BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-95-M2

"A Visual Interface for Producing Queries the AQUA Algebra"

by

Brian G. Anderson

A Visual Interface for Producing Queries the AQUA Algebra

Brian G. Anderson

Department Of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University

March 1995.

Stan Zdonik

Professor Stan Zdonik
Advisor

A Visual Interface for Producing Queries in the AQUA Algebra

Brian Anderson

bga@cs.brown.edu

Dept. of Computer Science

Brown University

Box 1910

Providence, RI 02912

Abstract

This paper presents a visual interface for generating queries on object-oriented databases based on the AQUA data model and query algebra. We introduce a display emphasizing the navigation of paths through the member functions of abstract data types and a procedural interaction technique based on line gestures. We explain how to use silVIA (Visual Interface to AQUA) and demonstrate an example query in a sample session.

1.0 Introduction

This paper presents silVIA (Visual Interface to AQUA), a program for visually producing object-oriented database queries in the AQUA [LMS+93] algebra. This interface offers a gestural interaction style to manipulate views of the database into the desired query. It addresses the issues of path navigation and the procedural nature of AQUA in ways not found in existing visual query languages. In conjunction with the GROOVE visualizer for query optimization, it forms a visual environment for query formation and optimization.

In the AQUA data model, named global sets provide access to the database; these global sets form the initial display in the silVIA interface. The user can navigate paths through the structure of the database schema, starting with the global sets and expanding member functions of abstract data types to reach the desired information. Gestures allow the user to alter the views into the database, creating new named sets. As the user manipulates the display of the schema, silVIA keeps track of the underlying AQUA syntax for the actions, so that when a displayed view matches the user's query, its AQUA representation can be submitted as a query to the query optimizer.

The display of the schema and views is based on a nested-tree structure that emphasizes path-navigation and many-to-one relationships. Expanded member functions are placed within the existing display structure to visually preserve transitive relationships, which sometimes are not apparent in graph-based visual languages.

Line gestures are based on metaphors representing the actions of selection, join, union, and extraction. A line crossing the display of a set generates a selection, just as a selection divides a set in half based on a predicate function. Joins and unions are accomplished by drawing a line that connects the sets. An instance deep in a path from the named global sets can be extracted to create a new view into the database by drawing a line from the instance out of its containing sets.

Section 2 compares the silVIA interface to related research in visual interfaces. A brief overview of the AQUA data model and algebra is given in Section 3. The silVIA interface's display and interaction techniques are described in Section 4. Then, a sample query produced with silVIA is demonstrated, with conclusions and future work proposed in the final section.

In this paper, text referring to silVIA's display and names for abstract data types appear in bold face, AQUA expressions appear in italics.

2.0 Existing visual languages

[VAO93] reported in 1993 that since the introduction of QBE, over fifty visual languages for querying databases have been developed. The majority of these languages are designed for the relational model In most cases the method for creating a query uses a declarative, as opposed to a procedural, language. The AQUA data model and syntax defines a procedural method of querying object-oriented databases, so a visual interface for AQUA needs to reflect this in its interaction techniques. The silVIA interface is distinctive in its display of path navigation through member functions of objects and its gestural interaction techniques.

2.1 Relational vs. object-oriented model

QBE [Zlo75] began the movement to create visual techniques for querying databases. Many of the visual techniques and display styles introduced in QBE have been continued in other visual query language designs, including the table based display and the declarative style of creating a query. In table-based displays, a table represents a relation, each field in a tuple placed in a column. Predicates on specific fields are entered within the column. Instance variables can be placed in the columns of two different relations to convey relationships across the database.

The PICASSO [Kim88] interface deals with the relational model by displaying the fields of the relations as areas on the screen with ovals representing the relations. An oval surrounds the fields that are contained in that relation such that relations that share a field are displayed with overlapping ovals.

Although some techniques from relational visual languages can be used in object-oriented visual languages, relational visual languages cannot properly display abstract data types or handle inheritance and path navigation.

2.2 Declarative vs. procedural models

Many visual languages are based on declarative styles of querying. Instead of instructing how to construct a query, the user specifies what the correct answer should look like. Declarative query languages such as DATALOG for relational models and F-logic [KL89] for object-oriented models form the basis for some visual languages; for instance, the DOODLE [Cru92] visual language is based upon F-logic queries.

Instance variables are used in declarative languages to relate between relations or sets of objects in the database. The variable may appear in columns of tables as in VQL [VAO93], or as nodes in a graph with arcs connecting the instance variable to the sets, as in GOOD [GPV90].

Though a declarative method is appropriate for certain query languages, the AQUA model is procedural, so the visual model for an interface for AQUA needs to address this in its interaction style. silVIA uses a set of gestures to modify views of the database, thereby constructing new views. At each step in the process, the displayed sets represent the result of applying AQUA procedures to the existing database global sets. By manipulating the display, the user instructs how the query should be constructed. This style of interacting with a display differs from existing visual languages, and better fits the AQUA model.

2.3 Direct manipulation

An alternative style of visual querying is direct manipulation[Shn93]. In this model, the values in the database are displayed in a format dependent on the data; queries are made by manipulating sliders and other input devices. The display is updated as the user manipulates the query; in order to maintain an interactive response time, query results must be produced at an update rate of 100 msec.

Since the display of the data is dependent on the values in the database; different application pro-

grams need to be written for each type of data being queried. Although input techniques can be transferred from one database type to another, a different application has to be written for each database schema. Because the silVIA display is dependent only on the structure of the schema, and not what types are actually in the database, the interface can be used on different schemas without changing the interaction method a user uses to create a query.

The AQUA data model does not assume that queries can be made quick enough for the direct manipulation method. With AQUA, a query is formulated and then submitted to a query optimizer, which considers the structure of the database. In some databases, including distributed databases, a interactive response time may not be feasible.

2.4 Table and graph displays

Visual query languages generally use one of two different techniques for display: tables or graphs. The table display originated in the relational model, although VQL [VAO93] is table-based and can be used to query object-oriented databases. Relationships between tables are implied through instance variables repeated in different tables. However, because of the complex relationships between objects in the object-oriented model, table-based displays are not necessarily the best way to convey these relationships.

Figure 1 shows an example VQL query illustrating a table-based display. The relationship between the manufacturer of vehicles **c** and **d** is implied by the repetition of the instance variable **m.** However, there is no visual cue to this relationship beyond recognizing that the letter m has appeared in two tables.

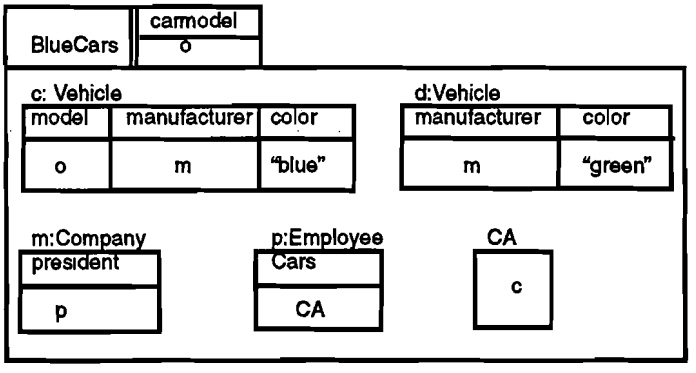


Figure 1: Example VQL query showing table-based display. [VAO93]

A graph based display is used in GOOD [GPV90] and G$^+$ [Cru89]. Instances are normally

expressed as nodes in the graph; member functions or fields in tuples are expressed by arcs. The GOOD method of display makes it hard to visually tell a "many to one" relationship, because the only visual cue is the a double headed arrow, but the basic arc diagram remains the same between many-to-one and one-to-one relationships. An arc in G^+ is labelled "instanceof" to signify one instance of a set. Figure 2 illustrates a G^+ query for the Blue Cars query that will be presented in the sample session part of this paper. Notice the need for nodes representing instance variables (w,x,y,z) in order to define relationships across sets. The arcs in this example define a better visual cue for these relationships than the VQL table-based display; however, when the size of the query increases, the transitive relationships, such as the cars owned by the president of a company (the arcs connecting COMPANY to w to z to y in Figure 2), may not be as easy to read.
The silVIA display introduces an alternative to the graph-based display which conveys these relationships through a nested-tree structure that does not require nodes and arcs.

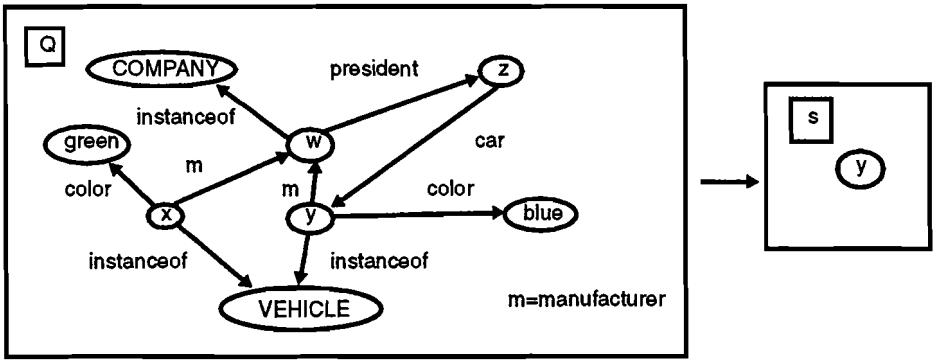


Figure 2: Example G+ query showing graph-based display.[VAO93]

The Hy^+ query language [CM92] displays the result of a query in a hygraph that can lead to crowded displays if there are many nodes and arcs. In addition, transitive relationships are sometimes hard to distinguish in complex hygraph displays. The silVIA display uses a nested tree diagram to display the schema, instead of the actual values in the database, that makes many-to-one and transitive relationships more apparent. Because of the importance of these relationships in object-oriented databases, the silVIA techniques may be more appropriate when navigating a path through object member functions.

2.5 Nested tree diagram of silVIA

Object-oriented databases are accessed through named global sets of objects. The objects within

these sets can contain member functions that return other objects, or sets of objects. These objects can in turn return objects in their member functions. In this way, a path can be constructed through the database, accessing certain objects through their relationships to others. This navigation forms the basis for silVIA's display technique.

In silVIA, the initial display of a named global set consists of the name of the set and a rectangle containing an instance of the type contained within that set. The rectangle expresses the many-to-one relationship of the multiple instances within the set to the one set name. The instance within the set rectangle can be expanded to display its member functions. These member functions and their return types are displayed within the original rectangle, preserving the many-to-one relationship. If **People** is a set of **Person**, and *name* is a member function on **Person** returning a string, **People** could be used to access a set of names. The nested nature of the silVIA display makes this apparent, whereas displays that use arcs to represent member functions [GPV90],[GSKZ85] sometimes lose this distinction.

