Using Escalante to build Visual Language Applications

Jeffrey D. McWhirter   Zulah K. F. Eckert
Gary J. Nutt

CU-CS-655-93   October 1993

University of Colorado at Boulder

Technical Report CU-CS-655-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Using Escalante to build Visual Language Applications

Jeffrey D. McWhirter   Zulah K. F. Eckert
Gary J. Nutt

October 1993

Abstract

Constructing visual language applications is a difficult task. The Escalante system facilitates the process of application construction by supporting the high level specification of a visual language and the generation of code that realizes the language within a working application. Using Escalante one can rapidly develop highly functional applications for a wide variety of visual languages with a minimal amount of manual coding. Escalante is written in C++ and runs under X Windows. This paper presents an overview of the Escalante system and a detailed set of examples that can guide the development of visual language applications using Escalante.

Contents

1 Introduction

The Escalante[1] system provides facilities to rapidly construct applications for visual languages that are based on graph models. Applications are developed by specifying a target visual language using Escalante's visual specification environment. From this specification language-specific software is generated and combined with preexisting Escalante software to realize the target application. The resulting application incorporates a broad spectrum of facilities for handling the specified language including the underlying language data structures and a comprehensive editing module. The application can also be expanded by manually embellishing the generated software to incorporate arbitrary functionality.

Visual languages within the domain of Escalante are characterized as *graph models*; however, this characterization is related more to the Escalante design principles than to one's intuitive nature of the language-based applications that can be implemented using Escalante. In general, Escalante is applicable to visual languages which are based on some notion of "things" (elements) and "connections between things" (relations). This approach allows one to cast a spectrum of languages as graph models, ranging from traditional graph models such as directed graphs to computer games.

Escalante is an evolving system, yet it has already been used to construct a wide variety of visual applications, some of which have essentially no manual embellishments and others of which have substantial manual additions. This technical report is intended to explain the principles that underly Escalante, to describe how the system implements the principles, and to illustrate how Escalante can be used to implement a variety of aspects pertinent to visual applications.

1.1 Basic System Architecture

Escalante is an object-oriented system composed of three components: a base language module, a base editor module, and the *GrandView* language specification editor. Figure 1 shows the development process and a conceptual view of the target application architecture. Applications built using Escalante are composed of a language (or data) module and an editor (or control) module. The *language module* encapsulates most of the language specific functionality required within an application, including the application data model and its representation. The *editor module* consists of a built-in editor model that offers a rich set of interaction mechanisms and can be adapted by the language designer to support language or application specific interaction techniques. We have taken a language-centered approach for the principles underlying Escalante, meaning that visual applications are defined around the underlying specification of the visual language; as a consequence, the system tends to focus on the language module rather than on the editor module (or other application-specific modules that might be added manually).

The language and editor modules are made up of a predefined base component coupled with generated and programmed language specific components. The predefined component

[1] **E**nvironment for the **S**pecification and **C**onstruction of visu**A**l **LAN**guage applica**T**ions and **E**ditors

encapsulates general functionality and behavior of visual language applications. The generated component is created from a language specification defined using GrandView; it adds language-specific functionality. The programmed components of the language and editor modules are created manually by the language application developer and are used to modify or extend the capabilities and functionality of the application that are not addressed by the language specification. It has been our experience that complex visual language applications can be created with minimal manual programming.

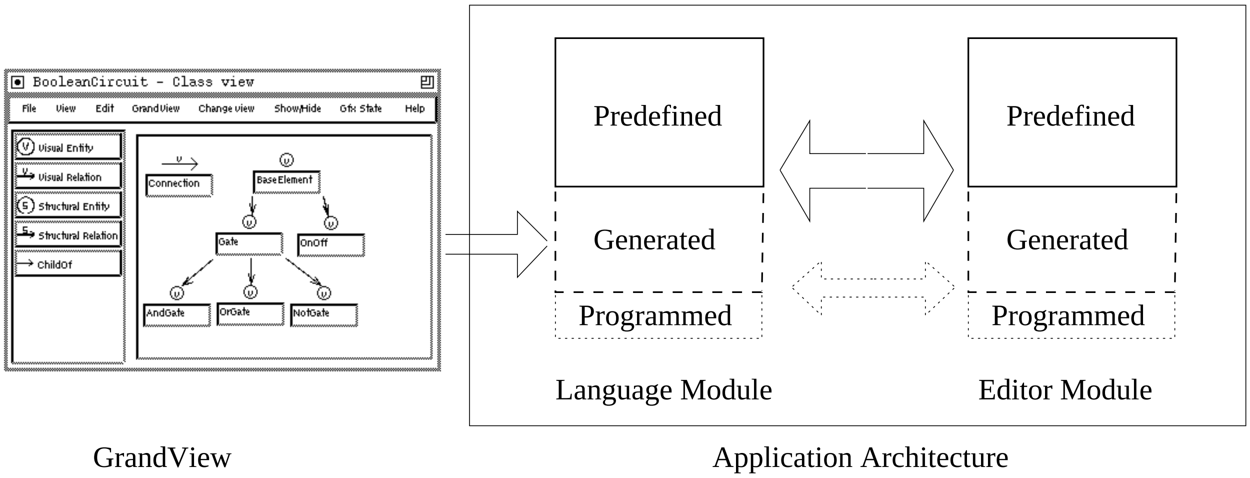


Figure 1: System Architecture

The process of constructing a visual language application consists of creating the language-specific editor and language modules using GrandView. The developer uses GrandView to specify the constructs and characteristics of the target visual language, then code is generated that realizes the specification. "Hooks" are provided to extend the functionality of this generated code if that is desired (e.g., additional language-specific semantics, different interaction techniques, or different look and feel). The generated editor module serves as a template and provides the developer a rich framework that can be tailored to construct the language-specific editor component.

The following section is a discussion of the underlying principles of Escalante including the structure of the language and editor modules and the interaction between them. Section 3 discusses how GrandView and Escalante are used to produce a working visual application, including a detailed description of the use of GrandView in the language specification process. The code that is generated by GrandView is described and the hooks that are provided to incorporate additional language or application functionality are discussed. In Section 4 we present an extensive set of visual application examples that have been built using Escalante. It is intended to demonstrate the diverse domain of visual languages to which Escalante is applicable, and to illustrate how Escalante can be used to realize these visual language applications.

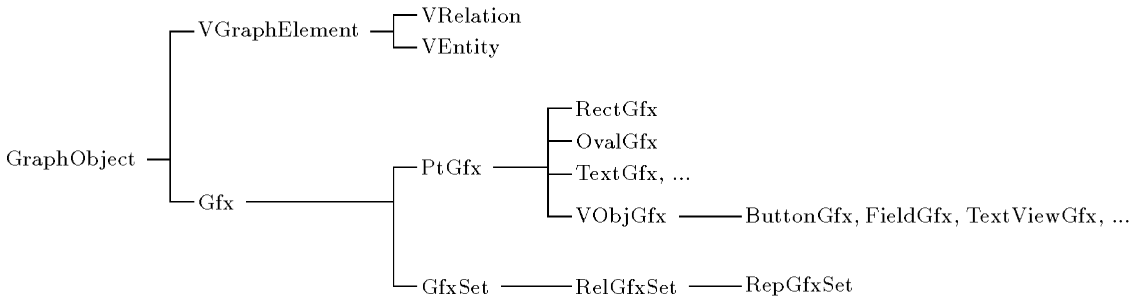


Figure 2: Base Hierarchy

2 Principles

To provide coverage for a broad spectrum of visual languages, we have developed a conceptual *language characterization framework*. The development of the language module has been based on this framework; it is used to describe the constructs and characteristics that make up the visual languages we address. It provides a very general view of graph model based visual languages. In this section we discuss the overall structure of the language module and the major points of the characterization framework that underly the module. We then describe the editor architecture and the interactions between the editor and language modules.

2.1 Language Module

In Escalante we implement the constructs that make up a visual language as a collection of static classes organized as a type hierarchy. A *visual program* is a collection of objects instantiated from these classes. That is, the class hierarchy defines the language, while an instantiation of objects defines a particular instance of that language. Figure 2 shows the base language class hierarchy within Escalante. There are two groups of basic classes: *visual graph elements* (*VGraphElement*) and *graphic primitives* (*Gfx*). The language developer defines the specific constructs of a visual language by deriving classes from the base visual graph element classes.

For example, Figure 3(a) shows the set of classes that are used to define the constructs of the Boolean Logic Circuit example shown in part (b) of the figure. Instances of the predefined graphics classes are used to specify the representation of the element. In the case of the Boolean Logic Circuit example the images of the VGraphElement classes are defined using instances of the BitmapGfx, ImageButtonGfx, PolyGfx, and TextGfx graphic primitives.

