K. Du, and W. Hou, "Time-Constrained Query Processing in CASE-DB," *IEEE Trans. Knowledge and Data Eng.,* pp. 865-884, Dec. 1995.
[40] G. Ozsoyoglu, V. Matos, and Z.M. Ozsoyoglu, "Query Processing Techniques in the Summary-Table-by-Example Database Query Language," *Proc. ACM Trans. Database Systems,* vol. 14, no. 4, Dec. 1989.
[41] G. Ozsoyoglu, Z.M. Ozsoyoglu, and V. Matos, "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions," *ACM Trans. Database Systems,* vol. 14, no. 4, Dec. 1987.
[42] G. Ozsoyoglu and H. Wang, "A Relational Calculus with Set Operators, Its Safety, and Equivalent Graphical Languages," *IEEE Trans. Software Eng.,* vol. 15, no. 9, Sept. 1989.
[43] G. Ozsoyoglu and H. Wang, "Example-Based Graphical Database Query Languages," *Computer,* pp. 25-38, May 1993.
[44] A. Papantonakis and P.J. King, "Syntax and Semantics of Gql, A Graphical Query Language," *J. Visual Query Languages and Computing,* vol. 6, pp. 3-25, 1995.
[45] D. Papadias and T. Sellis, "A Pictorial Query-by-Example Language," *J. Visual Languages and Computing,* vol. 6, pp. 53-72, 1995.
[46] G.H. Sockut et. al., "GRAQULA: A Graphical Query Language for Entity-Relationship or Relational Databases," *Data and Knowledge Eng. J.,* vol. 11, 1993.
[47] M. Stonebraker et. al., "Tioga: Providing Data Management Support for Scientific Visualization Applications," *Proc. Very Large Data Base Conf.,* 1993.
[48] A. Shoshani and H.K.T. Wong, "Statistical and Scientific Database Issues," *IEEE Trans. Software Eng.,* Oct. 1985.
[49] J.D. Ullman, *Principles of Database and Knowledge-Base Systems,* vol. 1, 1988.
[50] K. Vadaparty, Y.A. Aslandogan, and G. Ozsoyoglu, "Towards a Unified Visual Database Access," *Proc. ACM SIGMOD Conf.,* 1993.
[51] M.M. Zloof, "Query-by-Example: A Database Language," *IBM Systems J.,* vol. 21, no. 3, 1977.

Nevzat Hurkan Balkir received the BSc degree from Bilkent University, Ankara, Turkey, and the MS degree from Case Western Reserve University in 1993 and 1995, respectively. He completed his PhD degree in managing multimedia presentations from the Computer Engineering and Science Department, Case Western Reserve University, Cleveland, Ohio, in 1998. He is a solutions architect at Experian North America. His primary research interests are in the areas of object-oriented databases, multimedia databases, query optimization, and data distribution on very large databases.

Gultekin Ozsoyoglu received the PhD degree in computing sciencefrom the University of Alberta, Edmonton, Alberta, Canada, in 1980. He is a professor in the Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio. Dr. Ozsoyoglu has published many papers in database, security, multimedia computing, and real-time computing conferences and journals, such as *ACM Transactions on Database Systems*, *IEEE Transactions on Software Engineering*, *IEEE Transactions on Knowledge and Data Engineering*, and *Journal of Computer and System Sciences*. He has served on program committees and panels of database conferences, such as ACM SIGMOD, VLDB, and IEEE Data Engineering. He was an ACM national lecturer, program chair of the Third Statistical and Scientific Database Conference, workshops general chair of the CIKM '94, CIKM '96 Conferences, research prototypes chair of ACM SIGMOD '94 conference, a guest editor for *IEEE Transactions on Knowledge and Data Engineering*, general chair of the 11th Scientific and Statistical Database Management Conference in 1999, and co-chair of the NSF '2000 Information and Data Management Workshop. He is a member of the IEEE Computer Society.

Z. Meral Ozsoyoglu received the BSc degree in electrical engineering and the MSc degree in computer science, both from Middle East Technical University, Turkey. She received her PhD degree in Computer Science from the University of Alberta, Canada. She is a professor of Computer Science at Case Western Reserve University, Electrical Engineering and Computer Science Department, Cleveland, Ohio. Dr. Ozsoyoglu's primary research work and interests are in the areas of principles of database systems, database query languages, database design, object-oriented databases, and complex objects with applications in scientific, temporal, and multimedia databases. She has published several papers on these topics in computer science journals and conferences. She was ACM PODS program chair in 1997 and the 11th SSDBM Conference program chair in 1999. She has served on the organizing and program committees of several database conferences, including ACM SIGMOD, ACM PODS, VLDB, IEEE DE. She was the ACM SIGMOD vice-chair for 1997-2001. She has been a recipient of several awards, including an IBM Faculty Development Award, an US National Science Foundation Faculty Award for Women in science and engineering (FAW), and a University of Alberta Distinguished Alumni award. She is presently an associate editor of the *ACM Transactions on Database Systems* and *IEEE Transactions of Knowledge and Data Engineering*. She is a member of the IEEE Computer Society.