Chapter V: Implementation

Automatic
Visualization
Interface
Designer

In appendix E we evaluated a set of visualizations generated by our automatic design system, AVID (Automatic Visualization Interface Designer). The evaluation results (appendix E) show that expanding our understanding and vocabulary of visualization primitives to include data computation/transformation operators, perceptual or mapping transform operators and input device components, can enhance our ability to generate visual designs that are interesting and appropriate for our information tasks. This chapter describes how our automatic design system, AVID, is implemented based on the visualization functions framework described in chapters II and III and the visualization design heuristics and metrics described in chapter IV. The implementation of AVID shows that the theoretical concepts we developed previously are complete and specific enough to be applied to a real system. This chapter also highlights the system engineering issues that must be considered to capture the new function classes and heuristics we introduce in our work. Note that all the visualizations shown in this document are generated by AVID unless otherwise noted.

AVID, consists of three components corresponding to the three stages of the automatic design process (shown in Figure V-1):

1. *The task specification component*: Initially, a higher level agent (user or a domain specific system) that has a deeper understanding of the problem domain generates a set of tasks for AVID. Tasks are expressed using a simple language based on the EDA (Exploratory Data Analysis) task model first developed by Tukey [*Tukey, 1977*] and later refined by Casner [*Casner, 1991*] for automatic design. This language is relatively low-level and its purpose is to capture important components of a task that may affect the visual design process. We do not expect typical end users to specify tasks in this language; rather, specifications will most likely be generated by domain specific systems that use graphics to present and summarize their results to users, such as automatic planning systems, automatic information analysis systems, agent based information gatherers, etc. We described general concepts of our task language in appendix C-3. In this chapter we discuss the implementation details of the language and how it is interpreted by AVID. Specifically AVID deals with processing embedded tasks as well as *accuracy* and *iterative* special task conditions that are not dealt with in previous automatic systems but are crucial in our work because of their impact on data transform functions and input devices.

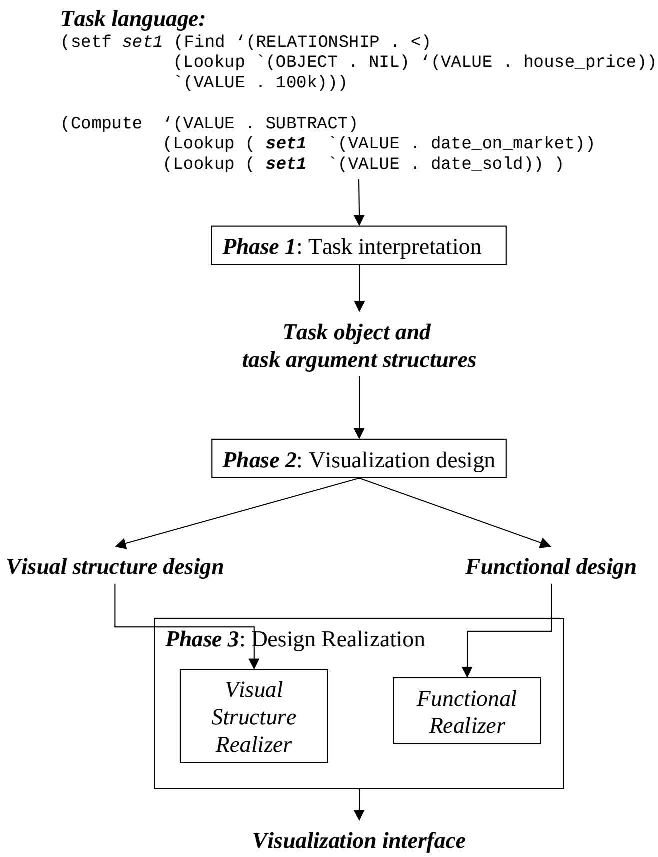


Figure V-1: Three components within AVID that correspond to the three stages in the automatic design process: 1) Task interpretation, 2) Visualization design, and 3) Design Realization

2. *The design component*: In the design component, AVID parses task objects and argument structures generated from the task interpretation component and converts them to design constraints and preferences. Based on these constraints, AVID explores the design space for the input tasks and automatically generates a set of visualizations ordered from best to worst. These output designs are expressed in a language that captures the visual structure of a visualization interface as well as any underlying transform functions and active interactive components. Visual structure descriptions have been developed in previous work [*Mackinlay, 1986a, 1986b; Chuah, 1995*]. This thesis develops a language for capturing the functions and active components within a visualization (described in chapters II and III).

One of the main contributions of our designer is in expanding the visualization design space to include data transforms, mapping transforms, and interactive components (i.e. input devices). Our GOMS evaluation tests in appendix E showed that this expansion allows us to more effectively address the class of analysis tasks considered in this work compared to previous automatic designs that solely rely on mapping operators. The expanded design space enhances human computer communication because a greater visual vocabulary allows more efficient communicative constructs to be generated. In addition, the effectiveness of AVID as a design assistant is also increased because it is able to provide more design alternatives and choices to users. To enable this expansion in the visualization design space, AVID incorporates new procedures in its search algorithm over what has been done in previous systems. Specifically, previous systems only considered how data attributes can be effectively mapped to graphical properties in a *data attribute mapping* procedure (section V-2.1.2). In AVID, we have an additional *task processing* procedure (section V-2.1.1) that considers whether to apply data transforms or mapping transforms to solving tasks, what hybrid data and mapping transforms are valid design alternatives, as well as how to address embedded tasks, object filtering, and unknown task arguments. Our design system also culls out bad designs (i.e. task inexpressive designs or designs that do not support the input task(s)) as well as duplicate designs (section V-2.1.3). This saves users from having to devote attention to these less appropriate visual representations while still having good coverage of the design space. This issue was also not considered in previous systems.

3. *The realization component*: The "realizer" component interprets design specifications generated by the design component and renders an active visualization interface. This component makes layout decisions and assigns default values to visual components that are left unspecified or unconstrained in the design specifications. Currently, AVID's realizer is capable of interpreting most of the selection, transformation, and translation functions described in this thesis (e.g. computations, set-operations, threshold operations, etc). By combining these primitives it can render a wide range of interactive behaviors such as aggregation, painting, dynamic queries, simple semantic zoom, SDM graphical manipulation operations [*Chuah, 1995*], navigation operations, etc. Previous systems could not render designs with data transform functions or designs that contain interactive components.

In the following sections of this chapter we describe how our automatic design system, AVID is implemented and how the concepts laid forth in the previous chapters are captured within its three primary components.

V-1 Task Interpreter Component

The task interpreter component accepts task descriptions as input, analyzes the tasks and their arguments for validity, and then produces a set of *task-class* (Figure V-4) and *task-argument* (Figure V-3) structures. These structures are passed to the design component that uses the information to guide its design strategy. The task interpreter accepts specifications that are in LISP form. An example task specification is shown in Figure V-2.

In this task we calculate the duration that houses under 100k in price, stay on the market. Each task within the specification has three parts: the *task class*, the *task input arguments*, and any special *task conditions*. For example, the top task in Figure V-2 can be decomposed into a *find* task class, a list of three task arguments, with no special task conditions. The bottom task in Figure V-2 can be decomposed into a *compute* task class, a list of three task arguments, and two task conditions. The embedded *lookup* tasks within the bottom *compute* can be decomposed in the same way.

```
(setf set1 (Find      '(RELATIONSHIP . <)
                      (Lookup '(OBJECT . NIL) '(VALUE . selling_price))
                      '(VALUE . 100k)))
```

1. Task operator *2. Task arguments*

```
(Compute     '(VALUE . SUBTRACT)
             (Lookup set1  '(VALUE . date_on_market))
             (Lookup set1  '(VALUE . date_sold)) )      :loop-type one-to-one
                                                       :accuracy 'neutral )
```

1. Task operator *2. Task arguments* *3. Task conditions*

Figure V-2: Example task specification

The three task parts encapsulate the following information:

1. *Task class*: The task class captures the main goal of the current analysis operation. Different task classes require different numbers and types of input arguments as is summarized in Table V-1. Currently AVID can interpret five different task classes: *lookup, find, AND, compare,* and *compute*. These task classes can commonly be used to describe problems that arise in data analysis [*Casner, 1991; Senay, 1994*].

2. *Task arguments*: Task arguments may specify single or sets of values. For example the arguments `'(VALUE . SUBTRACT)`, and `'(VALUE . 100k)`in Figure V-2 contain single values (*subtract* and *100k*). On the other hand the argument `'(VALUE .`(1 2 3 4 5))` contain a set of five values. Task arguments may come in one of two forms:
   a) *Associative value pairs*: An associative pair has two elements (e.g. `'(VALUE . SUBTRACT)`), the first describes the task argument type (e.g. *value*) and the second contains the actual argument value(s) (e.g. *subtract*). Currently we have three types of arguments: *value*, *object*, and *relationship*. *Value* arguments may be numbers or strings; *object* arguments refer to conceptual structures within the visualization such as a *data concept*, a *graphical object*, or a *chart region*; and *relationship* arguments describe relations that may exist among objects or values. Currently, we only consider simple value relationships such as >, <, and =.
   b) *Output results from other tasks*: Tasks may also accept output argument structures that are generated by embedded tasks. The compute task in Figure V-2, for example, accepts results from two embedded *lookup* tasks. Each of these *lookup* tasks produces a set of values corresponding to the *date_on_market* and *date_sold* attributes of the *house* data concepts associated with the visualization.

3. *Task conditions*: Apart from the regular task input arguments, we may also specify special task conditions. Currently, AVID can process two types of conditions, namely the *task loop type* and the *task accuracy level*. Details on these two task conditions can be found in appendix C-3.2.

The *task loop type* describes how the task input value sets should be iterated over. There are three iteration types in our task framework:

a) *One-to-one* is the default iteration type. It specifies that each value in the first set is processed with the corresponding values in all subsequent sets (e.g. the 1^{st} value in each set are processed together, and so are the 2^{nd}, 3^{rd}, 4^{th}, and *n-th* values);

b) *All-to-all* tasks require each value in an input set to be processed with each and every value in the subsequent sets.

c) *Previous-pair* tasks order the input value sets based on an ordering attribute, then applies each consecutive pair of values within the ordered set to the task function.

The *task accuracy level* describes the level of accuracy that is desirable for the task. Currently, there are three accuracy levels, *accurate, neutral* and *fuzzy*.

AVID's task interpreter possesses a LISP function corresponding to each task class (i.e. *lookup, find, AND, compare,* and *compute*). When activated, each LISP function analyzes the task input arguments to ensure their validity. This includes number-of-argument checks, argument-type checks, and argument-correspondence checks. The number and type of arguments required for each task class is summarized in Table V-1.

	Input arguments	Output argument types
Lookup	• *1 object* argument containing the set of objects to perform the lookup on. *E.g.* `'(object .(house-1 house-2 house-3))` Note that an empty object set defaults to all objects in the database. *E.g.* `'(object . nil)` • *1 value* argument containing the lookup attribute name. *E.g.* `'(value . date-on-market)`	• *1 value* argument
Compute	• *1 value* argument containing the compute operator to apply (e.g. *add, subtract,* etc). • *n value* arguments containing the data value sets to compute.	• *1 value* argument
Find	• *1 relation* argument containing the find relationship to apply (>, <, =, >=, <=). • *2 value* arguments containing the data value sets to search on.	• *1 object* argument
AND	• *n object* arguments containing the object sets involved in the *AND* relationship.	• *1 object* argument
Compare	• *2 value* arguments containing the data value sets to compare.	• *1 relation* argument

Table V-1: Task inputs and outputs

Argument-correspondence checks ensure that the input data types are consistent with the task. For example, only arguments of the same type can be added or subtracted from each other. It is not possible to perform *additions* and *subtractions* on a set of *price* data values and a set of *weight* data values. The same correspondence constraint applies to *find* and *compare* task classes.