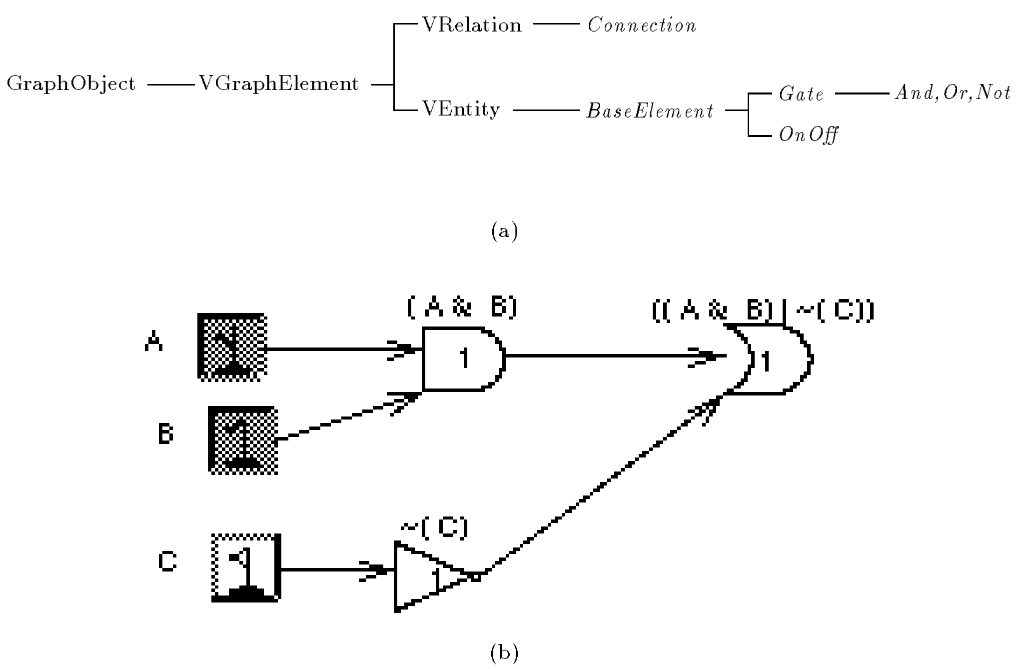


Figure 3: Boolean Logic Circuit

2.1.1 Graph Objects

The *GraphObject* class is the root of the language class hierarchy. This class implements an *attribute value mapping mechanism* to propagate attribute values from one object (source) to the attributes of other objects (targets). One can place any number of *attribute filters* along the attribute mapping path. These filters allow for the modification of the values and the control of the attribute mapping process by providing mathematical operations (e.g., addition, division, log), simple control flow, (e.g., if(attribute $<$ number)then), logical operations (e.g., $=, <, >$), type conversions and access to other attribute values of the source or target objects. The specific uses of this mechanism are discussed throughout this paper.

2.1.2 Elements and Relations

The core concept of the characterization framework (and the language module of Escalante) is the way we characterize or describe the constructs which make up a visual language. In the domain of graph model based visual languages, one encounters a wide variety of constructs (e.g., node, edge, port, graph, subgraph, aggregation). Each of these constructs may have its own particular characteristics and behavior. For example, a subgraph in a particular language may be viewed and manipulated in particular ways. The elements of the subgraph may be viewed in a separate window. The subgraph construct may be used as a visual abstraction mechanism (i.e., the elements in the subgraph are not shown when the subgraph

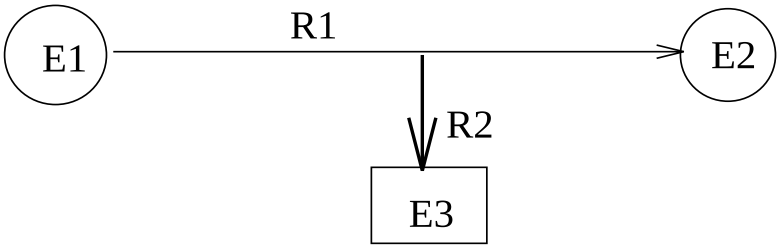


Figure 4: Element/Relation Connectivity

is shown). On deletion of the subgraph construct, the elements of the subgraph may also be deleted.

To provide explicit programmatic support for the diverse language constructs one may encounter we distinguish between the constructs and their characteristics (or behavior). All language specific constructs are generalized into a simpler form - elements and relations. An *element* represents a thing within a language (e.g., node, graph, edge, subgraph, aggregation). A *relation* is also an element (i.e., the relation construct is derived from the element construct) that is explicitly able to represent an arbitrary connection between two elements.[2] An instance of a relation is an object that connects two other elements, termed the *tail* and *head*. (The tail and head may be null in some cases, suggesting that the relation may be treated much like an element.) Figure 4 shows one representation of the connectivity of elements and relations. We use this particular representation for discussion purposes only. This graph could be thought of as G = {E1, E2, E3, R1(E1,E2), R2(R1, E3)}. The tail of the relation R1 is the element E1. The head of R1 is the element E2. R1 is considered an *out relation* of E1 and an *in relation* of E2. The *connected elements* of an element are the elements that are the tail (head) element of in (out) relations of the element. For example, element E2 is a connected (out) element of element E1. The tail or head of a relation can be a relation. The tail of the relation R2 is the relation R1. Within a tail/relation/head grouping there are six different relationships or *relation connections* that exist: tail → head, tail → relation, head → tail, head → relation, relation → tail and relation → head.

We use the element/relation characterization to make explicit and concrete the (possibly implicit or abstract) constructs and relationships that occur within visual languages. For example, in a visual language there may exist one element that spatially contains another element. In Escalante this implicit relationship between the two elements would be made explicit by a relation object. One of the characteristics of this relation would be to define the spatial constraint of containment between its tail and head elements. Another example of abstract constructs is the concept of a graph (i.e., a collection of nodes and edges). In Escalante a graph is an element; the implicit relationship that captures the notion of an element being a *member of* a graph is made explicit by a concrete relation that connects the graph element and the element which is the member of the graph (e.g., Figure 10). The implicit member of relationship defines certain behavioral properties of the graph and the

[2]The entity construct is used to disambiguate a relation from a construct that is not a relation.

member of the graph. For example, if one were to draw the graph this would imply the drawing of the member of the graph. If one were to delete the graph this would imply the deletion of the member of the graph. In Escalante the explicit member of relation concretely defines these implicit behaviors.

The structure of a graph can be defined using elements and relations, but the the behavior must be defined using other mechanisms. We have generalized the functionality and behavior one encounters in specific visual languages and applications, then encapsulated this knowledge as general mechanisms in the element/relation constructs. Many of these mechanisms can be defined within a relation and act with respect to one or more of the six relation connections described above. These mechanisms include propagation of events, propagation of attribute values, visual dependencies, and spatial constraints. These mechanisms are discussed in this and later sections.

Event Propagation Certain events (such as deletion, copying, and drawing), occur to elements of a visual language within the context of an interactive application. When an event occurs to an element it often causes other events to occur to related elements. For example, the deletion of an element may cause the incident relations of the element to be deleted. The deletion of a graph may cause the member of relations and the members of the graph to also be deleted. The *event propagation mechanism* allows one to define and control this propagation of events. Relations contain flags associated with event/relation connection pairs. The values of these flags determine the propagation of the events. In Figure 4, if one were to desire that on deletion of element E1 the incident relation R1 should also be deleted then one would define within the relation that the "deletion event" be propagated from the tail to the relation.

Relation Attribute Propagation One can make use of the attribute mapping mechanism to define mappings among the six relation connections. For example, one can define that changes to the width of the tail of a relation are propagated to the width of the head of the relation.

2.1.3 Visual Graph Elements

The three classes, *VGraphElement*, *VRelation*, and *VEntity* make up the basic set of visual element classes. These classes, in conjunction with the graphic primitives discussed below, encapsulate the state and functionality related to representing constructs on the screen.

Elements of type VGraphElement have a screen position which is defined by two points, P1 and P2, and a set of joint points. The VGraphElement attribute, *theimage*, contains a set of graphics objects which provide the representation of the element. The position and layout of these graphic objects are defined with respect to the points P1, P2 and the joint points. The VEntity class is used to disambiguate visual elements that are derived from the VRelation class from those that are not.

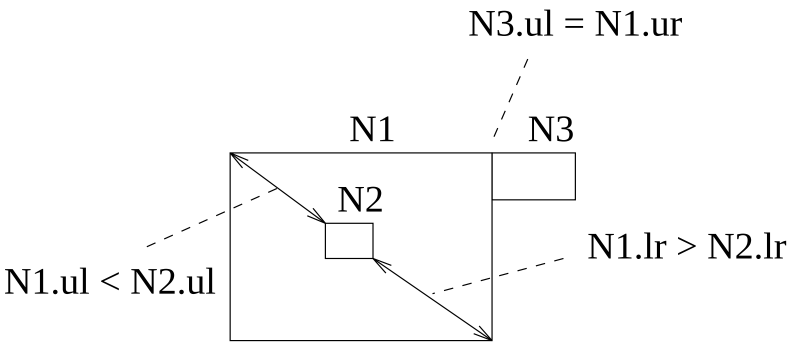


Figure 5: Location Constraint

The VRelation class is derived from VGraphElement. By default this class positions its points, P1 and P2, to the closest points on its tail and head element, respectively. This functionality can be controlled so that the points P1 and P2 are independent of the tail and head.[3] A visual relation may exhibit any type of graphical representation that a VGraphElement exhibits. It is not constrained to look like an edge.

Location Constraints *Location constraints* are used within visual relations to define spatial relationships between any of the relation connections described above, (e.g., tail and head, head and relation) The position of a point of the target element is constrained to be greater than, equal to, or less than a point from the source element. Figure 5 illustrates the use of location constraints to define containment and adjacency. To define the spatial relationship of containment two location constraints are used; the first defines that the upper left corner of N1 is less than the upper left corner of N2.[4] The second constraint is based on the lower right corner of N1 and N2. The spatial relationship of adjacency in Figure 5 is defined by constraining the upper left corner of the target node, N3, to be equal to the upper right corner of N1.

2.1.4 Graphic Primitives

The graphical depiction of an element is defined using the set of graphics classes shown in Figure 2. The *Gfx* class encapsulates graphics state such as color, fill, and pen width. The attribute mapping mechanism can be used to define a mapping between attribute values within a visual graph element and attribute values within the set of Gfx objects which form the image of the element.

The VGraphElement class attribute, *theimage* (mentioned above), is a pointer to a *GfxSet* object. The GfxSet class has a number of children Gfx objects. Figure 6 is an example of a set of Gfx objects and the image the objects produce. The children of a GfxSet object may

[3]Using the methods CalcHdPt(FALSE), CalcTlPt(FALSE).