If **Person** has a member function *cars* that returns a set of **Vehicle**, then another set rectangle is drawn. The instance drawn within cars is two set rectangles removed from the top level **People**. Therefore, **People** can be used to access a set of sets of **Vehicle**. Because of the nested tree display of silVIA, this transitive many-to-one relationship between **People** and **cars** is maintained. Section 4.1 will discuss the silVIA display in more detail.

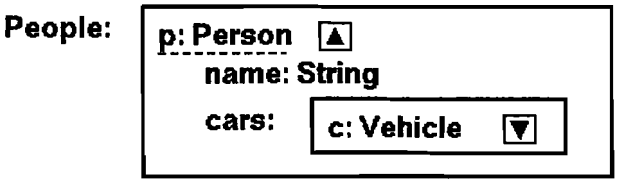


Figure 3: Nested-tree structure showing set People of type Person with name and cars as member functions of Person.

Because of the various models of database querying, many different visual interaction styles have been developed. The silVIA interface offers unique solutions for the issues of object-oriented navigation and the procedural nature of the AQUA data model and algebra.

3.0 AQUA Model and Syntax

In order to explain the results of actions in silVIA, the syntax of the AQUA query algebra is pre-

sented briefly here. A detailed explanation of the AQUA data model and algebra can be found in [LMS+93].

3.1 Types

Types in AQUA are built up from the set of four base types: integer, float, boolean, and string. New types can be built from combinations of these base types and other constructed types. Abstract data types are defined through a type definition and a collection of functions to access values within the type. Single inheritance between types is supported, so that an **Employee** type can be constructed by subclassing from an existing **Person** type.

3.2 Syntax

The syntax for AQUA consists of terms: either a variable, constant or function symbol, a lambda abstraction, or an application. In the AQUA expression *apply(λ(p) p.name)(Persons)*, *apply* and *name* are function symbols, *p* is a variable, and *λ(p) p.name* is a lambda abstraction. The **apply** function iterates through the set **Persons** assigning *p* to each element in the set and then invoking the function on *p*. In this case, the member function *name* is applied to each **Person**, returning a string. The expression *I.f* is a shortcut for the **invoke** function: *invoke(I,f)* calls function *f* on instance *I*. For example, *p.name* invokes the *name* function on the object *p*. The result of the complete expression is a set of strings representing the set of names of each person in the **Persons** set.

3.3 Operators

The set operators in AQUA commonly used by silVIA during gestures are introduced here. The **apply** operator was illustrated above; it takes a function, *f*, and a set, *A*, and applies *f* to each element in *A*. A new set of the type returned by *f* is produced.

The **select** operator chooses elements from a set, *A*, based on a predicate function, *p*. Each element in *A* is included in the resulting set if *p(a)* is true. A common predicate function used in selection is the **mem** operator. The arguments for **mem** are a set, *A*,an element, *x*, and an equality function defined on the type contained in the set. The **mem** function returns true if there is an element in *A* that is equal to *x*.

The **fold** operator reduces a set to a single value. It takes three arguments ($u, f, \oplus$) and a set, *A*. The function *f* is applied to each element in *A* and the results are combined using the function, $\oplus$. *u* is the result of applying the **fold** to the empty set. The operators **select** and **mem** could be

implemented using **fold**, for instance:

select(p)(A) = fold({}, λ(x) if p(x) then set(x) else {}, union(id,T))(A).

The binary set operators **union**, **intersection**, and **difference** differ from the traditional definitions in that they require a type parameter. This is the resulting type of the binary set operation. For **union**, this type must be some common supertype of the two inputs. For **intersection**, this type can be either of the two input types. For **difference**, this type must be the type of the first input set.

The **join** operator takes a predicate and a function. The predicate argument can be used to specify natural- or equi-joins, or to do a selection on the result of the join. The function argument defines how the two sets will be joined. The following expression demonstrates how to join **Hotels** and **Monuments** into a set of tuples so that each Hotel/Monument pair is in the same city.

join(λ(h,m) (h.address.city=m.address.city),λ(h,m) <<hotel:h,monument:m>>)
(Hotels,Monuments)

The expression *<<hotel:h,monument:m>>* creates a tuple from the variables *h* and *m*. The fields are named "hotel" and "monument" respectively. The operator **tup_select***(name)(T)* extracts out the field *name* from the tuple *T*. The operator **tup_concat***(T1,T2)* combines tuples *T1* and *T2* into one tuple.

The silVIA interface allows the user to produce AQUA queries without a complete understanding of the AQUA syntax. The operators described above can be executed through the use of mouse clicks and gestures.

4.0 silVIA Display and Interaction

After reading a pair of schema and global sets files, silVIA allows the user to display the sets in the database, navigate through the different types, and manipulate the display with a set of gestures to produce a query in AQUA syntax.

The silVIA interface consists of a global sets list, a message box, a work area, and buttons for deletion and applying functions. Selecting a set from the globals list displays the set in the work area. Within the work area, the displayed sets can then be expanded and manipulated to produce the desired AQUA query. Messages related to the actions being taken, and the AQUA syntax for a selected instance, are displayed in the message box below the work area.

The next section describes how the user can display and navigate through the schema representing the database. Then the set of gestures that manipulate the schema is introduced. Section 4.3 explains how functions and folds are applied to the sets. Lastly, two sample queries are presented to illustrate the full process of producing a query.

4.1 Display of the Schema

The first step in creating a query with silVIA involves choosing which global sets from the database will be involved in the query. A representation of each set is displayed in the work area. Figure 4 illustrates how **Tours**, a set of type **Tour** would be displayed.

Figure 4: Initial display of Tours, set of Tour objects.

The initial display of a set consists of the set name followed by a colon. The rectangle to the right of the name is the set marker. The lower case **t** represents the unique lambda name that will be applied to the set. After the lambda name is the type of the items within the set; in this case, objects of type **Tour**.

The set marker represents a distinction between single and multiple instances of objects. There is one instance of the set called **Tours**, so it is placed outside of the set marker. Within the set, there are multiple instances of objects of type **Tour**. The application of $\lambda(t)$ iterates over the set **Tours** so that **t** can be used to represent each object within the set.

The icon of a down arrow within a box signifies that the object type **Tour** has member functions that can be applied to it. The number of member functions for this type has been defined in the schema file that silVIA has loaded upon execution. Clicking on the down arrow icon expands the display of the **Tour** type, presenting the member functions.

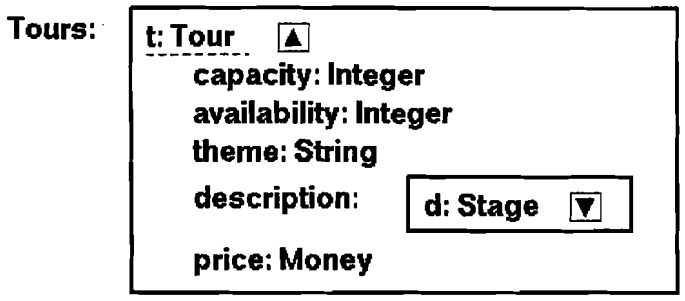


Figure 5: Type Tour has been expanded to show member functions.

In this example, the type **Tour** has five members functions: *capacity*, *availability*, *theme*, *description*, and *price*. After each of these member functions is the type returned by the function. Notice that **description** returns a set of type **Stage**, represented by another set marker. There is another arrow down icon after **Stage** because this type also has member functions. However, there is no icon next to **price** of type **Money**, because this type does not have any member functions defined in the schema file.

The user can then click on the icon next to **Stage** and display the member functions applicable to this type. In this way the schema can be navigated, creating a path through the object types. One of the member functions of type Stage is **what** of type **Place_TG** (place to go). The instances of type **Place_TG** are now two set levels deep from the set name **Tours**. Therefore, by applying functions to the set **Tours**, the user can create a set of sets of type **Place_TG**. Later in the paper, a method for extracting and flattening these places to go will be described, so that it will be possible to create a set of **Place_TG** that represents all of the places that some tour in the database visits.

The arrow icon next to **Tour** now contains an up arrow to represent that it has been expanded. Clicking on this icon will return the display to its previous appearance.

4.1.1 Inheritance

Sometimes in the schema, a type is defined as a subclass of another object type. When this type is expanded, the member functions for this type and the inherited functions for its parent type are all listed. In addition, the parent type is displayed in parentheses next to the original type.

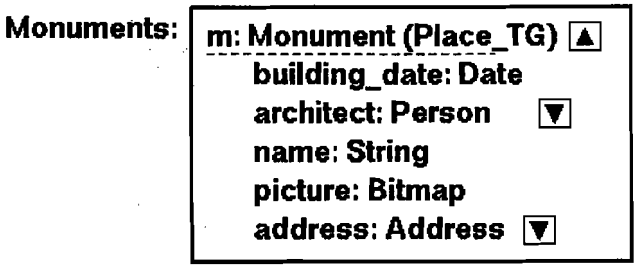


Figure 6: Type Monument inherits member functions from type Place_TG.

In this example, the set **Monuments** consists of a set of type **Monument**. **Monument** inherits functions from the type **Place_TG**. The functions *building_date* and *architect* are from type **Monument**, the others are defined in type **Place_TG**.

4.1.2 Active areas in the display

When the user clicks on an object in the work area, the item is highlighted with a dotted underline. In addition, the AQUA string representing the item is displayed in the message box below the work area.

There are two types of active areas in the display: an instance of an object or a set of objects. The set of objects is represented with the set's name and a rectangle surrounding the instances within the set. The display of an instance of an object consists of the name and type of the object. Instances are selected by clicking within the bounding box surrounding the name and type. A set can be selected by clicking either on the set's name or on any empty area within the rectangle marking the extent of the set.

4.1.3 Mouse click as a path

Each pixel on the display actually represents a path of objects. For instance, in the figure above, a mouse click on **name: String** constructs a path from the set **Monuments**, through the instance **m** of type **Monument** to **name** of type String. This path represents the AQUA steps required to access **name**: apply $\lambda(m)$ to the set *Monuments*, invoking the member function *name* on each element in the set. The result is a set of strings, the names of all of the monuments in the database.

4.1.4 Constructing AQUA

Each instance in the display keeps track of its portion of the AQUA query. When a path is constructed by the user's input, a traversal of the path produces the AQUA text that represents the

query. In the above example, **name:String** contributes its AQUA name to the query: *name*. Then **m:Monuments** adds *m* to the beginning of the AQUA string, resulting in *m.name*. Now the path crosses the set rectangle, so an **apply** function is added. The set **Monuments** knows that it represents the global set *Monuments*, so it adds to the AQUA text producing:

apply (λ(m) m.name) (Monuments)

The path could have crossed more than one set rectangle, so that more than one **apply** function could have been added to the AQUA text. Also, actions by the user could have changed the underlying representation of the set **Monuments**, changing the AQUA text for the set in the **apply** function. For instance, after a selection the AQUA text may appear as:

apply (λ(m) m.name) (select (λ(m) (m.architect.name = "Wright))

4.2 Gestures

The user manipulates the display with a series of gestures. By pressing and holding the mouse button down while moving the pointer in the work area, a line is drawn on the screen. Depending on the start and ending positions of the line, actions such as selections, unions, and joins are executed.

