Functional Unification Approach to Automated Visualization Design

Stephan Kerpedjiev[♣], Steven F. Roth[♣♦], Joe Mattis[♦]

MAYA Viz[♣]
2100 Wharton Street
Pittsburgh, PA 15203
{kerpedjiev, roth}@mayaviz.com

Robotics Institute[♦]
Carnegie Mellon University
Pittsburgh, PA 15213
jam@cs.cmu.edu

Abstract

A unification-based approach to visualization design provides a uniform way of representing user requirements, design knowledge, and graphic designs as well as algorithms for synthesizing graphic presentations. We demonstrate this approach on two types of requirements – structural in the form of sketches and functional in the form of tasks. With this approach we aim to achieve the following system design goals: expressiveness (the formalism can express the visualization design problem and its problem-solving algorithms), uniformity (the same formalism can be applied to different generation tasks), efficiency (graphics can be designed in a reasonable amount of time), and extensibility (the system can be extended with new types of requirements, design elements and design knowledge).

The visualization design problem

The visualization design problem is to synthesize a graphical presentation that expresses a set of data to satisfy given requirements. The requirements may come in different forms. For example, the user might sketch some elements of the visualization (e.g., a chart with an interval bar), possibly map some data attributes to graphical properties and request a graphic that contains all sketched elements. Alternatively, the request might specify the data exploration goals (e.g., find subsets of sales by product type, and within each group check if any correlation exists between buyer's age and sales volume). By addressing different design requirements we go beyond Mackinlay's (1986) graphical presentation problem, which did not consider any requirements other than the data, and beyond Casner's (1990) work, which considered only tasks. With the challenge of designing graphics that respond to various types of requirements, we turn our attention to using a formalism that can express the variety of requirements in a uniform way. We propose a unification-based approach to graphic design to achieve the following system design goals:

- Expressiveness - the formalism can capture various design requirements and the knowledge needed to generate graphics that satisfy those requirements.
- Efficiency - the system can produce designs in a matter of seconds.
- Uniformity - the same formalism can be used for all stages of the design process.
- Reusability and extensibility - the system can be extended with new design elements, constraints, and knowledge by reusing existing grammars.

We chose functional unification as a formalism to achieve our goals for the following reasons. It is a constraint-satisfaction method, which fits perfectly with the nature of the design task. It is fairly simple using only one type of data structure, the functional description, and one type of operation, unification (Shieber, 1986). It provides a common representation for multimedia generation, where the natural language and graphics can be coordinated by sharing common data structures.

Our graphic design system employs a wealth of knowledge gained in prior work on automated graphic design. This knowledge includes the definition of expressive and effective graphical languages (Mackinlay, 1986), task-driven graphic design (Casner, 1990), data characterization (Roth and Mattis, 1990), and visualization tasks (Zhou and Feiner, 1998). Work on user interfaces for graphic design brought a different perspective to the automated design problem. Interfaces such as SageBrush (Roth et al., 1994) and SageBook (Chuah et al., 1995) let the user express elements of the design in an intuitive visual way. This gives the user more control but also poses a greater challenge to the designer - SageBrush sketches need to be reconciled with designs generated automatically by Sage. This challenge was a major contributing factor for choosing functional unification - decisions informed by different sources such as data characteristics and user sketches need to be unified in order to produce desirable presentations.

This paper extends prior work on automated graphic design in three ways. First, we demonstrate how a functional unification approach can provide a common representation throughout all stages of the design process. This provides significant support when engaging the larger problem of automatically designing multimedia presentations with coordinated natural language and graphics. While functional unification has been used for generating coordinated natural language descriptions and device drawings (Feiner and McKeown, 1991), we extend it to generating data graphics as part of multimedia explanations.

Second, this paper contains previously unpublished details about the internal representation of graphical design requirements, search strategies, and completed designs, which can facilitate the implementation of new automated design systems.

Third, and most significantly, this paper introduces the concept of applying design strategies, which is a systematic approach that produces more coherent designs, more quickly, than previous automated design systems. In particular, both Mackinlay's APT system (1986) and prior Sage work (Roth et al., 1994) functioned by determining a maximally "effective" set of perceptual techniques (e.g., position, size, color) and then by exhaustively searching for a means of composing those techniques into a coherent design. Conversely, design strategies take a top-down approach, which improves system responsiveness, makes extensibility more tractable, and produces more coherent designs.

