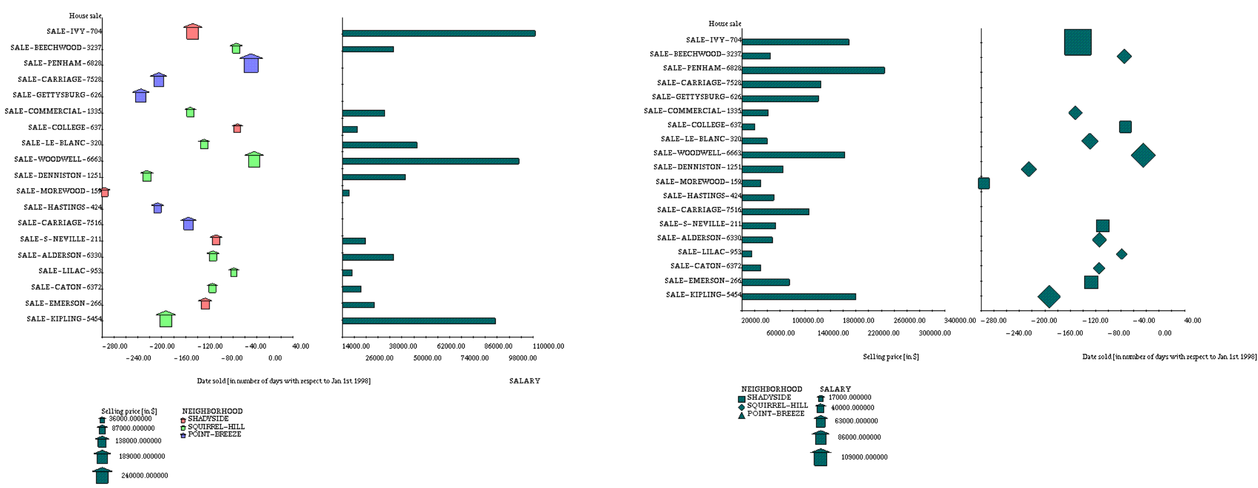
Appendix D

Appendix to Implementation (Chapter V)

D-1 Structural & Content Matching

Structural matching is based on the graphical properties used, the number of graphical objects, and the number of graphical regions in the design. For a structural match to occur, the number of graphical objects and the distribution of those objects across the different regions must be identical for the two visualization designs. For example Figure D-1a and Figure D-1b match structurally because both visualizations have two *chart* regions and a set of *mark* graphical objects in one region and *horizontal-bar* graphical objects in the other. Note that the order of the two regions is irrelevant in the structural match.



(a) **(b)**

Figure D-1: An example pair of visualizations that match based on both structure and content. Structurally both designs have two charts, one of which is a bar chart and the other a scatterplot. In terms of content, both designs contain the same data attributes (*object-name, selling_price, neighborhood, owner_salary*) and graphical property classes (*2 positionals,* and *2 retinals*).

Content matching, on the other hand, is based on the data contained within a visualization and how that data is mapped to graphical properties. Two designs match in content if the *data attributes* and *graphical property classes* contained within them are identical. There are three graphical property classes in AVID (based on Bertin's [*Bertin, 1983*] graphical property categorization): *positional*, *retinal*, and *labels*. *Positional* properties include *x-position, y-position*, and *z-position*, *retinal* properties include *size, hue, saturation, shape, thickness*, etc., and *label* properties include the use of *text labelling*.

Figure D-1a and Figure D-1b are also content matches because both visualizations have identical data content (*object-name, selling_price, neighborhood, owner_salary*) and identical graphical property class content. Figure D-1a uses *y-position, size, hue* and *x-position* respectively to encode its data attributes which translates to a graphical property class content of [*positional, retinal, retinal, positional*]. Figure D-1b uses *y-position, x-position, shape* and *size* to encode its data attributes which translates to a graphical property class content of [*positional, positional, retinal, retinal*]. Note that the graphical property class content of Figure D-1a and Figure D-1b match even though the actual graphical properties used may be different. Figure D-1b for example uses *shape* to encode *neighborhood* while Figure D-1a uses *hue* to encode the same data attribute. A match still occurs because both *hue* and *shape* belong to the *retinal* property class. Also note that the order in which data attributes are mapped to graphical properties does not affect the content match result. For example in Figure D-1a, a *retinal* property (*size*) is used to encode *selling_price* and a *positional* property (*x-position*) is used to encode *owner_salary*. On the other hand in Figure D-1b a *positional* property (*x-position*) is used to encode *selling_price* and a *retinal* property (*size*) is used to encode *owner_salary*. Even though the mappings are permuted in these two cases the graphical property classes used and the data attribute content within the two visualization designs are identical, thus Figure D-1a and Figure D-1b are considered content matches.

D-2 Translating a Functional Design from AVID's Design Component into a Complete Specification

The translation algorithm for AVID's functional design follows the *instantiation augmentation process* for interactive techniques described in chapter III. Initially we collect all functional operators in the *functional-operator-list* of a node and connect them based on the embedding structure of their related tasks. For example consider the design in *node-3*, which is a pure data transform design with five data transform operators corresponding to each of the input tasks. These operators are connected from innermost task to outermost task. Thus the *GetAttributeValue selling_price* operator for the innermost *lookup* task is connected to the *Threshold* operator of its parent *find* task. The *Threshold* operator is in turn connected to the *GetAttributeValue* operators for the *lookup-date_on_market* and *lookup-date_sold* tasks. Finally we end the functional design with the *BinaryCompute* operator related to the outermost *compute* task. This initial design structure is shown in Figure D-2. All functional operators are represented as rectangular boxes.