With these gestures, the user can produce new sets by manipulating the display of the global sets in the database. These new sets, or instances within them, can then be submitted as a query.

The gestures are designed as metaphors for the actions that they represent. Because a selection divides a set in two based on a predicate, the gesture for selection involves drawing a line that divides the set in half. Joins and unions take the contents of two sets and combine them into one, so the gesture involves drawing a line from one set to the other. The first is removed from the display to reinforce the idea that the contents of the first set are now added to the second. A member function of a type within a set can be extracted out of the set to form a new set of its own. For instance, the **name** field can be extracted from a set of **Person** to form a set of names. The gesture for this extraction involves drawing a line from the instance within the set to an icon on the top level of the display.

4.2.1 Selection

The gesture for selection on a set involves drawing a line that crosses the set. The line should begin and end outside of the set and the midpoint should be over the set.

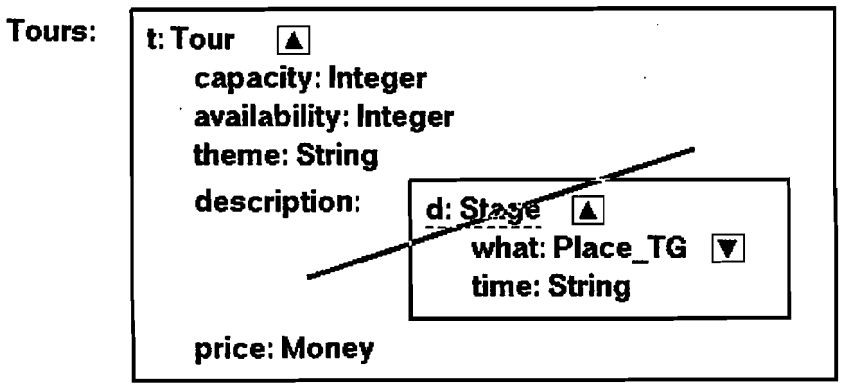


Figure 7: Selection on t.description, a set of Stage objects.

Once the set that is to be selected is determined by the gesture, and the predicate for the selection is then entered into a dialog box. While the dialog box is visible, the user can click on items in the display and the AQUA representation of the items will be automatically entered into the dialog box. The AQUA representation is produced in relation to the set being selected; for instance, if the selection is on **Tours**, a set of type **Tour**, a mouse click on **theme** will return *t.theme*, not *apply (lambda (t) t.theme) (Tours).*

The selection predicate dialog box has a row column of shortcut buttons. Clicking on one of these buttons enters the text on the button into the predicate. In this way, the user can click on an instance in the work area, click on the "=" shortcut button, and then click on the other instance in the work area; entering the predicate without having to use the keyboard.

Once the selection predicate has been entered, the display is updated so that the predicate is contained within the set rectangle, in italic letters to differentiate from the instances within the set. The underlying AQUA representation of the original set is altered to reflect the selection. If the underlying set had already been selected, then the two predicates are combined with the *and* operator. If the underlying set was the result of a join, then the selection is added to the join predicate.

4.2.2 Unions/Joins

When the user draws a line from one set to another, a number of actions are possible. The user's gesture defines a pairing between these two sets and silVIA then decides what actions are feasible based on the types within the sets. The contents of the sets are compared; a set can contain a single type, another set of types, or a tuple of types, sets, or other tuples. If the sets are both single

types, then the types are compared to see if they are compatible. For instance, a set of type **Person** is compatible with a set of type **Employee** because **Employee** is derived from **Person**. If both are sets of tuples, then the fields of the tuples are compared. If the sets are determined to be compatible, then the user is given a dialog box of join options. The options could include set union, intersection, difference, or a join. The display is altered after the user chooses the desired action. If the sets are not compatible, then a join is done automatically.

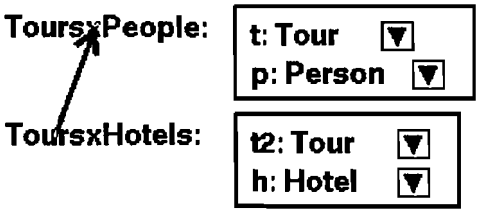


Figure 8: Join of two sets of tuples.

Joining sets of tuples can result in more options. Two sets of tuples can be unioned only if their field names and types are the same. However, compatible types within tuples can be form predicates for the join. A set of tuples of type *<<t:Tour,p:Person>>* can be joined with a set of tuples *<<t2:Tour,h:Hotel>>* such that the **Tour**s are equal. If this case is possible, then another option, **equi-join**, is added to the join action dialog box. The user can then select predicates to add to the join and decide which fields from the two tuples will be in the result. For this join example, the user could choose the predicate *tup_select(t)(x)=tup_select(t2)(y)* and choose not to include the field *t2* in the result. The resulting set would contain tuples *<<t:Tour,p:Person,h:Hotel>>*.

4.2.3 Creating Tuples

There is another way of creating tuples besides joining two sets. A set of **Tours** and a set of **Monuments** could be combined so that the resulting set contains a tuple consisting of a **Tour** and a set of **Monuments**. Each **Tour** would be paired with its individual copy of the set of **Monuments**, so that a selection can be done on the **Monuments** set depending on values in the **Tour** instance. The gesture for this action differs from the join gesture in its release point. A line from **Monuments** to the set **Tours** results in a join, but a line from **Monuments** to the instance **t:Tour** within the set **Tours** creates a set of tuples *<<t:Tour,Monuments:Set[Monument]>>*.

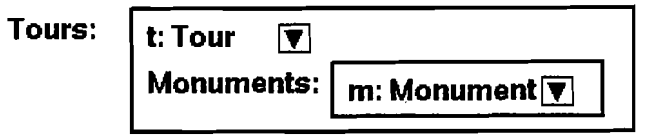


Figure 9: Set of tuples containing a Tour object and a set of Monuments.

4.2.4 Nested Queries

The display illustrated in Figure 6 can be used to represent a tuple or a nested query. If the set **Tours** is selected, then the displayed query is :

apply(lambda(t) <<t:t,Monuments:Monuments>>)(Tours)

However, if the set **Monuments** is selected, then the query is:

apply(lambda(t) Monuments)(Tours)

In this way it is possible to select on **Monuments** relative to each Tour, **t**, without creating a tuple in the query. After a selection on **Monuments**, the following query could be produced, selecting the Monuments that are in a the same city as the places to which each tour goes:

apply(lambda(t) select (lambda(m) (mem(=, m.address.city)(apply (lambda (d) d.what.address.city) (t.description))))
(Monuments))
(Tours)

This query produces a set of sets of Monuments, one set for each Tour. One of the example que ries at the end of this paper will give a another (perhaps better) example of the use of nested que ries.

4.2.5 Extraction and Flatten

When the user expands the member functions for types, she can navigate a path through the schema. By highlighting an instance deep within the path it is possible to create query results such as: a set of sets of facilities for each hotel in the database. The user however may want to construct a new set that represents this query so that she can work with it like the other global sets on the top level of the display. Also, the user may want to flatten this set so that it consists of only a single set of facilities. This action is accomplished by drawing a line from the instance to the New Set Icon in the upper left corner of the work area. The gesture stands for extracting the instance out of the path and up to the top level.

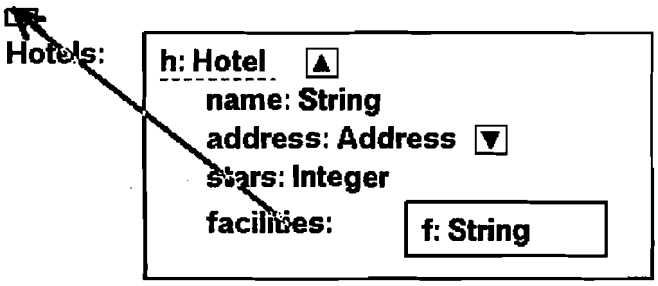


Figure 10: Extracting facilities from Hotel to the New Set Icon.

If the extracted set consists of nested sets (i.e. a set of sets of strings), then the user is given the option of flattening the set. A dialog box gives a list of types that are possible, specifying whether and how much to flatten the set.

4.2.5 Partial Extraction

The gestures for extraction can be combined with the gestures for joins and tuple creation. It is possible to extract an instance partially up the path, without creating a new set at the top level. For instance, it is possible to create a set of tuples <<*e:Employee,birthdate:Date*>> where the **Date** field is the employee's birth date. *birthdate* is a member function of **Employee**, so it will be displayed when the type **Employee** is expanded. If a line is drawn from **birthdate:Date** to **e:Employee**, then a tuple is created just as in the **Monuments** and **Tour** example. In this case, however, the AQUA that stands for birthdate is relative to **e:Employee**. If birthdate was extracted to the New Set Icon, then the AQUA would be *apply(lambda (e) e.birthdate)(Employees)*; but in this case the AQUA is *e.birthdate* because the gesture did not cross a set rectangle.

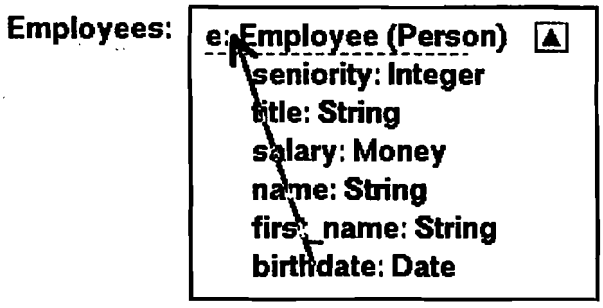


Figure 11: A partial extraction of birthdate, creating a set of tuples <<e:Employee,birthdate:Date>>

It is also possible to extract and do joins and unions at the same time. If the gesture line crosses a set rectangle, but the release point is on a set of instances instead of the New Set Icon, silVIA calculates the extraction and then looks for possibilities for unions or joins.

4.3 Apply Function/Fold

Below the schema list, on the right of the silVIA display, is a button that applies a function or fold to an object. The function or fold is applied to the highlighted object; a function is applied to an instance, a fold is applied to a set.

If the highlighted object is a set, then a dialog box is displayed that asks for the conditions for the fold. A fold requires three items: a null case, a function to apply, and a conjunction function. The fold starts with the null case and then iterates through the set, applying a function to each item in the set and then combining the results using the conjunction. The default case in the dialog box returns a count of the items in the set: *fold(0, lambda(x) 1, lambda(x,y) x+y)*. Figure 12 illustrates how a fold that counts the number of elements in a **Tour**'s *description* set is displayed as if it were a new member function, *number_of_stops*, of **Tour**.