[4]The origin of the coordinate system is the upper left corner. Positive X axis is to the right. Positive Y axis is down.

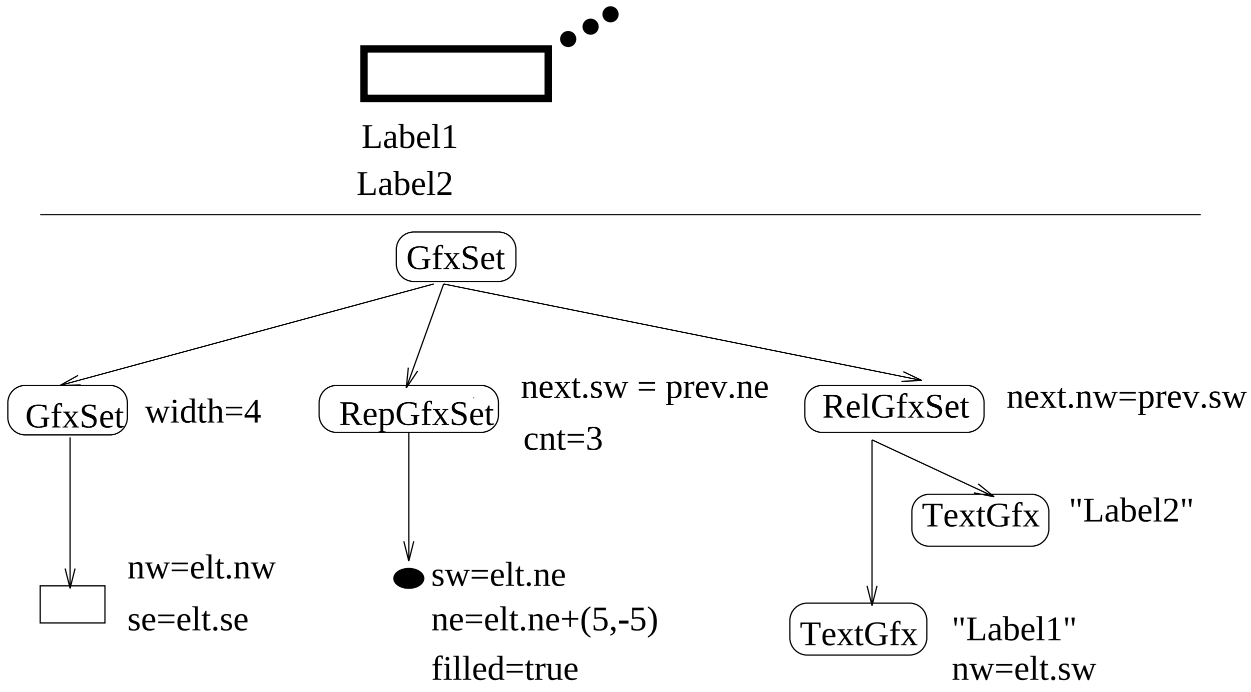


Figure 6: Example Gfx Instance

inherit graphics state associated with the parent GfxSet. The children of a *RelGfxSet* object are consecutively laid out with respect to one another. For example, in Figure 6 the two TextGfx objects, Label1 and Label2 are laid out so that the Northwest corner of the Label2 object is equal to the Southwest corner of the Label1 object. The *RepGfxSet* object repeats the display of its child Gfx object a certain number of times. In Figure 6 the OvalGfx object is repeated three times.

The *PtGfx* classes provide the actual image that is shown on the screen. Included in this set are classes for rectangle, oval, wedge, text, bitmap, and polygon. The size and position of these objects are defined with respect to the VGraphElement attributes, P1, P2 and joints. These points can be used directly as reference points or the bounding rectangle of the points, P1 and P2, can be used (e.g., North, East, Center). The position of the rectangle object in Figure 6 is defined using the Northwest and Southeast corners of the associated visual element.

The *VObjGfx* classes provide the ability to create representations that respond to direct user input. These classes include text fields, buttons, text views, and menus. The attribute mapping mechanism is used to define the mapping between input values, (e.g., button click, text, menu selection), and the attributes of the visual element associated with the Gfx object. This type of graphical representation is useful in raising the level of user/system discourse. One can directly manipulate the internal state of the language constructs in a modeless manner.

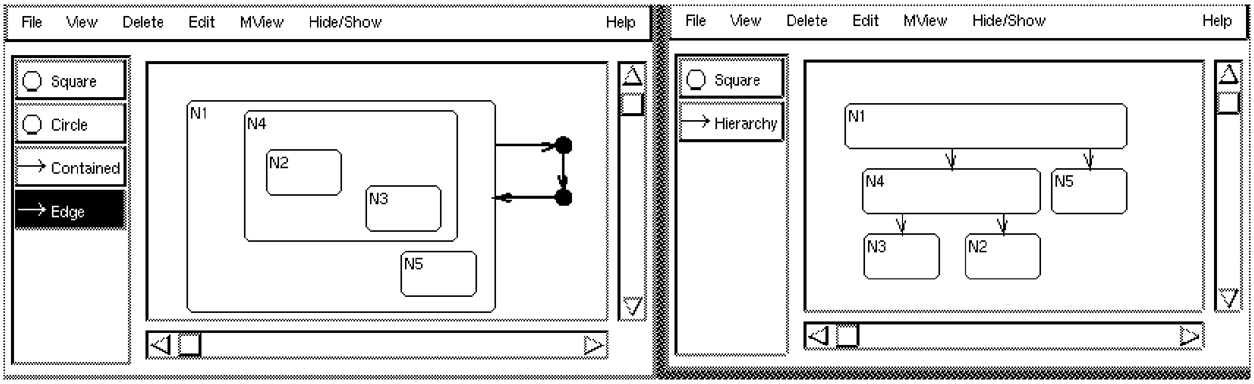


Figure 7: Multiple Representation Example

2.1.5 Multiple Representations using Structural Graph Elements

Some visual language applications may consist of two or more distinct, yet related, sets of visual elements (or visual graphs). Figure 7 shows an example of this *multiple representation* functionality. This application is made up of two groups of visual elements. Each of the visual elements in a window is related to visual elements in the other window. When adding an element to the visual graph in one window, elements of the appropriate type are added to the visual graph displayed in the other window. Likewise, when deleting a visual element the corresponding element in the other graph is also deleted. The set of corresponding visual relations connect corresponding elements. For example, the relation of type Hierarchy between the elements labeled N1 and N4 in the right window corresponds to a relation of type Contained between the elements labeled N1 and N4 in the left window.

To implement multiple program representations there must be some way of associating two or more distinct sets of visual elements. This association must be in terms of the overall structure of the graph (i.e., elements, relations, connectivity) and the internal attribute state of related individual visual elements (e.g., labels). To associate visual elements, Escalante provides a set of *structural element* classes (Figure 8). These classes, *SGraphElement*, *SRelation* and *SEntity*, mirror much of the basic functionality of their respective visual element classes. This common functionality includes basic element/relation connectivity, event propagation and attribute mapping. However, they differ from the visual element classes in that there is no functionality pertaining to the visual representation of the element. Rather, instances of structural element classes contain a list of their related visual graph elements and provide a common point through which the visual graph elements are related (e.g., Figure 11). The VGraphElement class contains a pointer to its structural element.

The 1:n relationship between structural and visual elements provides a simple and regular way in which multiple representation applications are built, allowing for a wide variety of approaches in its use. One could define a language initially as a set of structural elements and then define a set of visual elements to arrive at a representation of the structural elements. One could add a set of structural elements to a previously defined set of visual elements in order to realize a different representation of the original visual elements.

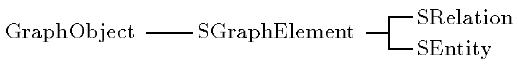


Figure 8: Structural Element Classes

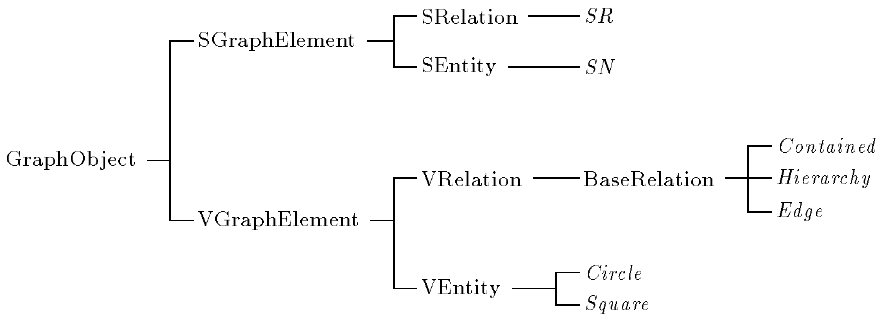


Figure 9: Multiple Representation Class Hierarchy

In the example shown in Figure 7, the left window is made up of visual entities of type *Square, Circle* and visual relations of type *Contained* and *Edge*. In the right window there are Square entities and *Hierarchy* relations. When adding an element of type Square to the graph in one window, an element of type Square is also created and added to the graph in the other window. When a relation of type Contained is added in the left window, a relation of type Hierarchy is created and added to the graph in the right window. Elements of type Circle and Edge in the left window have no corresponding elements in the right window. Instances of the structural element types SN and SR are used to relate the Square, Contained, and Hierarchy visual elements together. Figure 9 shows the hierarchical structure of these classes. Further examples of the use of multiple representations are given in Section 4.5 and 4.7.

The attribute mapping mechanism discussed in Section 2.1.1 is used to bidirectionally map attribute values between structural and visual elements. In the application in Figure 7, the Square visual class and the SN structural class, both have a character string attribute label. An attribute mapping is defined between each Square element and its corresponding SN element. Changes to the label attribute (e.g., through the TextFieldGfx) in a Square element are propagated to the label attribute of SN. This change is then propagated to the other visual elements associated with the SN element.