When a task is specified correctly (i.e. all its arguments are valid), the task LISP function generates:

1. *Output argument structures*: The output arguments generated have the same form as any input *task argument* structure (an example is shown in Figure V-3). It captures properties of the task results that are derived from the task class and the task input arguments. Association value pair arguments (e.g.`'(VALUE . SUBTRACT)`) are converted into task argument structures based on the pair values and the argument's parent task. For example, a value pair argument in a *lookup* task (e.g.`'(VALUE . selling_price)`) implies that the second value in the pair (*selling_price*) is an attribute name. On the other hand, a value pair in other task classes may imply a data value (e.g.`'(VALUE . 100k)`) or a relationship value (e.g.`'(VALUE . <)`) depending on the expected input arguments of the task class (as is shown in Table V-1). The output argument structure may be passed on to other tasks as inputs, which is what occurs when we embed one task within another. Input arguments from embedded tasks are already in the desired argument structure form (as was described above) and thus need not be further processed. For example in Figure V-2, the *find* task generates a task argument structure containing a set of data concepts and passes that on to the *lookup* tasks which extract the *date-on-market* and *date-sold* values from those concepts. These two sets of values are subsequently passed on to the *compute* task. Sometimes these output structures may contain newly generated derived or summarization attributes (within its *content* slot) that are used to store the results of a data computation function. For example, the *find* and *AND* tasks generate a boolean attribute (attribute containing T or F values), the *compute* task generates a value summarization or derived attribute, and the *compare* task generates a relationship attribute (attribute containing >, <, or = values).

```
(defclass task-argument (primitive-object)
   ( class         ;; Argument type: OBJECT | VALUE | RELATION
     parent        ;; all tasks that contain this argument
     within        ;; task which produces this task argument
     content       ;; Data attribute or value(s) associated with argument
     viz-function  ;; Internal function used to process results for this argument
    ))
```

Figure V-3: Task argument structure

2. A *task class* structure: An example *task class* structure is shown in Figure V-4. Each task within the input specification is translated into a *task class* structure. All input argument structures associated with the task are collected and placed within the *arg-list* field slot.

```
(defclass task-class (primitive-object)
   ( class        ;; task class: either [ LOOKUP, COMPUTE, COMPARE, FIND, AND ]
     arg-list     ;; input task arguments
     is-embedded  ;; whether task is embedded within another [ t | nil ]
     accuracy     ;; task result accuracy [t = accurate, nil = approximate]
     loop-type    ;; Loop method on input objects[one-to-one|all-to-all|previous]
     num-times    ;; task frequency: number of times a task is to be repeated
     output-arg   ;; output task argument structure produced by task
    ))
```

Figure V-4: Task class structure

V-2 Automatic Design Component

Our automatic designer is implemented using Common LISP (Allegro version 4.0), and a constraint satisfaction system called SCREAMER [*Siskind*]. The designer accepts a list of task class structures (shown in Figure V-4) and task argument structures (shown in Figure V-3) as input and produces a set of design specifications as output, ordered according to task effectiveness. Each design in the output set fulfills all the input task requirements. A design specification consists of two components: a) a structural description of the graphical components within a visualization and b) a description of the functional components within a visualization (this functional specification corresponds to the framework language described in chapters II and III of this thesis).

In the following sections we outline our strategy for exploring the space of visual elements and visualization techniques as well as describe how the heuristics provided in chapter IV can be encoded as design constraints and design costs. The constraint and cost structure directs the search algorithm and allows the AVID design component to generate an ordered list of designs that reflect cognitive, perceptual, and articulatory complexity with respect to the input task(s).

V-2.1 Search Strategy

AVID's search procedure has two primary phases: the *task processing* phase, and the *data attribute mapping* phase. These two phases are indicated on the search strategy flowchart in Figure V-5. The section to the left describes the *task processing* phase, the section to the right describes the *data attribute mapping* phase. The *data attribute mapping* phase is what was performed in previous automatic systems. **To enable the design of visualizations that contain data transforms and input devices, we added the *task processing* phase**. The *task processing* phase begins with the first outermost task and then proceeds to all embedded tasks within it. Consider the house task described previously (Figure V-2).

In this task we are interested in seeing whether houses costing less than 100k stay on the market for relatively short periods of time. The task specification is shown again below:

```
(setf set1 (Find       `(RELATIONSHIP . <)
                       (Lookup `(OBJECT . NIL) `(VALUE . selling_price))
                       `(VALUE . 100k)))


(Compute       '(VALUE . SUBTRACT)
               (Lookup set1  '(VALUE . date_on_market))
               (Lookup set1  '(VALUE . date_sold)) )       :loop-type one-to-one
                                                           :accuracy nil )
```

Before the task processing phase begins, AVID's design component orders its input set of tasks according to their embedding structure, from outermost to innermost. Based on this ordering method, the house task described above would be organized as follows (the numbers in the angle brackets "[]", indicate embedded tasks):

```
1. (Compute '(VALUE . SUBTRACT) [2] [3])
2. (Lookup [4] '(VALUE . date_on_market))
3. (Lookup [4] '(VALUE . date_sold))
4. (Find '(RELATIONSHIP . <) [5] '(VALUE . 100k))
5. (Lookup '(OBJECT . NIL) '(VALUE . selling_price))
```

Task processing starts with the *compute* task and proceeds until the *lookup selling_price* task. During the task processing phase, the designer decides what data to pre-process, what data to show to users, and how to constrain the mapping from data to graphics in order to facilitate perceptual processing. Once all the embedded tasks are processed, the search algorithm proceeds to the data attribute mapping phase, where all data attributes deemed necessary in the task processing phase are mapped to graphical properties. The mapping decisions are subject to the perceptual constraints placed during task processing.

Branching in the search procedure occurs when there are alternative methods for achieving the same goal. In Figure V-5, a black circle indicates these *branching* or *alternative* points. An important branching point, for example, occurs after the "*Process next task*" node. One alternative is to process the task with data transforms (i.e. have the system pre-compute the task results). Another alternative is to map the task data to graphical properties and let users derive the task results perceptually. Each of these alternatives causes a new path to be created in the search tree. To further differentiate these "*branching points*" in Figure V-5, we curve the arrows originating from them while leaving all other arrows rectangular. In the next sections we describe the two search phases, *task processing* and *data attribute mapping*. We will show how a search tree is constructed and what state information is stored within its nodes during each of these phases.

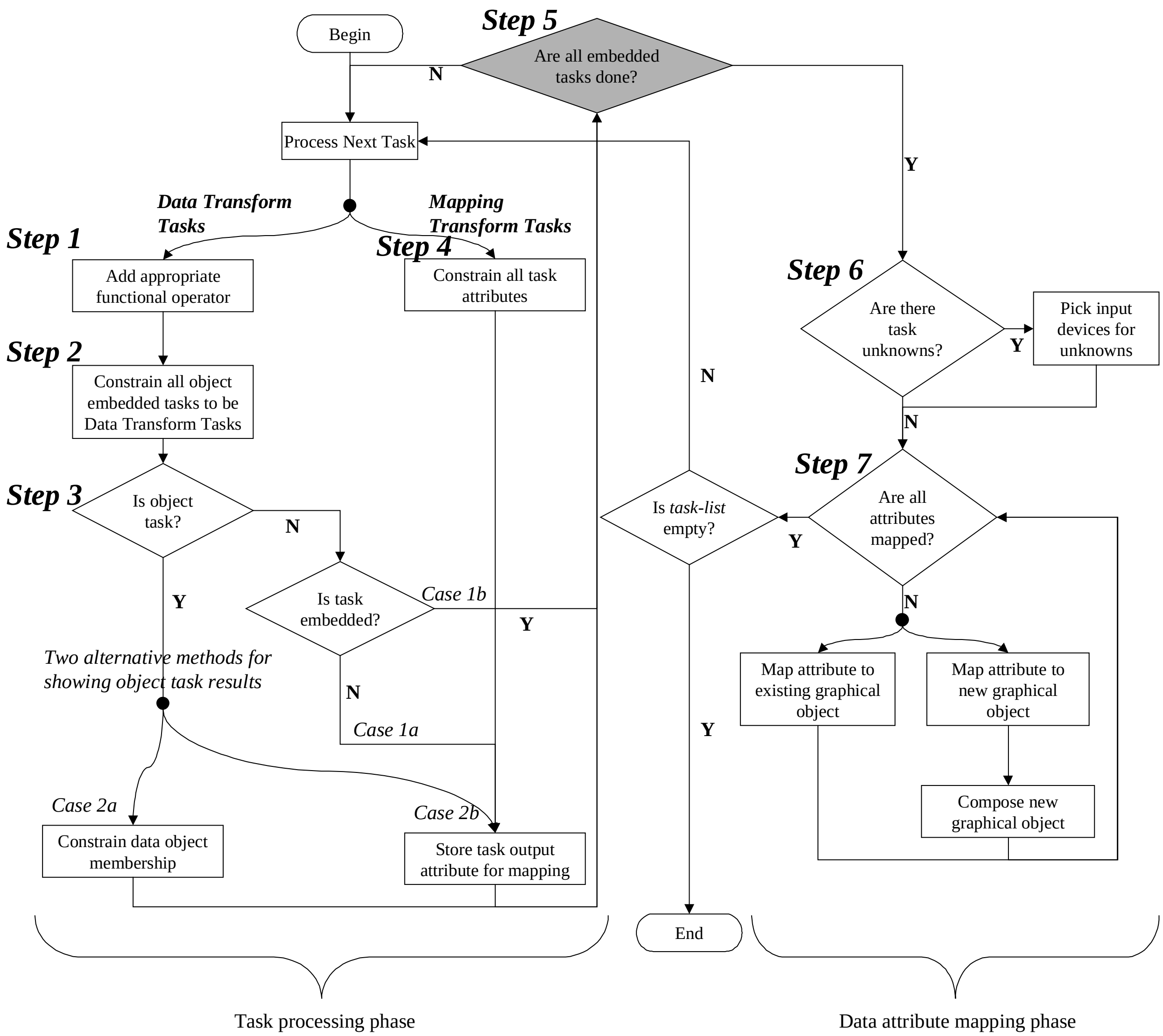


Figure V-5: Flowchart of AVID search strategy. Consists of two main phases: 1) task processing phase and 2) data attribute mapping phase

V-2.1.1 Task Processing

At the start of the search procedure the search tree contains a single root node that has a list of all outstanding tasks ordered from outermost to innermost. For example the partial search tree in Figure V-7 reflects the design space of the house example task described earlier in this section. The root node of this search tree contains a list of task structures related to the house example. Task processing begins with the first task in the root node, which is the *compute-subtract* task. Each task may be performed through data summarization and manipulation operators or by mapping data attributes of the task to appropriate graphics. As is shown in Figure V-7, each of these alternatives generates a new path in the search tree (*node-1* is the data transform alternative and *node-8* is the mapping transform alternative). New additions to the node state at each step are shown in **bold red letters**.

Note that we have cut out some nodes from the search tree in Figure V-7 because of space constraints. In particular, *node-1* and *node-8* are linked to the root node by dotted arrows to indicate that some intermediate node states along these two paths have been culled out. Both nodes show the state of the design after processing the first three tasks (i.e. the *compute* task and both of its embedded *lookup* tasks). We collapsed these two path segments because in both cases, the embedded *lookups* are constrained to the same task processing method as their parent *compute* task (i.e. both *lookup*s are constrained to data transforms in *node-1* and constrained to mapping transforms in *node-8*). As a result there is no branching in the tree within these segments and collapsing them does not remove any information.

In the following paragraphs we describe the steps associated with the data and mapping transform processing alternatives (as is shown in Figure V-5) as well as the changes they make upon the node states in Figure V-7.

Alternative 1: Data Transform Processing

Step 1: Add functional operator

When we decide to perform a task through data computation (i.e. system computation), a data transform operator corresponding to the task class is generated and stored within the node. For example in *node-1* of Figure V-7 the *compute* task is performed with data transforms, thus a *BinaryCompute* data transform operator is added to the *functional-operator-list* slot of the node. Similarly each of the other data transform tasks also adds a data transform operator to the node state. These transform functions are later connected and used to create a functional design for processing the data contained within the visualization.

There is currently a one-to-one correspondence between our task classes and the data transform primitives described in chapters II and III. Figure V-6 shows all our task classes and their corresponding data transform operators.

Task operator	***Corresponding data transform primitive***
Lookup	GetAttributeValue
Compute (*mean, min, max*)	UnaryCompute
Compute (*add, subtract, divide, multiply*)	BinaryCompute
Compare	GetValueRelation
Find	Threshold
And	SetOperation