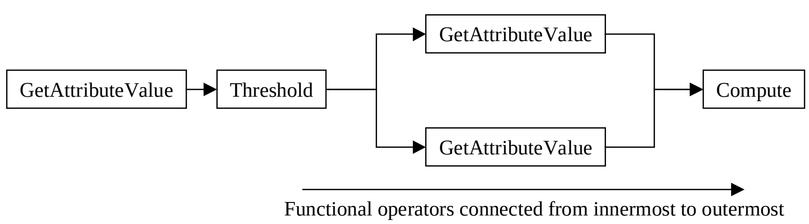


Figure D-2: Connecting all visualization functions within a node state from innermost task to outermost

Once we have constructed the general functional structure, we perform *task class* related modifications, as is shown in Figure D-3. Newly added function rectangles are highlighted to indicate changes made to the design specification at the current step. Primarily, a *find* task can either show its results through object filtering or through mapping its results to a graphical property. Depending on which of these alternatives is relevant to the current design, we either filter objects within the appropriate region with an *Add_object* and a *Delete_object* operator (*alternative 1* in Figure D-3) or we store the *Threshold* results within the visually mapped *find* attribute with an *Assign* operator (*alternative 2* in Figure D-3)[1]. In Figure D-3, an *Assign* operator is also connected to the *BinaryCompute* operator. This *Assign* operator stores processed time duration values within the *compute* task's related *duration_on_market* attribute that is subsequently mapped to graphics. Note that both *find* task design alternatives are shown in Figure D-3 for pedagogical purposes. Any single design, however, would only contain structures for **one** of these alternatives

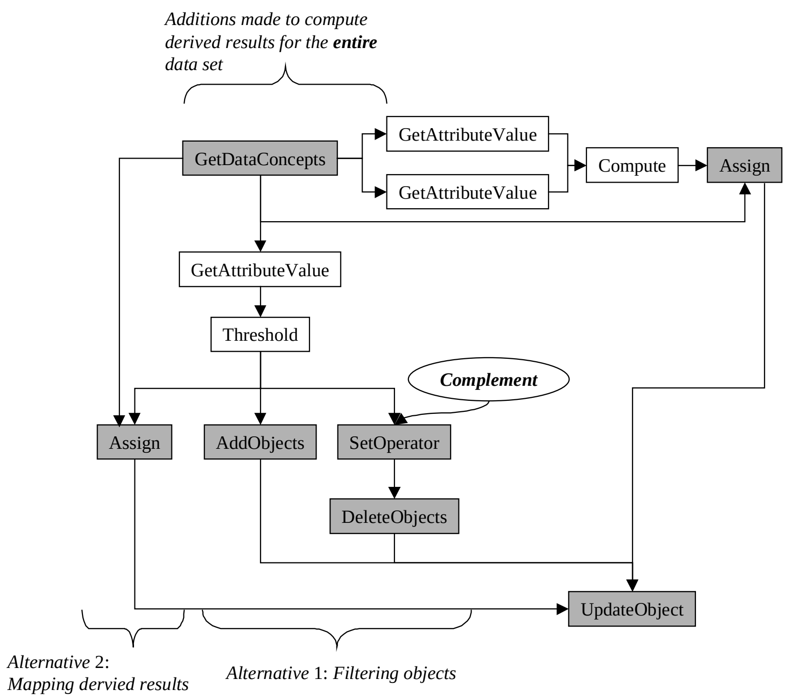


Figure D-3: Making task class modifications depending on whether we want to show the *find* task results either through object filtering or by mapping its results to a graphical property

Depending on how the *find* task results are shown, it may be necessary to show the computed time duration values only for the objects that fulfill the *find* task (*design-1*) or we may have to show the values for all house

[1] In general all visualization functions that provide values to mapped data attributes must be linked to an *Assign* operator so that newly computed values may be associated with their corresponding data attribute structure.

concepts (*design-2*). For design simplicity, we always calculate derived attributes for the entire set of related data concepts. To achieve this we disconnect all embedded object task operators from their parent task and connect them to a *GetDataConcepts* function instead. In Figure D-2, for example, the *Threshold* function results (for the *find* task) are piped into two date *GetAttributeValue* operators that limit the duration *compute* to only the threshold objects. To generate duration values for the entire house data set we disconnect the out-links from the *threshold* operator, and associate the two date *GetAttributeValue* operators to a *GetDataConcepts* operator. The *GetDataConcepts* operator accepts a set of data classes and extracts all data concepts belonging to those classes from the entire house data set. In this example derived values are calculated for all data concepts belonging to the house class[2].

Finally we must update the visualization design so that any newly computed results or change in values are reflected in the graphics. This is achieved by adding an *UpdateObject* operator at the end of the functional specification. We connect to the *UpdateObject* operator last to ensure that all computations and assignments are made before the object update occurs. The procedure thus far corresponds to step 1 of the *instantiation augmentation process* described in chapter III. In this step we define the object selection and transformation functions in the design[3].

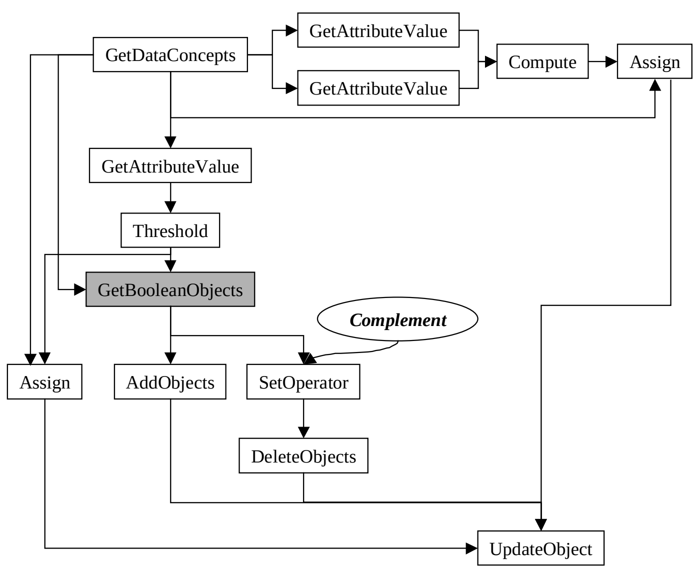


Figure D-4: Adding in any necessary translation functions

[2] Note that the *GetDataConcepts* operator also produces the input data concepts for the two *Assign* operators that are used to store computed functional results within the *duration_on_market* attribute.