2.2 Editor Module

Figure 10 shows a conceptual view of the objects and classes that make up the editor component. The *EscalanteManager* object contains a set of *EscalanteDocument* objects. In turn

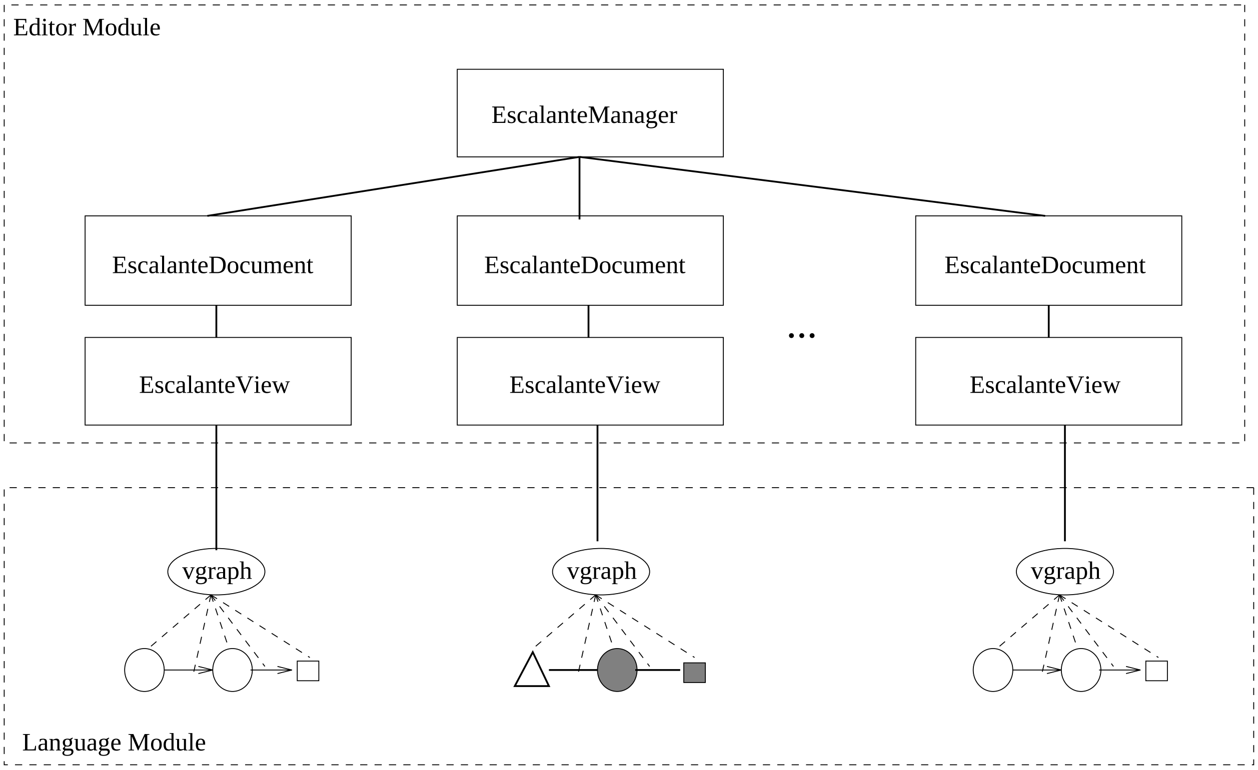


Figure 10: Editor Architecture

these document objects each contain an *EscalanteView* object.[5] A globally defined variable, *gEscalanteManager* is used to access this hierarchy of objects. Section 4.6 describes an example application that uses the gEscalanteManager to access all documents and all views of an application.

Much of the base editor functionality is encapsulated within the EscalanteView class. This class has an attribute, *vgraph*, which is a pointer to a VGraphElement object. It is through this element that the editor accesses and manipulates the graph as a whole. The EscalanteView class encapsulates a wide range of visual program editing capabilities including: the creation, deletion, and copying of language elements; graphical editing capabilities such as moving, resizing, scaling, alignment and simple layout; and grouping and manipulating groups of elements. There is a framework provided for creating online help. N-level undo/redo of element creation, deletion and movement is supported. One can copy/paste and export/import components of a graph. Very flexible mechanisms also exist for multiple views, viewing subgraphs and filtering out the display and selection of elements.

[5] An application would have application-specific Manager, Document, and View classes derived from these base classes.

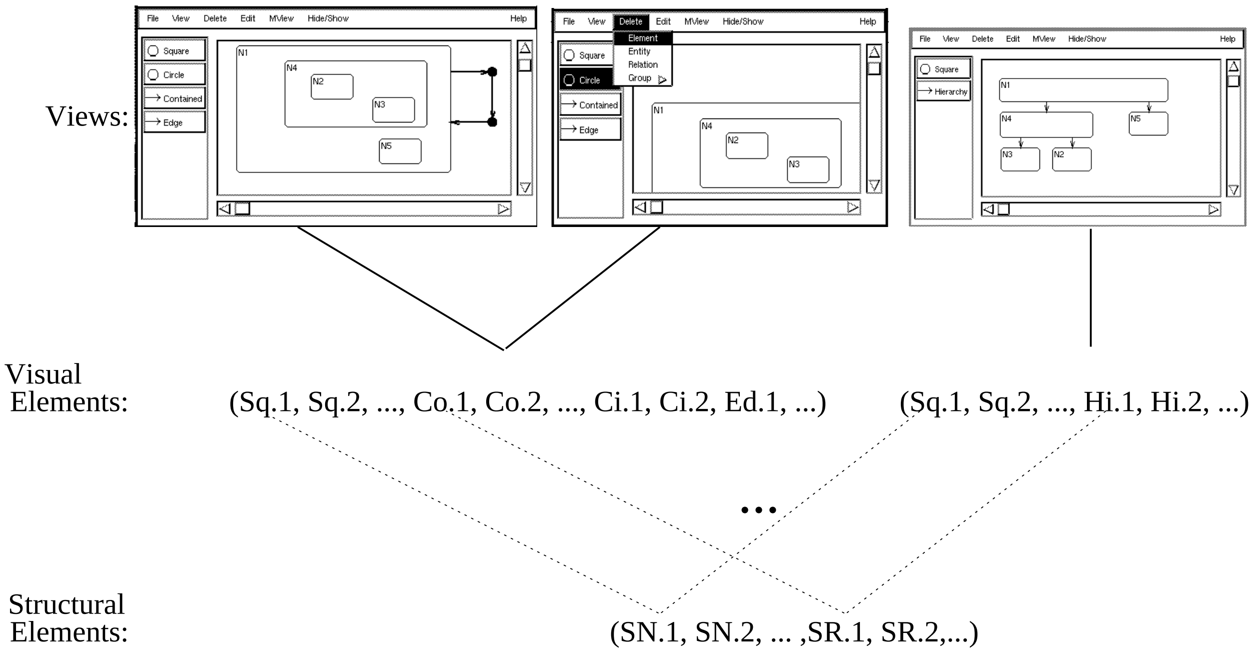


Figure 11: Editor/Language Configuration

2.3 Editor/Language Configuration

Escalante supports multiple model representations and multiple windows or views of those representations. The overall architecture is very flexible in how a system of structural elements, visual elements, and views is configured. Figure 11 shows one particular application configuration. (This example is the same as the one used in Figure 7.) The elements of the structural graph are related to two sets of visual elements (i.e., two visual graphs). The set of visual elements on the left are displayed in two separate views. The set of visual elements on the right are displayed in one view.

During a run of an application, one may dynamically create any number of views (using the View/Views/New View menu command). For example, in Figure 11 the set of visual elements on the left are displayed in two views. Likewise one can dynamically copy a set of visual elements, maintaining the connection to the underlying structural elements (Using the View/Views/Copy Graph menu command). A new view is created for this new visual graph. One can have any number of visual graphs associated with a common structural graph. Those visual graphs may be made up of elements of the same type or of different types. The elements of a visual graph may or may not have corresponding elements in the structural graph. Likewise, the elements of the structural graph may not have corresponding elements in a visual graph. It is not necessary to use a structural graph. There can be any number of views in which a visual graph is accessed and displayed. These views may display the visual graph as a whole or components of the visual graph. A view may define certain filters which control what elements are displayed.

3 Creating a Visual Language Application

The principles that underly the design of Escalante have been explained in Section 2; in this section we focus on how to *use* Escalante to build a visual application. First, note that Escalante is intended to be used for a spectrum of visual applications ranging from simple to complex ones; any specific application is not likely to use all aspects of the system as they are described in Section 2. That is, the system has been designed with the intent that simple visual applications be relatively easy to build, and so that increasingly complex applications use increasingly sophisticated aspects in Escalante; for example, it is not necessary to understand the principles behind multiple program representations unless one intends to build an application that uses them.

The principal tool used to create a visual language application is the GrandView specification environment and its concomitant visual language *Grand*. Through GrandView one defines the target visual language using Grand constructs and generates the C++ code that realizes the language module. We have taken a language-centered approach in the development of Escalante. We provide, through GrandView, extensive support in defining and constructing the language module (i.e., the data model) with less specific support in the construction of the editor module (i.e., the control model). GrandView produces a template of the target editor module and provides limited support in tailoring the target editor module (e.g., menu definition).

There are three steps in creating a visual application with Escalante: first, one must create a Grand specification of the target visual language using GrandView; second, through GrandView the generated editor and language modules are created; finally, application-specific functionality can be manually added by extending the generated editor and language modules. The following subsections provide a detailed description of each of these steps. Section 4 illustrates each of the concepts described in this section with detailed examples of visual applications that have been built using Escalante.