Functional unification

Functional unification (Kay, 1979) is an approach to natural language processing that assumes the functional perspective to language rather than the more common structural perspective. The functional perspective reflects the role of the constituents of a message while the structural perspective cares exclusively about the well-formedness of the messages. In generation, the functional perspective is crucial as it deals with the proper mapping from communicative goals to components of the message. We intentionally replaced the terms "sentence" and "words" with the more general concepts "message" and "components" to include graphical communication into the discussion. Indeed, just like language achieves communicative goals using discrete messages in the form of sentences and words, graphical languages use discrete messages in the form of charts and graphemes to achieve communicative goals.

Unification is a simple formalism. It requires just one type of data structure called functional description (FD) for representing both the input (the requirements for a sentence or a graphic) and the grammar (the knowledge that guides generation). An FD describes an object via an unordered set of attribute-value pairs, where each value is an atom, another FD, or a path (pointer) to another value. For example, in FD (1) below, the value of *a* is the atom *x*, the value of *b* is another FD with two attribute-value pairs, and the value of *c* is a pointer to another value obtained by following the path `{b m}`. Each component of an FD is "addressed" by the path that leads from the root to that component by following the attributes in the path. For example, the address of the whole FD is the empty path, which is denoted `{}`. The address of the value of attribute *a* is the path `{a}`. The use of a pointer indicates shared representation. In FD (1), the components `{c}` and `{b m}` should always have a common value. Therefore, if a constraint is imposed on `{c}`, it will be imposed on `{b m}` as well and vice versa. Components can be accessed using relative paths as well. A relative path begins with the special character '^' and each such character indicates going one level up the FD structure. For example, we could modify FD (1) into an equivalent one by replacing the absolute path `{b m}` with the relative path `{^ b m}`, which means go up one level to the root and then follow attributes *b* and *m*.

(1) Sample FD: (2) Sample grammar: (3) Enriched FD:

```
((a x)        ((c v)              ((a x)
 (b ((m v)     (alt (((a y)        (b ((m v)
     (n w)))          (p s))           (n w)))
 (c {b m}))          ((a x)        (c {b m})
                      (p t)))))    (p t))
```

The attribute-value pairs in an FD are conjunctive, i.e. the object described by the FD must possess all the properties specified in it. Grammars, however, use disjunction as well - for a string to be accepted by a grammar, only one of its rules need to be satisfied. To accommodate disjunction in functional unification grammars (FUGs), the formalism has been extended with the alternation construct specified by the attribute "alt" followed by a list of alternative FDs. For example, grammar (2) above consists of the pair `(c v)` in conjunction with one of two alternatives: the pairs `(a y)` and `(p s)` or the pairs `(a x)` and `(p t)`.

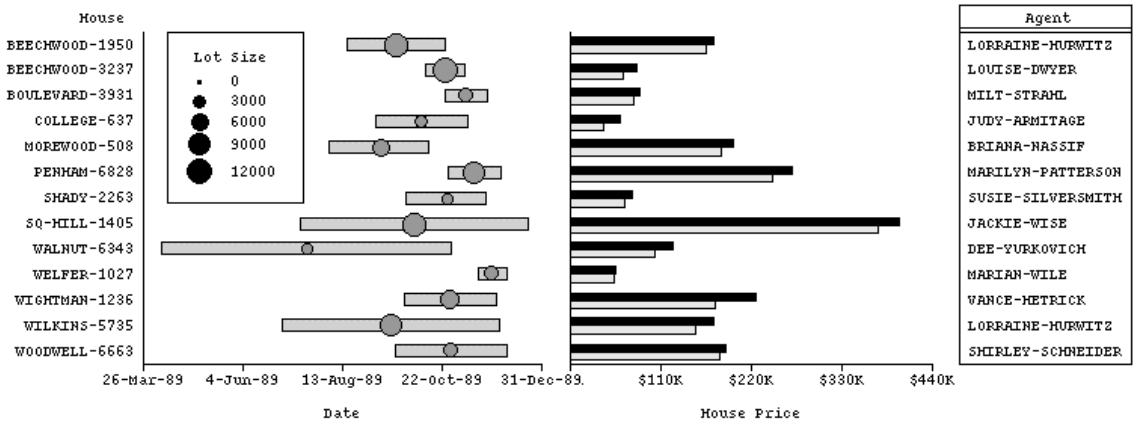
The only operation defined on FDs is unification. The unification of two FDs either produces a new FD that is compatible with and more specific than the input FDs, or fails. The unification fails if at least one attribute has incompatible values in the input FDs. Two values are incompatible, if one is atomic and the other is an FD, if they are two different atoms, or recursively, if they are two incompatible FDs. Compatible FDs are merged into one FD much like set union except that it is recursive. In the case of alternation, only one of the alternative FDs needs to unify with the other FD.