[3] In Figure D-2 and Figure D-3 we include *GetAttributeValue* functions that are technically translation functions but since they are related to the *lookup* tasks we found it useful to add them to the design at this initial step.

In step 2 of the *instantiation augmentation process* we add in translation functions to ensure that the inputs to a visualization function match the output of its source function. In this case only one translation function is added, namely the *GetBooleanValues* function for converting boolean values from the *Threshold* function into an object set so that it can subsequently be processed by the *Add_object* and *Delete_object* mapping functions (Figure D-4).

Next, we determine the input arguments to the visualization operators based on the tasks and task arguments that they are associated with. This corresponds to steps 3 and 4 of the *instantiation augmentation process*. These input values are represented in Figure D-5 with oval boxes and ***bold italicized*** text. Most of these values can be directly extracted from the task arguments (e.g. *selling_price, 100k, date_on_market,* and *date_sold*). The two derived attributes *duration_on_market* and *houses_with_selling_price_less_than_100k* can have their names specified in the task description or automatically generated based on related task operators and arguments. The region name(s) for the *Add-object* and *Delete-object* functions is derived from object constraint information stored within the node state during the search procedure. In general all regions containing attributes from an object task (e.g. *find* or *AND*) are constrained by the results of that object task and thus become inputs to its corresponding *Add-object* and *Delete-object* functions. In *design-1*, for example, the *interval-bar region* contains the *duration_on_market* derived attribute, which is a parent attribute to the *find* task. Thus the *interval-bar region* gets assigned as the input region to the *Add-object* and *Delete-object* functions in Figure D-5. Finally the entire visualization design (*design-1*) is updated using the *UpdateObject* function.

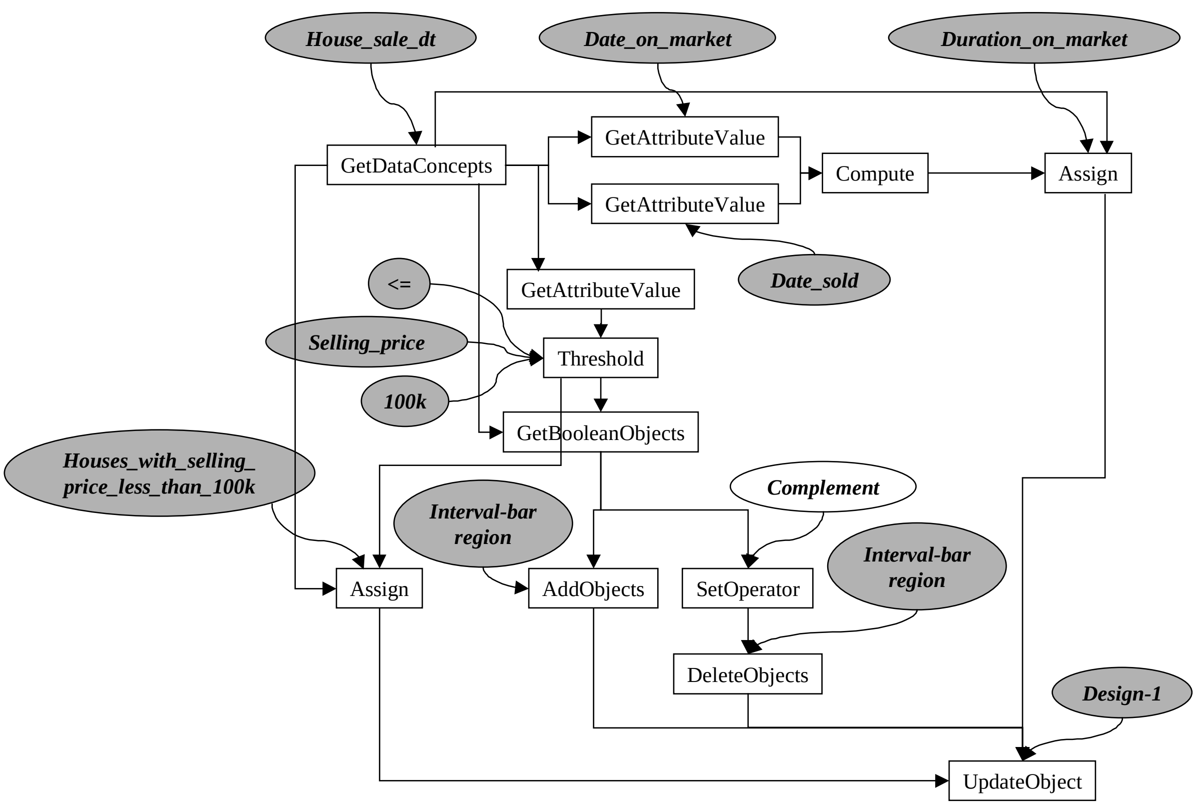


Figure D-5: Adding in input arguments to the visualization functions

In step 5 of the *instantiation augmentation process* we add in all the input devices stored within the node state and initialize them. The house purchasing task in chapter V, Figure V-2, does not require the use of any input devices, thus all designs (*design-1,2,3,4,5,6*) generated have none. For illustration purposes however, suppose that the *find* threshold value is tied to an input device instead of a pre-specified constant of *100k*. In this case, a *slider* input device would be associated with the *find* task and stored within the *input-device-list* of the node state. We therefore add the device to our functional specification and augment it with any initialization functions necessary.

For each input device class, AVID contains knowledge on the set of device attributes that must be initialized (this is based on the input device description in chapter IV). For a *slider*, we need to initialize three attributes: its *label name*, as well as its *min* and *max* values. Figure D-6 shows the initialization specification for the *slider* input device. To the right are the three device attributes that must be initialized. To the left we extract relevant values from the task argument(s) related to the input device. In this example the related task argument is associated with the *selling_price* attribute and the house object class. As shown in Figure D-6 (highlighted ovals), these values are piped into the input device initialization specification. The rest of the functions and inputs within the initialization structure are stored as design knowledge within AVID. Similar to the *slider*, other devices have initialization structures associated with them as well, including entry point rules for connecting task argument information.