It is difficult to describe the complete human-computer interface for any visual application, e.g., GrandView. In the remainder of this technical report, we assume that the reader has a copy of Escalante; the reader is strongly encouraged to use GrandView to explore the concepts and to provide the inevitably missing context for many of the explanations.

3.1 ET++

Escalante is built using the ET++ application framework toolkit [1]. ET++ provides an extensive class hierarchy that encapsulates: a multi platform user interface toolkit; an extensive set of collection classes (e.g., lists, dictionaries, sets); a meta-class class; and support for arbitrary object I/O. Escalante has been developed so that one can build visual language applications with little knowledge of ET++. However, there are some aspects of ET++ that are important to understand including the meta-class class Class and the ETRC resource file.

The meta-class class (called *Class*) contains information about other classes. To make use of Class one places the *MetaDef(Class_Name)* macro in a class definition and the *NewMetaImpl(Class_Name, Parent_Class_Name)* macro in the source file of the class. These macros create an instance of the meta-class Class for the defined class. The methods Class* IsA() and bool IsKindOf(Some_Class_Name) are created by these macros and are used to determine the class of the defined class and whether the defined class is derived from Some_Class_Name. The *Meta* macro is used to access the instance of the Class class for a particular class (e.g., Class * c = Meta(Some Class Name))

ET++ supports the specification of resource values using the resource file ETRC. In the ETRC file there are resources defined for both ET++ and Escalante. Using this one can change the look and feel of the interface. This includes the scroll bars of the view, palette structure and layout, creating palettes in separate windows and the creation of menus. One can add their own application specific resources to the ETRC file.

3.2 GrandView Overview

GrandView is a visual language environment for specifying and generating the language specific module of a visual application. Escalante encapsulates the predefined language module in a C++ class hierarchy. A particular visual language is defined as a set of classes derived from the predefined class hierarchy. An application is built by combining the resulting language classes, the language specific editor module and the base modules provided within Escalante. Since Grand is a visual language, this process is itself an instance of visual (meta) programming. The human-computer interface is based on *views* that represent the various aspects of the Grand specification, and *menus* to invoke operations on that state.

Views. Users interact with GrandView through a set of views of a specification. The fundamental views within GrandView are the *Class View* (Subsection 3.2.1) and the the *Prototype View* (Subsection 3.2.2).

GrandView supports many other views of the Grand specification, depending on the aspect of the specification on which the designer focuses at any given time. The *Change View* menu in the Class View invokes alternate views that support the definition of various characteristics and behavior of the selected class specification; Figure 12 lists the alternative views, the functionality within each view, and the type of class specification object required for that view. Each of the views will be described in more detail in the remainder of the section.

Menus. Menus are the primary mechanism for invoking operations on the specification; GrandView provides the *GrandView*, *Change View*, *Show/Hide*, and *Gfx State* menus among others. The GrandView menu contains entries to prototype a specification and to generate the visual application software. The Change View menu is used to invoke alternate views from the Class View, depending on element selection in the Class View. In alternate views, a Change View command changes the view for the current element. The Show/Hide menu

View	Functionality	Type
Class	Define language class construct hierarchy	
Attribute	Class attributes	All
Gfx	Graphical representation of a class	Visual elements
Event	Event mapping	Relations
Relation attr map	Attribute maps in relations	Relations
Check tail/head	Legal tail/head combinations for relations	Relations
$S \times V$ attr map	Attribute mapping between visual and structural elements	All
Location constraint	Spatial constraints within a relation	Visual relation
Group	Grouping of incident relations or elements	All
Define menu	Define menu entries	None
Default relations	Define default relations	None

Figure 12: GrandView Views

turns the visibility on or off for various components of an element's image. Using this menu, it is also possible to control the visibility of components in a subtree (e.g, Child Of relations in the ClassView). The middle mouse button turns off the visibility and left mouse button turns on the visibility. Depressing the shift key while turning on the visibility of a subtree causes only one level of the tree to become visible. The Gfx State menu lets one define graphics state of GfxSpec elements in the GfxView.

GrandView also makes use of default menus and commands provided by the base editor module. These include the *File*, *View*, *Delete* and *Edit* menus. The File menu contains commands for: loading and saving a specification; printing the screen; and exiting GrandView. The View menu contains commands that enable one to change various aspects of the view. The View/Views menu allow one to create new views or windows of a specification. Using the View/Flags menu one can set flags that control certain aspects of the interaction mechanisms. For example, in GrandView turn the Move Hints flag on. This causes the propagation of the movement of an element to other elements in various views in Grandview. The Delete menu allows one to delete individual elements, elements in the selected group or all elements. The Edit menu contains commands that support copying, moving, reshaping and picking elements. One can also change the size and position of individual Gfx primitives with the Edit/Gfx menu. The Edit/Align menu lets one layout the elements of the selected group in various ways.

Help. Much of the information in this section is available as online help in GrandView. Use the *help* menu within the context in which assistance is required.

3.2.1 Class View

The default view of a specification is the Class View (see Figure 13). There are four class specification elements: *Visual Entity, Visual Relation, Structural Entity*, and *Structural Re-*

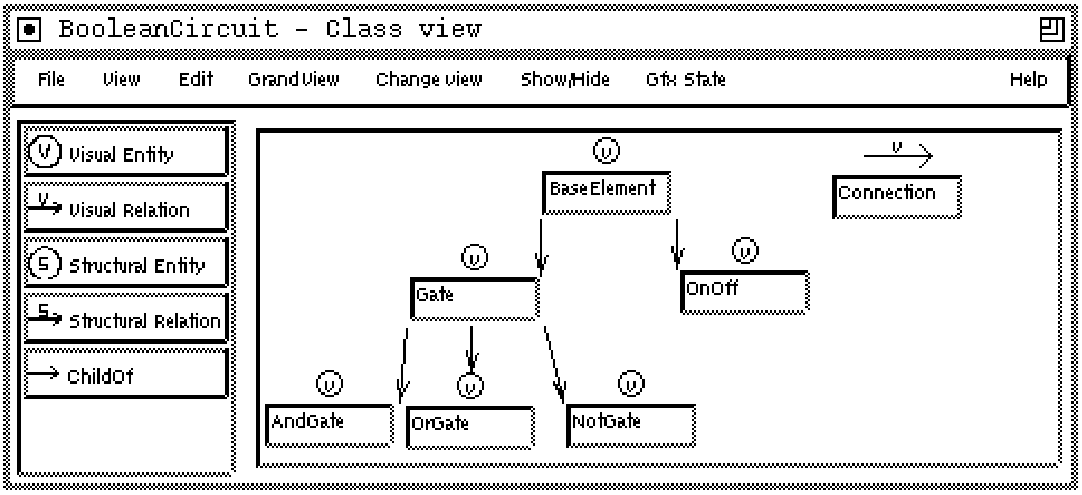


Figure 13: Class View

lation used in this view. Each of these class specification elements contains a TextField that allows one to set the name of the class. The Visual Entity and Visual Relation specification elements have a detailed image, accessed through the Show/Hide menu, which allows one to define the initial size of the element. The *Child Of* relation is used to construct a hierarchy of the class specification elements. Only legal connections of class specification elements are allowed with the ChildOf relation. The hierarchy defined with the ChildOf relation is reflected in the generated C++ code. By default, if a class specification has no parent class specification it is derived from one of the base classes: VEntity, VRelation, SEntity, or SRelation.

Figure 13 is the Class View of the Boolean Circuit example discussed in Section 2. In this example, all entities are derived from *BaseElement* which, by default, is derived from VEntity. The *Gate* and *OnOff* classes are derived from the BaseElement class. The *AndGate*, *OrGate*, and *NotGate* are subclasses of *Gate*. The *Connection* class is, by default, derived from the VRelation class. Most of the examples in Section 4 provide a discussion about their Class View specifications.

3.2.2 Prototype View

A *prototype* approximates the behavior of a specified language construct. Prototypes differ from the actual realization of the generated language construct in that certain aspects of the specification are not implemented in the prototype even though they would be provided in the generated program. The Prototype View window allows the user to see the results of an evolving class specification, including approximate behavior, prior to generating the C++ visual application. Prototypes are added to the prototype view using the *Add prototype* option in the GrandView menu. The *Clear prototypes* menu entry empties the list of prototypes in the palette of the Prototype View. Much of a specification can be prototyped, including class structure (i.e. inheritance), attributes, graphics, attribute mapping, event mapping and location constraints. However, it is not possible to prototype specified functionality that makes use of object type, (e.g., Check tail/head, Grouping). Specifications of structural elements cannot be prototyped (e.g., structural classes, $S \times V$ attribute mapping).

