Exploring Terra Incognita in the Design Space of Query Devices

Christopher Ahlberg & Staffan Truvé

Department of Computer Science and SSKKII
Chalmers University of Technology
S-412 96 Göteborg
Phone: +46 31 772 5410
Email: {ahlberg,truve}@cs.chalmers.se

June 6, 1995

To be published in”Engineering for Human-Computer Interaction”,
C. Unger, ed., IFIP Transactions series, Chapman & Hall, 1995

Exploring Terra Incognita in the Design Space of Query Devices

C. Ahlberg & S. Truvé

Department of Computer Science & SSKKII
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Phone: +46 31 7725410
Fax: +46 31 165655
Email: {ahlberg,truve}@cs.chalmers.se

Abstract

This paper introduces query devices, an extension to the widget concept. Query devices are graphical database interaction objects such as rangesliders and toggles. The design space of query devices is structured and it is shown how the design space can be explored so that consistent and distinguishable query devices are created. Existing query devices are placed in the design space and new ones suggested. A set of query devices aimed for dynamic queries systems is presented.

Keywords

Dynamic queries, visual query language, visual information seeking, query device, widget, design spaces, morphological analysis, human-computer interaction.

1 INTRODUCTION

Searching large amounts of information is an increasingly important task in today's society. Computers have made it possible to store massive amounts of information effectively. However, users of this information are left behind. User interfaces to online services, information retrieval systems, and database front-ends all too often suffer from problems with inappropriate presentation of information, complex query languages, and little guidance for novice users.

Recent advances in information seeking has emphasized the use of visual presentations of databases and query results in information retrieval systems (Ahlberg & Shneiderman, 1994)(Shneiderman, 1994)(Robertson, Card, and Mackinlay, 1993)(Eick, Nelson, and Schmidt, 1994). Visual presentations provide many advantages:

- Query results can quickly be judged for relevance.

- Very large databases can be overviewed in one single screen.
- Visualizations provide users with a well defined context and basis for the often highly interactive and iterative search.
- Allows for exploration of data, discovery of trends and patterns, detection of anomalies in complex datasets, etc.

In direct manipulation based database systems, interactive methods for controlling query results are as important as visualizations themselves. A particular approach to direct manipulation database search, dynamic queries, makes heavy use of widgets such as the rangesliders and toggles in Figure 1 (Ahlberg, Williamson & Shneiderman, 1992)(Ahlberg & Shneiderman, 1994). In the particular example in Figure 1 users manipulate rangesliders and buttons and can immediately (within 100 ms) observe query results (homes in Washington D.C) in a geographic map.

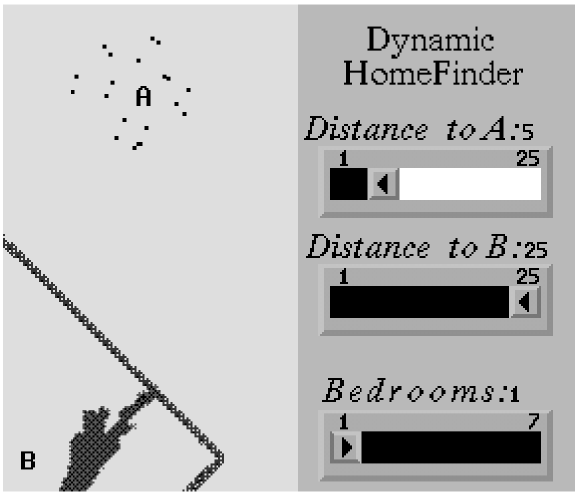

Figure 1 Examples of query devices used in the Dynamic Homefinder dynamic queries prototype (Williamson & Shneiderman, 1992).

The aim of this paper is to provide structure to the design space of such graphical interaction objects used in direct manipulation database systems. A notion for graphical database interaction objects is introduced, query devices – an extension of the widget concept. Widgets are the traditional parts of graphical user interfaces, such as scrollbars, pushbuttons, and sliders. When there is a close relationship between widgets and some underlying data structure, such as a database, we conjecture that it is fruitful to introduce a class of objects which includes both graphical appearance and relations to the database (e.g. selection behavior). Typical query devices are the rangesliders for range selection in Figure 1.

The paper makes two contributions:

- The design space of query devices is structured using primitives such as widget behavior, selection behavior, and graphical appearance, which can be composed using a small number of composition operators. The design space description guides the design, and improves consistency and distinguishability between query devices.

- The design space of query devices is explored - by fitting in existing ones into a taxonomy and suggesting new ones. These query devices are intended for use in dynamic queries interfaces.

1.1 Exploring and describing design spaces

Exploring and describing design spaces can be done in several ways:

- Task driven – tasks ask for certain designs and when the need for a new design is encountered it is created – i.e. largely an ad-hoc approach.
- Analytically driven – the space of possible designs is structured and explored in some appropriate way.

The task driven approach to creating user interfaces is usually recommended in the literature (Diaper, 1989). A task driven approach relies on task analysis and will in principle guarantee that a new design will fit the task. But this approach can be quite slow, and maybe more importantly, without keeping other possible variations in mind, there is a risk that the designs created will not be consistent with each other. Especially in human-computer interaction this is serious, as we usually want to strive for both consistency and distinguishability in our designs.

Mackinlay (1986), Mackinlay, Card & Robertson (1990), and Card, Mackinlay & Robertson (1991) argue for exploring a design space by organizing emerged designs in terms of abstractions that give insight into the design space. They focus on input devices (Mackinlay *et.al*, 1990)(Card *et.al*, 1991) and graphical presentations (Mackinlay, 1986) and describe designs as points in a parametrically described design space. To do this, parametric representations are determined which represent the central idea of particular designs. Similar approaches have been taken by Bertin (1983) when describing graphics and Zwicky (1967) in the generation of the design space of jet engines.