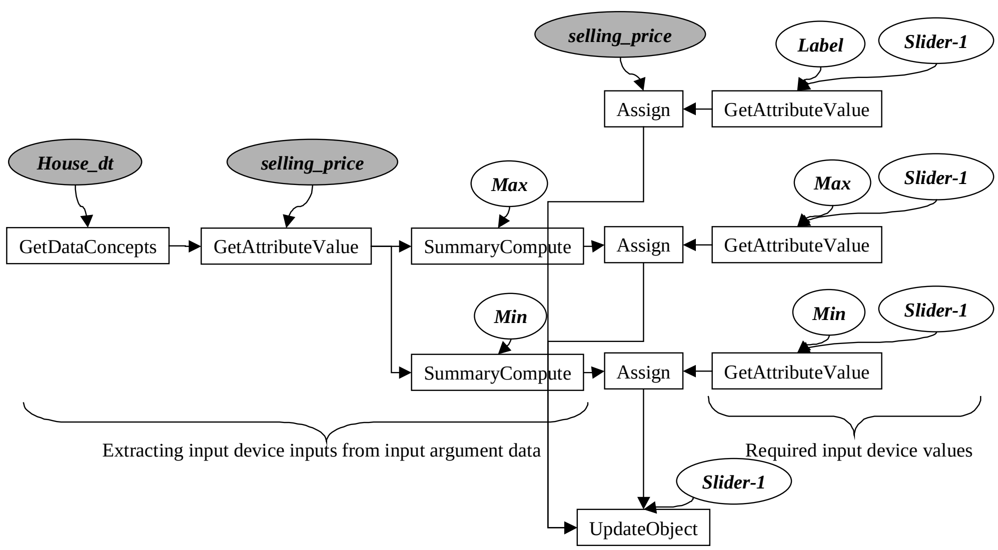


Figure D-6: Initialization functions for the slider input device

D-3 Visualization Realizer Component

AVID's realizer is divided into two components: 1) the graphical object realizer and 2) 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).

D-3.1 Graphical Object Realizer

The graphical object realizer is implemented in C++, using SGI's Inventor toolkit, which provides a framework for organizing and rendering the graphical objects within the visual structure specification. Initially, C++ functions are in place to interpret the input design and convert it into one or more Inventor nodes. These Inventor nodes can then be manipulated or rendered onto the display using a set of Inventor functions.

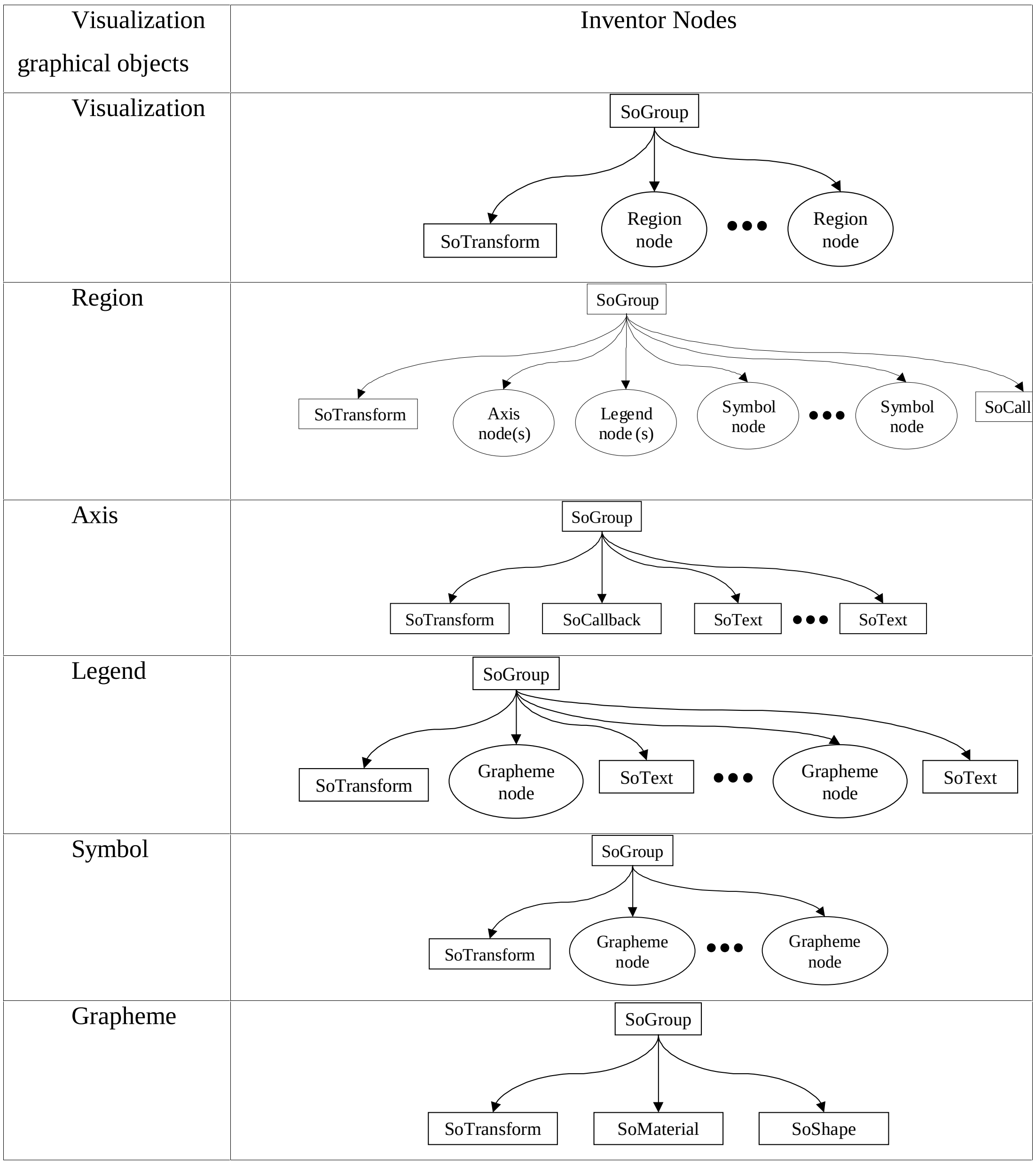


Figure D-7: Visualization graphical objects and their corresponding Inventor nodes

Figure D-7 shows all the graphical object classes in our AVID system and their corresponding Inventor nodes or sub-trees. Details on these object classes can be found in chapter III-1.1.2. Primitive Inventor nodes are represented in Figure D-7 as square boxes and their names are preceded with *"So"*. Oval boxes represent AVID graphical object classes.