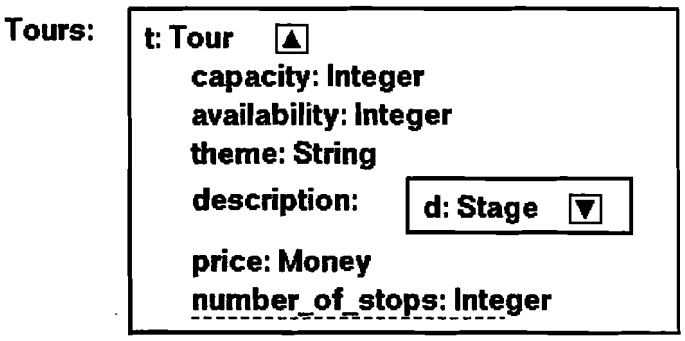


Figure 12: A fold counting the elements in description appears as a new member function, number_of_stops, for Tour

If the highlighted object is not a set, then a dialog box asks for the name of the function to apply to the instance. In this way, new member functions can be added to a type. For instance, a string length function could be applied to the *theme* of a **Tour**, creating a new member function representing the string length of the tour's theme. A selection could then be made on the set of tours based on the string length of the theme.

5.0 Sample Queries

To illustrate the use of silVIA to create a query, a sample session is described based on an example proposed in [Cru89], a variation of the Blue-Cars query from [KKD89]. The query is *Get all the blue cars that are driven by the president of the company that also manufactures green cars.* The relevant parts of the schema for these objects are:

```
Class Vehicle{
   color : String
   manufacturer: Company
}
Class Company{
   president: Person
}
Class Person {
  cars: Set [Vehicles]
}
```

To start the query, the user chooses the set "Vehicles" from the global list. The set **Vehicles** is displayed in the work area. Because the type Vehicle has member functions, a down arrow icon appears next to the type. When the user clicks on the arrow icon, the member functions *color* and *manufacturer* are displayed.

The user then selects the green cars from this set by drawing a line that crosses the **Vehicles** set. At the selection predicate dialog, the user clicks on **color:String**. The AQUA expression *v.color* is entered into the dialog box and the user then adds *="green"* to the predicate.

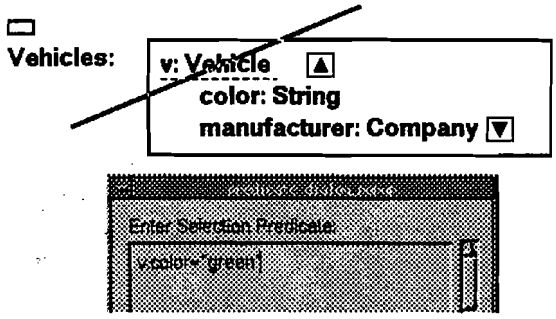


Figure 13: Selection on Vehicles such that color of vehicle is green.

The selection predicate is added to the display and the underlying AQUA for the **Vehicles** set is changed. The displayed set now represents a set of green cars.

The user now clicks on the arrow icon next to **Company** which displays the member function: **president:Person**. The user then clicks on the arrow icon next to **Person** and displays the member function **cars**, a set of **Vehicle**. This set represents the cars owned by the presidents of companies that make green cars. A click on the arrow icon next to Vehicle within this set of cars shows the color and manufacturer for each car within this set.

It is possible for the president to own cars that are not manufactured by his company, but the Blue-Cars query that this example is based on requires that we only get the cars that are manufactured by the company for which the president works. Therefore, we must select from the **cars** set such that the manufacturer of the cars within the set under **president** is the same as the manufacturer of the **Vehicle** in the original set. The user draws a line that crosses the set **cars** and clicks on the two manufacturers when the predicate dialog is displayed, creating the predicate *c.manufacturer=v.manufacturer.*

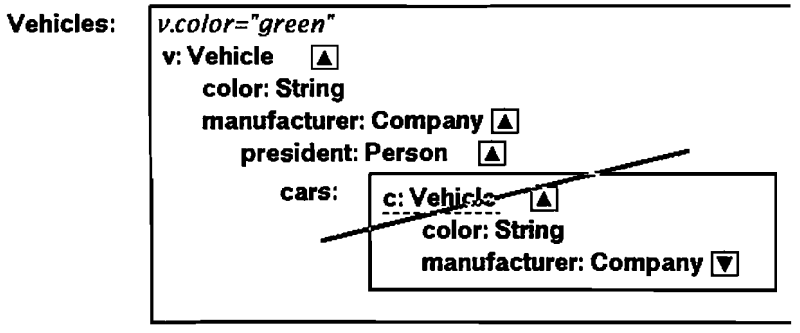


Figure 14: Selecting set "cars" so that c.manufacturer=v.manufacturer.

The set **cars** is imbedded with the original set vehicles, so we really have a set of sets of vehicles. The result of the query should be a single set of vehicles. To extract out the set of cars, the user draws a line from **cars** to the New Set Icon at the top of the work area. This gesture extracts out the cars set to the top level of the display. A dialog box is displayed that asks the user if the set of sets of **Vehicle** should be flattened to a single set of **Vehicle**. In this example we want to flatten the set so the user selects "Set[Vehicle]".

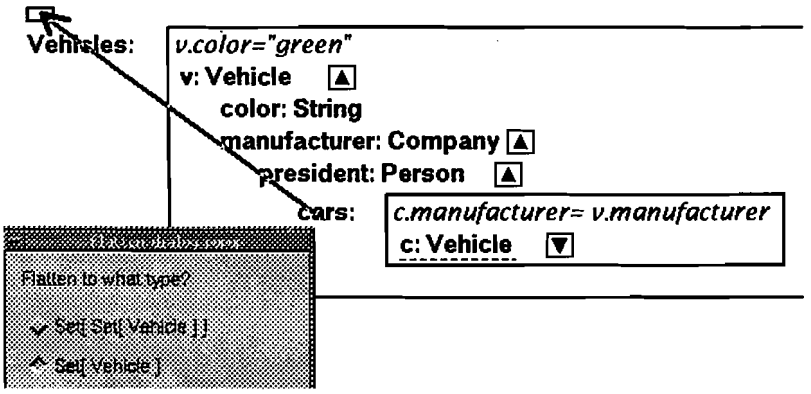


Figure 15: Extracting and flattening "cars" to produce a new set of type Set[Vehicle].

A new set is created and displayed in the work area. This is the set of cars that are owned by the president of the company that manufactures it, provided that the company also manufacturers green cars.

The relationships between the company and the president and the cars are all included in the path nature of the silVIA display. Declarative based visual interfaces require the user to specify these relationships through instance variables; however, in silVIA the display handles these relationships automatically.

The only thing left is to get all of the blue cars out of the new set. One last selection is performed on the new set such that the Vehicle's color equals "blue". The user can then highlight the new set and the AQUA expression of this set will be displayed in the message area. This expression can then be submitted to the query optimizer. The AQUA expression for this query, as produced by silVIA is:

select (lambda(c2) (c2.color="blue"))
(flatten(apply (lambda (v) select (lambda(c) (c.manufacturer= v.manufacturer))
(v.manufacturer.president.cars))
(select (lambda(v) (v.color="green"))
(Vehicles))))

5.1 A Second Sample Query

The second sample query is based on the TravelScema file that is used in the /pro/oodb directo ries. The query is *For each place to go, find all the tours with capacity more than 20 that have the same theme as some tour that already visits the place to go. The result should be a set of tuples <<place: Place_TG, tours: Set[Tour] >>.*

This query illustrates the use of nested queries. There are many ways to produce this query, the one illustrated here will produce the shortest AQUA string.

First, the sets **Places_to_go** and **Tours** are displayed from the global sets list. Then the set **Tours** is put inside the **Places_to_go** set, producing a nested query.

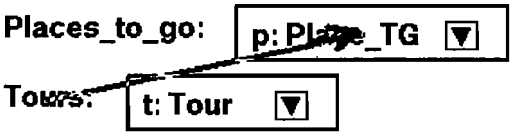


Figure 16: Nesting Tours within Places_to_go

Then the **Tours** set can be selected such that p is in the description set of t. The user clicks on the "mem(=,)()" shortcut button on the selection predicate dialog box, and positions the cursor between the comma and the end parentheses. The user clicks on **p:Place_TG**, and "p" is inserted into the dialog box. Then the user positions the cursor between the two parentheses at the end and clicks on **what:Place_TG** within the description set. The final predicate is:

mem(=,p)(apply (lambda (d) d.what) (t.description))

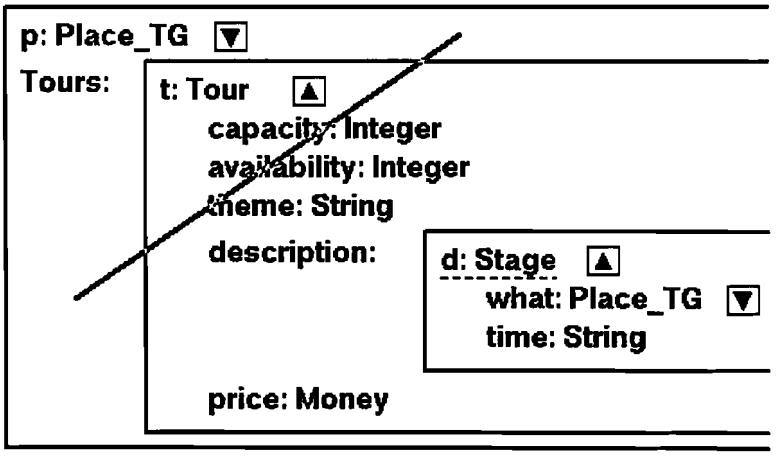


Figure 17: Selecting the Tours set relative to p:Place_TG

Now a new set of **Tours** is chosen from the global sets list. This set is selected so that its capacity is greater than 20, and then placed within the **Places_to_go** set. Doing the selection before adding the set to the tuple produces a shorter query.