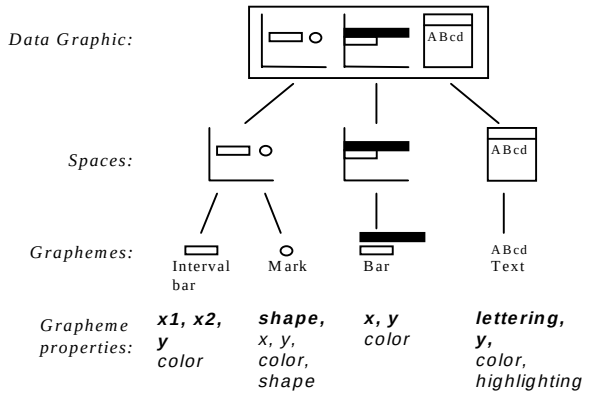
Here is how the unification of FD (1) with grammar (2) would proceed. The first pair of the grammar is `(c v)`. The value of attribute *c* in the input is accessed following the path `{b m}`, which yields the atom *v*. The two values are equal and therefore the matching is successful. The next construct of the grammar is a set of alternatives, the first one being the FD `((a y) (p s))`. The first pair of this FD, `(a y)`, does not unify with the pair `(a x)` of the input FD. Therefore, this alternative fails. The first pair `(a x)` of the next alternative unifies with the pair `(a x)` of the input. The second pair, `(p t)`, does not have a counterpart pair in the input and therefore is added to the result. Finally, the pair `(b ((m v) (n w)))` has no counterpart in the grammar and is added to the result to yield FD (3).

Having presented the basic concepts of functional unification, we turn to the heart of the problem - analysis of the design problem, identifying its ingredients, and representing them by FDs and unification procedures.

Ingredients of visualization design

At the highest level, the visualization design problem deals with three types of objects: requirements, design





knowledge, and individual designs. We considered two types of requirements – sketches and tasks. The user sketches, created with a specialized drawing editor called SageBrush (Roth et al, 1994), are parsed and represented as constraints on the design. Tasks are produced by decomposing a data exploration or communicative goal. Our task language is based on work by Casner (1990) and Roth and Mattis (1990), as well as some recent work on multimedia generation (Kerpedjiev et al., 1997).

A formal representation of graphic designs facilitates the proper communication between the system modules and supports the reasoning of the designer. It captures the graphical elements, their relationships, and the mapping of data objects to graphical objects. The design representation (explained in the next section) is based on prior work in the Sage group (Roth and Mattis, 1990, Roth et al., 1994, Chuah et al., 1995).

The design knowledge is modularized into the following sub-grammars applied to the input in this order: mapping design requirements into constraints on the design (grammars `SKETCH-MAPPINGS` and `TASK-MAPPINGS`); creating the skeleton of the symbols that express data elements (grammar `DESIGN-STRATEGIES`); merging designs (grammar `COMPOSITION`).

Design representation

The design representation specifies visualizations so that other components of the system can use it to perform tasks such as rendering, explaining or supporting interaction. Figure 1 shows a sample graphic, which consists of three horizontally aligned spaces. Figure 2 (from Chuah et al., 1995) decomposes it into design elements. The main design components are spaces (charts, maps, networks, tables), encoders (axes, color keys), symbols, graphemes (marks, bars, lines), and their properties.

The space is a container for symbols and imposes a layout discipline via its encoders (e.g., `X` and `Y` axes for charts). Each space is represented by an FD, with attributes for its type and one or more positional encoders. For example, a space of type chart would be represented as follows:

```
((type chart)
 (x-axis ((g-type x-position)
          (data-type date)))
 (y-axis ((g-type y-position)
          (data-type address))))
```

An encoder maps values from a data type such as date to graphical values such as x-position. Hence, its representation consists of those two elements.

A symbol provides an integrated view to a data object by presenting several of its attributes via the graphical properties of one or more graphemes, all of which are co-located in space. The main components of the symbol description are its determinant - the attributes that determine the position of the symbol, and a pointer to the space in which the symbol resides. A sample FD of a symbol is given below:

```
((det ((y address)
       (x1 date-on-market)
       (x2 date-sold)))
 (of-space {chart1}))
```

The path `{chart1}` points to a space description like the one given above. A symbol with this description occupies a horizontal interval location within the chart pointed to by the `of-space` attribute. The symbol's lo-

cation is determined by the triple of attributes: *address*, *date-on-market*, and *date-sold*.

The grapheme is an atomic graphical object such as mark, bar, line or text that conveys information through its properties (e.g., x-position and color). Each grapheme is described by its type, a pointer to the symbol it is part of, and the graphical properties that encode data attributes. For example, the mark described below conveys information via its x-position, y-position, and size:

```
((type mark)
 (of-symbol {symbol1})
 (x ((attr date-sold)
     (encoder ((data-type date)
               (g-type x-position)))))
 (y ((attr address)
     (encoder ((data-type street-address)
               (g-type y-position)))))
 (size ((attr lot-size)
        (encoder ((data-type square-feet)
                  (g-type size))))))
```

The FDs of graphemes and symbols serve as definitions for the rendering component, which applies them to data objects to produce individual symbols and graphemes like those in Figure 1.