By parametrically exploring the design space of some class of designs of user interface objects, consistency and distinguishability will be much easier to reach. Designs can be grouped into families and new designs can be suggested. However, it is important to realize that even if the design space is explored in a formal way it is still necessary to design the graphic appearance and behavior of the particular points in the design space suggested by the formalization – a non-trivial, traditional human-factors task.

To parametrically describe a design space we need both a set of primitives and a set of composition operators which can be used to construct complex objects from the primitives. For example (Mackinlay *et.al* 1990) in their analysis of the design space of input devices introduce 1) simple input devices characterized by a six tuple describing attributes such as how the device is manipulated and the resolution of the device and 2) composition operators which can be used to combine simple input devices into complex ones.

2 GENERATING THE DESIGN SPACE OF QUERY DEVICES

Query devices select tuples from databases based on query criteria (Figure 2). Users manipulate a widget part of a query device to trigger the selection of database subsets. Query devices have a graphical appearance and behavior, both static and dynamic – depending on how they select tuples and what selection criteria can be specified interactively.

Query devices will be described as compositions of three types of primitives, widget primitives **W** which abstractly describe the interaction possible with the query device, selection primitives **S** which do the actual database filtering, and graphical primitives **G**, i.e. the actual look and feel of the query device. Sample compositions are presented in Figure 6.

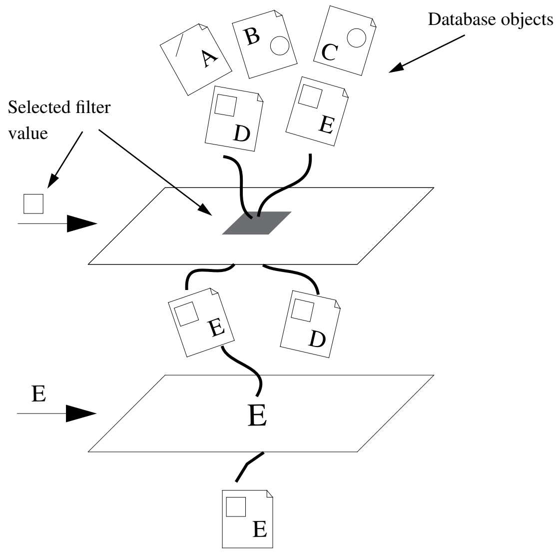

Figure 2 Model of query devices in context. Each query device selects a subset of a set of database tuples.

2.1 Basic definitions

Query devices select subsets of databases. Only databases consisting of a single relation are considered here. This is a limitation – however, for many cases databases holding multiple relations can be transformed into one single universal relation – which actually is considered to be a desirable approach from the user perspective (Kim, Korth, and Silberschatz, 1988).

Accordingly we define a database as a relation *r* of a relation schema *R*. A relation is defined as a subset of $A_1 \times A_2 \times ... \times A_n$, where A_i is some domain (*Int*, *Real*, *String*, or *Set* of either *Int*, *Real* or *String*) (Elmasri & Navathe, 1989). A database object is a tuple $t_i \in A_1 \times A_2 \times ... \times A_n$.

2.2 Query device construction primitives

Selector function S

S is the selector function which do the actual database filtering, given as a function type and a predicate. For example the selector of the top rangeslider in Figure 1 selects those homes *t* for which the attribute *t.DistanceFromA* are less than the value selected with the slider, *r*.

Selector functions **S** take a value (a *Basic* ranging over {*Int*, *Real*, *String}*, or a *Range* ranging over <*Basic, Basic*>, or a *Set of Basic,* or a *Set of Range*) and a *Set of Tuple* (i.e. a database) and return the set of tuples fulfilling the criterion set by the query device. The following selector function types exist:

$(Basic \times Set\ of\ Tuple) \rightarrow Set\ of\ Tuple$
$(Set\ of\ Basic \times Set\ of\ Tuple) \rightarrow Set\ of\ Tuple$
$(Range \times Set\ of\ Tuple) \rightarrow Set\ of\ Tuple$
$(Set\ of\ Range \times Set\ of\ Tuple) \rightarrow Set\ of\ Tuple$

The type of **S** is not enough to describe the selecting behavior. Already for a query device allowing users to select a single integer numerous semantics are possible. For a richer description of ***S***, predicates will describe the exact behavior of the selector function.

The *BasicQuery* corresponds to the simplest query, i.e. selection of those database tuples t where the attribute $t.A$ equals to a user selected value c, or those where the attribute $t.A$ is not equal to c. A *BasicQuery* might be performed with the widgets in Figure 4.

$$
\begin{aligned}
BasicQuery ::= \; & t.A = c \mid \\
& t.A \neq c
\end{aligned}
$$

The *RangeQuery* corresponds to various queries where a range is specified. Typical range queries are performed with the query devices in Figure 1. Below, $min(t.A)$ and $max(t.A)$ correspond to the minimum and maximum values of the attribute $t.A$.

$$
\begin{aligned}
RangeQuery ::= \; & min(t.A) \text{ op } t.A \text{ op } r \mid \\
& r \text{ op } t.A \text{ op } max(t.A) \mid \\
& r_1 \text{ op } t.A \; op \; r_2 \mid \\
& \neg\, (r_1 \text{ op } t.A \text{ op } r_2) = (min\,(t.A) \text{ op } t.A \text{ op } r1) \vee (r_2 \text{ op } t.A \text{ op } max(t.A))
\end{aligned}
$$

$$op ::= \; < \mid \leq \mid > \mid \geq$$

The *RangeSetQuery* corresponds to queries where multiple ranges can be selected (*rs* is the set of ranges selected with the query device).