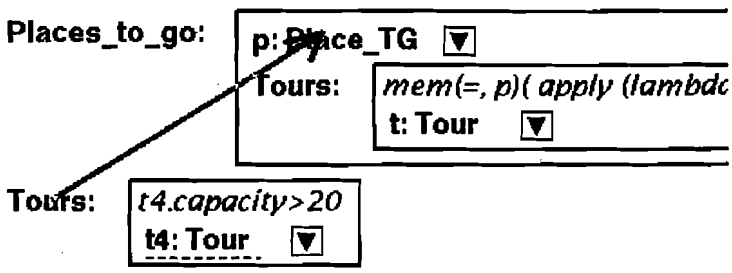


Figure 18: Adding a second set of Tours to the tuple

When the second Tours set is added to the tuple there is a duplicate name, so the second Tours becomes "t5". The user could have renamed either tours set before this gesture to avoid the renaming. However, in this example we will rename the tuple fields at the end of the query. Now the themes of the two sets of tours need to be compared. The user selects the "t5" set so that t4.theme is in the set of themes in the "Tours" set. Again the user uses the "mem(=,)()" shortcut button on the selection dialog box; this time clicking on the theme member function in each of the sets of tours. When the user clicks on the theme item in the set "Tours", the AQUA representation for the tuple field "Tours" is included in the **mem**. This enables us to delete the field "Tours" now.

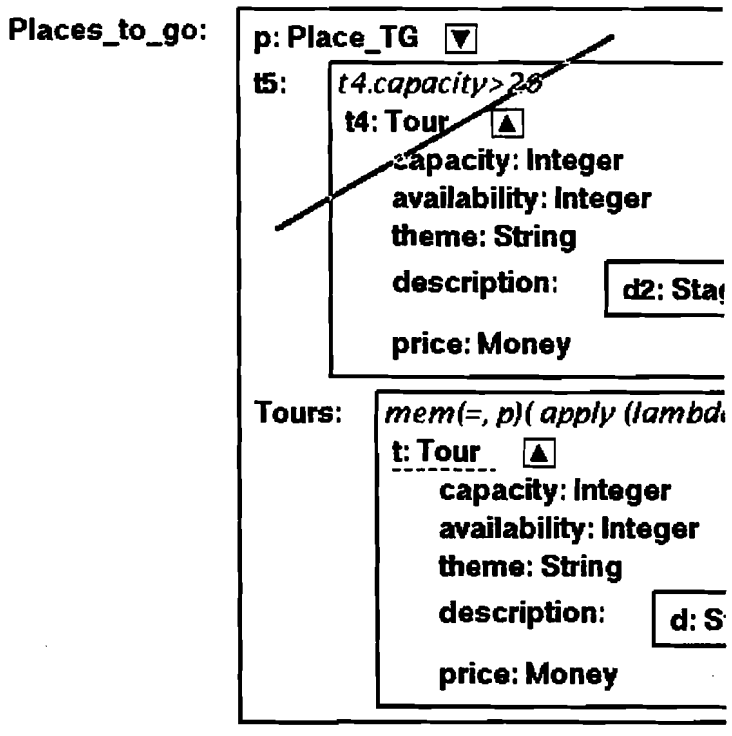


Figure 19: Selecting "t5" set of tours comparing themes with Tours

The set is deleted by first clicking on the "Tours:" string, the set is then highlighted with an dotted underline, and then pressing the trashcan button in the bottom right of the display.
Only two fields are left in the tuple, "p" and "t5". The user renames them to be "place" and "tours"; and then clicks on the outside set "Places_to_go." The AQUA syntax for this query is displayed in the message box and can be output to the query visualizer.

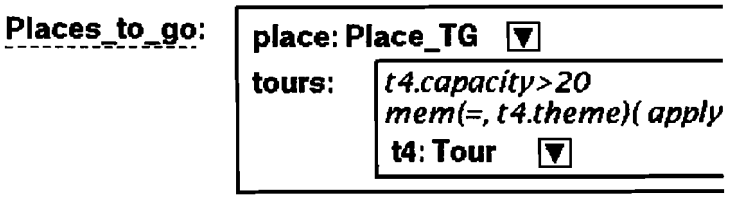


Figure 20: The final tuple

The AQUA query produced by silVIA for this example is:

```
apply(lambda(p) <<place:p,
   tours:select (lambda(t4) (mem(=, t4.theme)( apply (lambda (t) t.theme)
        (select (lambda(t) (mem(=,p )(apply (lambda (d) d.what)  (t.description) )))
         (Tours)))))
       (Tours)>>)
(Places_to_go)
```

6.0 Conclusions/Future Work

[BWFH92] writes that "For the casual user, the greatest obstacle to query formulation is typically poor knowledge of the database's constituent structure. If one does not understand what is in the database, then it is difficult to anticipate the correct table and attribute headings."
An advantage of a visual interface is that it helps the user understand the database's structure. The database's objects and member functions can be displayed so that the user can see her options and experiment with different techniques for getting at the information she wants. Even without producing a query, the silVIA interface could be used to explore paths through the database, and see how objects and instances are related. When trying to formulate a query, an exact understanding of the AQUA syntax is not necessary. As the AQUA syntax continues to evolve, only changes to the program's output have to be made, the gestures that a user has learned can remain the same.

Further work on silVIA could introduce new visual cues for the varied types of bulk types supported by AQUA. Multisets, lists and trees maintain the many-to-one relationship of sets, but their special characteristics warrant unique visual representations.

To complete the visual environment offered by the combination of silVIA for creating queries and GROOVE for visualizing the query optimization, a method should be developed to display the result of the query. [Cru93] describes a system for user-defined visual languages for the display of data. In this way, the user can define rules for how to display the results of the query. An alternative style of display could be based on the type of the result. Each type within the schema would have a function that displayed its data. For instance, a instance of the **Place_TG** type contains a name, a picture, and an address; so its display function would create a form to arrange the information and then call the display functions for types **String, Bitmap**, and **Address**. A **String** may be displayed as a text field, a **Bitmap** as a pixmap, and the **Address** field would create another form to display its results. The inheritance and abstract data type principles of object-oriented paradigms could be used in the display of information.

The silVIA interface expands the visual environment that began with the GROOVE optimization visualizer. Further work could lead to unified visual system for query formulation, submission and display of results.

References

[BWFH92] Hans Brunner, Greg Whittemore, Kathleen Ferrara, Jianiene Hsu. An Assessment of Written/ Interactive Dialogue for Information Retrieval Applications. In *Human Computer Interaction*, 7, 1992, pp.197-249.

[CM92] Mariano P. Consens, Alberto O. Mendelzon. HY$^+$:A Hygraph-based Query and Visualization System. In *ACM SIGMOD*, March 1992, pp.511-516.

[Cru89] Isabel F. Cruz. Declarative Query Languages for Object-oriented Databases. F. H. Lochovsky, editor, *Office and Database Systems Research*, 1989.

[Cru92] Isabel F. Cruz. Doodle: A Visual Language for Object-oriented Databases. *Proceedings of the ACM SIGMOD*, 1992, pp. 71-80.

[Cru93] Isabel F. Cruz. User-Defined Visual Languages for Querying Data. Technical Report CS-93-58, Dept. of Computer Science, Brown University, 1993

[GPV90] Marc Gyssens, Jan Paredaens, Dirk Van Gucht. A Graph-Oriented Object Model for Database End-User Interfaces. *ACM SIGMOD Record*, May 1990, pp. 24-33.

[GSKZ85] K.J.Goldman, S.A. Goldman, P.C. Kanellakis, and S.B. Zdonik. ISIS: Interface for a Semantic Information System. In *ACM-SIGMOD International Conference on Management of Data*, 1985, p. 328-342.

[Kim88] H.J. Kim, H.F. Korth, and A. Silberschatz. PICASSO: A Graphical Query Language, *Software: Practice and Experience* 18, 3, 1988, pp.169-203.

[KKD89] K. Kim, W. Kim, A.Dale. Cyclic Query Processing in Object-oriented Databases. In *IEEE Intl. Conference on Data Engineering*, 1989.

[KL89] Michael Kifer, Georg Lausen. F-Logic: A Higher-order Language for Reasoning about Objects,

Inheritance, and Scheme. In *ACM-SIGMOD*, June 1989, pp.134-146.

[LMS+93] T.W. Leung, G.Mitchell, B. Subramanian, B. Vance, S.L.Vandenberg, S.B.Zdonik. The AQUA Data Model and Algebra. In *Proc. 4th Int'l. Workshop on Database Programming Languages*, New York, New York, August 1993, Springer-Verlag.

[MS95] Kevin Mullet, Darrell Sano. *Designing Visual Interfaces, Communication Oriented Techniques.* SunSoft Press, Mountain View, California, 1995.

[Shn93] Ben Shneiderman. Dynamic Queries for Visual Information Seeking. Technical Report SRC-TR-93-3, Dept. of Computer Science, University of Maryland, 1993.

[VAO93] K.Vadaparty, Y.A. Aslandogan, G. Ozsoyoglu. Towards a Unified Visual Database Access. In *ACM-SIGMOD*, June 1993, pp.357-366.

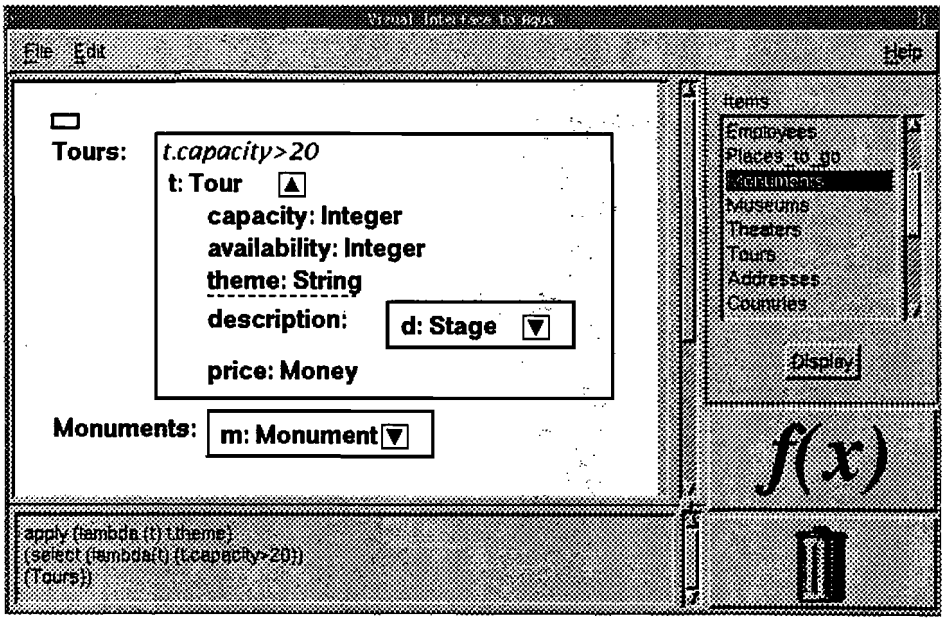
[Zlo75] M. Zloof. Query-by-Example. *Proc. 1975 National Computer Conference*, June 1975.

silVIA - Visual Interface to AQUA

Design Document

Brian Anderson
March 21st, 1995

This document describes the objects, data structures, and flow of control for the Silvia program. I will give an overview of the program design and help point out what code is in each of the files, so that if you need to find something in the code, this paper should help you locate it and understand what the existing code is doing. I am assuming that the reader has used Silvia before, this paper describes how the program works; there is a user's manual in this directory that explains how to use the program.