To summarize, the design is an interlinked collection of spaces, symbols, graphemes, and encoders. Each grapheme is a part of exactly one symbol, each symbol can reside in exactly one space, and each space imposes a layout discipline by its positional encoders.

Design strategies

Design strategies represent high-level organizations of graphics. The basis for a design strategy is the mapping of data attributes to the positional properties of symbols. Thus, a design strategy prescribes how the graphic will use the space to structure the information but leaves out any other details such as how many and what kinds of graphemes will constitute the symbol, or if and what retinal properties (e.g., color and shape) will be used for encoding data attributes. For example, at least the following two strategies could be adopted for presenting data about four attributes of house sales: the date the house was put on the market, the date it was sold, address, and selling price.

(strat1) By symbols that mark the intervals each house was on the market. External to the strategy might be a text annotation of price.

(strat2) By symbols whose spatial distribution conveys the correlation between date-on-market and date-sold. External to the strategy might be the size of a mark for price and a text annotation for address.

The strategies are represented in the grammar by their types and determinants. The strategy type (explained below) is a convenient abstraction that is used throughout the grammars for various types of decision making. However, the key component of any strategy is its determinant. It specifies the attributes that determine the location of the symbols designed by the strategy. For example, the FDs below illustrate strategy **strat1**, which is of type disjoint interval (`DI`), and **strat2**, which is of type correlation (`CORR`):

```
((type DI)
 (det ((y address)
       (x1 date-on-market)
       (x2 date-sold))))

((type CORR)
 (det ((y date-on-market)
       (x date-sold))))
```

The strategy type, defined solely on the basis of characteristics of the data, asserts some positional features of the symbols. The following strategy types are used:

Functionally-independent attribute (`FIA`) - the *x* or *y* attribute of the strategy functionally determines all other attributes. A `FIA` strategy guarantees a unique strip (horizontal or vertical) for each symbol.

Relation (`REL`) - the *x* and *y* components of the determinant together functionally determine all attributes in the data. This type guarantees a unique point location for each symbol in a chart-like space.

Location (`LOC`) - the pair of attributes bound to the *x* and *y* components of the determinant form a geographic location. The symbols are shown on maps.

Disjoint interval (`DI`) - the triple of attributes bound to the *x*, *y1*, and *y2* components of the determinant are characterized to form a disjoint interval. This strategy guarantees a unique interval location for each symbol.

Correlation of two, three or four attributes (`CORR`, `COOR3`, `CORR4`, respectively) - there are no functional dependencies from the attributes in the determinant to other attributes. Strategies of type `CORR` are realized by symbols that occupy a point in the space, `CORR3` strategies are realized by horizontal or vertical interval bars, and `CORR4` strategies are realized by lines. Since none of those strategies guarantees unique positions the symbols may overlap.

Why use design strategies? The positions of the symbols determine how the space is utilized. By selecting a strategy the designer makes an important decision about the main view to the data and will stick to this view as long as there is no evidence that the user needs a different one. For example, if at a certain point of the design process, the designer decides to organize the graphic around location, it will try to realize all subsequent constraints within the `LOC` strategy unless this proves impossible or ineffective. Using multiple views requires establishing a link between them to make the presentation coherent.

How do design strategies work? When a grapheme is instantiated as the result of satisfying some constraint, that grapheme is unified with grammar `DESIGN-STRATEGIES`. This unification instantiates a symbol and coordinates the positional properties of the grapheme with the determinant of the symbol. Grammar `DESIGN-STRATEGIES` has the following structure:

```
((grapheme
   ((of-symbol
       ((alt <design-strategies>)))
    (alt (((type 1D-POINT)
           (y ((attr {^^ of-symbol det y}))))
          ((type 2D-POINT)
           (x ((attr {^^ of-symbol det x})))
           (y ((attr {^^ of-symbol det y}))))
          ((type HORIZONTAL-INTERVAL)
           (y ((attr {^^ of-symbol det y})))
           (x1 ((attr {^^ of-symbol det x1})))
           (x2 ((attr {^^ of-symbol det x2}))))
          ((type VERTICAL-INTERVAL)
           (x ((attr {^^ of-symbol det x})))
           (y1 ((attr {^^ of-symbol det y1})))
           (y2 ((attr {^^ of-symbol det y2}))))
          ((type LINE)
           (x1 ((attr {^^ of-symbol det x1})))
           (x2 ((attr {^^ of-symbol det x2})))
           (y1 ((attr {^^ of-symbol det y1})))
           (y2 ((attr {^^ of-symbol det y2}))))
          ((x NONE)  (y NONE)))))))
```