$$
\begin{aligned}
RangeSetQuery ::= \; & \exists (r_1,r_2) \in \; rs \mid r_1 \text{ op } t.A \; op \; r_2 \mid \\
& \exists (r_1,r_2) \in \; rs \mid \neg(r_1 \text{ op } t.A \; op \; r_2)
\end{aligned}
$$

The *SetQuery* corresponds to queries where multiple values can be selected, either of the *Int*, *Real*, or *String* types. A typical widget which allows for a *SetQuery* would be the group of checkboxes in Figure 3.

$$
\begin{aligned}
SetQuery ::= \; & s \subseteq t.A \mid \\
& s = t.A \mid \\
& s \cap t.A \neq \varnothing \mid \\
& s \cap t.A = \varnothing
\end{aligned}
$$

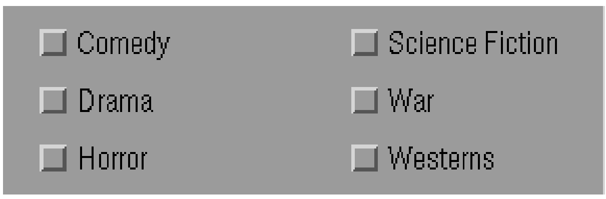

Figure 3 Checkboxes allowing for selection of several film categories.

Notice the similarity between *RangeQueries* and *SetQueries*. Both actually allows users to select a set of values which then is used to decide which database tuples that fulfill the query. However, the distinction is motivated by the fact that ranges can be used to select a large set of values easily. More variations are obviously possible, the given expressions are somewhat arbitrarily selected to be a minimal, intuitive set of selectors which has grown out of four years of research on dynamic queries. By composition of these primitives other query types can be created.

Widget function ***W***

Selector functions **S** need a value to be able to the determine which database tuples should be selected. Users specify this by interacting with the query device – i.e. by manipulating the widget part of the query device, denoted by **W**. **W** is an abstraction (a function) of the actual widget, and several different widgets might have the same widget function (Figure 4).

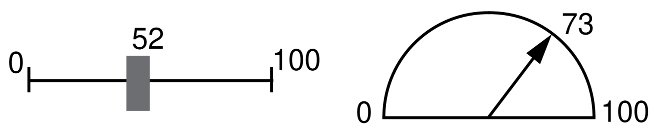

Figure 4 Both the slider and dial widgets deliver an integer when users interact with them.

Widget manipulation creates events. Events of the following types are introduced for now:

Event ::= SelectEvent | *HorizontalDragEvent* | *VerticalDragEvent*

Obviously this is not a very rich description of possible events in a window system. However, only events leading to meaningful actions in a query device are considered, i.e. either a button part of a query device is selected/deselected or a dragable part dragged. More event types can easily be added to provide a richer description. Possible widget function types are:

Event → *Basic*
Event → *Range*
Event → *Set of Range*
Event → *Set of Basic*

Graphical appearance ***G***

G can partially be derived from **S** and **W**, but the design is not a function of the requirements stated by **S** and **W**. Therefore, although the pair <***S,W***> fully describes the functionality of the query device and defines a point in the design space of query device functionality, **G** is made part of the definition of points in the design space. **G** will be given as an image – for a more concise description Interface Object Graphs could be used (Carr, 1994).

Many of the designs proposed below are based on variations of slider widgets. These are quite useful in direct manipulation based database systems (Ahlberg *et.al*, 1992). However, other variations are obviously possible, the dial in (Figure 4) which can substitute for the slider in the same figure is just one example.

2.3 Composition operators

The set of primitives provides the basic language for describing query devices and their behavior. To create the actual query devices, composition operators are introduced. Also, query devices can be composed into groups of query devices for use in database query systems - e.g. Figure 1 and Figure 5.

The following composition operators are introduced:

- $\otimes$: composes a selector function **S** and a widget function **W** into a query device. Legal combinations are those where **S** and **W** can be type-correctly composed by function composition.
- $\times_W$: composes two widget functions into a complex widget allowing users to specify criteria for several attributes simultaneously.
- $\times_S$: composes two selector functions in a complex selector function setting criteria for several attributes.
- &: composes two query devices into a query group which can be used in a database system. A composition *A & B* can be combined with a query device *C* into a larger query group (Figure 5).

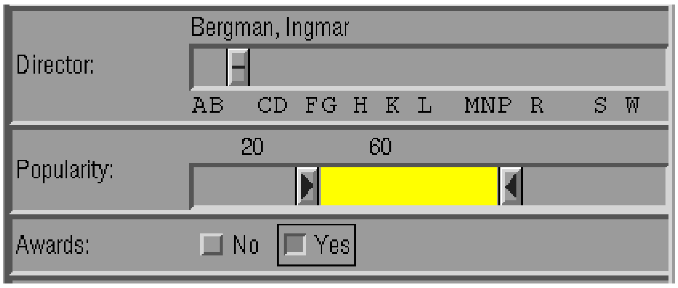

Figure 5 Group of query devices composed by the & composition operator.

Sample compositions can be found in the taxonomy in Figure 6. Widget abstractions are on the left and selector abstractions are on the right. Selector and widget composition with the $\otimes$ operator is represented by a thick black line between selectors and widgets. Compositions of widgets and compositions of selectors are represented by dashed lines.