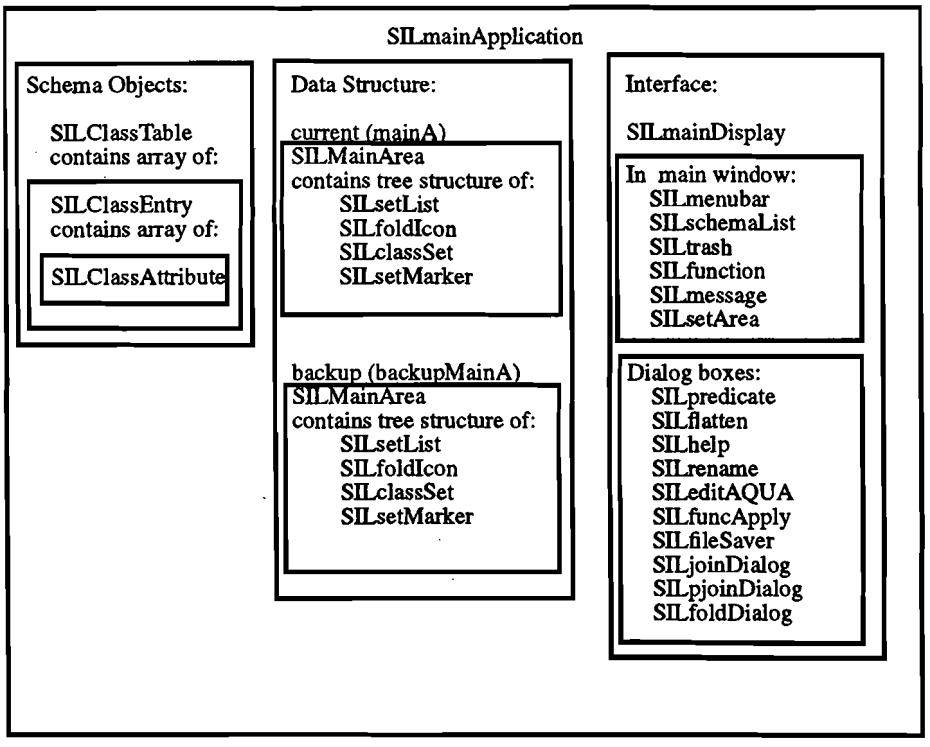
Overview of Classes and Objects

The main class in Silvia is a SILmainApplication. It contains all of the other objects and controls what actions should be taken at what time. The other objects each have a pointer to the one SILmainApplication. The other objects in Silvia are organized into three main groups: schema objects, interface objects, and the tree data structure.

When Silvia starts, it parses two files in the directory defined by EPOQ_SCHEMA_INI: TravelSchema and TravelGlobals. The data from TravelSchema is kept in the schema objects (SILClassTable), and the globals sets are kept in SILschemaList (the selection box on the right of the display).

The interface objects control the initial Motif appearance and the dialog boxes that appear along the way. They pass mouse events to the SILmainApplication, which actually performs actions on the data structure.

The main data structure is in the form of a tree. Within one SILMainArea object, a list of pointers to the top level sets in the work area is contained. These top level sets point to lists of items in the set. These, in turn, point to lists of member functions which could point to new objects and sets. The actual classes contained in this data structure will be described later.



Schema Objects

The code for the objects that keep track of the schema are in SILschema.C The SILmainApplication contains one SILClassTable, which contains an array of SILClassEntry. Each SILClassEntry represents one type defined in the schema file.
A SILClassEntry contains the name of the type, a parent type that it inherits member functions from (if there is one), and an array of SILClassAttribute, one for each member function on this type. Each SILClassAttribute contains the name of the member function, the type returned by this function, and whether the return type is a set.
When the user expands a type in the work area display by clicking on the arrow icon, the ClassTable tells how many member functions are defined for this type and what the member functions are. The SILmainApplication calls the function SILclassSet::Expand on the instance being expanded and passes in a copy of the SILClassTable. The SILclassSet can then get the information out of the SILclassTable.

SILClassTable : represents the schema with an array of SILClassEntry, one for each type in the database

SILClassEntry : represents one type in the database; has fields for name of the type and name of the parent type if one exists. Contains array of SILClassAttribute, one for each member function on this type.

SILClassAttribute : represents a member function; contains the name and the return type of this function.

An example TravelSchema file is included in the Appendix.

Interface

The Motif interface is separated out in the SILmainDisplay object.
To start the application, SILmainApplication calls SILmainDisplay::OpenApplication. This creates the main interface widgets. The code for these objects is in the file SILdisplay.C, except that the code for the work area (SILsetArea) is in SILsetArea.C. Here is a list of which objects control what part of the interface:

SILmenubar : the main menubar at the top of the display (File, Edit, Help menus)
SILschemaList : the global sets list on the right of the display
SILtrash : the trashcan button in the bottom right
SILfunction : the function/fold button above the trashcan button
SILmessage : the message box at the bottom of the display
SILsetArea : the work area where the tree structure is displayed

When a dialog box needs to be displayed, the SILmainApplication tells the SILmainDisplay object to pop up the appropriate dialog box. The SILmainDisplay object then routes the message to the object responsible for that type of dialog box. Here is a list of the dialog box objects and which file they are in.

SILpredicate : used for entering a selection predicate (SILdisplay.C)
SILflatten : asks the user whether and how much to flatten a set of sets (SILdisplay.C)
SILhelp : displays list of help topics, with text and a pixmap for each topic (SILhelp.C)
SILrename : asks for new name for a set or tuple field (SILdialogs.C)
SILeditAQUA : allows user to edit AQUA string before saving, or change AQUA representation of an instance in the tree structure (SILdialogs.C)

SILfuncApply : displayed when f(x) button is pressed when an instance is highlighted, asks for function to apply and the name and type of the result (SILdialogs.C)

SILfoldDialog : displayed when the f(x) button is pressed when a set is highlighted, asks for conditions of fold to apply and name and type of result. (SILdialogs.C)

SILfileSaver : file selection box that chooses filename in which to save AQUA query. (SILdialogs.C)

SILjoinDialog : asks what type of action to take when pairing two sets: union, intersection, difference, cross-product, or equi-join (SILdialogs.C)

SILpjoinDialog : If equi-join was chosen from the SILjoinDialog, this is displayed, asking for predicates for the join and which fields to include in the resulting tuple (SILdialogs.C)

Tree Data Structure

The display in the work area is based on a tree data structure of objects that are subclassed from the SILset class. There are three subclasses: SILfoldIcon, SILclassSet, and SILsetMarker. Each one of these classes contains an object of type SILsetList, which contains an array of pointers to more SILset instances.

The base of the tree data structure is an object of type SILMainArea. The SILmainApplication contains two SILMainAreas, one for the currently displayed structure, and one for a backup so that the user can undo an action.

The top level of the tree structure is a list of global sets. The children of these sets represent the type of objects within the sets. If the type has member functions defined in the schema file, then these functions are placed in the data structure as children of this type. These member functions can return sets or other abstract data types which, in turn, contain their own subtrees.

SILfoldIcon

The New Set Icon in the upper left of the work area display is represented by an SILfoldIcon. There will only be one SILfoldIcon in the tree structure, and it is the first item in the SILMainArea's list of children. It is drawn as a small, empty rectangle.

SILsetMarker

A SILsetMarker represents a set of objects. Its SILsetList points to the instances within the set. The SILsetMarker is drawn by writing the set's name and then a rectangle that surrounds the contents of the set. The children of the SILsetMarker are drawn within this set rectangle.

When an SILsetMarker is first created, it has one child representing the type of object within the set. After a join or the creation of a tuple, the SILsetMarker will have more than one child, one for each field in the tuple.

SILclassSet

A SILclassSet represents a single instance of a type. It is drawn by writing its name, followed by a colon, and then its type. The children of a SILclassSet represent the member functions for this type. The children are drawn below the name and type of the SILclassSet.

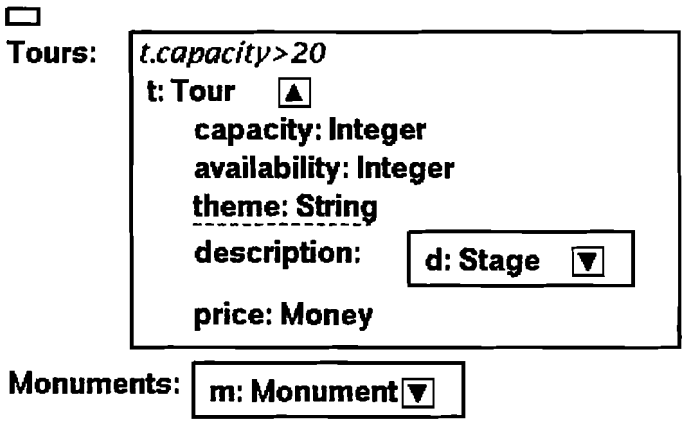
When the SILclassSet is the only child of a SILsetMarker, its name field is the lambda name applied to each element in the set and its type field is the type of the objects within the set.

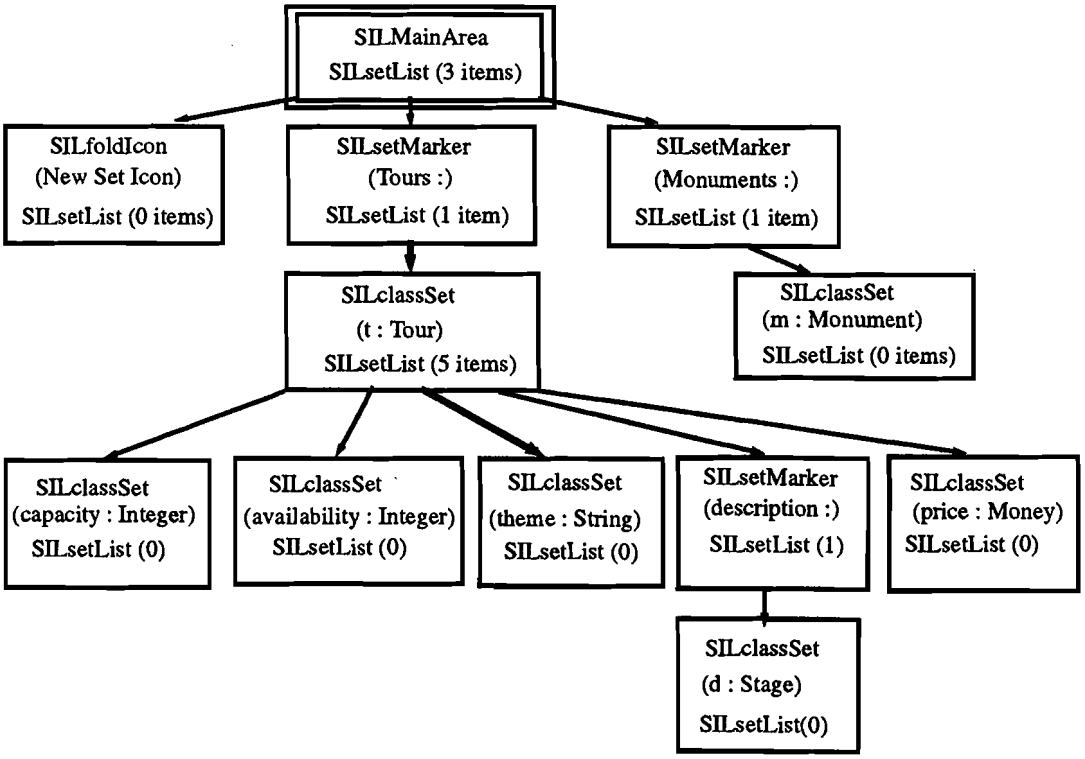
When the SILclassSet is a child of a SILclassSet, its name field is the name of a member function and its type field is the return type of the function.