All graphical objects in the visual structure specification are converted into at least a *SoGroup* and a *SoTransform* Inventor node. The *SoGroup* node is used to collect all other nodes associated with the graphical object together under a single root node. This makes it easier to organize the objects within a visualization as well as to access and manipulate their appearance. Every *SoGroup* node has a *SoTransform* node as its first child. This *SoTransform* node allows the realizer to translate, rotate, or scale graphical objects. A change in the transformation parameters of a *SoTransform* node not only changes the appearance of the current graphical object but also the appearance for all its children. For example, increasing the scale of a *SoTransform* node for a *region* object increases the region bounds as well as the size of the *axes*, *legend*, *symbol*, and *grapheme* objects within it. Changing the scale of a *SoTransform node* for a *symbol* object will affect the size of the *grapheme* objects it contains but not have any effect on the *region*, or *visualization* objects which contain the changed *symbol*.

In addition to the *SoGroup* and *SoTransform* nodes, graphical objects may also contain other graphical objects. In fact the graphical object classes within AVID have a hierarchical relationship with *visualization* objects at the top of the hierarchy followed by *region* objects, *symbol* objects and finally *grapheme* objects. Depending on the region class (i.e. *chart, map, table* or *grid*) and the *grapheme* properties used, regions may also contain positional *axis* objects and *legend* objects that capture how *grapheme* property values can be converted back into data values. *Grapheme* objects form the leaves of the hierarchy, and as such they only contain primitive Inventor nodes, unlike the other graphical objects. *Grapheme* objects have three Inventor nodes: a *SoTransform* node that determines the position, orientation and size of the object, a *SoMaterial* node that determines the color of the objects, and a *SoShape* node that determines the appearance of the *grapheme* object. The appearance of a *grapheme* object is based on its class (*mark, bar, text, interval-bar* or *line*) and its *shape* graphical property.

Certain graphical objects also contain Inventor *SoCallback* nodes. *SoCallback* nodes are free-form nodes that allow specialized drawing functions to be executed. In the *region* object, for example, this node is used to call functions for texturing maps, drawing table columns, and other region rendering operations. Similarly the *SoCallback* node for the *axis* object is used to draw the axis line, and the axis tick marks.

When we connect the Inventor nodes for the different graphical objects together, we get a hierarchical *scene graph* of Inventor nodes, as in Figure D-8. This scene graph is rooted at the *SoGroup* node of a *visualization* object. Every *visualization* object in a design gets converted into a separate hierarchy of objects. Once the realizer converts all objects within the visual structure specification into a *scene graph*, Inventor functions are available to render the scene onto a display screen and ensure that all object transforms are applied in proper order.

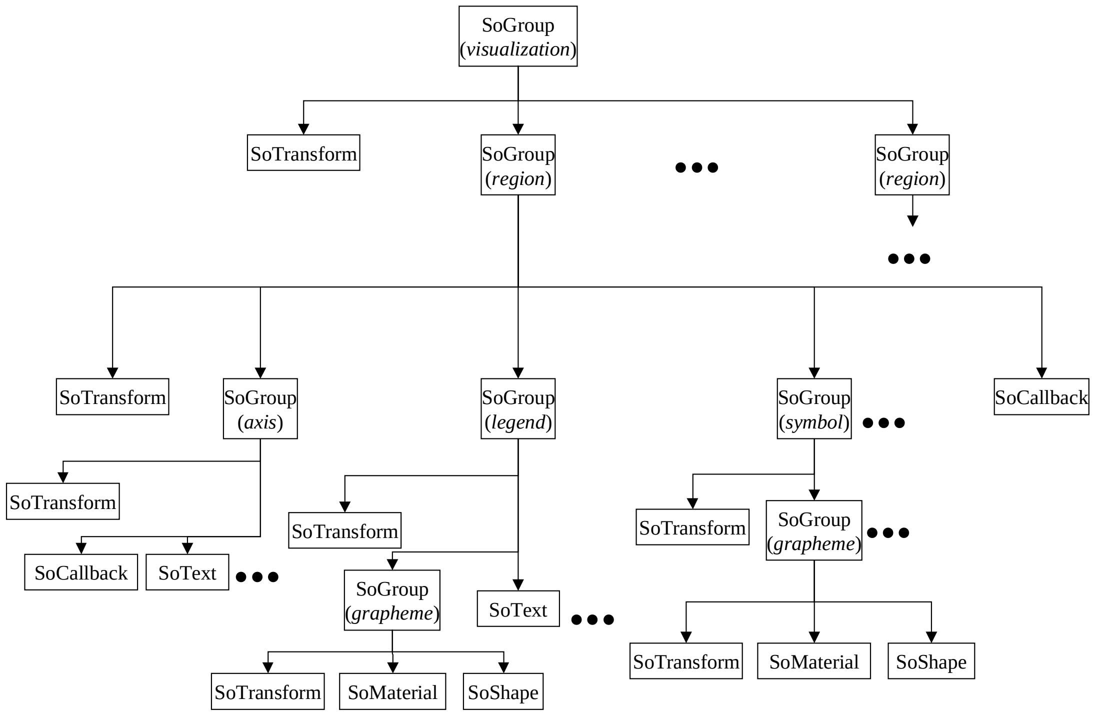
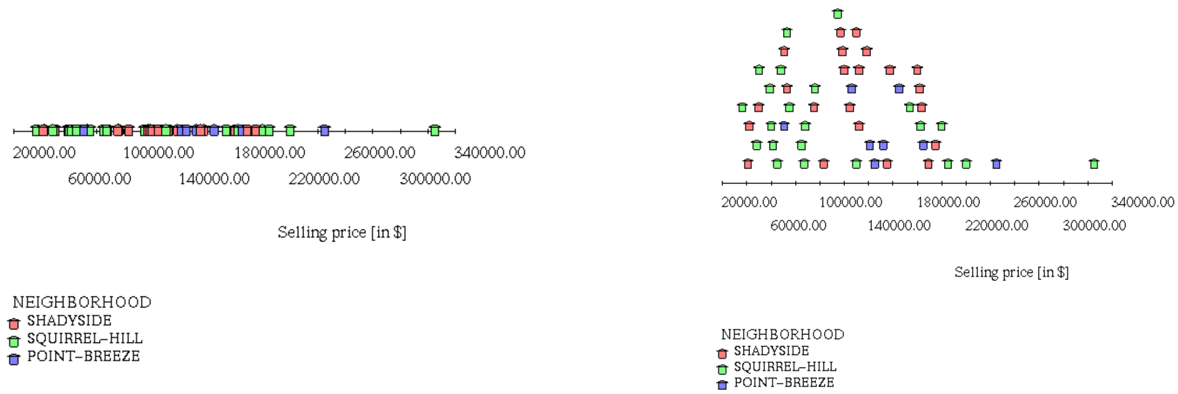