Alternation `<design strategies>`, which consists of the FDs of all potential strategies, is generated on the fly by analyzing the characteristics of the data, which include functional dependencies, data types (e.g., nominal vs. ordinal vs. quantitative), and composite data types such as location and interval. The second alternation coordinates the positional properties of the grapheme with the determinant of the symbol. The six alternatives represent a point in a 1D-space (e.g., a table), a point in a 2D-space (e.g., a chart or a map), a horizontal interval in a chart, a vertical interval in a chart, a line in a 2D space, or a satellite (a grapheme that does not have its own positional properties).

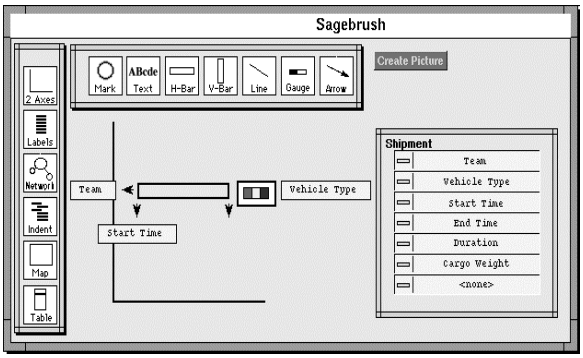
Sketches

SageBrush (Roth et al., 1994) is an interface in which users express their graphics needs by sketching design ideas from primitive elements such as spaces, graphemes and data attributes. The graphemes are placed within spaces while the data attributes are mapped to grapheme properties such as position or color, to space encoders such as `X` or `Y` axes, or are left unbound. Figure 3 shows a sketch with a chart drawn from the palette on the left side of the interface and a bar dragged from the top palette. Three attributes are mapped to the bar's *y* and *x1*-positions, and its color, while the rest (end time, duration, and cargo weight) are not bound.

Four types of constraints represent a sketch:

An empty space - created for each space in the sketch that has no graphemes in it and no attributes mapped to its axes. The type of the space imposes a constraint on the type of the strategy. For example, a map can be satisfied only by a strategy of type `LOC`, a table - of type `FIA`, and a chart - by any type of strategy but `LOC`.

An attribute on an axis - created for any attribute dropped on the axis of a space. This constraint can be satisfied by any symbol whose determinant has the attribute dropped on the axis as a value of the corresponding symbol position.



A grapheme in a space - created for each grapheme placed within a space. The grapheme type imposes constraints on the strategy type (e.g., an interval bar can only realize strategies of type `DI` and `CORR3`). Any mappings of attributes to positional properties of the grapheme impose constraints on the symbol's determinant.

A free (unbound) attribute – does not impose any constraints on the strategy.

The description of the four types of constraints summarizes the `SKETCH-MAPPINGS` grammar, which translates sketches into the common language of the design representation. To illustrate the grammar, consider the two FDs below representing the grapheme-in-space constraint from Figure 3 and a relevant fragment of grammar `SKETCH-MAPPINGS`, respectively.

```
((cat grapheme-in-space)
 (grapheme ((type horizontal-interval-bar)
            (y ((attr team)
            (x1 ((attr start-time)))
            (color ((attr vehicle-type)))))
 (space ((type chart))))

((cat grapheme-in-space)
 (grapheme
   ((of-symbol
     ((of-space {^ ^ ^ space})))))
 (alt (((space ((type chart)))
        (grapheme
          ((alt
           (((type mark)
              ...)
            ((type horizontal-interval-bar)
             (of-symbol
               ((of-strategy
                  ((alt (((type DI))
                         ((type CORR3)))))))))
            ((type vertical-interval-bar)
              ...)
            ...)))))
       ((space ((type map)))
        ...)
       ...))))
```

Grammar `SKETCH-MAPPINGS` hooks the grapheme's symbol to the space in which the grapheme was placed and for the combination of a space of type *chart* and a grapheme of type *horizontal-interval-bar* constrains the strategy to types `DI` or `CORR3`. Then, the FD is unified with grammar `DESIGN-STRATEGIES`. The constraints im-

posed so far make possible the unification only with strategies of types DI and CORR3 and whose *y* and *x1* determinants are bound to the attributes *team* and *start-time*, respectively. Similar rules guide the realization of the other types of sketch constraints.