Figure V-6: Task operators and their corresponding visualization functions

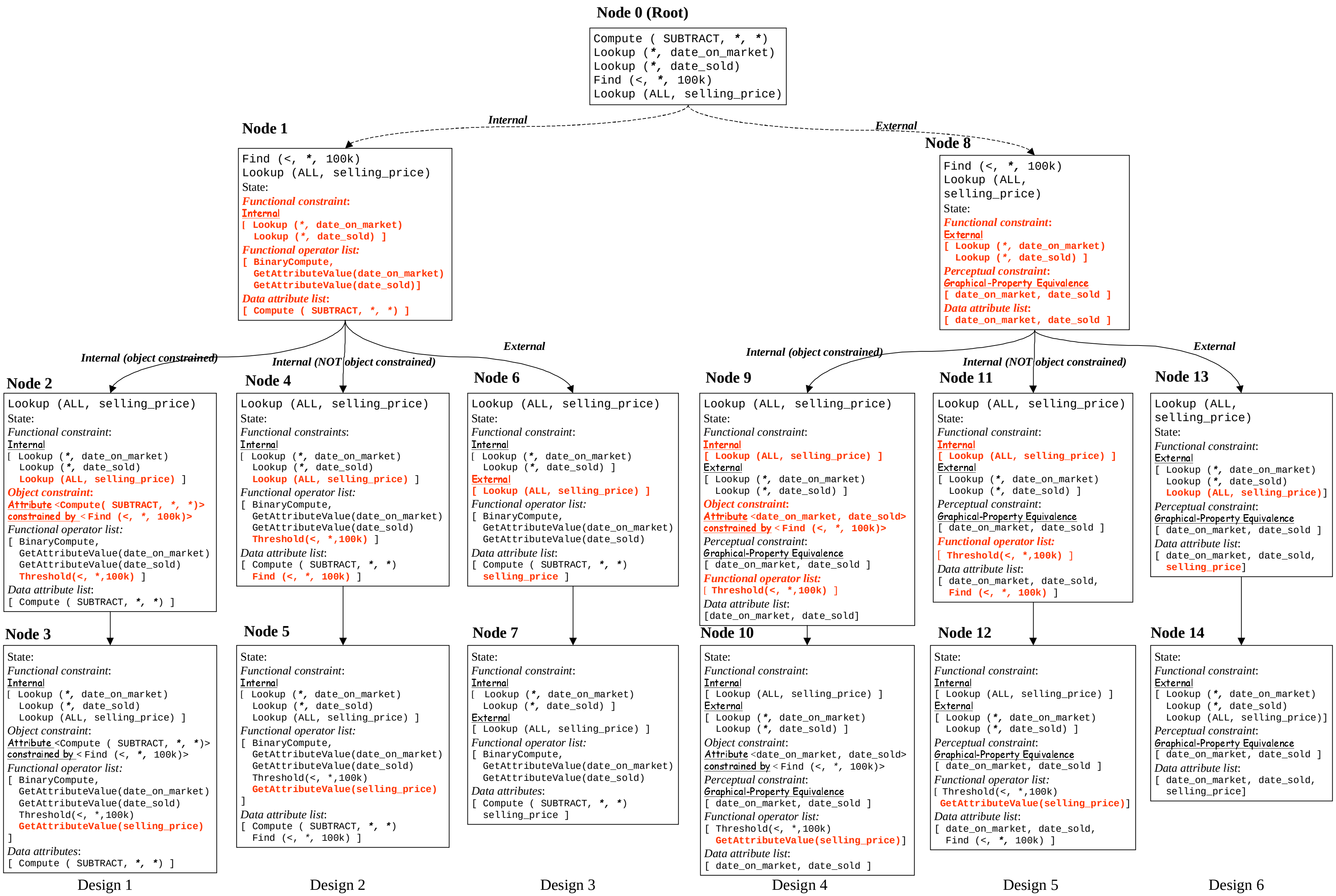
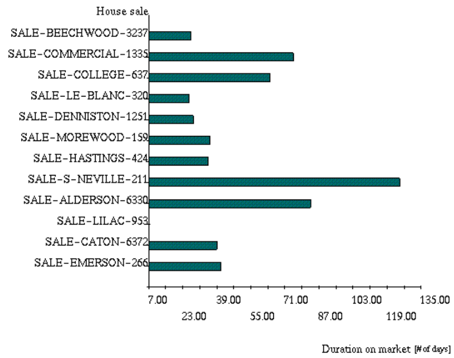
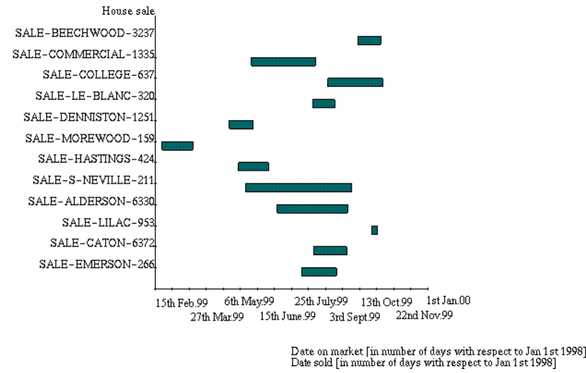


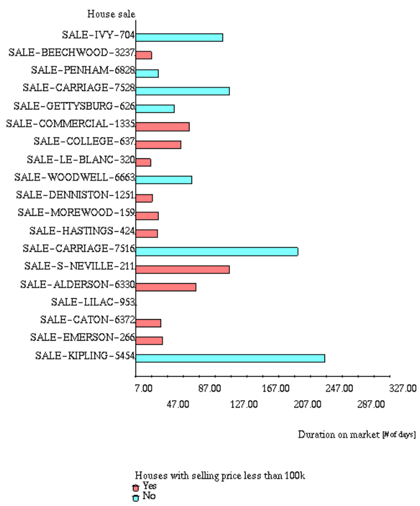
Figure V-7: Partial search tree of house example task



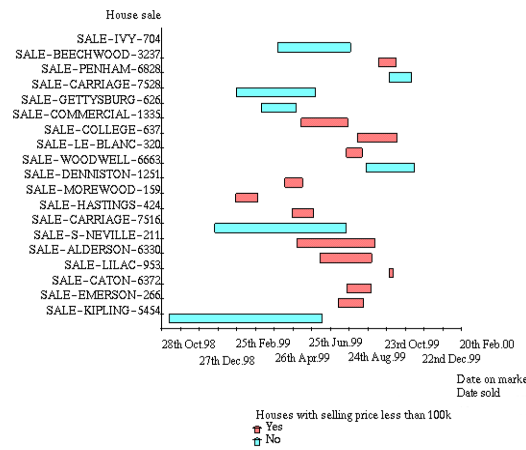
Design 1: Pure data transform design with object filtering (node-3). Duration on market is computed and mapped to the *x-axis* and only houses costing less than 100k are shown



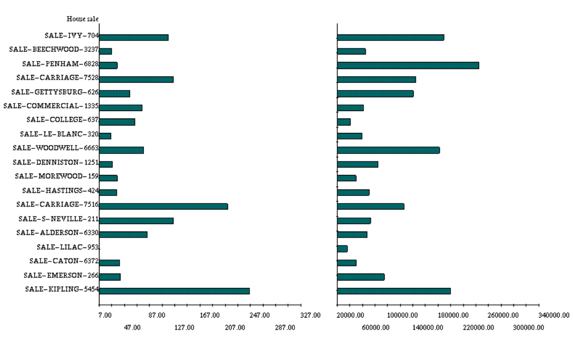
Design 4: Hybrid design with *date_on_market* and *date_sold* mapped to the *x-axis* so that the duration on market can be determined perceptually. The *find* selling-price task is computed with data transforms and only those houses costing less than 100k are shown (node-10)



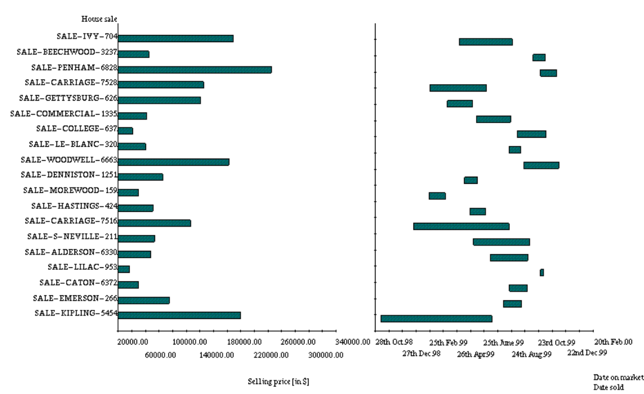
Design 2: Pure data transform design with NO object filtering (node-5). Duration on market is computed and mapped to the *x-axis*. Houses costing less then 100k are computed and shown in *red*.



Design 5: Hybrid design with *date_on_market* and *date_sold* mapped to the *x-axis* so that the duration on market can be determined perceptually. The *find* selling-price task is computed with data transforms and shown using *hue* (node-12)



Design 3: Hybrid design with duration on market computed and mapped to the *x-axis* of the left chart. The *find* selling-price task however is performed perceptually and *selling-price* is shown on the *x-axis* in the right chart. (node-7)



Design 6: Pure mapping transform design, i.e. all tasks are mapped to graphics. *Selling-price* is mapped to the *x-axis* of the left chart and *date_on_market* and *date_sold* are mapped to the *x-axis* of the right chart (node-14)

Figure V-8: Example designs generated corresponding to the 6 terminal nodes in the search tree in Figure V-7

Step 2: Constrain embedded tasks based on whether they have an object or non-object output argument

In this step we determine whether the tasks embedded within the current task are *non-object* or *object* tasks. In the first case, a data transform task constrains all of its **embedded non-object tasks** (i.e. *lookup*, *compute* or *compare*) to data transforms as well. For example *node-1* in Figure V-7 performs the *compute* task with a data transform. Since the two embedded *lookup*s within the *compute* are value tasks (i.e. non-object) they are constrained to data transform functions as can be seen in the *functional-constraint* list of the node. This constraint is in place because to fulfill a task with data transform functions when its embedded tasks are performed through mapping transforms, users must perform those embedded tasks perceptually and than convey their results to the parent data transform task. Commonly this requires great precision in the mapping transform task and significant articulatory costs in conveying its results. For example in the house task, if we were to perform the *compute* with data computation but the embedded *lookup'*s with mapping transforms, the user must enter in two sets of values, one corresponding to the *date_on_market* lookup and another corresponding to the *date_sold* lookup, resulting in very significant articulatory costs.

In the second case, **embedded object tasks** (*find, AND*), do not have this data transform task constraint. Consider *node-6* in Figure V-7, the outer *lookup* task is performed with data transforms but its embedded *find* task can be performed with a mapping transform because it is an object task. While articulatory costs tend to be large for multi-value entry, they are significantly smaller for selecting a set of objects. Input devices such as *lassos* and *bounding-boxes* allow multiple objects to be selected simultaneously, while value entry must be performed individually, incurring a cost for each input value. *Lassos* and *bounding-boxes* are especially effective in the case where the resulting object set is dependent on the objects' positions. In addition input devices are often not even needed for embedded object tasks because users can perceptually identify those objects of interest and then just lookup their attributes. There is no need to convey to the system which objects fulfill the *find* task conditions unless we want to filter the objects within the display. For example consider *Design 3* which allows users to perceptually search for houses under 100k in price by looking at the *selling-price* bar chart to the right. Once those houses are identified, users may look up their *computed* duration on market. In this design, the *find* task is mapped to graphics, while the *compute*, *lookup date_on_market* and *lookup date_sold* tasks are data transform computed, however, no system input is required of the user.

*Step 3: Process **current** task based on whether it is embedded and whether it has an object or non-object output argument*

Once we have added appropriate transform operators and constrained all **non-object embedded tasks** to data transforms, we check whether the task itself is embedded and whether it is an object task. Table V-2 summarizes the different actions that get carried out based on these two conditions.

Task type	**Task embedding**	**Action taken**
Non-object	***(1a) Not-embedded***	Map to graphical property.
	(1b) Embedded	Do nothing.
Object	***(2a)***	Filter visualization objects.
	(2b)	Map to graphical property.

Table V-2: Summary of actions taken based on task output and embedding status

First we consider non-object tasks (*case 1*). For non-object, non-embedded tasks (*case 1a*), (like the *compute* task in the house example above) we assume that there is an implicit outer *lookup* task around it. This is because in data analysis we perform tasks to gain insight from their results and not for the exercise of performing the task itself. Thus for a non-embedded task, we need to map its resulting derived attribute to a graphical property. In *node-1*, for example, we add the derived attribute for the *compute-subtract* task to the *data-attribute-list* of the node for future mapping. In *design 1*, this *compute* attribute later gets mapped to *x-position*. Embedded tasks (*case 1b*), however, pass their results onto higher level tasks for subsequent processing, thus it is less important to show their intermediate results (refer to chapter IV-3.2). For embedded, non-object tasks we take no action and proceed to the next task.

When the task being considered is an object task (*case 2*), we have two alternatives. Either we can filter the graphical region(s) so that only those objects that fulfill the object constraints of the task are shown (*case 2a*) or we map the task results (a boolean attribute) to a graphical property (*case 2b*). These two alternatives result in *node-3* and *node-5* of the search tree in Figure V-7. In *node-3* an object constraint is applied while in *node-5* the *find* task result attribute is added to the *attribute list* of the node for subsequent mapping.

Choosing the filter alternative may significantly reduce the number of objects that need to be shown and thus reduce clutter (as is shown in *Design 1*). On the other hand such a design makes it harder to maintain data context because when we alter the conditions of the *find* task the data membership of that visualization changes and causes objects to shift around to fill in empty spaces or to make new spaces for additional objects. The object filter decision also constrains data membership for a set of graphical objects, and this may preclude the results of other related tasks with different data membership requirements from being shown with the same graphical objects. As a result object filtered visualizations usually tend to be less integrative, spreading the related data attributes over more objects and possibly more regions.

The second object task alternative (*case 2b*) maps the task results to a graphical property. This does not create context maintenance and integration problems, as is the case with filtering the objects in the previous case (*case 2a*).

However, this design decision results in greater perceptual complexity because more graphical objects are shown and an additional graphical property must be used to show the results of the search.

AVID takes the costs associated with these two alternatives into account during the design process. A cost is associated with mapping the task results to a graphical property. Less integrated designs that result from different object membership requirements also incur a design cost. AVID's cost structure is described in greater detail in section V-2.3.

Alternative 2: Mapping Transform Task Processing

Step 4: Add perceptual constraints

An alternative to performing a task with data transforms is to perform the task through mapping transforms which encodes task related data attributes with graphical properties. Mapping transform tasks commonly impose perceptual constraints on the data to graphical encodings to facilitate perceptual processing. To perform a *compute-add* task with mapping transforms, for example, we constrain all input child attributes to be mapped to stackable graphical objects and properties, the *compute-subtract* task, on the other hand, constrains all data attributes to be mapped to the same graphical property so that value comparisons are facilitated. In Figure V-7, *node-8*, for example, performing the *compute-subtract* task with mapping transforms causes a "*simple graphical-property*" constraint to be added. This constraint restricts both the *date_on_market* and *date_sold* data attributes to be mapped to the same graphical property. We see in *designs 4, 5,* and *6*, that both attributes are mapped to the *x-position* graphical property and this facilitates the perceptual *subtraction* task. More details on perceptual constraints are provided in section V-2.2.