A few other distinctions are also made in Figure 6:

- Widget and selector abstractions are categorized into different types they affect (*Int*, *Real*, and *String*).
- Widgets are informally categorized in terms of their resolution. By resolution we mean how many distinct values widgets can select from. The resolution of a toggle is one or two, a slider being 200 pixels wide has a resolution of 200 and a multigranularity alphaslider can have a resolution of over 10’000 (Ahlberg & Shneiderman, 1994).

Selector and widget composition

To create a query device which allows users to select all the tuples with an attribute *A* being in the range *<Int,Int>*, a selector function **S** $((\langle Int,Int\rangle \times \textit{Set of Tuple}) \rightarrow \textit{Set of Tuple})$ and a

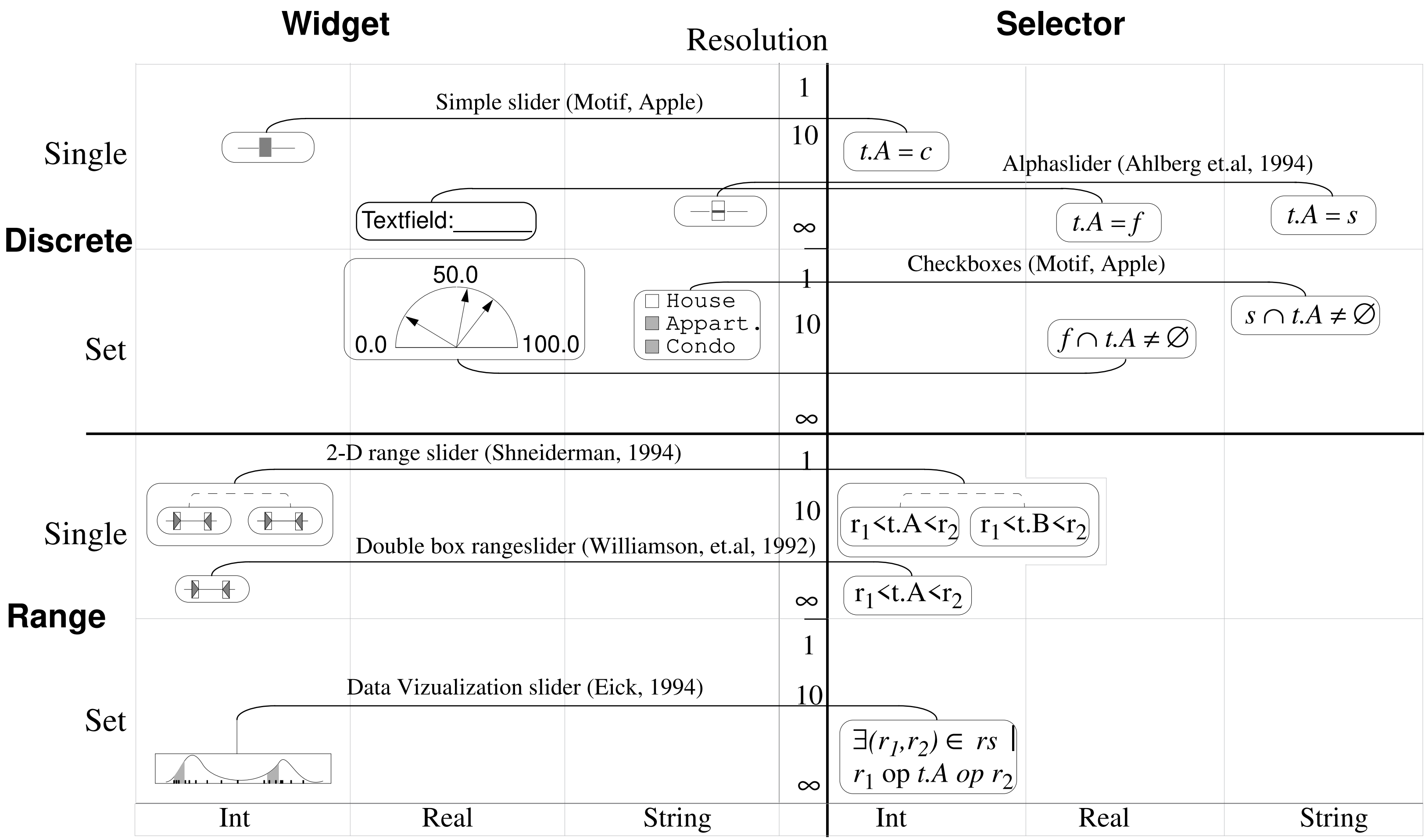

Figure 6 Taxonomy of query devices. Widgets are composed with selectors to create query devices which are categorized into those selecting discrete values and those selecting ranges. Those subcategories are divided into those selecting single values, ranges, or sets of values or ranges.

widget function **W** (HorizontalDragEvent $\rightarrow$ <Int,Int>) can be combined to create a query device with the type:

S $\otimes$ **W**: *(HorizontalDragEvent* $\times$ *Set of Tuple)* $\rightarrow$ *Set of Tuple.*

Composition of widgets into complex widgets

The $\times_w$ composition operator is introduced for creation of complex widgets allowing users to specify criteria for more than one attribute simultaneously. For example, consider two widget functions, one being $W_1 = HorizontalDragEvent \rightarrow Int$, and one being $W_2 = VerticalDragEvent \rightarrow Int$. These two can be composed into a complex widget function $W = W_1 \times_w W_2$ (Figure 7).

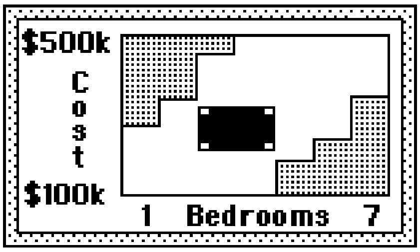

Figure 7 Two dimensional rangeslider widget (Shneiderman, 1994).