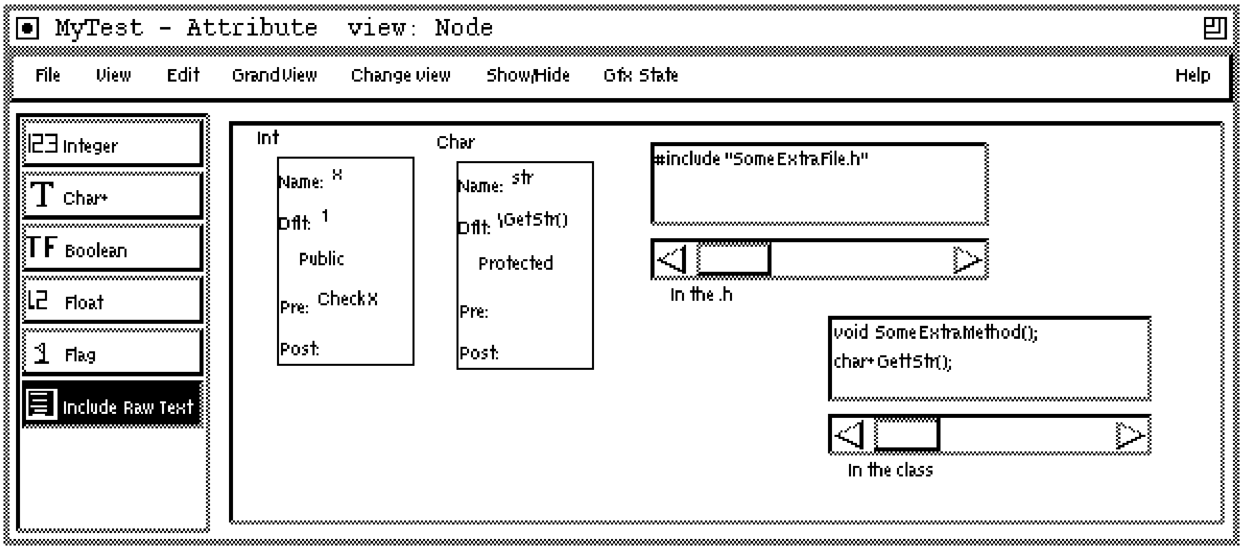


Figure 14: Attribute View

3.2.3 Attribute View

The *Attribute View* is used to inspect, modify, and add attributes to any class specification. An attribute can be an integer, character, boolean, floating point, or flag variable (i.e., a 1 bit boolean variable). An attribute specification consists of a name, default value, a C++ protection class (one of **public**, **private**, or **protected**), and pre and post functions. The pre function acts as a a precondition when setting the value of an attribute. The pre field is taken to be the name of a boolean function that is used to determine whether or not the value of an attribute is set. If defined, the pre function is called before the attribute is set. The post field is also taken to be the name of a procedure. If defined, the post function is called after the value has been set. The pre and post functions allow one to *tap into* the default control and data flow among a set of objects. One can insert user defined procedures into this predefined control flow. The signature of the pre and post functions are:

```
bool  pre(GraphObject * ptr, int attribute_id, data_type input_arg);
bool  post(GraphObject * ptr, int attribute_id, data_type attr);
```

The ptr parameter is the pointer to the element. The attribute_id parameter is the unique identifier for the attribute. Escalante uses the macro ATTRID(class_name, attr_name) to access the unique attribute identifier. The pre function takes the input argument that the attribute is to be set to. The post function takes the actual attribute. These two parameters can also be defined as aliases (i.e., data_type & input_arg).

The *Text Include* element is used to manually include *raw* text in the generated .h and .C files. The Menu at the lower left corner of the Text Include element determines where the raw text is placed in the generated code. It can be in the class definition, in the .h file or in the .C file (we discuss this feature in more detail in Section 3.4.1).

Figure 14 shows the Attribute View for an example class *Node*. The attributes specified are an integer attribute, x, and a character string attribute, str. The default value of x is 1. A pre function, CheckX(), has been specified for the attribute x. The character string attribute, str, is a protected attribute and has a default value derived from a call to some method, GetStr(). It should be noted that the value of fields that specify some value that in the generated code will be taken as a text string is written out in quotes. To write out text unquoted the first character in the field should be a \. This is the case with the dflt field of the CharAttr element.

Two Text Include elements are used to include raw text into the generated code. The first causes the inclusion of "#include "SomeExtraFile.h"" in the generated .h file of the class Node. The second Text Include specification causes the inclusion of new method definitions, void SomeExtraMethod(); and char* GetStr();, in the generated class definition of Node. See the Boolean Circuit example in Section 4.1 for a further example of the use of the Attribute View.

3.2.4 Gfx View

The graphics to be associated with a class can be added using the *Gfx View*. GrandView supports the specification of both basic graphical shapes (e.g., rectangle, line, bitmap) and various types of widgets that support direct manipulation of the internal state of an element. These elements are used together with grouping, positional, state, and attribute mapping elements to define a desired graphical outcome. Relations are used to establish a correspondence between the various specification elements.

The Gfx View, shown in Figure 15[6], is made of a set of GfxSpec elements that lets one define images and groupings of images. There is also a set of elements that lets one define characteristics of the Gfx objects (e.g. text state and position). Each of the GfxSpec elements consists of a bitmap that represents the type of Gfx. Some of the elements also show a rectangle which represents simple graphic state of the element (pen width, fill, pen color, and fill color). This can be changed using the *GfxState* menu. The GfxSpec elements also have a detailed image and a GfxBase image. The visibility of the detailed and the GfxBase image can be controlled through the Show/Hide menu (left mouse button to show and the middle mouse button to hide).

The GfxBase image of a GfxSpec consists of a Trans: menu, Id: field, and Border, Shown, Attachable and Pickable buttons. The use of the Trans menu is deferred to the discussion for defining Gfx locations. The Id field is used to define an integer identifier for the Gfx object.

[6]The two column layout of the palette is specified in the ETRC file.

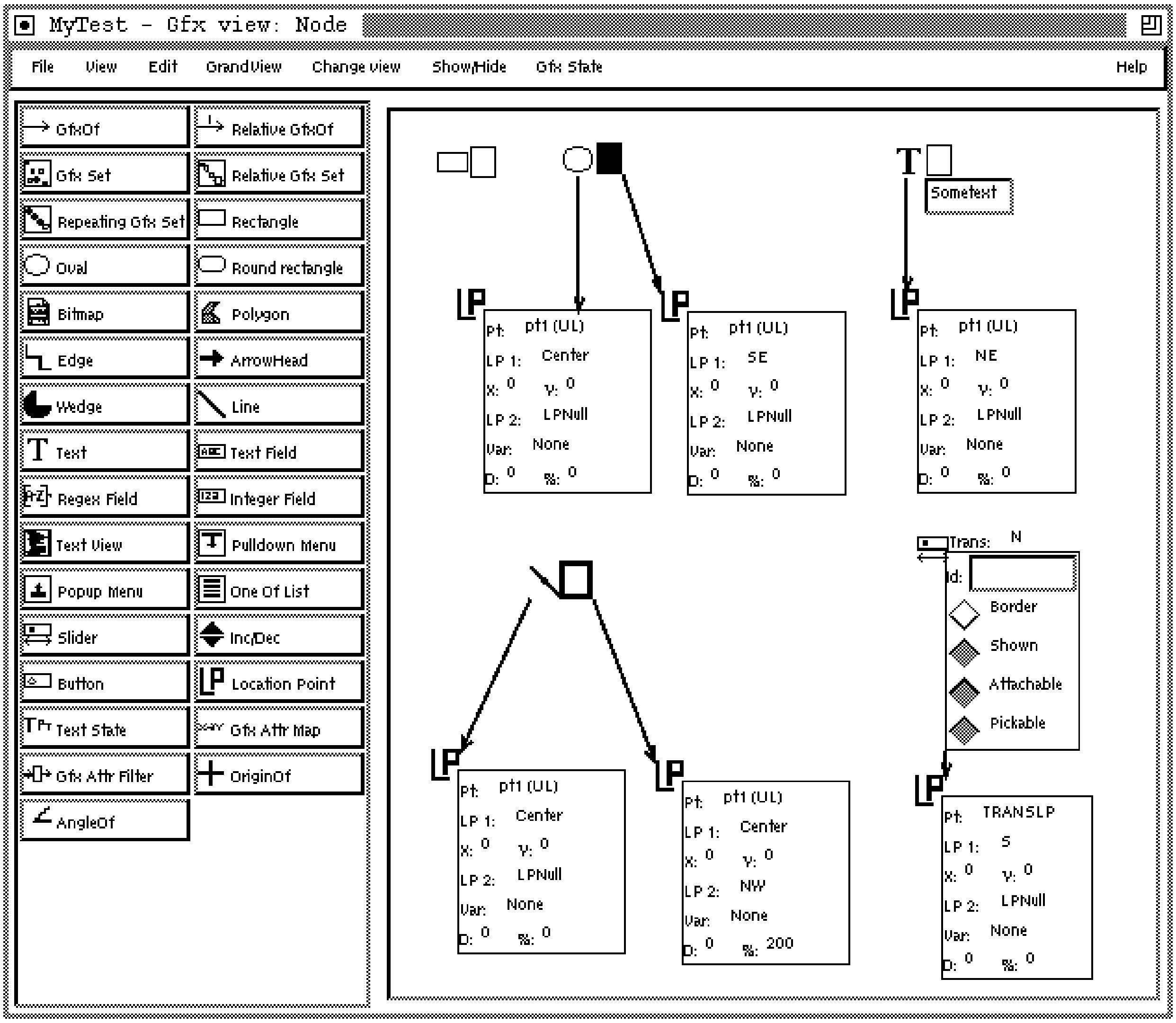


Figure 15: Location Points Specification

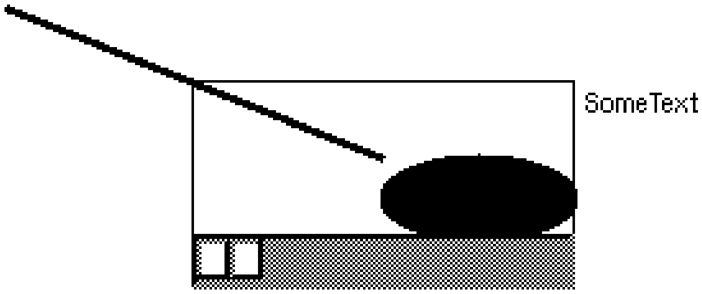


Figure 16: Result Gfx

Using this identifier one can access a specific Gfx object from its element. When the Border button is true, it indicates that the bounding rectangle of this Gfx object is drawn. The Shown button determines whether this Gfx is initially shown or not. The Attachable button determines whether a relation can be attached to this Gfx. The Pickable button determines whether this Gfx can be used when picking the element.