Figure D-8: Inventor scene graph of visual structure design

Another important function of the graphical object realizer is in making layout decisions. Note that the visual structure specification provided to the realizer does not contain details on exact object positions. Instead, the specification contains mapping information such as which data attribute should be mapped to which graphical objects and properties, object information such as what objects should be in the visualization, and roughly how objects should be laid out in relation to one another, as well as containment relationships. It is up to the realizer to set the actual pixel or color ranges used for mapping, the actual object sizes and positions in pixels as well as default object property values. To avoid certain occlusion problems among the graphical objects and to reduce perceptual clutter, the realizer has a set of algorithms for arranging graphical objects. For example, white-space is added between *region* objects, and *regions* are never placed on top of one another. *Grapheme* objects may also be subject to special layout algorithms. This occurs in *single-axis charts* or when a chart maps non-unique, non-continuous data attributes to positional axes.

D-3.1.1 Single axis layout

For single axis charts, occlusion may be significant because the objects are only distributed across a single positional dimension. For example Figure D-9a shows a single axis visualization with *selling_price* mapped onto the *x-axis*. Because of the great concentration of objects around particular ranges of values there is significant occlusion in those areas which in turn reduces readability of the visual display. The AVID realizer reduces occlusion in this case by utilizing the other, unmapped positional dimension (i.e. the *y-positional* axis) to spread out the high object concentration areas. This is achieved by adding an offset value to the unmapped positional dimension of all occluding objects. We repeat this positional offset step until there is no longer any occlusion. This pushes occluded objects up as far as necessary, as is shown in Figure D-9b which has the same data and design as Figure D-9a, but with the *single axis* layout algorithm applied.

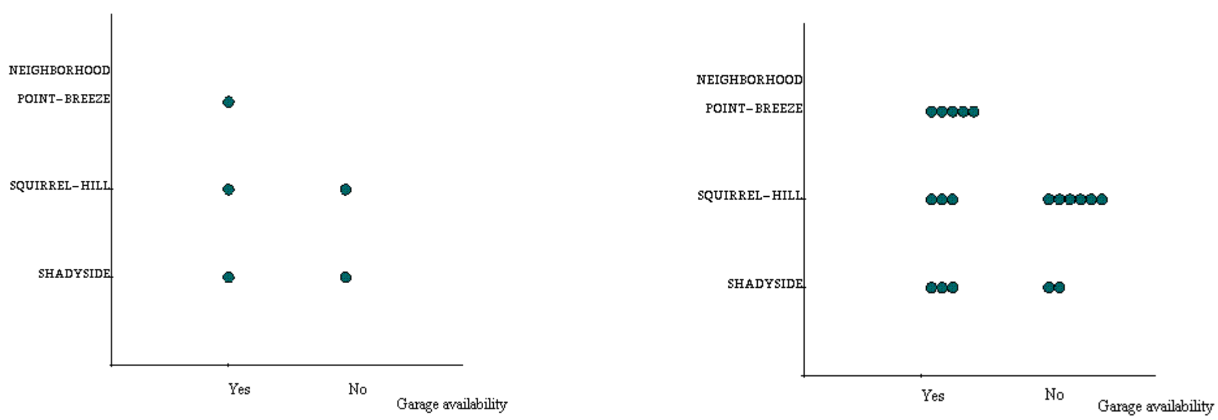


(a) without layout **(b) with layout**

Figure D-9: Single axis visualization design with and without realizer layout algorithm

D-3.1.2 Non-unique positionals

Significant occlusion may also occur among *grapheme objects* when a chart only contains non-unique, non-continuous positionals. For example, in Figure D-10a there is significant occlusion because two non-unique data attributes, *neighborhood* and *garage_availability* are mapped to the *y-axis* and *x-axis* respectively. As a result all houses that fall within the same *neighborhood-garage_availability* category will have exactly coincidental positions. In such cases the graphical object realizer offsets the positions of objects within each category so that they do not occlude one another as in Figure D-10b which contains the same data and design as Figure D-10a but with significantly less occlusion.



(a) without layout **(b) with layout**

Figure D-10: Non-unique positional visualization design with and without realizer layout algorithm

D-3.2 Functional Realizer

The functional realizer takes a functional design from the automatic designer as input and produces active visualization techniques for manipulating data or graphical objects. AVID's functional realizer is implemented in C++ and Motif. Initially C++ functions are used to transform the objects within the functional specification into networks of data transform functional nodes. Each network is a *directed acyclic graph*, and may look like the specifications in Figure D-5 or Figure D-6. The Motif toolkit is used to integrate virtual input devices and handle events from physical devices such as keystrokes, mouse operations, etc. These Motif events serve as alerts and may cause one or more functional networks to be executed. When a functional network is executed, we traverse down the acyclic graph and execute the transform functions accordingly.