Step 5: Determine if all embedded tasks have been processed

After we have finished processing a task we remove it from the *task-list* field of a node and continue to process the next task in that list. Processing continues until all embedded tasks have been visited. Once this is done we proceed to the *data attribute mapping* phase where we consider how the data attributes collected in the *data-attribute-list* slot in the node state can be mapped to graphical objects and properties.

V-2.1.2 Data Attribute Mapping

In this phase we consider all the data attributes in the *data-attribute-list* slot of a node and explore the different ways in which these attributes can be mapped to graphical properties. Preference in mapping is given to graphical properties that allow parallel processing and integrated designs with low perceptual complexity.

Step 6: Add input devices for unknown task arguments

Before we begin mapping the data attributes we first populate the visualization design with input devices for all unknown task arguments (i.e. task arguments that do not have associated data values because the user is unsure which value(s) are most suitable for the task). Unknown arguments are specified in AVID by using a '**?**' symbol in place of an object or data value set. For example if we were unsure of what *selling_price* threshold to use in the house example, we would specify the task as follows:

```
(setf set1 (Find       `(RELATIONSHIP . <)
                       (Lookup `(OBJECT . NIL) `(VALUE . selling_price))
                       `(VALUE . ?)))


(Compute        '(VALUE . SUBTRACT)
                (Lookup set1  '(VALUE . date_on_market))
                (Lookup set1  '(VALUE . date_sold)) )       :loop-type one-to-one
                                                            :accuracy nil )
```

In this newly modified task, AVID will attach input devices to those design alternatives where the *find* task is performed with data transforms so that users may change the *selling_price* filtering threshold similar to a dynamic query interface [*Ahlberg, 1994*]. Input devices need not be added when tasks are mapped to graphics because unlike the data transform case, all the data for the task is shown and users can just perceptually process the data differently based on changing task conditions.

Step 7: Map all data attributes

The attribute mapping process is similar to the mapping process used by Casner and Mackinlay. Attributes are mapped to properties of an existing graphical element or to properties of a new graphical element that is then composed with current graphical objects. There are four types of composition methods, each of which produces a new branch in the search tree: *cluster*, *double axis*, *single axis alignment* and *no composition*.

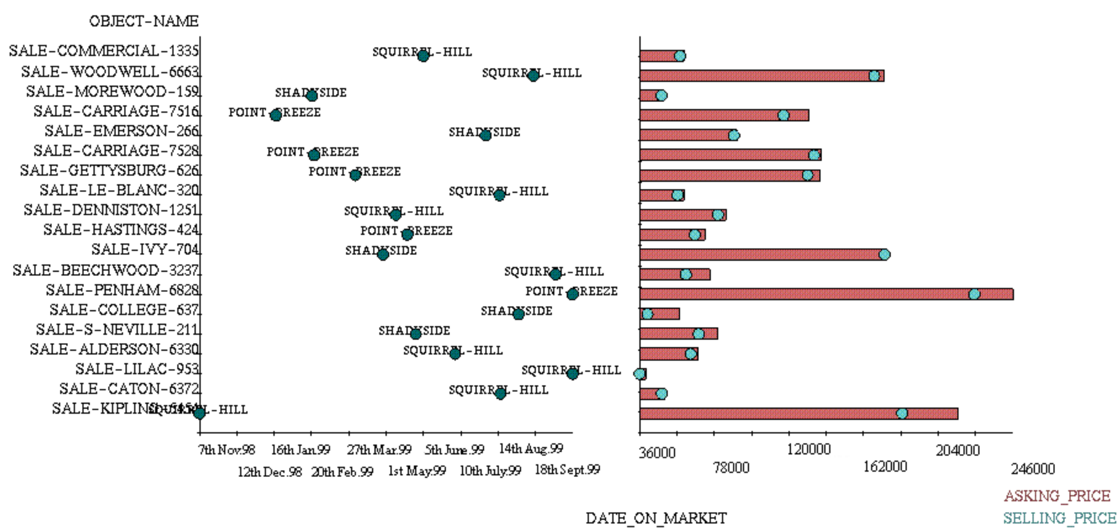


Figure V-9: Visualization design illustrating the different composition types. There is *cluster* composition in the left chart between the *labels* and the *marks*. There is *double-axis* composition in the right chart between the *marks* and the *bars*. There is *single-axis* alignment between elements in the left-chart and those in the right-chart. These composition types were first introduced by Mackinlay [*Mackinlay, 1986a, 1986b*].

Clustering ties a new object positionally to an existing object in the partial design. For example in Figure V-9, a label showing house *neighborhood* is clustered with a *mark* graphical object. *Double axis composition* adds a new object into an existing space but does not tie it positionally to any other object within that space. For example the *mark-grapheme* and the *horizontal-bar-grapheme* in the right region of Figure V-9 are *double axis composed* because even though they reside in the same region, they do not share the same *x-position*. *Single axis alignment* adds a new object in a new region that shares at least one positional axis with an existing region. The two regions in Figure V-9 are *single axis composed*, sharing their *y-positional* axis. It is important to note that to share a positional axis both axes must have the same data type and the same min-max range. In Figure V-9, both regions are aligned on the *object-name* data type. Finally *no-composition* shows the data within a new visualization window. Different costs are associated with different graphical property mappings depending on the task and the data attribute being mapped. Costs are also different for the various object composition methods. Cost details are discussed in section V-2.3.

V-2.1.3 Post Design Processing

Once all tasks are processed and all data attributes are mapped, the visualization design is complete. During post design processing, AVID's designer performs two primary tasks:

1. *Culls out similar designs that have been generated previously*

   A completed design is compared to all previous designs to determine whether there is similarity in its *structure* and *content*. If there is a structural and content match[1], then the newly completed design is culled from the design space and a new design is generated. Otherwise, the new design is transformed into our design specification language and sent to the realizer component for rendering.

2. *Transforms the visualization design into a visual and functional specification language*

   Once a design is completed and determined to be unique (i.e. does not match based on structure or content to any previous design), AVID's designer translates the completed solution into a visual structure specification language and a functional specification language (developed in chapters II and III of this thesis). The translated design is then passed on to the realizer component. In appendix D-2 we describe the functional translation process, i.e., how the visualization operators within the *functional-operator-list* slot of a node are connected together and populated with sufficient information for subsequent rendering. The translation process from visual design to visual structure specification language is fairly straightforward and has been explored previously [*Mackinlay, 1986a, 1986b; Chuah 1995*], thus we will not describe this process again here.

V-2.1.4 Summary

In this section we described the search procedure used in AVID's design component. This procedure consists of two main phases: task processing phase and data attribute mapping phase. The *attribute mapping* phase was adopted from previous work on automatic systems. However, we created and added the *task processing* phase to address data transform and input device issues. Specifically we describe how to address embedded tasks and utilize

---

[1] Refer to appendix D-1 for details on structural and content matching.

the task embedding structure to achieve more effective data transform designs, how to filter objects within a visualization with data transforms and its impact on the graphical representation, how data transform functions are combined and attached to task parameters, and how to address unknown task arguments by attaching input devices.

During the two search phases costs are assigned to the partial visualization designs within each node of the search tree and constraints are placed on the various design elements (as is shown in Figure V-7). These costs and constraints help guide the search so that inexpressive design paths (i.e. designs that are not appropriate for the input tasks) are abandoned and promising design alternatives are explored first. Many of these cost and constraint heuristics are based on previous research that explore how visualizations can be used to amplify cognition. Card et al. [*Card, 1999*] summarizes these findings very well and we show them in Table V-3.

Increased Resources	
1. High-bandwidth hierarchical interaction	The human moving gaze system partitions limited channel capacity so that it combines high spatial resolution and wide aperture in sensing visual environments (Resnikoff, 1987).
2. Parallel perceptual processing	Some attributes of visualizations can be processed in parallel compared to text, which is aerial.
3. Offload work from cognitive to perceptual system	Some cognitive inferences done symbolically can be recoded into inferences done with simple perceptual operations (Larkin and Simon, 1987)
4. Expanded working memory	Visualizations can expand the working memory available for solving a problem (Norman, 1993)
Reduced Search	
5. Locality of processing	Visualizations group information used together reducing search (Larkin and Simon, 1987)
6. High data density	Visualizations can often represent a large amount of data in a small space (Tufte, 1983)
7. Spatially indexed addressing	By grouping data about an object, visualizations can avoid symbolic labels (Larkin and Simon, 1997)
Enhanced Recognition of Patterns	
8. Recognition instead of recall	Recognizing information generated by a visualization is easier than recalling that information by the user.
9. Abstraction and aggregation	Visualizations simplify and organize information, supplying higher centers with aggregated forms of information through abstraction and selective omission (Card, Robertson, and Mackinlay, 1991; Resnikoff, 1987)
10. Visual schemata for organization	Visually organizing data by structural relationships (e.g. by time) enhances patterns.
11. Value, relationship, trend	Visualizations can be constructed to enhance patterns at all three levels (Bertin, 1977/1981)
Perceptual Inference	
12. Visual representations make some problems obvious	Visualizations can support a large number of perceptual inferences that are extremely easy for humans (Larkin and Simon, 1987)
13. Graphical computations	Visualization can enable complex specialized graphical computations (Hutchins, 1996)
14. Perceptual Monitoring	Visualizations can allow for the monitoring of a large number of potential events if the display is organized so that these stand out by appearance or motion.
15. Manipulable Medium	Unlike static diagrams, visualizations can allow exploration of a space of parameters values and can amplify user operations.

Table V-3: How Information amplifies cognition (from Card et al.[*Card, 1999*])

In the following sections we describe the constraint and cost structures used in AVID as well as discuss their use, limitations, and how they relate to Table V-3.

V-2.2 Design Constraints

We divide AVID's design constraints into two classes according to the two primary design phases: *task processing* and *data attribute mapping*. *Task processing constraints* ensure that the visualization can be effectively controlled (small *articulatory* distance) and parsed (small *observational* distance) by the user. In addition, the visual design must be capable of expressing or processing all the information required by the input task(s) (*expressive* and *functional* distances = 0)[2].

Data attribute mapping constraints, on the other hand, ensure that the structure of the visualization is valid. For example, graphical objects within a *map* region must have their *positions* mapped to *longitude* and *latitude* while objects within a *grid* have no positional mappings. In addition, *regions* may only be aligned if they share at least one axis with the same *data type*. These *data attribute mapping constraints* follow established information design rules and their application to automatic visualization design have been explored in previous work [*Mackinlay, 1986a, 1986b; Casner, 1991; Roth, 1994*]. Therefore, in this section we will focus only on *task processing constraints*.

Task processing constraints are characterized based on three dimensions: *softness, scope,* and *constraint-condition*. We describe each of these dimensions next, as well as detail the primary areas in the task processing phase where these constraints get imposed.

V-2.2.1 Constraint Dimension 1: Softness

Constraints may be applied as *hard* or *soft constraints*. *Hard constraints* cannot be violated. Any search path that violates a hard constraint is considered a failure and abandoned. In AVID we use hard constraints to prevent the designer from generating visual representations that are not functionally and/or visually expressive of the input tasks.

Specifically, hard constraints are applied in AVID so that designs with positive task expressive or functional distances are never generated because they do not provide users with sufficient information to solve the input tasks. Note, however, that hard constraints are not applied to all the expressive distance measures. In chapter IV-1.2, we listed four expressiveness measures: task expressiveness, data expressiveness, correctness, and data presence. Task expressiveness and correctness are necessary conditions in any AVID generated design because they determine whether a task can even be performed and if so whether it can be performed correctly. As a result these measures are implemented as hard constraints. Data presence and data expressiveness restrictions, on the other hand, are not crucial to completing a task, thus they are implemented as soft constraints. In fact, data summarization, which

[2] Articulatory, functional, expressive, and observational distances were all described in the metrics framework in chapter IV-2.

decreases data expressiveness, can be a powerful tool for reducing graphical complexity and improving perceptual processing [*No.9, Table V-3*].

Hard constraints are also applied to prevent perceptually inexpressive designs from being generated. For example in Figure V-10, the *date_on_market* and *date_sold* attributes are mapped to two different graphical properties (*x-position* and *saturation*) making it very difficult to determine the duration on market without resorting to cognitive calculation. In this case the perceptual mappings are a hindrance because they do not enable any perceptual operators for performing the *compute* duration task (i.e. because the graphical properties used to represent the data are inexpressive, it is not possible to offload the cognitive operations onto the perceptual system [*No.3, Table V-3*]). As a result the graphical property values must first be converted back into data values and then the computation task must be performed cognitively. To prevent this, we enforce a *graphical-property* constraint (which restricts both *date_on_market* and *date_sold* to the same graphical property) as a *hard* constraint. Consider all the designs resulting from the subtree at *node-8* that has this *graphical-property* constraint (i.e. *Design 3*, *Design 4*, *Design 5*). All these designs allow the *compute* duration task to be performed perceptually by mapping both *date_on_market* and *date_sold* to the same graphical property class, namely *x-position*.