The relations *GfxOf* and *RelGfxOf* are used to relate Gfx objects (the latter also allows ordering of elements of a Relative Gfx Set).

Defining the Gfx Location. As discussed in Section 2.1.4, the position of a Gfx object is defined by a set of *location points* derived from the VGraphElement associated with the Gfx object. The available VGraphElement points are the points, P1 and P2, the bounding rectangle of P1 and P2 (e.g. NW, SE, Center), the joint points and the points of the refrect (e.g., Ref Rect N, Ref Rect SW) which is discussed in Section 3.2.9. For the case of the Gfx for a visual relation, one can also use the points of the tail and head to define the image of the relation. Most Gfx objects, (e.g. rectangle, oval, and widgets) require two points that define a rectangle in which the Gfx object is displayed. The TextGfx object requires one point, the upper left corner of the text. The PolyGfx object requires n points.

GrandView provides default location points for the Gfx objects. One can use the *Location Point* element in the Gfx View to override these defaults. When adding a Location Point element, a relation of type GfxOf is automatically added to the nearest Gfx element (if one is found). A Gfx object can be directly related to a Location Point with the GfxOf relation from the palette.

Figure 15 shows a set of GfxSpec elements with a set of Location Point specifications. Figure 16 shows the resulting graphical image. A Location Point specification consists of the particular Gfx point being specified (e.g., Pt: pt1, Pt:pt2). The LP1 field defines what point is used from the VGraphElement, (e.g., P1, NW, J1). The X and Y fields are offsets from that point. The LP2 field lets one define a second point from the VGraphElement. This point is used in conjunction with the D: field (fixed distance along line) and the %: field (percentage along line) so that the result point is derived by the following pseudo code:

```
if(LP2 != LPNull){  //If LP2 has been specified
     if (percent  != 0.0)  //If % not 0
//point = percent along line (LP1,LP2)
          result point = percent*(LP1->LP2) + Point(x,y);
     else
//point = distance, D, along line (LP1,LP2)
         result  point = D(LP1->LP2) + Point(x,y);
}
else   //LP2 has not been specified
     result point = LP1 + Point(x,y);
```

As shown in Figures 15 and 16 the Rectangle Gfx has no connected Location Points so it makes use of the default location points of (NW,SE). The Oval Gfx points are defined to be (Center, SE). The point for the Text Gfx is defined as NW+(5,0). The Line Gfx goes from the Center to 200%(Center, NW). The Slider Gfx makes use of the TRANSLP point. When set, this Location Point defines a point to which the Trans: point of the bounding box of the Gfx (from the GfxBase image) is translated. The Trans: point can be one of N, NE, E, SE, S, SW, W, NW and CTR. In the case of the Slider Gfx the default Location Points are used to determine its size. Once the size is determined, the SliderGfx is translated so that the N point of the Gfx bounding box of the Slider Gfx is set to the S point of the VGraphElement.

One can use the *OriginOf* and *AngleOf* elements in the GfxView to define a rotated coordinate system. This is useful for creating different styles of arrowheads. In Section 4.9.5 we discuss the use of these elements.

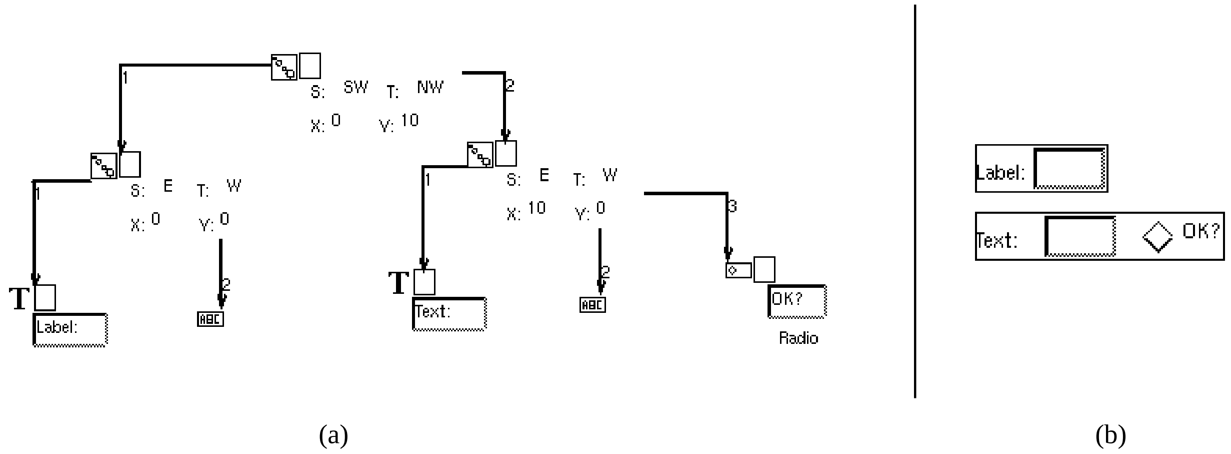


Figure 17: Relative Gfx Set

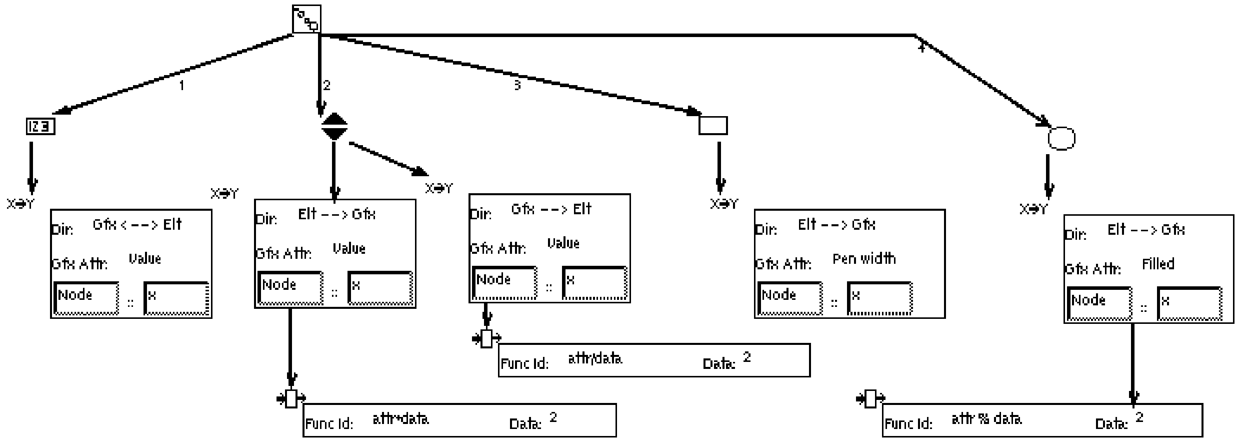
Gfx Set, *Relative Gfx Set*, and *Repeating Gfx Set* are used to create groups of Gfx objects which can inherit state defined in a parent GfxSet and allow for the control of the visibility of a group of Gfx objects. Children Gfx of a Gfx Set inherit any state (e.g. visibility and pen width) defined in the parent GfxSet and not defined in the child Gfx. The Relative Gfx Set enforces an ordering for the layout of the consecutive children Gfx. The detailed image of this element contains a source point menu (S:), a target point menu (T:) and an offset specification (X:,Y:). The RelGfxOf relation is used to provide an ordering of the layout of the children Gfx. The Relative Gfx Set lays out the first child Gfx at the location specified for the Gfx. Successive children Gfx are laid out with respect to the previous child Gfx. The target point specifies a point on the bounding box of the current Gfx that is set equal to the source point of the bounding box of the previous Gfx (plus the offset). The Repeating Gfx Set repeats a single child Gfx a specified number of times. It lays out these repeated Gfx in the manner of the Relative Gfx Set. See Section 4.9.4 for an example using a Repeating Gfx Set.

Figure 17a shows a Gfx specification that consists of a set of RelGfxSet, Text, TextField and Button elements. These are grouped together with the RelGfxOf relation. The RelGfxOf relation has an integer field that allows one to define the layout order of the connected elements of the RelGfxSet specification element. The head of the RelGfxOf relation with layout order = 1 is taken to be the first Gfx. The location of this Gfx is defined as usual. All successive Gfx of a RelGfxSet are laid out with respect to the first Gfx. Figure 17b shows the result of this specification. The Border has been turned on for the two lower Relative Gfx Sets using the Border button of the Gfx Base image. The TextGfx, "Label:" and the TextFieldGfx are members of a RelGfxSet that lays out its members West = East. The "Text:" TextGfx, TextFieldGfx, and ButtonGfx are members of a different RelGfxSet that lays out its members West = East + (10,0). These two RelGfxSets are themselves members of a third RelGfxSet that lays out its members NW = SW + (0,10).

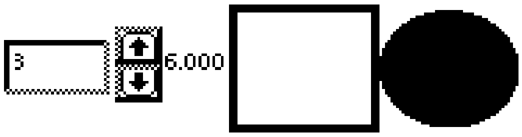
Gfx Attribute Mapping. Figure 18 shows an example Gfx specification that makes use of the *GfxAttrMap* and *GfxAttrFilter* elements. One can define an attribute mapping between a visual element and the Gfx objects that make up the image of the element using these elements. When adding these elements they will both be connected by a GfxOf relation to the nearest GfxSpec element found. If none are found then they will created without the GfxOf relation. One can then manually connect them with the GfxOf relation.