The functional network structures in the specification are based on the framework described in chapters II and III of this thesis. There are four types of objects in this network:

D-3.2.1 Primitive visualization functions

A *visualization function* (VF) structure describes the primitive building blocks of a visualization technique. Figure D-11 shows an example visualization function structure in our functional language.

```
(DEFSCHEMA visualization-function-741
    (class BINARY_COMPUTE)
    (defaults
      (0 0 0 0 0 VALUE IV_STRING SUBTRACT)
      (0 0 0 0 1 VALUE IV_STRING ONE-TO-ONE)
       )
     (to con-743    )
     (composite cvi-748)
     )
```

Figure D-11: An example description of a visualization function

The *class* field captures the primary processing operations of the visualization function. As was described in chapter II, visualization functions may belong to the object selection class (*object-definition* or *enumeration*) or the transformation class (*data, mapping, graphical,* or *rendering* transforms). Tables III-1, III-2, III-3, and III-4 in chapter III summarize all the visualization function classes defined in our AVID realizer. Each function class definition contains the number and type of input arguments required, the number and type of output arguments generated, and a functional description (in C++ code) of how the input arguments are processed. When a visualization function is activated, its input arguments are processed using the functional description code defined within its function class.

A system designer may provide input arguments to a visualization function as default values. For example the visualization function in Figure D-11 has two default input value strings, *subtract* and *one-to-one*. Input arguments may also be generated by other visualization functions. This is achieved by attaching both the source and destination visualization functions together using a *connector* object. For example, in Figure D-11, *visualization-function-741* is connected to the connector object, *con-743*. Figure D-12 shows that this connector object routes the output of *visualization-function-741* to *visualization-function-742*.

D-3.2.2 Connectors

A *connector* routes output arguments from one visualization function as input arguments into another. Figure D-12 shows an example connector object. A *connector* object contains a source and a destination visualization function in its *visualization-function* slot. This information alone however, is insufficient to fully specify the connection. A visualization function commonly has multiple input and output arguments, thus to fully specify a connection, we must not only declare the source and destination functions, but also the specific output and input argument positions. This information is stored in the *connections* field. In Figure D-12, *con-743* links the first output argument of *visualization-function-741* to the second input position of *visualization-function-742*. The first number in the *connections* field represents the number of connections there are in the field. The subsequent numbers in the field are pairs of source argument and destination argument positions.

```
(DEFSCHEMA con-743
    (instance bvi-connection)
    (visualization-function  visualization-function-741
                             visualization-function-742)
    (connections 1 1 2 )
    )
```

Figure D-12: An example description of a connector

When the *connections* field of a connector object is left unspecified, the functional realizer will try to infer which argument(s) of the source function best matches any unspecified input arguments in the destination function based on the type and properties of those arguments.

D-3.2.3 Input devices

An example input device structure is shown in Figure D-13. The input device *class* describes the type of input device to use, which could be a physical device, e.g. *mouse, light-pen* or a virtual device *option-menu, scroll list* or *slider*. In addition to the input device class, we must also declare the *trigger visualization functions*. The *trigger visualization functions* are the functions that get executed whenever certain *trigger events* are sensed by the device. Trigger events commonly differ based on the input device class. For example the trigger event for a *scroll list* is a double click on a menu choice, the trigger event for an *option button* is a mouse release on one of the button choices and the trigger event for a *bounding-box* is a mouse release within the visualization window. The types of input devices available on our AVID system and their corresponding trigger events are shown in Table III-5, chapter III. Finally, the input device structure also contains information on which visualization window the input device should be attached to. In Figure D-13, the input device *id-500* is attached to the visualization *design-338*.

```
(DEFSCHEMA id-500
    (instance inputDevice)
    (class BOUNDING_BOX)
    (trigger bvi-500)
    (operation-within design-338)
    (composite cvi-501)
    )
```

Figure D-13: An example description of an input device

For each input device class, AVID's realizer contains information on its trigger event, its appearance, its properties as well as a functional description of how the device's properties and state change with respect to user inputs.

D-3.2.4 Composite visualization functions

As is shown in Figure D-5 and Figure D-6, a set of visualization functions and input device objects may be combined together through a set of connector objects to form a functional network (which is a directed acyclic graph). Each functional network is called a *composite visualization function*. A visualization technique may consist of a single composite or multiple interacting composites. In the example in section D-2 there are two composite functions, one for providing the inputs to the slider input device and the other for describing how the visualization graphical objects are altered when the user enters different values through the slider. Figure D-14 shows an example composite visualization function (*composite-501*). This composite object contains all the source or origin visualization functions within the functional network. Source visualization functions are operators that only contain pre-defined input arguments, i.e. operators that do not accept inputs from other visualization functions. When a composite visualization function is activated, processing begins with these source functions.

```
(DEFSCHEMA composite-501
     (instance composite-function)
     (source-nodes visualization-function-734
                   visualization-function-726
                   visualization-function-751)
     )
```

Figure D-14: An example description of a composite visualization function

Based on the structural descriptions of these four object types within the input functional specification, the functional realizer builds one or more networks of connected *visualization functions* and *input device objects*.

Initially, all functional networks are activated once beginning with their source visualization functions. In addition to this initial activation, functional networks may also get reactivated as a result of trigger events from input devices. When a *visualization function* is activated its input arguments are transformed based on its function class. The newly generated outputs are piped into subsequent visualization functions that in turn get activated. Note that a visualization function can only be activated when all of its input functions (i.e. the functions that generate the inputs to the current function) have also been activated. This ensures that all of the input data are up to date before processing begins. If an error in the design specification results in un-updated or missing input arguments, then processing of the error visualization function, as well as its parent composite function is halted.

D-3.3 Summary & Scope

This section describes the realizer component of AVID, which is divided into two parts, the graphical object realizer and the functional realizer. The graphical object realizer deals with rendering the graphical components within a visualization design while the functional realizer deals with constructing and executing the selection and transform visualization functions associated with a design.

The selection and transformation functions captured within our functional realizer are merely a subset of the wide range of possible useful data analysis operations. In fact as we discussed in earlier chapters, it is never possible to guarantee complete coverage of the functional design space nor the visual design space. However, the designer and realizer components within AVID are modular in their implementation, and it is not difficult to add in new visualization function classes, input device classes, graphical object classes and region types.