Tasks

Conceptual tasks are the operations that the user should perform on some, yet undefined, representation of the data to achieve a given data exploration goal. An example of a data exploration goal is "find the addresses of all houses that were on the market within a given time interval." For such a goal, the user will need to search the set of houses by inspecting their *date-on-market* and *date-sold* attributes to find a house that was on the market in the specified interval, and then look up its *address* attribute. By designing a graphic, those conceptual tasks will be realized as concrete perceptual and cognitive operations. The designer's goal is to make those operations maximally effective.

Our analyses revealed that conceptual tasks are composed of operations (or subtasks) at two levels: value accessing and entity manipulation; a distinction not evident in Casner's work. Each value-accessing task produces a value in one of three possible ways: e*valuate a constant* (e.g., evaluate the date April 16, 1999); a*ccess the value of an attribute* (e.g., evaluate the date-on-market of a given house); c*ompute a value* by applying some arithmetic operator such as total and max to other values (e.g., the difference between asking-price and selling-price).

The entity manipulation tasks work at the level of objects and result either in identifying objects by conditions imposed on some of their attributes (the SEARCH task) or in asserting some predicate about objects that are already identified (the LOOKUP, COMPARE and CORRELATE tasks). For the SEARCH task to be effective, the attributes should be mapped to properties that allow direct access from the attribute value to a narrow space where the object's symbol is located. Positional attributes are most suitable but retinal properties processed pre-attentively (such as color) are also good candidates; text labels are ineffective for search. The attribute of a LOOKUP task should be mapped to a graphical property that allows easy decoding from graphical values to data values. Labels and positional properties are good candidates as well as color and shape for attributes that have a small number of values, such as sex or race. Two conditions should be satisfied for COMPARE tasks: the attributes should be mapped to graphical properties using the same encoding rule (e.g., a common axis); and the graphical property should allow effective comparison (e.g., position, but not text). All attributes of a CORRELATE task must be mapped to positional or retinal properties of the same symbol.

Tasks have a hierarchical structure obtained by decomposing the goal into entity manipulation and value accessing subtasks. The entity manipulation tasks dominate the value accessing subtasks in the sense that the former are performed as part of the latter. For example, in the example above, the SEARCH task for houses dominates the access task on the *date-sold* attribute. In addition to dominance, there are dependency relations between the entity manipulation tasks. For example, before looking up the attribute of a house of interest, the user will have to find this house. In this case, the LOOKUP task depends on the SEARCH task. Such dependencies are captured by three organizational operators: SEQUENCE (each subtask depends on the previous one and therefore has to be executed after it); DISJOINT (the subtasks are independent of each other and can be executed in any order); CONJOIN (each subtask depends on the other subtasks and therefore all subtasks have to be executed in parallel). An example of mutually dependent subtasks is a pair of SEARCH tasks for the <u>same</u> object by two different attributes. We believe our treatment of the structure of tasks is more principled than Casner's (1990) vector representation based solely on the co-occurrence of objects and attributes in different tasks.

The input FD for each task specifies its type, subtasks (or operands) and any relevant data characteristics. For example, the following FD represents the sample task from the beginning of this section:

```
((cat sequence)
 (sub1 ((cat conjoin)
        (sub1 ((cat search)
               (op1 ((cat attr-value)
                     (attr date-on-market)
                     (object ?house)
               (op2 ((cat value)))))
        (sub2 ((cat search)
               (op1 ((cat attr-value)
                     (attr date-sold)
                     (object ?house)))
               (op2 ((cat value)))))))
 (sub2
   ((cat lookup)
    (op ((cat attr-value)
         (attr address)
         (object ?house))))))
```

The realization of a SEQUENCE task should enable the user to connect the graphical symbols of the object(s) that are common to any pair of consecutive subtasks (e.g., ?house). This can be achieved effectively in one of three ways. (1) The two symbols are identical. (2) The two symbols are different but realized by the same FIA strategy (in this case the user will be able to match the symbols by virtue of the fact that they lie in the same strip). (3) The symbol realizing the first subtask has a label for the functionally independent attribute of the data set while the symbol realizing the second subtask is realized by a FIA strategy whose determinant is the same functionally independent attribute. The third alternative, which is least efficient, requires that the users look up the label in the first symbol, and then using its value find the strip that contains the second symbol. A CONJOIN task requires that the common objects in its subtasks be realized by the same symbol. This constraint stems from the fact that tasks are performed in parallel

only if their operands are simultaneously in the user's focus. Disjoint tasks do not impose any constraints.