SILsetList

Each of the three classes above contain a SILsetList object. This class does not contain any information about the database or the queries, it simply manages the list of pointers to child SILsets.

To illustrate how the display in the work area is represented in the tree structure; here is an example work area display and the underlying tree structure.





Each node in the tree keeps track of its own representation in the AQUA syntax. A path in the tree creates an AQUA query. When actions are taken, the AQUA representation for the sets at the top level are changed. For instance, in the diagram above, a selection has been taken on the Tours set; so the AQUA representation of the set is now: *select(lambda(t) t.capacity>20)(Tours).* A path from theme to the SILMainArea would produce the query: *apply(lambda (t) t.theme)(select(lambda(t) t.capacity>20) (Tours).*

Mouse Events

When a mouse event occurs in the work area, the SILsetArea passes the X event to SILmainApplication::SetAreaEvent(). The main application then passes the x and y locations to either SILMainArea::MouseDown() or SILMainArea::MouseUp(). This causes a traversal of the tree to find on which SILset the mouse event occurred. Each SILset keeps track of its upper left point in the display and how much space it needs to draw itself. When the MouseDown function is called, the SILset checks to see if the point is within its area. It also passes the point to its children to see if the mouse event happened on one of the children.
A SILaction structure is passed along with the x and y locations to these functions. The traversal of the tree fills in the fields of this structure. On a MouseDown event, the traversal creates the AQUA string for the path from the down instance to the top level of the tree. This AQUA string is displayed in the message area and can be saved as the query.
When the gesture is interpreted to be a selection, then one more traversal, MouseMidpoint(), is made to find the set that is underneath the midpoint of the line. This is the set that will be selected.

SILaction

The SILaction structure records the information about a mouse gesture that the SILmainApplication needs to decide what action to take. Since this is an important structure in the understanding of the Silvia code, a description of the fields is given. The section "Interpreting the Gesture" will illustrate how these fields are used. I will use the phrase "down instance" to refer to the SILset on which the down event occurred, (and "up instance" for the up event's SILset).

SILset *up : a pointer to the up instance
SILset *down : a pointer to the down instance
SILset *mid : a pointer tothe SILset that is to be selected (midpoint of the gesture line)
SILsetList *downparent : the SILsetList that points to the down instance
SILsetList *upparent : the SILsetList that points to the up instance
SILsetList *midparent : the SILsetList that points to the SILsetMarker that is to be selected
SILsetList *splitpoint : the SILsetList where the path to the up instance and the path to the down instance split. This can be NULL if the up and down instances are the same, or if they are different lengths down the same path.
SILsetMarker *lastSet : on the path from the top level to the up instance, this is the last SILsetMarker on the path.
int child_of_set : if the up instance is an immediate child of an SILsetMarker, this is set to 1; this is 0 if there was a SILsetMarker on the path but the up instance is not an immediate child of a SILsetMarker; this is -1 if there was no SILsetMarker on the

path.
int down_sets : the number of SILsetMarkers crossed on path from splitpoint to the down instance
int up_sets : the number of SILsetMarkers crossed on path from splitpoint to the up instance
int expand_icon : this is set to one if the mouse event was on the arrow icon of the down instance
char down_type[50] : the type of the down instance
char up_type[50] : the type of the up instance
char down_name[128] : the name of the down instance
char up_name[128] : the name of the up instance
char split_name[128] : the name of the midpoint (selected) set
SILpt upPt : the x and y location of the up event
SILpt downPt : the x and y location of the down event
char AQUAname[1024] : the AQUA string representing the query for the path taken to the down instance
int AQUA_insert_pos : gives the place to insert more AQUA code, when traversing up the path

When SILmainApplication::SetAreaEvent() gets a mouse motion event, it draws a line from the down point to the current mouse location. It uses an XOR style line so that the old lines can be erased when the mouse moves. When the mouse up event occurs, the line is written to the pixmap so that expose events will not erase it while a dialog box is being displayed.

Interpreting the Gesture

The data in the SILaction structure allows the SILmainApplication to decide what action to take. In general, the gesture involves extracting the down instance up to the splitpoint, adding sets for each SILsetMarker that it crosses on the way up and then pairing this new set with the up instance. Sometimes a traversal of the structure is done again from a different starting point to create a subtree path, and therefore get the AQUA string for an instance relative to another within the tree structure. I will describe the flow of control for each type of gesture, explaining how the AQUA representation of the tree is changed. After simple cases are explained, more complex gestures can be described in terms of combinations of the simpler actions.
The SILaction that the SILmainApplication uses with gestures is called "action", so from now on I will refer to fields in the SILaction as "action.up", "action.upparent", etc.

If the Up and Down Instances are the Same

If the user clicks on an instance, without dragging the line, the action.up and action.down fields will be equal, and action.splitpoint will be NULL. The SILset is highlighted by passing its pointer and its parent to SILmainArea, it will be displayed with a dotted underline. Also, the AQUA string in action.AQUAname is displayed in the message box, using SILmainDisplay::DisplayMessage().
If the type clicked on is a classSet, then the value of action.expand_icon is checked. If this is equal to 1, then the SILclassSet is expanded (compressed if already expanded.)
Expanding a set involves passing the SILClassTable to the SILclassSet. The SILclassSet looks up

its type in the SILClassTable and finds how many member functions its type has (if any). It then tells its SILsetList to create a new SILset for each member function. It passes the name, type, and a field for whether the function returns a set, to SILsetList::CreateItem(). If the set has already been expanded, then the field "expanded" is set so that the children are no longer drawn.

Selection

It is possible to draw a gesture line and still have the up and down instances equal. If the up and down events occurred on the same SILsetMarker, or both on the toplevel (action.up = action.down = NULL and action.up_type = action.down_type = "Toplevel"), then the midpoint of the gesture line is checked using MouseMidpoint. If the midpoint is over a set that is below the up instance in the tree structure, then this set is selected.
The SILmainApplication tells the display object to display the selection predicate dialog box (SILpredicate). The user enters a selection predicate and then the SILpredicate calls SILmainApplication::DoSelection(), passing the predicate. DoSelection passes the predicate to the SILsetMarker, which adds to its AQUAname so that the selection is recorded. The predicate is also stored in a character string array to be displayed in italics when the set is displayed.
While the predicate dialog box is displayed, the SILmainApplication sets its field "waiting" to signify that further mouse events will create AQUA text that should be passed to the SILpredicate. In this way, the user can click on items on the display and have the AQUA text appear in the dialog box. The AQUA representation is calculated relative to the selected set using SILMainArea::SubtreeMouseDown().

Extracting (Up on the New Set Icon)

A SILset within the tree structure can be pulled out to the top level by drawing a line from the instance to the New Set Icon in the upper left corner of the work area. The field action.AQUAname has already calculated the AQUA representation for this SILset relative to the toplevel during the MouseDown traversal. All that has to be done is to add a copy of the instance to the top level and set its AQUA representaion to equal action.AQUAname.
When extracting an instance to the top level, we need to preserve how many sets deep the instance is in the tree structure. The field action.down_sets keeps track of how many SILsetMarkers have been crossed on the way to the instance. If this is more than one, then the user has the option of flattening the set of sets to just a set of a type. SILmainApplication tells the display object to display the SILflatten dialog box. This dialog box will return how much to flatten the set to SILmainApplication::DoFlatten(). If the set is flattened, the action.AQUAname is changed to reflect the application of "flatten()".
Now we know how many sets to surround the instance that we are copying. The function SILsetList::AddItem is called on the SILsetList of the SILMainArea, passing a pointer to the down instance and an integer for how many sets to place around the copy of the instance.
If the mouse up event occurred on the New Set Icon the action is finished. However, this step of extracting the down instance to the top level is done whenever action.splitpoint does not equal NULL. The actions for union, join, and apply tuple all involve getting of a copy of the down instance to the top level temporarily. Then this set is incorporated into the up instance according to the type of the up instance, and the temporary set is deleted.
The exact flow of control involves calling SILmainApplication::AddDownSets() to get a copy of the up instance to the top level. This function may display the SILflatten dialog box if necessary. When the down instance has been brought to the top level, then the function SILmainApplica-

tion::RouteAction() looks at the SILaction structure to see what action to take next. In the case of the up event on the New Set Icon, the action is finished and the pixmap is refreshed.

Joins and Unions (Up on an SILsetMarker on the top level)

If the up event occurred on a SILsetMarker, then the types of the up set and the copy of the down set are compared in SILmainApplication::AskJoinType(). In this function, SILset::QueryType() is called on each of these SILsets. The type is returned as a string in a format like : "Set[city:City, hotels : Set[Hotel]]". This string would represent a set of tuples consisting of a City named "city" and a set of Hotel named "hotels". The file SILviewcode.C handles the parsing of this format; all functions is this file start with "VC "(ie. VCisTuple()). It could be possible in the future to save a query for later use by writing this format to a file, along with the set's name and its AQUA representation.

The function SILmainApplication()::AreSetsCompatible() is called to see how compatible these sets are. The functions for calculating the compatiblity are in the file SILcompatible.C

If the sets are compatible enough for a union, the supertype of the set types is returned so that it can be used in the application of the "union" function.

Then the sets are compared to see if an equi-join is possible. If both of these sets are tuples, then certain fields may be of compatible types without the whole set being compatible enough for a union. The function SILmainApplication::PartialJoinPossible() (in SILcompatible.C) calculates which fields are compatible enough and constructs a list of strings that could be used in the equi-join. The equi-join dialog box (SILpjoinDialog) will display these possible predicates and a list of the tuples that could be in the resulting tuple. These two character string arrays are passed to the SILpjoinDialog to set up its data, even if the dialog never ends up being displayed.