Composition of selector functions into complex selector functions

When complex widgets allowing for specification of multiple criteria simultaneously are introduced, the corresponding complex selector functions must be allowed too. Complex selector functions are created by composition of basic selector functions, using the $\times_s$ composition operator.

Composition into a query group

Each query device QD can be composed with another query device into a query group which will allow a *Set of Tuple* to be filtered through a series of query devices, i.e. reflecting the view in Figure 2. The composition

QD_{range} & QD_{range} & QD_{set}

yields a query group, filtering database tuples through two range selecting query devices and one set selecting query device.

The queries performed with individual query devices in a query group need to be combined appropriately, i.e. by allowing arbitrary boolean combinations of the results of query devices.
Possible queries should be:

$Query ::= BasicQuery \mid RangeQuery \mid RangeSetQuery \mid SetQuery \mid \neg Query$

A query created by the manipulation of several query devices is given as:

$CombinationQuery ::= (q_1 \wedge q_2 \wedge ... \wedge q_n) \vee (q_1 \wedge q_2 \wedge ... \wedge q_n) \vee$

where q_i is a Query, reflecting the possibility to create arbitrary combinations of query device results.

3 EXPLORING THE DESIGN SPACE OF QUERY DEVICES

The approach to exploring the design space presented below does not provide a one-to-one mapping between points in the design space and final query devices. Quite the contrary, many solutions are usually possible. But the types and predicates indicate crucial points for design, where there is risk of confusion between query devices and where consistency is important.

For the *BasicQuery* and *RangeQuery* intuitive mappings can be found relatively easily – some of which have already been tested in controlled experiments (Ahlberg *et.al*, 1992)(Ahlberg & Shneiderman, 1994)(Eick, 1994)(Williamson & Shneiderman, 1992). These designs and other published variants are placed in the design space and new query devices are suggested.

3.1 General approach

Generation of query devices will be demonstrated by an example with a range selecting query device. Assume that the type of the needed selector function has been decided to $(\langle Int,Int\rangle \times Set\ of\ Tuple) \rightarrow Set\ of\ Tuple$ and the corresponding query predicate to $r_1 \leq t.A < r_2$, i.e. a query device allowing users to select tuples with the attribute *t.A* within the range greater than or equal to r_1 and smaller than r_2.

First the selector function is used to decide that the query device needs a widget supplying two integer values – and accordingly the widget function $HorizontalDragEvent \rightarrow \langle Int,Int\rangle$ is selected. A graphical interaction object corresponding to this type is a double box slider widget (Figure 8), another would be two grouped text fields.

Figure 8 Double box slider allowing users to select two Int:s.

From the type of the widget function, $HorizontalDragEvent \rightarrow \langle Int,Int\rangle$, it can also be concluded that the range selected with the widget should be reflected graphically in the widget to convey the sense of range and not two single points, and textually to give the exact numbers. But the type does not provide enough information to conclude what range the two integers indicate. From the query predicate, $r_1 \leq t.A < r_2$, it is concluded the query range should be given both graphically in the widget and textually, with the bright color between the dragboxes indicating the range (Figure 9).

Figure 9 Slider extended with cues for range.

The query device does still not convey whether the end points of the range are included or not. This information can also be concluded from the query predicate and cues for this are added to the query device (Figure 10).

Figure 10 Cues added to convey whether end points are included or not.

Following this simple approach, query devices existing within the given framework will be presented. A large number of the query devices are quite similar to each other and accordingly only interesting variations will be presented. The four following subsections each corresponds to one of the main rows of Figure 6.

3.2 Discrete single value selecting query devices

The simplest query devices are those allowing users to select a single value, with the composed type (*Basic* × *Set of Tuple)* → *Set of Tuple* (these can be found in top of (Figure 6)). For the *Int* type and the query predicate, $t.A = c$, the design is simple. The query device allows users to select a single integer which is indicated by the single drag box and the textually given current value (Figure 11). A variation would be the dial widget in Figure 4.

Figure 11 Query device allowing users to select a single integer

The negation of the former query predicate, $t.A \neq c$, allows users to select all the values but c, which is reflected in Figure 12.

Figure 12 Query device allowing users to select all the values but c.

The *Real* and *String* types pose similar problems. For the type *Real* users want to be able to select numbers with different granularities, i.e. be able to set both the integer and the fractional part. If both the range and the needed granularity is large, the mapping of mouse movements to selection of reals *Event* → *Real* becomes too cumbersome for users (if the range is 0 to 1’000 and the needed granularity 1/100, users would have to move the mouse 100’000 screen units to move from the left to the right end of the slider).

The same situation exists for query devices for string selection if the list of strings is large. A solution to this problem is to provide users with different granularities of movement. This is elaborated in (Ahlberg & Shneiderman, 1994) – where four designs were compared in a controlled experiment. A design where users selected granularity by initiating dragging in different parts of the slider drag box was found to be the most effective.

Accordingly the following design is proposed for a query device for the type *Real* (Figure 13). In this case a real number between 0 and 100 can be selected, and by initiating dragging in the upper part of the slider drag box the number is increased or decreased by 1 for each mouse movement. By initiating dragging in the lower part the number is increased or decreased by 0.1 for each mouse movement. Notice that this is just a partial solution to the problem, *Real* numbers can still not be selected with arbitrary granularity. However, more or less obvious extensions of this design could be a step in that direction.

Figure 13 Query device allowing users to select a Real.