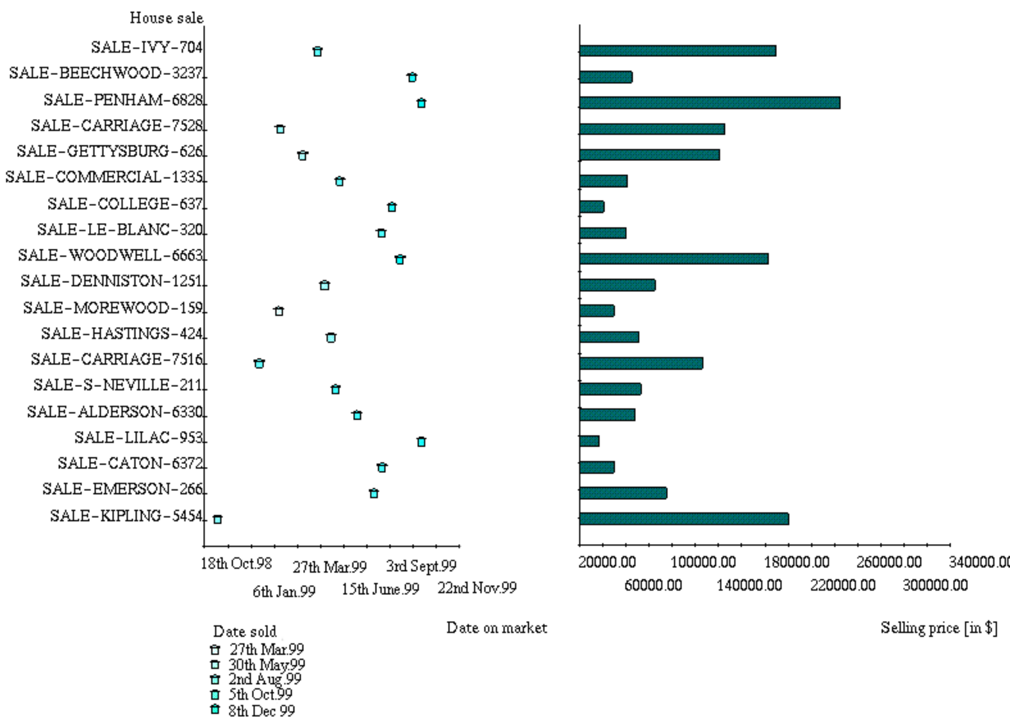
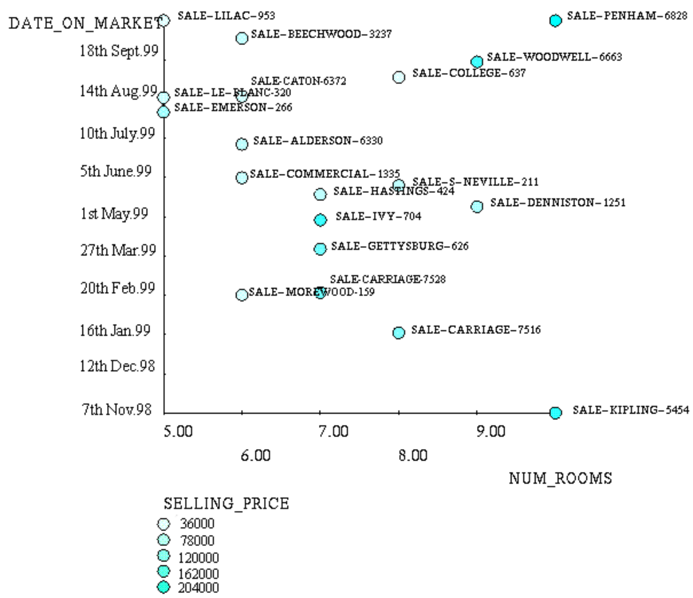
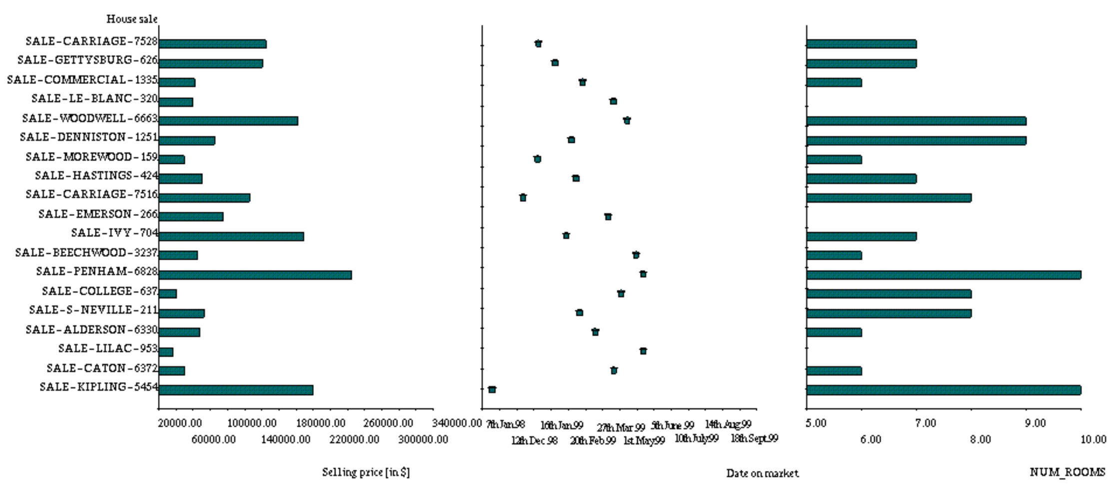


Figure V-10: Perceptually inexpressive design of the house task in Figure V-2. This is because *date_on_market* is mapped to the *x-axis* of the left chart and *date_sold* is mapped to *saturation* on the left chart. This makes it difficult to compute the duration on market because there is no perceptual operator for comparing the difference between *positional* and *saturation* values.

Soft constraints, unlike *hard constraints*, may be violated, but they incur a violation cost. Soft constraints help direct the designer to choose more effective designs (i.e. designs with smaller observational and articulatory distances) over less effective ones. For example there are *soft* integration constraints which try to direct the designer to show all task related data within the same region [*No.5, Table V-3*]. Suppose we are searching for houses based on three properties, *selling_price*, *number_of_rooms* and *date_on_market.* Figure V-11 shows mapping transform solutions to the task in which all three *find* attributes are mapped to graphical properties. In Figure V-11a all the information is displayed in the same *chart* area while in Figure V-11b the information is separated over three different regions. Both visualizations can be used to solve the house search task however Figure V-11a is more effective because it requires fewer eye movements due to the more integrated design. The soft integration constraint reflects this preference.



(a) Integrated Design: *Selling_price, number_of_rooms* and *date_on_market* are all mapped onto a single region thereby facilitating the house search task.



(b) Less Integrated Design: *Selling_price, number_of_rooms* and *date_on_market* are each mapped onto a different region thereby making the house search task less efficient compared to design (a).

Figure V-11: Mapping transform designs for house search task on *selling_price, number_of_rooms* and *date_on_market*

V-2.2.2 Constraint Dimension 2: Scope

Constraints may have *local* or *global* scope. Local constraints only affect the current design decision. Once the decision is made, any related local constraints are discarded (i.e. local constraints are not propagated from one decision point to another). For example, AVID's designer has a local constraint that ensures a task can only be graphically mapped if there are appropriate graphical representations for it. Some complex tasks such as *log* or *exponent* have no appropriate mappings and thus cannot be achieved with mapping functions. This constraint only has *local* scope because once we decide that the *compute-log* task must be performed with data computation, we need not check this condition again later in the design process. I.e. the constraint only affects the current decision.

Global constraints, on the other hand, must be propagated through the design states because they may affect multiple design decisions spanning different time periods. For example the *graphical-property* constraint applied at *node-8* is a *global constraint*, and it gets propagated to all child nodes. Even though this constraint is generated in the task processing phase (i.e. during processing of the *compute*-duration task), it affects multiple mapping decisions in the data attribute mapping phase.

Before a node is processed, all global constraints stored within it are instantiated to set the current constraint context of the search path. Once we have finished processing a node (i.e. generated all of its children) we remove its constraint context and replace it with the context of the next node.

V-2.2.3 Constraint Dimension 3: Constraint Condition

Each constraint has a test condition. Inability to pass the test condition causes a violation of the constraint. This may result in the abandonment of the current design path (in the case of a *hard* constraint) or in a cost increment for the current partial design (in the case of a *soft* constraint). Constraint conditions may be placed on various elements of the visualization design or of the input task(s). The elements that may be used in a constraint differ based on the scope of the constraint. *Local* constraints can only be applied to elements and properties that are locally accessible while *global* constraints can be applied to any element.

There are three groups of constraint conditions that commonly appear in AVID: mapping constraints, task constraints, and object membership constraints.

1. *Mapping constraints*

Mapping constraints are the most common type of constraint. These constraints restrict how data attributes may be mapped onto graphical properties and graphical objects. There are two classes of mapping constraints, *simple constraints* and *complex constraints*. Simple mapping constraints are equality constraints that restrict a single aspect of a mapping to be identical with that of another. For example in *node-8* of Figure V-7 a simple *graphical-property* mapping constraint gets applied to the *date_on_market* and *date_sold* attributes. This constraint restricts the *date_on_market* and *date_sold* mappings to have identical destination graphical properties. Simple mapping constraints may also restrict a mapping property to a named constant value. For example we may constrain the *date_on_market* attribute so that it is mapped to the *mark grapheme class* or to a specific region in the visualization, e.g. *region_324*.

AVID has seven different simple mapping constraints, the first five are object constraints while the last two are property constraints:

a) *Cluster constraint*: data attribute constrained to a given cluster of graphical objects.

b) *Graphical object constraint*: data attribute constrained to a given graphical object.

c) *Graphical object class constraint*: data attribute constrained to a graphical object that is of a given class (e.g. *mark*, *horizontal bar* or a *line* object class).

d) *Region constraint*: data attribute constrained to reside in a given graphical region.

e) *Region discipline constraint*: data attribute constrained to a particular region discipline (e.g. *chart, table, grid, map*).

f) *Segment constraint*: data attribute constrained to a given graphical property segment. A graphical segment links a set of data attributes to a graphical property and a graphical minimum and maximum for that segment. For two data attributes to be encoded in the same segment they must be of the same *data type* and they must share approximately equal data minimum and maximum ranges. Some example segments include *positional-axes* and *color, size* or *shape* legends.

g) *Graphical property constraint*: data attribute constrained to a particular graphical property (e.g. *position-x, position-y, shape, size, hue*, etc.).

Complex mapping constraints, unlike the simple constraints may restrict multiple graphical properties simultaneously and/or restrict the *relationships* between multiple graphical objects or properties. The primary complex constraints in AVID are the graphical property relationship constraints: *integral, conjoint* and *separable*. *Integral properties* refer to graphical properties that cannot be perceptually separated from one another. An example is *hue* and *saturation.* Both these properties determine the *color* of an object and it is difficult to perceptually separate out the *saturation* component and the *hue* component. *Integral constraints* are used to restrict the mapping of data attributes to graphical properties that have *integral* relationships. For example we may constrain the *house_neighborhood* and the *selling_price* data attributes to share *integral* graphical properties (such as *hue* and *saturation*) as in Figure V-12. This constraint facilitates our ability to search for houses with the combined attributes of low price and good neighborhood (e.g. *Shadyside*) because we only need to focus on one aspect of the graphical object, its *color* (e.g. *light pink*).

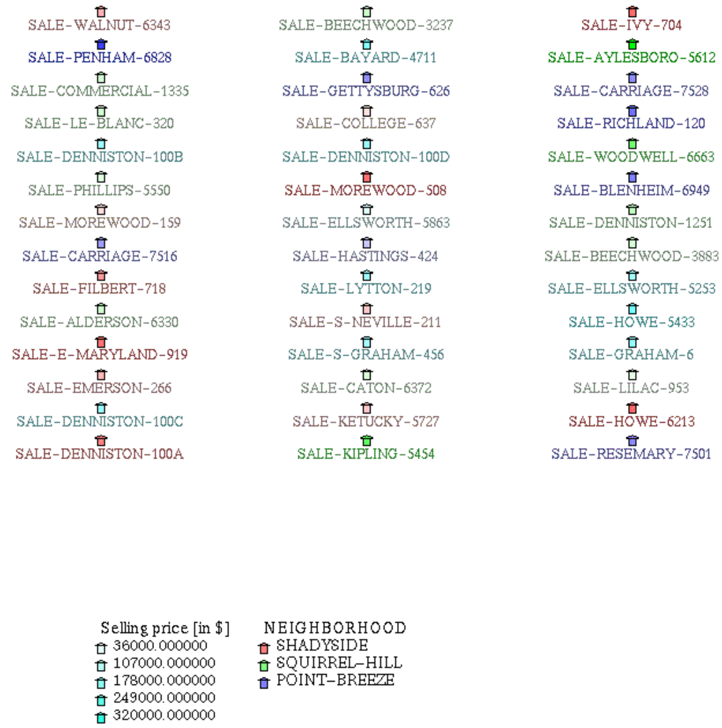


Figure V-12: Design with *neighborhood* and *selling_price* mapped to integral properties (*hue* and *saturation*). This makes combined search on both these data attributes easier because only a single emergent property (i.e. *color*) needs to be attended to.