After grammar `TASK-MAPPINGS` imposes its constraints according to the rules described verbally in this section, each grapheme instantiated by an attribute access task gets unified with the grammar `DESIGN-STRATEGIES`. The following realizations are possible: (1) one symbol with one grapheme of type horizontal-interval-bar: the *y*-position encodes *address*, *x1* and *x2* encode *date-on-market* and *date-sold*; (2) one symbol with two graphemes: a mark whose *x* and *y* positions encode *date-on-market* and *date-sold*, and a label, which encodes *address*; (3) one symbol with two graphemes: a mark whose *x* and *y* positions encode *date-on-market* and *address*, and a *label*, which encodes *date-sold*.

The first design (Figure 4) is definitely the most effective one. It exploits a `DI` strategy, which allocates a unique interval location for each symbol. The `CORR` strategy of design (2) does not guarantee unique locations for the symbols, which may cause some labels to overlap. The third one (Figure 5) is ineffective because it employs text to encode the attribute of a search task. While the designer does not have any graphic critiquing capability, the ordering of the alternatives guarantees that strategies ensuring uniqueness of position such as `FIA` and `DI` will be explored first.

Composition

Composition merges design elements instantiated in response to different requirements. It makes visualizations more compact, coherent, and effective. We use the four types of composition proposed by Mackinlay (1986). The corresponding composition grammars (described below) apply to two graphemes, grapheme-1 and grapheme-2, where grapheme-1 is the grapheme just instantiated and grapheme-2 varies among previously instantiated graphemes.

Merging graphemes. The two graphemes can be merged into one. Merging graphemes is expressed by the following grammar:

```
((cat merging-graphemes)
 (grapheme-1 {^ grapheme-2}))
```

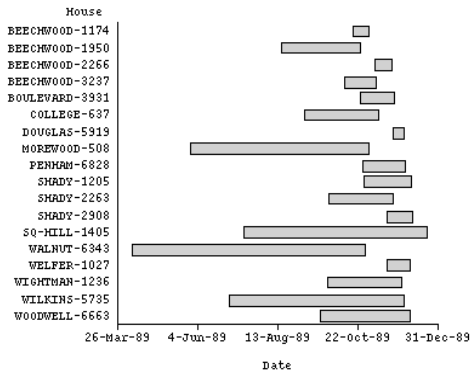
Clustering. The symbols of two distinct graphemes can be unified (e.g. a mark and a text annotation to it):

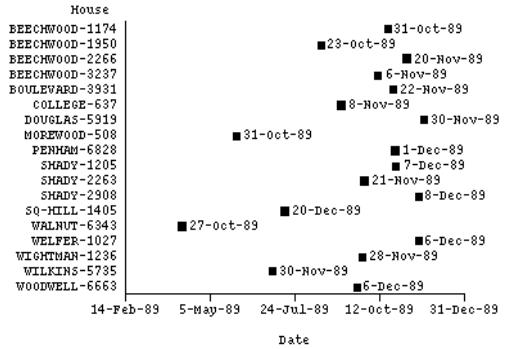
```
((cat cluster-composition)
 (grapheme-1 ((of-symbol {^ ^ grapheme-2
                          of-symbol}))))
```

Double axis composition. The distinct symbols of two graphemes can be placed in the same space:

```
((cat double-axis-composition)
 (grapheme-1
   ((of-symbol
      ((of-space {^ ^ ^ grapheme-2
                  of-symbol
                  of-space}))))))
```

Alignment. The distinct spaces of two graphemes share a common axis. The alignment can be horizontal (shared *y*-axis, cf. Figure 1) and vertical (shared *x*-axis):






```
((cat horizontal-alignment)
 (grapheme-1
   ((of-symbol
      ((of-space
         ((y-axis
            {^ ^ ^ ^ grapheme-2
                     of-symbol
                     of-space
                     y-axis}))))))))
```

Similarly, vertical alignment unifies the *x*-axes of the two spaces.

Figure 1 illustrates clustering (the marks attached to the interval bars in the left chart), double-axis composition (the two horizontal bars in the middle chart), and horizontal alignment of the two charts and the table. In the case of clustering, the mark and the interval bar form one symbol. In the case of double-axis composition, the two horizontal bars share the same space. And in the case of alignment, all spaces share the same *y*-axis.

Discussion

Using functional unification for graphic design offers some clear benefits, the most important one being that it supports thinking about the design in a systematic way. Every factor that contributes to the selection of graphical techniques is considered from the perspective of imposing constraints on some design elements. Those constraints are expressed declaratively as FUGs. Our approach was enabled by analysis of the requirements and the elements of graphic designs. In particular, it was informed by Bertin's (1983) semiological analysis of

graphics, Mackinlay's (1986) relational approach, Casner's work on task-driven graphic design, and the long-term research and development effort of the Sage project (Roth et al., 1997).

We looked at two radically different ways of expressing user needs - sketches and tasks. Although sketches convey the user needs in the form of graphical elements and relations, the designer still needs to reason about proper and consistent mapping of free attributes. On the other hand, tasks are goal and process oriented rather than graphics oriented. The designer needs to reason about what graphical techniques would support the tasks and the relations between them.