For selections of type *String* the alphaslider from (Ahlberg & Shneiderman, 1994) can be utilized (Figure 14). Note the index below the slider which is spaced proportionally to the number of strings in the attribute which start with a particular character, i.e. if 20% of the strings start with ”S” then the ”S” part of the index occupies 20% of the space. A variation is a scrolling list.

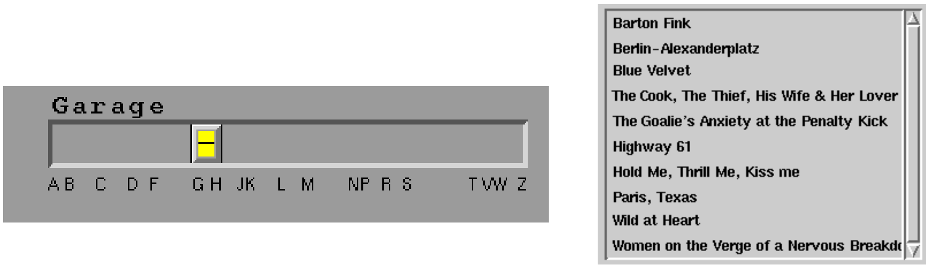

Figure 14 Query devices for selection of a String.

3.3 Discrete set selecting query devices

For set selecting query devices with the type

$$(\textit{Event} \rightarrow \textit{Set of Basic}) \otimes ((\textit{Set of Basic} \times \textit{Set of Tuple}) \rightarrow \textit{Set of Tuple})$$

the following four predicates were suggested above:

$$\begin{aligned} \textit{SetQuery::=} \quad & s \subseteq t.A \ \mid \\ & s = t.A \ \mid \\ & s \cap t.A \neq \emptyset \ \mid \\ & s \cap t.A = \emptyset \end{aligned}$$

The set selecting query devices allow users to select a set of values of the type *Basic*. Widgets with the type *Event* $\rightarrow$ *Set of Basic* are for example groups of checkboxes allowing users to select a number of settings (Figure 3) and scrolling lists allowing for multiple selections. A set selecting query device should communicate to users how this set is used to determine which tuples fulfill the criteria.

Below a query device allowing users to manipulate and grasp a set selecting query dynamically is presented, using Venn diagrams (Figure 15). Variations of Venn diagrams have been used with success in other query systems (Michard, 1982)(Spoerri, 1993), and can also be used to effectively communicate the set selecting predicates.

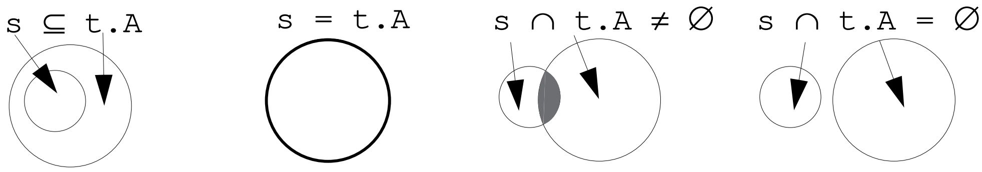

Figure 15 Venn diagrams illustrating simple logical predicates used for the query device in (Figure 16).

The query devices allows users to select a set of strings from an attribute and also to decide how this set should decide which tuples are selected. Users select strings as with the alphaslider, with the addition that by pushing the ”+” button another string can be selected. By selecting one of the four buttons to the right, the relation that should hold between the set selected with the query device and the attribute *t.A* in a tuple *t* can be set.

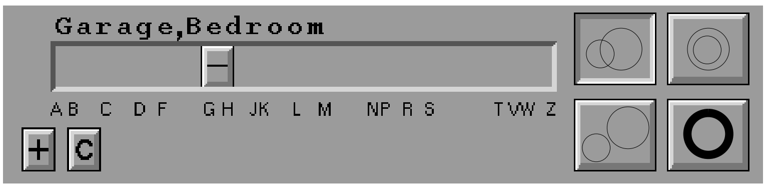

Figure 16 Query device allowing users to select a set of strings.

The following variation is presented for selection of *Sets of Real*. Using a dial widget, users can activate one or more selector arrows which each can be used to specify an value. An advantage of this approach is that users can significantly increase the resolution of the widget by moving the mouse away from the arrow being manipulated while specifying a number (Figure 17). The interaction objects called boxes in Feiner and Besher’s n-vision system (Feiner & Besher, 1990) are an interesting variation (Figure 17). Users manipulate a widget with the type:

$(XDragEvent \rightarrow Float) \times_w (YDragEvent \rightarrow Float) \times_w (ZDragEvent \rightarrow Float)$

to interactively select three values in a n-dimensional space. This widget can be combined with an appropriate selector predicate to create a useful query device.

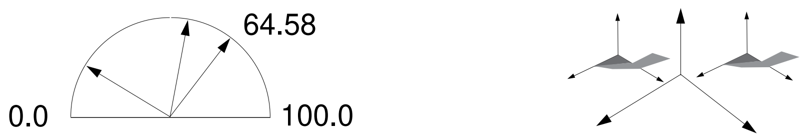

Figure 17 Query devices allowing users to select a set of *Real*. To the left a multiple dial query device and to the right a mock-up of Feiner & Besher’s boxes (1990).

3.4 Single range selecting query devices

Many variations of range selecting query devices exists – some can be found in the taxonomy in Figure 6. Design challenges are:

- How to indicate the selection of a range.
- How to indicate whether end points of a range are included or not.
- How to select several ranges for the same attribute.