Separable *properties* are the exact opposite of *integral* properties, in that they can only be perceived separately. An example is *size* and *hue*. Both properties do not combine to form an emergent property as in the previous case where *hue* and *saturation* combine to form *color*. As a result it is easy to view each of these properties independently. By the same token it is more difficult to perform combined property searches (e.g. find objects that are large and colored purple) compared to integral properties. Finally *conjoint properties* are both *separable* and *integral*. I.e. they can be perceived separately but at the same time they combine to form an emergent property. Examples of conjoint properties are *width* and *height*. Both properties can be perceived separately but at the same time they combine to form an emergent property namely *area*. The same is true of *x-position* and *y-position*.

Another complex constraint in AVID is the *stacking* constraint. The *stacking* constraint restricts one or more graphical objects to be positionally laid out one on top of each other. This *stacking* constraint is most commonly applied by the *add* operator to facilitate performing the summation task perceptually.

Note that mapping constraints are also special in that they get propagated from parent to child tasks. For example a simple graphical-property constraint that gets applied to the *compute-add* task shown next gets propagated to its child tasks as well so that in the pure mapping solution, all embedded data attributes (i.e. *full_compensation*, *full_salary*, *assoc_compensation*, *assoc_salary*, *asst_compensation*, *asst_salary*) are constrained to the same graphical property (e.g. in appendix E-2.6)

```
(Compute '(VALUE . ADD)
          (Compute '(VALUE . SUBTRACT)
                    (Lookup (OBJECT . NIL)  full_compensation)
                    (Lookup (OBJECT . NIL)  full_salary) )
          (Compute '(VALUE . SUBTRACT),
                    (Lookup (OBJECT . NIL)  assoc_compensation)
                    (Lookup (OBJECT . NIL)  assoc_salary) )
          (Compute '(VALUE . SUBTRACT),
                    (Lookup (OBJECT . NIL)  asst_compensation)
                    (Lookup (OBJECT . NIL)  asst_salary) )
```

2. *Task constraints*

Task constraints may include conditions set upon *task processing methods* (either data transform processing or mapping transform processing), *task classes* (*lookup*, *compute*, *compare*, *find*, AND), *task operators* (e.g. *add*, *subtract*, *multiply* or *mean*), or task input data concepts and values.

Task processing constraints are often applied to prevent bad combinations of data transform/mapping transform hybrid designs from getting generated. Step 2 of the search procedure places a task processing constraint on all embedded non-object tasks of a parent data transform task so that they are computed through data transforms as well (refer to section V-2.1.1 for details). Another interesting instance where task processing constraints are used is for imposing similarity among embedded child tasks. A good example is the *AND* task, which searches for objects fulfilling a set of data conditions. Even though it is possible to solve the task if we processed the *AND* child tasks differently (i.e. some with data transforms and some with mapping transforms as in Figure V-13) such designs are not effective.

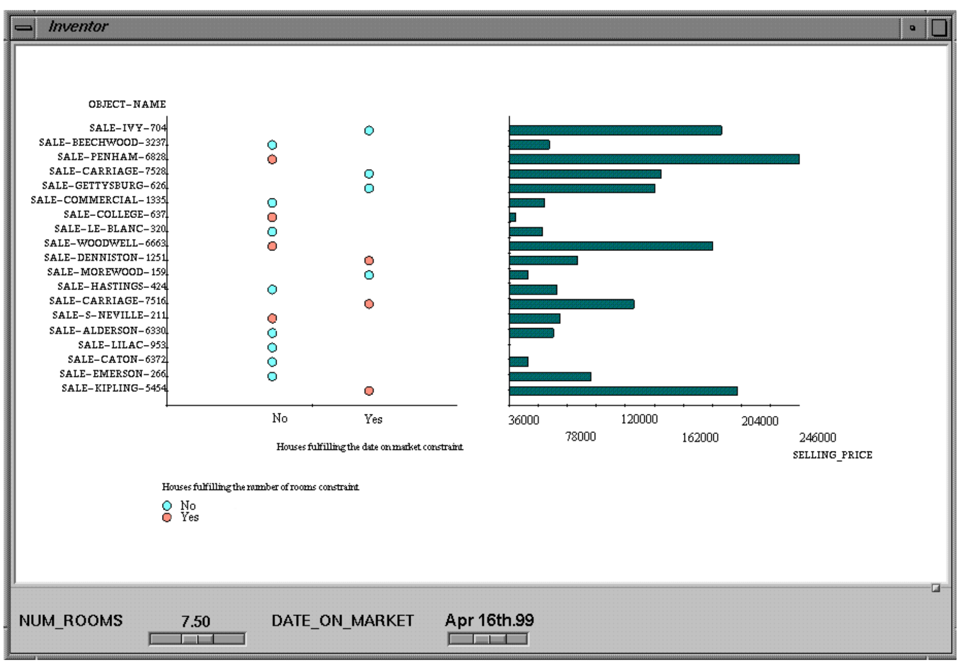
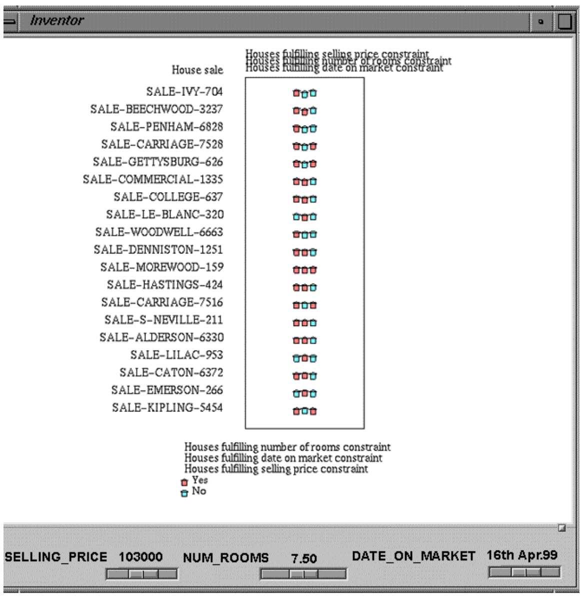
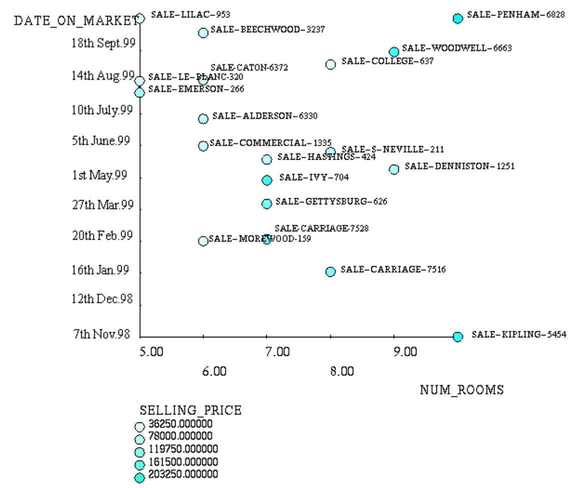


Figure V-13: Mixed task processing methods for the *AND* operator in the house search task. The *date_on_market* condition is pre-computed and mapped to the *x-axis* of the left chart, the *num_rooms* condition is pre-computed and mapped to *hue* on the left chart, however, the *selling_price* condition is performed perceptually by mapping *selling_price* to the *x-axis* on the right chart.

For example suppose we want to search for houses based on their *number_of_rooms*, *date_on_market*, and *selling_price* attributes. In Figure V-13 we must shift models from data computed conditions (the *number_of_rooms* and *date_on_market* conditions are system computed and mapped to *color* and *x-position*) to mapping conditions (*selling_price* is mapped to *y-position*). Pure designs such as the ones shown in Figure V-14 are much easier to understand and interpret. In Figure V-14a, each *mark* object corresponds to a search condition and whether that condition is fulfilled is indicated by its *color*. In Figure V-14b all of the raw data attributes are mapped to graphical properties. In both cases there is no confusion as to which condition is summarized (data computed) and which is not.



(a) Pure Data Transform Design. All three *AND* conditions are pre-computed and mapped to *hue* in each of the three clustered *marks*. The first mark encodes the *num_rooms* condition, the second the *date_on_market* condition, and the third the *selling_price* condition



(b) Pure Mapping Transform Design. All three *AND* conditions are represented perceptually. *Date_on_market* is mapped to the *y-axis*, *num_rooms* is mapped to the *x-axis*, and *selling_price* is mapped to saturation,

Figure V-14: Using similar task processing methods for the *AND* task

Apart from constraining the task processing methods used, we may also set task property constraints on the *task class* or *operator* so that addressing a task with mapping transforms fails if there are no appropriate graphical representations for the task, as is the case with the *log* and *exponent* operators. Task property constraints may also be applied to task arguments. For example, a mapping transform task constrains all embedded *lookup* tasks to be mapped to graphics unless there are unknown arguments associated with those tasks. This is because the data transform lookup function is only useful when it is connected to a subsequent processing function (i.e. when it is embedded within a data transform parent task) or when it is used to limit or interactively change the value set we are interested in (i.e. when it is attached to an unknown argument). Otherwise having a data transform lookup function within a mapping transform parent task has no effect because the act of mapping a lookup attribute to a graphical property implicitly extracts the data values needed from the data objects within the visualization (i.e. implicitly performs the data transform lookup function). In *node-8* both *lookup date_on_market* and *lookup date_sold* tasks are constrained to mapping transforms (in the *functional-constraint* list) because their parent compute task is mapped to graphics and the lookup tasks have no unknown input arguments.

3. Object constraints

Object constraints restrict the data object content of different regions within a visualization. Object constraints commonly get generated to encode the results of a data computed object task (e.g. *find* or *AND* task). For example, *node-2* and *node-9* perform the *find* houses task with data transforms and show the results of the *find* to users by placing an object constraint on the region containing the *compute-duration* or *date_on_market* and *date_sold* attributes so that those regions can only contain houses that cost less than 100k. It is important to note that currently AVID does not allow a region to have inconsistent object contents. For example a region cannot contain houses that cost more than 100k as well as houses that cost less than 100k. In the future, we plan to reduce the granularity of this restriction so that it only applies to graphical object sets and not to entire regions. Object constraints commonly cause designs to be less integrated, but they have the advantage of reducing visual clutter or visual density because fewer objects are shown within each region, as can be seen in *design 1* and *design 4*.

V-2.2.4 Summary

In this section we defined task processing constraints (constraints generated during the task processing phase) based on three dimensions: *softness*, *scope* and *constraint-condition.* Table V-4 summarizes the constraints applied by each data transform task class and Table V-5 summarizes the constraints applied by each mapping transform task class.

Task (Data Transform case)	Constraint Condition	Soft/ Hard	Scope	Description
All	*Task processing constraint*: Non-object child tasks MUST be constrained to data transforms.	hard	global	This is to avoid the high articulatory costs associated with value entries.
Lookup	*Task argument constraint*: Task MUST have unknowns or *Task processing constraint*: Task MUST be embedded within a parent data transform task.	hard	local	If task does not have unknowns or is not an intermediate operation then a data transform lookup is unnecessary.
Compute, Compare, Find, AND	None			

Table V-4: Data transform constraints for each task class

Task (Mapping transform case)	Constraint Type		Soft/ Hard	Scope	Description
All	*Mapping constraint*: Simple-region or Simple-visualization		soft	global	Preference towards integrating all data attributes related to task within the same region or visualization window. This reduces number of eye-movements required to solve the task (**locality of processing [***No 5, Table V-3***])**.
Lookup	*Task argument constraint*: Task MUST NOT have unknowns		hard	local	If task has unknowns, then mapped lookup fails because the *GetAttributeValue* function must be used to adapt the lookup results according to changes made to the task inputs. (**manipulable medium [***No 15, Table V-3***])**.
Compute	All	*Task operator constraint*: Compute operator MUST be simple	hard	local	If compute task operator does not have perceptual parallel then mapped compute fails. (**offload to perceptual system [***No 3, Table V-3***])**
	All (except ratio)	*Mapping constraint*: Simple-graphical-property	hard	global	Ensures that data attributes are mapped to the same graphical property to facilitate perceptual computation. (**offload to perceptual system [***No 3, Table V-3***])**
	Add	*Mapping constraint*: Complex-stack	soft	global	Preference for *add-compute* attributes to be mapped to stacked objects.
	Subtract	*Mapping constraint*: NOT Complex-stack	hard	global	Ensures that data attributes for *subtract-compute* are NOT mapped to stacked objects.
	Ratio	*Mapping constraint*: Simple-graphical-object	hard	global	Ensures that *ratio-compute* attributes are mapped to the same graphical object.
		Mapping constraint: Complex-conjoint	hard	global	Ratio values must be deducible from emergent conjoint property, e.g. from combined *x* and *y position*.
Compare	*Mapping constraint*: Simple-graphical-property		hard	global	Ensures that data attributes are mapped to the same graphical property to facilitate perceptual comparison.
Find	*Mapping constraint*: Simple-graphical-property		hard	global	Ensures that data attributes are mapped to the same graphical property to facilitate perceptual comparison.
AND	*Task processing constraint*: Task processing equivalence		hard	global	Ensures that all child tasks uses the SAME task processing methods (e.g. either all data transforms or all mapping transforms).