D-4 Interactive Functions Editor

In the previous section we described AVID, our automatic visualization designer that can generate a set of ordered visualization designs (consisting of both visual and functional components) based on input task specifications. AVID makes design decisions about what graphical object and visualization function primitives to use and how they should be combined. AVID’s designer component, however, does not utilize the entire visualization techniques design space that is captured by our framework in chapters II and III. In AVID’s designer component we mainly focussed on heuristics for integrating data manipulation, data summarization, and mapping techniques. We chose to leave graphical and rendering heuristics for future work, as this area is both rich and complex, making it difficult for us to give a complete or reasonable treatment of it in this dissertation. In appendix F we discuss graphical and rendering techniques in greater detail and outline how they may be integrated into the automatic design process in future work.

To more widely test the flexibility of our functional framework and its usability as a prototyping and customization language we implemented an editor for manually constructing visualization techniques. This editor allows users to build functional networks using the primitives described in chapters II and III. It is built upon two graph packages developed at AT&T-Bell Labs: *GraphViz* and *JavaApp* [*Ellson*]. *GraphViz* is used to improve the readability of a graph by organizing it so that there is little or no occlusion among its nodes and links. *Java App* allows visualization techniques to be constructed by adding visualization function nodes (indicated by rectangles), default value nodes (indicated by ovals), and connector objects; changing visualization function properties (using property sheets); deleting or moving existing nodes and other graph editing operations. We made some modifications to the *Java App* interface to better suit our purposes, such as allowing group selection, group moves, group deletes, as well as automatic resizing of the graph canvas depending on the current graph size. Figure D-15 shows a screen-shot of our functional specification editor. The top window allows us to load and save functional specification files, remove or add *cvi-components* to the current design, or layout chosen function networks. The bottom window shows the network diagram for a particular *cvi-component* (*cvi-4*) and a property sheet of one of the *visualization functions* within it. Once a functional specification is completed, it is transformed into the functional specification language that we described in section D-3.2. This specification may be saved as a file and/or piped into AVID's realizer component that instantiates the designs and combines them within a visualization interface.

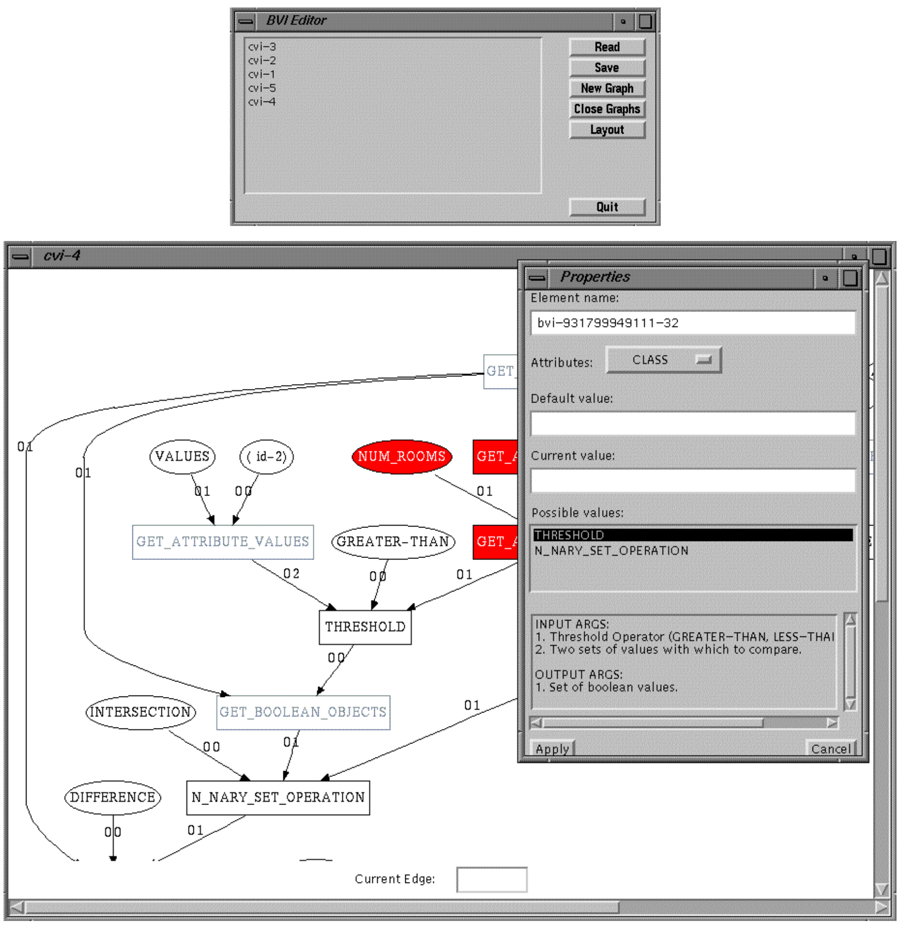


Figure D-15: Functional specification editor

By using this editor we generated all the example visualization techniques shown in chapters II and III including *painting* [*Becker, 1987*], *dynamic queries* [*Ahlberg, 1992*], *aggregation, Visage drag-and-drop* [*Roth, 1996*], *semantic zooming* [*Bederson, 1994*], etc. These examples illustrate the wide range of visualization techniques that may be captured by our framework and created using our functional editor. In chapter III-2, we used this interface to customize the *dynamic query slider* interactive technique and change its behaviors and effects. Each change is simple and time effective, showing how our visualization functions framework may simplify the customization, and prototyping of interactive visualization interfaces. By building these examples with our editor we show that our visualization functions framework is not merely a theoretical description of visualization techniques, but is complete and specific enough to be implemented and realized as active systems for creating and prototyping visualization behaviors and metaphors. Depending on the complexity of the design and the amount of structure sharing that is possible with previous techniques, creating each of these examples may take from 20 minutes up to several hours. In either case, the time taken is still insignificant compared to writing code from scratch. The editor interface can be further improved by considering the specification by example methods suggested by Myers [*Myers, 1991*].