The type of range query devices,

$$(DragEvent \rightarrow Range) \otimes (Range \rightarrow Set\ of\ Tuple \rightarrow Set\ of\ Tuple)$$

indicates that two selected values are to be used as range end points. However, by also examining the possible selector predicates for a *RangeQuery* it is apparent that at least eight variants are possible. The predicate $r_1 \leq t.A < r_2$ was examined in the introduction to this section. Whether or not end points were included in the range was communicated to users through the color of the arrows in the dragboxes indicating r_1 and r_2.

The negation of this predicate,

$$\neg(r_1 \leq t.A < r_2) = (min(t.A) \leq t.A < r1) \vee (r2 \leq t.A \leq max(t.A))$$

lets users select all the tuples with a value of *t.A* not in the range *(r1,r2)*. The following query device lets users select those tuples, indicating that two ranges actually being selected (Figure 18).

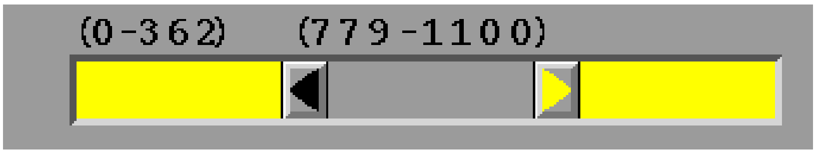

Figure 18 Query device allowing users to select tuples where $t.A < r_1$ *or* $t.A \geq r_2$.

For the predicates,

$(min(t.A)\ op\ t.A\ op\ r)$ and $(r\ op\ t.A\ op\ max(t.A))$

combined with the type,

$$(Event \rightarrow Range) \otimes ((Range \times Set\ of\ Tuple) \rightarrow Set\ of\ Tuple)$$

the following design is proposed:

Figure 19 Query device allowing users to select tuples where *t.A* is less than or equal to r_1.

The designs given above has all been for the *Int* type. For the *Real* type similar designs follows by adding functionality for selection with different granularities.

Complex range selecting query devices

A useful query device would allow users to select two ranges simultaneously (Shneiderman, 1994), i.e. allowing for manipulation of two widgets

$$DragEvent \rightarrow Range$$

simultaneously. Such a composed query device can be found in the taxonomy in (Figure 6). Two widgets accepting *DragEvents* in the horizontal and vertical direction are composed with the $\times_w$ composition operator into a complex widget:

$$(HorizontalDragEvent \rightarrow Range) \times_w (VerticalDragEvent \rightarrow Range)$$

To utilize such a widget in a query device, an accompanying complex selector function is needed. The selector function is created by composition of two selector functions into a complex selector function, utilizing the $\times_s$ composition operator:

$$(Range \rightarrow Set\ of\ Tuple \rightarrow Set\ of\ Tuple) \times_s (Range \rightarrow Set\ of\ Tuple \rightarrow Set\ of\ Tuple).$$

The complex widget and the complex selector can be combined into a complex query device similar to Figure 7.

3.5 Selection of sets of ranges

Selecting sets of ranges is sometimes useful, which can be performed with a query device with the type:

$$(Event \rightarrow Set\ of\ \langle Int,Int \rangle) \otimes ((Set\ of\ \langle Int,Int \rangle \times Set\ of\ Tuple) \rightarrow Set\ of\ Tuple)$$

An interesting design is proposed by Eick (1994). The traditional slider is enhanced by introducing a plot of the distribution of the data underlaying the query device into the slider area. Users select multiple parts of this slider by ”brushing” arbitrary areas (”brushing” here refers to users painting an area) (Figure 20).

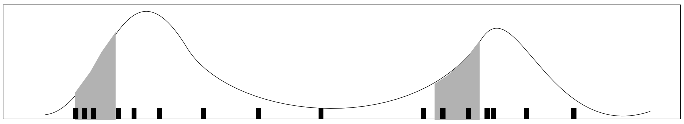

Figure 20 Mock-up of Eick's data visualization slider (Eick, 1994).

3.6 Boolean combinations of single, range, and set queries

Finally we need to cover the queries associated with compositions created with the query group composition operator *&*. To achieve an expressiveness equal to the one given above,

$$CombinationQuery ::= (q_1 \wedge q_2 \wedge ... \wedge q_n) \vee (q_1 \wedge q_2 \wedge ... \wedge q_n) \vee ...$$

where users can formulate queries with combinations of results from query devices on disjunctive normal form, several designs are possible. The InfoCrystal (Spoerri, 1993) allows users to easily overview and select all the boolean combinations of up to five variables, but needs too much screen space to be effective. Furthermore presenting more than five variables creates a cluttered screen unless a hierarchical structure of InfoCrystals is introduced.

The combination of dynamic queries and the Filter/Flow metaphor (Degi *et.al*, 1993) has been proposed to allow users to compose and understand complex queries (Shneiderman, 1994). The Filter/Flow metaphor visualizes a query utilizing a metaphor of water flowing through filters (Figure 21).

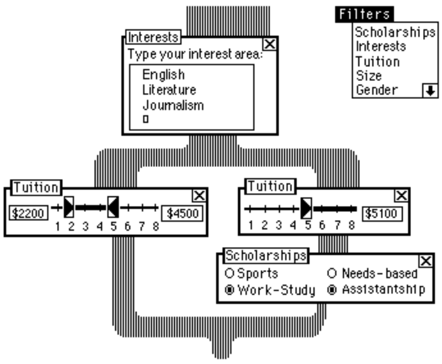

Figure 21 Mock-up of a filter/flow boolean query combined with dynamic queries (Shneiderman, 1994).