Table V-5: Mapping design constraints for each task class

Apart from the constraints shown in Table V-4 and Table V-5, that get assigned based on the *task processing method* used and the *task-class*, AVID also contains a small set of design-wide constraints that get enforced in all visualization designs. Two important instances where design wide constraints are applied include:

1. Complex-type relationships

Mapping constraints may get imposed as a result of complex-type relationships within the data [*Roth, 1990*]. For example to express an *interval* complex type relationship between *date_on_market* and *date_sold* both data attributes must be mapped to graphical properties and graphical objects that can reflect this *interval* relationship (as in *design 4* and *design 5*). The designer in AVID has knowledge about the complex relationships that are expressible by different classes of graphical objects. Figure V-15, for example, shows the description for the *interval-bar graphical object class* within AVID's designer. Included within this description is information on the types of complex relationships that it can express. At the start of the data attribute mapping phase, the designer determines if there are any complex-type relationships in the *data-attribute-list*. If so, the designer tries to find a graphical object and graphical properties that are capable of expressing these relationships. Complex-type constraints are declared as soft constraints.

```
(make-instance
     'grapheme-class
     'name "interval-bar-grapheme"

     ;; interval bars are capable of expressing the interval complex type with its position-x1 and position-x2
     ;; graphical properties
     'complex-types
         (list (make-instance 'complex-type
                              'type 'interval
                              'required t
                              'parameter-list (position-x1 position-x2)))

     ;; x-position can only be mapped to a quantitative data attribute
     'position-x1
       (make-instance  'grapheme-class-parameter
                        'element-type 'QUANTITATIVE)
     'member-parameters'(position-x1
                          position-x2
                          position-y1
                          hue
                          saturation)
     )
```

Figure V-15: Example graphical object class specification for interval bar grapheme

2. *"Objectness" constraint* (**spatially indexed addressing [***No 7, Table V-3*])

Figure V-16 shows an example visualization that is not expressive of the house task. In this visualization the *price* and *duration* information are separated into different spaces and it is difficult to identify which *selling_price* corresponds to which *duration* data. Our designer has a hard *"objectness" constraint* that restricts all task-related information to an object cluster or constrains each task attribute to be clustered with an *object-identifying* attribute (e.g. *object-name, house-address*). For example in *design 6* the data attributes are all tied together through the common house *object-name* attribute. It is also important to notice that object identification is more easily achieved when the identifier attribute is mapped to a *positional* property (as in *design 6*), rather than to a *label* because in the latter case we need to match *text* labels and this is en expensive perceptual operation.

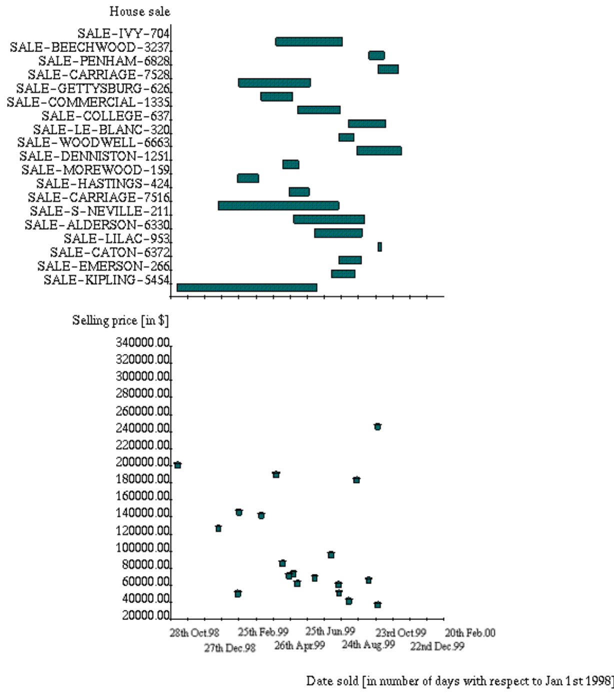


Figure V-16: Visualization design with no "objectness" constraint. I.e. it is not possible to associate which house *mark* in the bottom chart corresponds to which house *bar* in the top chart.

In summary this section describes the *task processing* and *design wide* constraints that are applied by AVID's designer. AVID also has a set of *data attribute mapping* constraints that are adapted from previous work [*Mackinlay, 1986a, 1986b; Casner, 1991; Roth, 1994*] and thus we do not describe them here. Based on the input task and data set, AVID establishes a network of constraint conditions that limit the output designs that are generated so that end users need not go through bad or inexpressive designs and can focus on a wider range of valid design alternatives.

V-2.3 Design Costs

The designer in AVID is faced with the problem of a large design space and having to present design solutions from that space to users in a timely and effective manner. In order to direct the search and generate design solutions according on their effectiveness with respect to the task we assign costs to different design decisions based on their effects to cognitive, perceptual, and articulatory complexity. The cost of each node within the search tree is the sum of its current cost and its expected future costs. The current cost of a node is the accumulated costs of the partial design at that node. The estimated cost for a node is based on the number of tasks that still needs to be processed and the number of data attributes that still needs to be mapped. Based on this cost structure AVID uses the A* search algorithm to explore the design search space.

AVID's cost structure is based on the design metrics we presented in chapter IV-1. The design metrics framework has four different goodness measures: *articulatory distance*, *functional distance*, *expressive distance*, and *observational distance*. As we stated in the previous section, *functional* and *expressive* distances are primarily represented in AVID through the use of hard constraints. I.e. designs that have *functional* or *expressiveness* discrepancies with respect to the input task(s) are culled out. Thus our cost structure is primarily based on *articulatory* and *observational* distances.

V-2.3.1 Articulatory Cost Structure

Articulatory costs of a visualization interface come from the frequency and complexity of input device manipulations that need to be performed by the user. Our AVID system adds input devices to data transform tasks in two cases:

- when there are unknown arguments in the task(s), or
- when results from a mapping transform task need to be conveyed to a data transform parent task.

Input devices are added in these cases because data transform functions summarize the task data and only presents a subset of the original data values to the user. When task conditions change, the data values of interest may change as well. In order for the system to reflect these changes, users must convey the new task conditions to the system through input devices. When a task is performed with mapping transforms, however, input devices are not needed because all of the original values are displayed to the user. When task conditions change, a user perceptually filters out segments of the display that are not pertinent and only processes the relevant elements. Changing task conditions will cause a change in user focus towards different display elements, but requires no argument input.

According to the metrics framework, articulatory costs depend on the following conditions:

1. The complexity and appropriateness of each input device with respect to the task,
2. The number of input devices within the design, and
3. The number of times we expect each input device to be used.

In the following sections we describe how AVID takes each of these conditions into account in its design component.

1. *Complexity and appropriateness of input device with respect to task*

AVID attaches input devices to visualization designs so that users may change the input data values or objects to tasks. An appropriate input device is determined by analyzing properties of the data values or objects that must be conveyed and then matching that with a device that can best provide those inputs. Specifically, input devices are evaluated based on the following properties:

a) *Continuous/Non-continuous outputs*: Whether the input device is capable of expressing continuous data or graphical values. Sliders, dials, *bounding-boxes*, *lassos*, and *text windows* can be used to define continuous values while menus, and buttons are inherently discrete. Therefore, if we wanted to pick a reasonable *selling_price* value for the task in Figure V-2, we must use a continuous device such as a slider because *selling_price* is a continuous data attribute.

b) *Visual representation*: Whether the input device requires visual representations of particular data objects or data attributes for it to operate. For example, devices such as the *bounding-box*, *lassos*, and *mouse clicks* are applied to visual representations of data within the visualization design. Before we can use such devices we must ensure that visual representations of the data concepts or values we want to select are available. On the other hand, devices such as the *option-button*, *scroll-list*, or *text window*, are not tied to objects within the visualization design. For example, we may select a house by entering its *address* through a text window irrespective of whether that house object is mapped to graphics. However, we can only select a house using a *bounding-box* if that house is represented and visible in the visualization.

c) *Number of objects*: The number of values, objects, or relationships that can be specified with each invocation of the device. *Bounding-boxes*, *lassos,* and *double ended sliders* for example can specify multiple objects or values. On the other hand, *text windows*, *option buttons,* and *dials* can only be used to define one value or object at any one time. Depending on the task we may only need to pick one value or one object per operation or we may need to specify sets of values or objects. For example, to enumerate all houses costing more than 100k we would need to define a set of objects thus a set input device such as a *bounding-box* would be appropriate here. On the other hand, to pick a threshold price (e.g. 100k) for filtering houses we only need to specify a single value to the system thus a *dial* or a *slider* would be sufficient here.

d) *Spatial/Non-Spatial*: Whether the input device is a spatially based device (i.e. the device defines a spatial region within the visualization window). *Bounding-boxes* and *lassos* are examples of spatial devices because they define a graphical region within the rendered visualization space. This property is important when we need to apply rendering transforms that take graphical regions as input.

Table V-6 shows all the input devices we consider in AVID as well as their status for the evaluation properties listed above. To pick an input device for a task argument, we consider the requirements of that task argument based on the properties above and match that to the input devices within AVID. We then choose a device that has all the required properties of the task argument.

	Continuity	Vis. Rep.	#-of- values	Spatial
One-ended slider	T	*	Singular	F
Two-ended slider	T	*	Plural	F
Mouse click	F	T	Singular	F
Bounding-box	T	T	Plural	T
Option Menu	F	*	Singular	F
Scroll List	F	*	*	F

Table V-6: Input devices considered in AVID with their cost properties
(* indicate no constraints on an input device property)

For example, consider the house example presented earlier in this chapter. In this example, we were interested in determining the period that less expensive houses stay on the market. Suppose due to changing economic conditions, we are no longer sure what constitutes a good “expensive” house threshold. We would then alter the task specification so that instead of setting the price threshold at 100k we use an "unknown" argument ("?"). The new task specification is as follows:

```
(setf set1 (Find '(RELATIONSHIP . <),
                 Lookup '(OBJECT . NIL), house_price),
                 '(VALUE . ?)))

(Compute '(VALUE . SUBTRACT),
         Lookup( set1, date_on_market),
         Lookup( set1, date_sold) )        :loop-type one-to-one
                                           :accuracy nil)
```

One way to fulfill this task specification is to perform the *find* task with data transforms and link the "unknown" threshold value for the *find* task to an input device. The input device properties required in this case is based on the *selling_price* data attribute, and is shown below.

	Continuity	Vis. Rep.	#-of- values	Spatial
Selling_price threshold properties	T	F	*Singular	*F

The two important constraints in this case are that the *selling_price* attribute is continuous and that the *selling_price* data attribute is not mapped to a graphical representation. The other requirements are soft requirements (indicated with an "*") because an input device that does not fulfill these conditions can still generate the types of values required for this task. AVID infers this information from the data characterization of the *selling_price* attribute and the task specification. Based on these requirements we pick the one-ended slider as the best match because it fulfills all the input device constraints including the soft requirements. Note that the two-ended slider also fulfills the two hard constraints (*continuity* and *vis.rep*), but it does not fulfill the *#-of-values* soft constraint and is therefore only a second choice.

Currently we pick the input device that fulfills all hard requirements and the greatest number of soft requirements as the best match. In AVID, we only use this "best match" input device and do not consider alternative designs that differ only in terms of the input devices used. This is because the focus of our work is not so much on choosing between multiple input device alternatives but rather on the choice of visualization functions, and how they can be used effectively in the visualization design process. By limiting our designs to only the best input device match, we limit our design search tree and increase responsiveness of our system.

2. Number of input devices within the design

Once an appropriate input device is chosen based on the selection process described above, a constant input device cost is added to the current design path. As a result, nodes or partial designs with a greater number of input devices will have a higher cost. This is because the more devices there are in a visualization interface, the greater the cognitive load placed upon users for understanding how to operate those devices. In addition, articulatory load is also increased because the *task specificity* is low and a greater amount of information needs to be conveyed to the system each time we want to test out a different set of task conditions. The situation worsens when there are constraints or relationships among the different input devices, i.e. changes in one input device cause state changes to occur in other input devices. Currently, however, we do not deal with such input device relationships.

3. *Number of times we expect each input device to be used.*

The number of times input devices are used within a data analysis session depends on the number of times a user wants to vary the current task conditions. This in turn could be affected by the task operator, the data associated with the task, the importance of the task, user preferences, user experience, and the difficulty of using each input device. Dealing with many of these issues is beyond the scope of this thesis. In AVID we only focus on the extreme repetition cases. Specifically we identify tasks or task properties that will likely result in high input device use and for those instances we either abandon the design path or add a commensurate cost to the design alternative. The data transform non-object constraint described in *step 2* of the search procedure in section V-2.1.1 is an example of high articulatory cost resulting from highly repetitive input requirements.

In this section we described how input devices are chosen by AVID's design component and how costs are associated with different input device decisions. Our input device selection strategy and cost structure is simple, but sufficient to capture the design differences and illustrate the design issues we are interested in pursuing in this thesis, such as dealing with unknown task arguments, linking input devices to visualization function primitives, and capturing how articulatory costs of input devices can affect the choice of using *data transform* vs. *mapping transform* task processing strategies. A simplified input device selection strategy allows us to:

- Limit the design search procedure and simplify implementation of the system
- Focus on developing heuristics for functional operators like the ones described in chapters II and III rather than on heuristics for choosing input devices.
- Not duplicate previous work that already deals with expressiveness and effectiveness of input devices [*Card, 1990*].