The functional unification approach described in this paper has been employed in the development of two systems that include automated graphic design. Sage automatically generates graphics that satisfy user's sketches. Sample visualizations designed by Sage can be found at http://www.cs.cmu.edu/~sage/sample.html. AutoBrief is an automated multimedia explanation system (Kerpedjiev et al., 1997). It employs communicative planning, media allocation, text and graphic microplanning, text realization, and graphic design. The graphic microplanner maps communicative goals allocated to graphics into conceptual tasks (Kerpedjiev and Roth, 2000), which the graphic designer realizes using the grammars described in this paper. Sample visualizations are available at http://www.cs.cmu.edu/~sage/abtour/start.html. We used FUF (Elhadad, 1992) as a functional unification engine for both systems.

For future work we plan to explore how context affects presentations. By context we mean features of the environment that influence the way users interpret graphics. For example, the size of the display or any previous visualization in the current session might affect the choice of graphical techniques.

Returning to our design goals, our system development effort confirmed that functional unification is a good formalism for tackling the visualization design problem. In both types of design requirements we were able to formulate the design knowledge in the form of FUGs and both systems generate graphical presentations in about 5-10 seconds. Compared to the older version of Sage, the unification-based one is able to complete a much larger number of design requests imposed by user sketches. Those observations rate the system pretty well on the scale of expressiveness and efficiency. We achieved uniformity by representing all the knowledge employed by the designer as FUGs. We gained some confidence about the extensibility of the grammars after members of our group requested incorporating specific design knowledge and we were able to fulfill those requests in half to one hour. However, to better evaluate the extensibility of the grammars, we would like to extend our design languages with new types of layout disciplines (e.g., polar charts) and new types of graphemes (e.g., tick marks).

Acknowledgments

This work was sponsored by DARPA contract DAA-1593K0005 to Carnegie Mellon University when the authors were there and by contract DARPA-N66001-99-D-8618 to MAYA Viz.

References

Bertin, J. 1983. *Semiology of Graphics: Diagrams, Networks, Maps*. Madison, Wisconsin: The University of Wisconsin Press.

Casner, S.M. 1991. A Task-Analytic Approach to the Automated Design of Information Graphic Presentations. *ACM Transactions on Graphics*, 10(2), 111-151.

Chuah, M., Roth, S., Kolojejchick, J., Mattis, J., and Juarez, O. 1995. SageBook: Searching Data-Graphics by Content. In: *Proceedings SIGCHI '95*, Denver, CO, 338-345.

Elhadad, M. 1992. *Using Argumentation to Control Lexical Choice: A Functional Unification Implementation.* Ph.D. dissertation, Computer Science Dept, Columbia University.

Feiner, S., and McKeown, K. R. 1993. Automating the Generation of Coordinated Multimedia Explanations. IEEE Computer, 24(10), 33-40.

Kay, M. 1979. Functional Grammar. In *Proceedings of the 5th Meeting of the Berkeley Linguistics Society*. Berkeley Linguistics Society.

Kerpedjiev, S., Carenini, G., Roth, S. F., and Moore, J.. 1997. AutoBrief: a multimedia presentation system for assisting data analysis. *Computer Standards and Interfaces*, 18, 583-593.

Kerpedjiev, S. and Roth, S. F. 2000. Mapping Communicative Goals into Conceptual Tasks to Generate Graphics in Discourse. In *Proc. Int. Conf. on Intelligent User Interfaces*, New Orleans, LA, (in print).

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

Roth, S. F., and Mattis J. 1990. Data Characterization for Intelligent Graphics Presentation. *Proc. SIGCHI'90*, Seattle, WA, ACM, 193-200.

Roth, S. F., Kolojejchick, J., Mattis, J., and Goldstein, J. 1994. Interactive Graphic Design Using Automatic Presentation Knowledge. *Proc. SIGCHI'94*, Boston, MA, ACM, 112-117.

Roth, S. F., Chuah, M. C., Kerpedjiev, S., Kolojejchick, J. A., and Lucas, P. 1997. Towards an Information Visualization Workspace: Combining Multiple Means of Expression. *Human-Computer Interaction Journal*, Vol. 12, Numbers 1& 2, 131-185.

Shieber, S. 1986. *An Introduction to Unification-based Approaches to Grammar*. Center for the Study of Language and Information. Stanford, CA, 105 p.

Zhou, M., and Feiner, S. 1998. Visual Task Charactererization for Automated Visual Discourse Synthesis. *Proc. CHI-98*, Los Angelos, CA, 392-399.