If the sets were not compatible at all, a cross product of the two sets is done automatically. If the sets were compatible, or an equi-join was possible, the join dialog box (SILjoinDialog) is displayed, asking the user which action to take.

When the user makes a selection, the SILjoinDialog calls the appropriate function in SILmainApplication. There are six possible functions that it could call:

DoUnionExactMatch() : the sets had exactly the same type, so the information from the down set is added directly into the up set.

DoUnionCompatible() : a new set needs to be created of the supertype of the two sets, then the information from the up and down instances are added to this set.

DoBinarySetOp() : an intersection or difference is done, the name of the set operator is passed in to this function. The resulting set needs to be the same type as the down instance, so the information from the up instance is added to the down instance.

DoPartialJoin() : display the equi-join dialog box and construct a predicate for the join.

DoJoin() : do a join with the default predicate "true"

AbortJoin() : the user pressed cancel, remove the temporary set.

The three functions DoUnionExactMatch, DoUnionCompatible, and BinarySetOp do similar things. They copy the predicates from the sets into whichever set will remain, change the set name for the resulting set to reflect the operation, remove the extra sets, and call SILset-Marker::AddUnionAQUA() passing in the binary set function name, the union type, and the AQUA name for the set to union with. Then the SILmainApplication function changes the AQUAname for the SILsetMarker to reflect the binary set operation.

Do PartialJoin() displays the equi-join dialog box, which will call DoJoin with the predicate that

the user has constructed.
DoJoin() takes three arguments: the predicate function ("true" if a cross product), an array of 1s and 0s for which fields to include in the resulting tuple (NULL if a cross product), and a count of the number of elements in this array of 1s and 0s. It joins the SILsetLists from the two sets, removing elements later if there are 0s in the array. It then constructs a join function based on the elements in the 1s and 0s array. It passes the predicate and the join function to SILset-Marker::AddJoinAQUA() which updates the AQUAname. It also changes the name of the set to represent the cross product and removes extra sets.

Applying Tuples (Up on a SILclassSet that is the immediate child of a SILsetMarker)

If the up event is on a classSet whose parent is an SILsetMarker, then the down instance is added to the set and a tuple is created. Again, the down instance is brought to the top level using SIL-mainApplication::AddDownSets(). Then SILmainApplication::RouteAction() passes the temporary copy of the down instance to SILmainApplication::DoApplyTuple() if action.child_of_set equals 1. At this point, action.lastSet points to the set that is the parent of the up instance. By calling SILsetMarker::GetSubObjects() on action.lastSet, we can get the SILsetList that contains the tuple fields. The name of the down copy is checked against the name in this tuple list to make sure there will be no duplicate names. Then a "name:AQUA" string is created for the down copy with "name" as the name of the down copy (or newname if there was a duplicate) and "AQUA" as action.AQUAname. This string is passed to action.lastSet, so that it can add the application of the tuple function to its AQUAname. The copy of the down instance is added to the tuple list (SILsetList::AddItem()) and its AQUAname is set to a unique lambda name.

Pushing the Data Out

Since the majority of the AQUA representation of the display is kept at the top level sets, it is sometimes necessary to push AQUA string information out to the top level. This happens in two situations, when a set in a tuple is selected and when a tuple is created (or added to) within a set in a tuple.
The function SILmainApplication::PushDataOut() takes the SILsetMarker, whose AQUA representation has changed, as a parameter. It checks to see if this set is an immediate child of another SILsetMarker. If this is the case, then it calls the function SILsetMarker::SelectionOnSetInTuple() on the parent set. This function looks at its down child and incorporates the new AQUA string for the down child into its own representation. Then PushDataOut() removes the AQUA string from the original set, replacing it with a unique lambda name. This process continues until the data is pushed out to the top level, or to a set that is a member function of a SILclassSet.
The tree data structure has no parent pointers, so the only way to tell if an SILset is the immediate child of a SILsetMarker is the action.child_of_set field. PushDataOut() uses a temporary SILaction structure to traverse the tree for the original changed set, looking for its parent set.

How the SILsetMarkers Change their AQUAname

The actual changing of AQUA strings occurs in the SILsetMarkers. The SILmainApplication simply says to add text for a union, selection, join, etc. and passes in the predicate and/or the AQUA string to union or join with. Most of the functions are straight forward additions to the existing AQUAname using sprintf, but there are a few occasions that are worth special mention.

Selection Shortcuts

If a set is selected, normally we just create a new statement: "select(lambda() predicate)(old AQUAname)" string putting in the predicate and old AQUAname. However, in some circumstances, the predicate can be added directly into the existing AQUAname.
If the set had been selected previously, then there already is a select operator in the AQUAname, so we can just add the new predicate in with the old by adding "and" and the new predicate.
Also, if the set is the result of a join, sometimes we can add the predicate to the join predicate that already exists in the AQUAname. If the previous predicate was "true" then we remove the "true" and add in the new predicate. In order to add in the predicate, we have to make sure that the lambda names for the fields are bound. This happens if there are only two fields in the tuple, otherwise one of the lambda names is bound to a tuple, and we cannot access member functions in fields in this tuple directly. The next section will discuss the problem of binding lambda names in tuples further.

Shortcut for Applying tuples

When we create a tuple through the gesture where the up instance is the child of a set, we normally add AQUA such that an application of the tuple function ("<< >>") is applied to the previous set. However, if we are adding a field to an already existing tuple, the "<<" and ">>" strings are already in the AQUA string (from either a join or a previous apply tuple).
We can then search for the "<<" string and add the new tuple field directly into the old AQUA name at this position. This shortcut is done in SILsetMarker::AddApplyTupleAQUA().

Selection after Apply Tuple

If we have just done an apply tuple, and the user selects the set within the tuple, then we should be able to put the new AQUAstring into the tuple definition. For instance, if we have this tuple:

apply(lambda(h2) <<h2:h2,Clients:Clients>>) (Hotels)

and the user then selects the Clients set within the tuple, we should be able to change the text after the "Clients:" string. The resulting AQUAname would be:

apply(lambda(h2) <<h2:h2,Clients:select (lambda(c2)
(c2.address.city=h2.address.city))
(Clients)>>)
(Hotels)

This shortcut is done in SILsetMarker::LookForSelectInTupleShortcut().

Shortcut for tup_select

When doing nested queries, the Silvia program generates tuples by default. However, a tuple may not be necessary for completing the query. We can do nested queries by looking for times that we are doing a tup_select immediately after doing an apply tuple. For instance, one way to get the Clients field out of the query above is to say "apply(lambda(x) tup_select(Clients)(x)) (... the query listed above...)". However, we could also just recognize that the AQUA for Clients is written right after the "Clients:" string within the tuple definition. So we could remove the tuple definition (from "<<" to ">>") and just leave the definition for Clients. The resulting AQUA string would be:

apply(lambda(h2) select (lambda(c2) (c2.address.city=h2.address.city))
(Clients))

(Hotels)

We now have produced a nested query without applying the tuple function. The function that looks for this shortcut is SILsetMarker::LookForAddTupSelectShortcut().

Deletion

There are some circumstances where an SILset cannot be deleted from the display; for instance, it does not make any sense to delete a member function from a type (although in the future a way to hide it, to compress the display, may be a good idea). Also, fields in tuples may not be able to be deleted if the AQUAname cannot be altered easily to reflect the deletion.
I decided to allow deletion of sets on the top level, and tuple fields if the tuple has more than two fields in it. Being able to delete tuple fields allows the user to create a tuple by extracting the name and age field from a student and then delete the original "Student" type. This will leave just a <name,age> tuple.
SILsetMarker::RemoveTupleField() searches the AQUA string for the definition of the field within "<<" and ">>". It then removes the section of AQUA text that belongs to that field. Removing a set at the top level involves removing the SILsetMarker from the SILMainArea's SILsetList using SILsetList::ExtractItem().

Menu items

The choices in the menubar follow a similar pattern. The SILmenubar calls the appropriate function in SILmainApplication, which usually tells the SILmainDisplay object to display a dialog box. The SILmainApplication sets its "wiating" field so that mouse events will be ignored when the dialog box is displayed. The dialog box calls a function in SILmainApplication when OK or Cancel is pressed. The SILmainApplication then changes the data structure accordingly. I will give an example for "Rename" in the "Edit" menu; the other menu items respond in similar ways. Clicking on "Rename" calls the Edit_cb() callback, passing in that Rename was chosen. Edit_cb() calls SILmainApplication::PopRename() passing the name of the highlighted SILset. PopRename calls SILmainDisplay::PopRename(), which sets up the SILrename dialog box and then displays it. The application returns to the XtMainLoop(), until the user clicks on OK or Cancel. Then the SILrename callback calls SILmainApplication::Rename() passing in the new name. Rename() changes the name for the highlighted SILset.

The apply function/fold button above the trashcan button follows a pattern similar to the menubar items. The only special concern of applying a function is that if the highlighted instance is the child of a set, then a tuple is created. This is handled using the same function as when the user makes an apply tuple gesture: SILmainApplication::DoApplyTuple().

Conclusion

This paper has just given an overview of the objects and the flow of control for the Silvia program. A more detailed explanation of the actual steps followed in each function is contained in the comments within the code itself. This paper should help point you to the right files and func-

tions, and the overview should help you to understand what the code is doing when you look at the *.C files yourself.

Appendix

Format for the "TravelSchema" file:

Class Place_TG : none (3 attr)
attribute name : none none String
attribute picture : none none Bitmap
attribute address : none none Address

Class Monument : Place_TG (2 attr)
attribute building_date : none none Date
attribute architect : none none Person

Class Bitmap : none (0 attr)

Class Country : none (2 attr)
attribute name : none none String
attribute cities : set none City

The string after "Class" defines the name of the type; after the colon is the parent type or "none". In this example Monument inherits from Place_TG. In parentheses is the number of member functions. The name of the member function is listed after "attribute", then a colon, then what type of bulk type the function returns ("none" if a single instance). The last item on the line is the base type of the result. For instance, Country.cities returns a set of type City.
This file is read and parsed in SILClassTable::ReadFile().

Format for the "TravelGlobals" file:

Cities :
Type : set none City
Size : 100 100
SKS : NIL
Range : -1 -1
SizeSSet : -1 -1

Monuments :
Type : set none Monument
Size : 10 10
SKS : Places_to_go
Range : -1 -1
SizeSSet : -1 -1

Most of the information in this file is for the query optimizer. The only information that the Silvia program uses is the name of the set and what type the set is. For instance, for "Cities :", Silvia would find that this is a set of City called "Cities". The rest of the information is skipped over.
This file is read and parsed in SILschemaList::LoadSchema().