4 FUTURE WORK

- More parts of the design space should be explored and other existing query devices should be placed in it. The taxonomy in Figure 6 leaves obvious "holes" in the design space that should be filled.
- The presented query devices need further refinement and human-factors testing.
- More characteristics of the design space should be added, such as speed of possible input

(important for dynamic queries) and resolution of query devices.

- Many of the described query devices are part of a generalized dynamic queries tool. This tool, where databases with hundreds of attributes can be explored, needs to address issues as how to select from hundreds of query devices and how to semi-automatically infer which query devices should be used for which datatypes.

5 CONCLUSIONS

This paper makes two contributions:

- The design space of query devices is structured and explored.
- A set of query devices considerably increasing the expressiveness of dynamic queries is presented.

The presented approach to finding query devices does not tell designers exactly how query devices should be designed, but does point out where careful design is necessary to achieve both consistency and distinguishability. More query devices can straight forwardly be added and thereby a basis for the necessary human-factors and design work is formed. The set of query devices presented is ready to use, although in some aspects somewhat crude. Refinement and human-factors testing as done in (Ahlberg & Shneiderman, 1994) is necessary.

6 ACKNOWLEDGMENTS

Dynamic queries is originally an idea of Ben Shneiderman and his contributions to the concept of dynamic queries can not be overstated. Richard Chimera, Andrew Moran, Lars Pareto, Ben Shneiderman, Anselm Spoerri, and Erik Wistrand all provided useful comments on this paper. This work was in part supported by NUTEK, grant no: 5321-93-2760.

7 REFERENCES

Ahlberg, C., Williamson, C., Shneiderman, B. (1992) Dynamic Queries for Information Exploration: An Implementation and Evaluation. *Proceedings ACM CHI'92: Human Factors in computing Systems*, 619-626. Also in Shneiderman, B. (1993) *Sparks of Innovation in Human-Computer Interaction*, Ablex Publishing Corp., Norwood, N.J.

Ahlberg, C., Shneiderman, B. (1994) Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays. *Proceedings ACM CHI'94: Human Factors in computing Systems*, 313-317. Also in Baecker, R., Grudin J., Buxton, W., and Greenberg, S., *Readings in Human-Computer Interaction: Toward the Year 2000 (2nd Edition)*, Morgan Kaufmann Publishers, San Francisco, CA, 1995.

Ahlberg, C., Shneiderman, B. (1994) The Alphaslider: A Compact and Rapid Selector. *Proceedings ACM CHI'94: Human Factors in computing Systems,* 365-371.

Bertin, J. (1983) *Semiology of Graphics*, University of Wisconsin Press, Madison, Wis.

Buja, A., McDonald, J. A., Michalak, J., and Stuetzle, W. (1991) Interactive data visualization using focusing and linking, *Proceedings IEEE Visualization '91*, 156-163.

Carr, D. (1994) Specification of Interface Interaction Objects, *Proceedings CHI'94: Human Factors in computing Systems,* 372-378.

Diaper, D. (1989) Task Analysis for Human-Computer Interaction, Ellis Horwood, Chichester.

Eick, S. (1994) Data Visualization Sliders, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology'94.*

Eick, S., Nelson, M., Schmidt, J. (1994) Graphical Analysis of Computer Log Files, *Communications of the ACM*, **37(12)**.

Elmasri, R., Navathe, S. B. (1989) *Fundamentals of Database Systems*, Addison-Wesley.

Feiner, S., Beshers, C. (1990) Worlds within worlds: Metaphors for exploring n-dimensional virtual worlds, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology'90*, 76-83.

Kim, H., Korth, H., Silberschatz, A. (1988) PICASSO: A Graphical Query Language, *Software – Practice and Experience*, **18(3)**, 169-203.

Mackinlay, J. (1986) Automating the Design of Graphical Presentations of Relational Information, *ACM Transactions on Graphics*, **5(2)**, 110-141.

Mackinlay, J., Card, S., Robertson, G. (1990) A Semantic Analysis of the Design Space of Input Devices, *Human-Computer Interaction*, **5(2-3)**, 145-190.

Michard, M. (1982) Graphical presentation of Boolean expressions in a database query Language: design notes and an ergonomic evaluation, *Behaviour and Information Technology,* **1(3)**.

Robertson, G. G., Card, S. K., and Mackinlay, J. D. (1993) Information visualization using 3-D interactive animation, *Communications of the ACM,* **36(4)**, 56-71.

Shneiderman, B. (1992) *Designing the User Interface: Strategies for Effective Human-Computer Interaction: Second Edition*, Addison-Wesley Publ. Co., Reading, MA.

Shneiderman, B. (1994) Dynamic Queries for Visual Information Seeking, *IEEE Software,* (November 1994).

Spoerri, A., (1993) InfoCrystal: A visual tool for information retrieval & management, *Proceedings ACM Conference on Information & Knowledge Management*'93, Washington D.C.

Young, D., Shneiderman, B. (1993) A graphical filter/flow model for boolean queries: An implementation and experiment, *Journal of the American Society for Information Science* **44(4)**, 327-339.

Zwicky, F. (1967) The morphological approach to discovery, invention, research, and construction. In *New Methods of Thought and Procedure*, Zwicky, F., Wilson, A. G., Eds, Springer-Verlag, 273-297.

8 BIOGRAPHY

Christopher Ahlberg is a graduate student in human-computer interaction at the department of computer science at Chalmers University of Technology, Sweden. He has worked as a visiting researcher at the Human-Computer Interaction Laboratory at University of Maryland.

Christopher Ahlberg has authored a number of papers on dynamic queries, a novel concept for visual information seeking. He is the creator and designer of the IVEE visualization system. He has consulted and lectured extensively in human-computer interaction and visualization for industry, academia, and research institutes.