V-2.3.2 Observational Cost Structure

Observational costs reflect the ease with which users can interpret a visualization interface. There are two classes of observational costs in AVID, corresponding to the two main phases of design: task processing observational costs and data attribute mapping observational costs.

V-2.3.2.1 Task processing observational cost

Task processing observational costs are accrued when tasks are mapped to graphics to account for the perceptual load placed upon users compared to data transform processing where the load is transferred to the computer system. Observational costs in this case are based on the task class, the task operator, the task input arguments, and the task properties.

1. Task class

Certain task classes are more difficult to perform perceptually than others. For example, for the task shown in Figure V-2, it is more difficult to *compute* the difference between *date_on_market* and *date_sold* than it is to *find* the houses costing more than 100k (assuming that we are using the best possible graphical properties for both tasks). Thus when we process these tasks with mapping transforms a higher cost is associated with the *compute* task compared to the *find* task. The task classes ordered according to decreasing perceptual difficulty are as follows: *Compute, Compare, AND, Find, Lookup*.

2. Task operator

The costs associated with a mapping transform task class may also vary based on the specific task operator used. This condition pertains mainly to the *compute* task that has a wide range of operators (e.g. *add*, *subtract*, *multiply, divide, mean, log, exponent*). Based on the *operator availability* heuristic in chapter IV-3.3, a higher cost is placed on compute operators that have less appropriate perceptual parallels (e.g. *mean* and *divide*) while a smaller cost is placed on those that have very effective perceptual parallels (such as *add* and *subtract*). Compute operators that have no perceptual parallels (such as *log* and *exponent*) have a hard constraint that only permits data computation.

3. Task input data

The perceptual load of a task may also depend on the task input arguments. For example *compute*, *compare* and *find* tasks are easier to perform with respect to constant input values (e.g. *find* all houses with *selling_price* greater than 100k) than with multiple data attribute value sets (e.g. *find* all universities with *avg_math_SAT* scores greater than *avg_verbal_SAT* scores). We illustrated this with the example visualizations in appendix C-4. Similarly we also showed that *enumerated* input data attributes may simplify perceptual processing of a task. This is because *enumerated* values can be represented both pre-attentively and accurately, unlike other attribute types. Thus lower task processing costs are assigned to mapping transform tasks when they have constant input arguments or enumerated input attributes.

4. Task conditions

As we described earlier in this chapter there are two task conditions: task accuracy level and task loop type.

- According to the accuracy heuristic in chapter IV-3.1, a higher cost is associated to mapping transform tasks compared to data transform tasks when *accuracy* is required. When *fuzzy* accuracy is explicitly called for, the data transform solution becomes invalid.
- According to the loop type heuristic in chapter IV-3.4, *all-to-all* tasks incur a heavy cost when we perform them with data transforms because more objects need to be shown (n^2 objects where n is the number of objects in the input data set) compared to the mapping transform alternative which only requires $2n$ objects. The cost added in this case is based on the number of data objects in the task input set.

V-2.3.2.2 Data attribute mapping observational cost

There are three classes of data attribute mapping observational costs: mapping costs, composition costs, and perceptual complexity costs.

1. Mapping cost

The mapping cost structure used in AVID is similar to the cost structures used by Mackinlay and Casner. Graphical properties are assigned costs based on how effective they are at showing different data attribute types. Preference is given to graphical properties that allow parallel perceptual processing [*No. 2, Table V-3*]. Figure V-17 shows how different graphical property classes are ordered (from most effective to least effective) based on their data attribute class (i.e. *data type*). For details on data characterization, refer to previous work by Roth [*Roth, 1990*] and Mackinlay [*Mackinlay, 1986a, 1986b*].

Data attribute class properties		**Accuracy**	**No-Accuracy**
Nominal	Enumerable	*1.* differential retinal, *2.* position, *3.* label	*1.* differential retinal, *2.* position, *3.* label
	Unenumerable	*1.* position, *2.* label	*1.* position, *2.* label
Ordinal	Enumerable	*1.* extent retinal, position, *2.* label	*1.* extent retinal, position, *2.* label
	Unenumerable	*1.* position, *2.* label, *3.* extent retinal	*1.* position, *2.* extent retinal, *3.* label
Quantitative		*1.* label, *2.* position, *3.* extent retinal	*1.* position, *2.* extent retinal, *3.* label

Figure V-17: Mapping costs ordered based on data attribute class and graphical property class

The most favored graphical property is *position* because it allows pre-attentive (parallel) perceptual processing as well as affords a relatively high degree of accuracy compared to retinal properties (such as *saturation* or *size*) which are pre-attentive but less accurate or *labels*, which are not pre-attentive. The only exceptions are in the nominal-enumerable and quantitative-accurate categories. Enumerable attributes commonly consist of only a few different values, and retinal attributes such as *hue* or *shape* can represent such attributes accurately, and pre-attentively, while requiring less display space compared to a *positional*. In the quantitative-accuracy category, *labels* are preferred because it can express the data more accurately than positionals, especially when the data range represented is large.

In Figure V-17 there are two types of retinal properties, *extent retinal* describes retinal properties capable of expressing ordered values (*saturation* and *size*) and *differential retinal* describe retinal properties that can only express unordered values (*hue,* and *shape*) [Card et al.]. Of the *extent retinal* properties we prefer *saturation* over *size* because *size* may result in occlusion problems or in expanding the display space required (less information presence). Of the *differential retinal* properties, *hue* is preferred because it is easier to pre-attentively process *hue* compared to *shape*. Note that the retinal property class is not present in the nominal-unenumerable category because data attributes of this type have too many values that have to be differentiated and it is difficult for users to associate these many values with an unordered retinal such as *hue* or *shape*.

2. Composition cost

The composition costs in AVID are assigned based on spatial locality [*No. 5, Table V-3*]. Higher costs are assigned to less integrated designs such as Figure V-11b and lower costs are assigned to more integrated designs such as Figure V-11a. Integrated designs are preferred because they require less eye-movement and visual search by the user. In addition less display space is required for the visualization, allowing more data to be shown at any one time. This is especially important for larger data sets. Based on this graphical integration rule, composition costs are ordered as follows from least cost to highest cost: use of existing graphical object, *cluster* composition, *double axis* composition, *single axis* composition, no composition (i.e. use of separate visualization window). Details on each of these composition types were discussed previously in section V-2.1.2.

3. Perceptual complexity cost

Finally costs are also added for each new set of graphical objects used. A higher cost is applied if we add new objects to a region that already contains many objects. Adding graphical objects into a visual design increases its complexity, requiring a steeper initial processing cost to learn the design structure. In addition added graphical objects may distract the user and cause perceptual interference, making it more difficult to find task related objects or identify interesting perceptual patterns.

In this work we experimented with a cost structure that seems to order the designs in a meaningful way for the classes of tasks we are interested in. We illustrated this in the GOMS analysis described in appendix E. This cost structure is just one instance of all possible cost assignments; in the future we hope to determine the costs statistically, as in a neural net. We suspect, however, that a single cost structure may not be applicable across different problem spaces and domains. The solution may lie in identifying different classes of cost structures that perform well with particular domains and tasks or letting users manipulate different cost classes manually. It is important to stress that the contribution of this work lies not in the exact cost structure provided in the expert designer but rather in identifying important aspects of the task data, task structure, and visualization design that we should attend to while assigning costs and in developing heuristics that provide general guidelines for determining which function operators and interactive devices to use, when to use them, where to use them, and what constitutes an effective combination of data transform, mapping transform and input device primitives.

V-2.4 Summary

The designer in AVID is driven by a search procedure consisting of two primary phases: the task processing phase and the data attribute mapping phase. During these two phases, there are multiple branch points that create alternate design paths in the search space. In order to direct the designer towards promising design alternatives and away from bad designs, AVID has a constraint and cost structure.

Currently AVID uses an A* branch and bound method to explore the design search space. We have also experimented with other search methods such as DFS, HILL, and BEAM. We found that although these methods can generate designs more quickly, their design quality is significantly inferior to those generated using the branch and bound method. Depending on the complexity of the task, AVID's designer may take from minutes up to several hours to come up with a design. Linear increases in the complexity of the tasks cause an exponential increase in the search space and the generation time. Optimizing the search space is ancillary to testing the theoretical concepts in this thesis, however, so we have decided not to focus on that particular aspect of the designer.

V-3 Visualization Realizer Component

The last component of the AVID system is the visualization interface realizer. The realizer accepts a visualization design specification from the automatic design component, and based on this design, instantiates or generates an active visualization interface. Each design specification has two parts:

a) *Visual structure design specification*: This specification captures the general look or structure of the graphical components within the visualization. It contains the number of graphical objects of each type (e.g. *grapheme, region, axes, legend,* etc), their object classes (e.g. *mark, bars,* and *lines* for *grapheme* objects or *charts, tables, grids* and *maps* for *region* objects), the distribution of *grapheme* objects across various *regions*, containment relationships among the graphical objects, etc.
b) *Functional design specification*: This specification describes which data, graphical, mapping, and rendering transforms are used, how these various transforms are composed, which objects they are applied to, which input devices they are linked to, etc.

The AVID realizer is divided into two components based on the two specification types described above: the graphical object realizer and the functional realizer. The graphical object realizer accepts visual structure design specifications and converts them into graphical element renderings. The functional realizer accepts functional design specifications and converts them into visualization techniques (e.g. dynamic query sliders, painting). Input device events and virtual input devices (e.g. *scroll-lists*, *option-buttons*, *sliders*) may be attached to the visualization techniques as necessary. Every visualization window is divided into two sections as is shown in many of the visualizations in this chapter (Figure V-13, Figure V-14). The top portion of the window contains graphical renderings of the data, which is generated by the *graphical object realizer*, and the bottom portion of the window contains all input devices that are generated by the *functional realizer*. Details on our AVID realizer are given in appendix D-2.

V-4 Conclusion

To showcase the applicability of our visualization framework and heuristics in "real" systems, we implemented an automatic design system (AVID) consisting of the three components: the *task interpreter, the automatic designer*, and the *visualization realizer*. The task interpreter accepts input tasks in LISP form and transforms them into a set of task and argument structures. Our task interpreter is able to deal with task embedding structure and special task conditions such as accuracy and task iteration, which was not taken into account in previous automatic visualization research. Task structures generated by the task interpreter are used by the automatic design component to guide its search of the visualization design space. In AVID we have expanded the visualization design search algorithm over previous systems to include a task processing phase for addressing data transform and input device decisions as well as a post processing design phase for culling out duplicate designs. The task processing phase deals with new issues that are unique to creating data transform designs such as processing embedded task structure, object filtering, addressing unknown task arguments, and pre-processing the input data set. In this search algorithm the task information is translated into a set of design constraints and costs that stop the designer from going down unpromising paths and direct the designer towards more effective design paths first. The translation of task information into design constraints and costs is based on the guidelines and metrics laid out in chapter IV of this thesis. Once the automatic design component finds an interesting unique solution to the input tasks, it transforms the design into a visual structure specification and a functional specification. These specifications are taken as input by the realizer component that translates the visual structure specification into a hierarchical scene graph of Inventor nodes and the functional specification into a set of acyclic functional networks. The Inventor scene graph is rendered onto the display using functions from the Inventor toolkit, and the functional networks are activated beginning with their source functions. During activation the C++ procedures associated with each visualization function in the network are executed on the input values of the functions. Functional networks may also get reactivated based on trigger events from input devices associated with the visualization design.

AVID and our interactive functional editor[3] are used to generate most of the visualization designs shown in this document. All visualizations not generated by our systems are annotated with their original sources. The wide range of example visualization designs generated by our systems shows the flexibility and generality of our theoretical framework and heuristics. Our ability to translate the theory into active systems indicates that our theory is relatively complete. In addition, the previous evaluation chapter shows that AVID produces practical design results that do indeed conform to cognitive, perceptual and articulatory complexity. The functional editor is also practical for manually creating and prototyping visualization techniques because it takes less time compared to using low-level code. Thus the implemented systems described in this chapter are good illustrations of the *generality, completeness* and *practicality* of our visualization framework and heuristics presented in previous chapters.

---

[3] Our interactive functional editor allows us to manually build visualization techniques by creating the node-link specification diagrams shown in chapters II and III. Details on this editor are given in appendix D-3.

With AVID we have expanded the design space of automatic visualization design systems. Previously, only mapping transformations were considered in the design process, but AVID is able to reason about many data manipulation and summarization operators as well as composite hybrid designs that use both mapping and data transformations to solve tasks. While we have expanded the functional design space from previous work, we can augment AVID further. An important area that AVID currently does not address is in integrating *graphical* and *rendering* transformation decisions into the design process. We will show in appendix F that *graphical* and *rendering* transformations can be very useful for solving *readability* issues that may arise in visualization designs and how our current system can be augmented with these graphical and rendering transforms. *Readability* refers to problems arising from constraints of the output medium and its interactions with our perceptual system that impede the optimal use of a visual design. Examples of *readability* impediments include occlusion among objects, display space that is too limited to show all the necessary design objects, or overly high ink density, producing visual interference.