The Gfx Attr Map element consist of a direction menu, a Gfx Attr menu and element class and attribute fields. The direction menu lets one define the direction of the mapping (i.e., element to Gfx, Gfx to element, bidirectional). Figure 19 lists the available Gfx attributes. The element class and attribute fields lets one define the class name and attribute name of the element. This can either be a user defined attribute from the Attribute View or one of a set of predefined element attributes as shown in Figure 20. To make use of these predefined element attributes, one needs to define the attribute name field using those names shown in Figure 20. The class name field can either be as shown, or the shortcut names, GO and VG can be used respectively for GraphObject and VGraphElement.

The Gfx Attr Filter specification element consists of an operation and a data field. Any number of filters can be concatenated together. Initially each filter takes the value of the source attribute and applies the prescribed operation to the attribute with the value specified in the data field used for binary operations. The result value is passed to the next filter which in turn applies its operation, etc. The operations available include simple mathematical and



(a)



(b)

Figure 18: Gfx/Element Attribute Mapping

Gfx Attribute	Description
Value	Value of fields, views, buttons, etc.
Label	The label of a button. The text entry of a menu. etc.
Shown	Is this Gfx object shown.
Not Shown	Inverse of Shown.
Outline	Do we draw an outline around this Gfx .
Pen Width	The width of any line.
Filled	Is this Gfx filled.
Pen color,Fill color	What color (integer, colors defined in src/gfx/CommonGfx.h).
Pen color_map,Fill color_map	You can use the gColorMap to map from integer to specific color.
Pen grey, Fill grey	What level of grey scale do we use (float).
Text grey, Text font	
Text size, Text color,Text color_map	Various text states
Text face	Doesn't really work.
Width	The width of the Gfx (if applicable).
Width left, Width right	When changed the Gfx moves left (right).
Height, Height up, Height down	Just like width.
Area	Only Gfx to Element.
Slider value	The current value of the slider.
Slider min, Slider max	Lower and upper bounds of the slider.
Slider percent	What percent is the slider value between the lower and upper.
Lp1 perc, Lp2 perc	The percent factor for the location point.
Lp1 dist,Lp2 dist	The distance factor for the location point.
RepGfx::howmany	How many of the child gfx are shown.
RepGfx::maxreps	Maximum number of child gfx shown.
BitmapGfx::filename	Bitmap file name.
WedgeGfx::start	Where does a wedge start.
WedgeGfx::length	How much of the wedge is shown.

Figure 19: Gfx Attributes

logical operations (e.g., addition, log, not). Some operations let one define primitive control flow, passing on values or aborting the mapping depending on the operation. One can also refer to the value of the target attribute (e.g., if(target.attr > attr) then attr).

One can also set future data references to be the target attribute with the *future data = target.attr* operation or the source attribute with the *future data = attr* operation. This overrides any later data settings and uses the specified value as the new data setting. One can then clear this setting of future data references with the *clear future data* operation. One can also set the future data or the attr value with other attribute values from the source and target object. This is accomplished with the *data = target.attr(data), data = source.attr(data), attr = target.attr(data)* and *attr = source.attr(data)* operations. This functionality cannot be prototyped in the current version of GrandView. The data value is taken to be an integer value which is the identification of the desired attribute. From the specification editor it is best to use the from ATTRID(class name, attribute name) as the data field.

One can define their own attribute filters using the cUserDefined[1-10] operation specification. See Section 3.4.2 for more details on user defined attribute filters.

Figure 18a shows a specification of a set of Gfx objects for an element with integer attribute x. Figure 18b shows the result of the specification. This image consists of (from left to right) an IntFieldGfx, IncDecGfx, RectGfx and OvalGfx. The value of the IntFieldGfx is bidirectionally mapped to the attribute x. When one types into the field that value is mapped to the attribute x. Changes to the attribute x are mapped to the value displayed by the field. The value of attribute x is mapped through a multiply by 2 filter to the value displayed by the IncDecGfx object. The value of the IncDecGfx is mapped through a divide by 2 filter to x. The attribute x is mapped to the penwidth attribute of the RectGfx and is also mapped through a mod by 2 filter to the filled attribute of the OvalGfx. As shown in the figure the current value of x is 3. The value displayed by the IncDecGfx object is 3 (i.e., 2 * x). The pen width of the RectGfx is 3. The Ovalgfx is filled because x modulo 2 = 1.

3.2.5 Event View

The *Event View* is used to define event propagation among relations as discussed in Section 2.1.2. For example, we might want a relation to *die* (or be removed), whenever the tail of that relation dies. The user specifies an event type, direction for propagation, and state. Figure 21 describes a few event propagation examples: the first (*Die*) specifies that when the tail of the relation dies the relation also dies; the second (*Move by*) states that when the relation is moved, the head of the relation is also moved (events can go from a relation to the tail or head); the third (*Die hint*) states that when the global flag, gDoDieHints, is true on the death of the tail the head also dies. Hint flags can be set through the *View/Flags* menu. Figure 22 lists the events supported in the current Escalante release. During the run of an application one can directly set or check the value of an event propagation flag using the following methods:

```
bool    GetEventDep(Events event, DepTypes deptype);
```

Class	Attribute	Access	Type	Description
GraphObject	this	r	GraphObject*	Ptr to the object
GraphObject	id	rw	int	Id of the object
GraphObject	existence	rw	bool	On write this kills the element
GraphObject	ClassName	r	char*	classname
VGraphElement	name	rw	char*	name attribute
VGraphElement	shown	rw	bool	Is this element visible?
VGraphElement	p1	w	Point	
VGraphElement	p2	w	Point	
VGraphElement	angle	rw	double	The angle the points p1 and p2 make
VGraphElement	west	rw	int	X coordinate of bbox west side
VGraphElement	north	rw	int	Y coordinate of bbox north side
VGraphElement	east	rw	int	X coordinate of bbox east side
VGraphElement	south	rw	int	Y coordinate of bbox south side
VGraphElement	area	r	int	area of bbox
VGraphElement	height	rw	int	height of bbox
VGraphElement	width	rw	int	width of bbox
VGraphElement	length	r	double	distance(p1, p2)
VGraphElement	originx	w		
VGraphElement	originy	w		
VGraphElement	doriginx	w		
VGraphElement	doriginy	w	int	Taken as a delta, not as an absolute pt.
SGraphElement	name	rw	char*	name attribute

Figure 20: Predefined Element Attributes

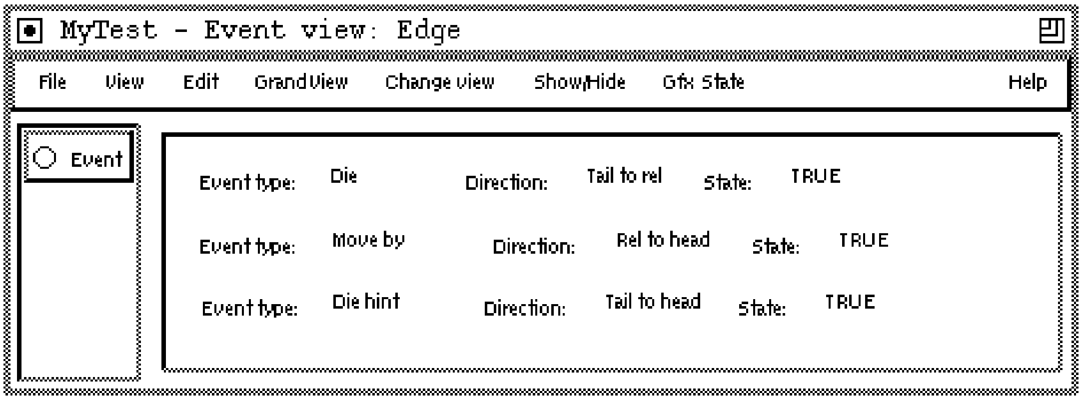


Figure 21: Event Specification

```
void    SetEventDep(Events event, DepTypes deptype, bool value);
```

Where the event and deptype are one of those listed in src/gm/CommonElt.h. See Section 4.8 for an example of using these methods.

3.2.6 Relation Attribute Map View

The *Relation Attribute Map View* allows the user to define attribute mappings between the six element pairs of relation connections (i.e., (tail $\rightarrow$ head),(tail $\rightarrow$ rel), (head $\rightarrow$ tail), (head $\rightarrow$rel), (rel $\rightarrow$ tail), and (rel $\rightarrow$ head)). In addition, an attribute value can be filtered (as discussed in Section 3.2.4).

Figure 23 is an example of the Relation Attribute Map View. The *Relation Attr Map* element defines the direction of the mapping and the attributes to be mapped. The *Attr Filter* element defines a function to be applied to an attribute (in the *funcId* field) and data to be used by the function (in the *data* field). The attribute filter mechanism is described in more detail in Section 3.2.4. The *Filter Of* relation is used to associate filters with Relation Attribute Maps. One uses the FilterOf relation to add an Attr Filter to a Relation Attr Map and to connect consecutive filters together.

In this example the relation Edge maps the attribute Node::x from the tail to the head of the relation. An attribute filter is defined that adds 1 to this value. The Filter Of relation defines a location constraint that causes the Attr Filter to be positioned under the Relation Attr Map. See Section 4.1 for further examples of the use of the relation attribute mapping functionality.

3.2.7 Check Tail/Head View

The *Check Tail/Head View* allows the user to specify the possible legal head and tail elements of a relation. This is accomplished by defining a set of boolean expressions graphically. The specification produces a method in the generated code that is invoked whenever the head or tail of a relation is about to be set.

The specification shown in Figure 24 is for the Connection relation of the BooleanCircuit editor described in 4.1. The *BoolSet* element defines the operators for an expression (these are *and* and *or* with negation). The *BoolOf* relation is used to structure an expression as a tree. The result value of a BoolSet specification is the operator applied to the result value of each of the connected elements of the BoolSet element. The *EltIsKindOf* element consists of a negation field, a class field and a position field (i.e., tail or head). The result of a EltIsKindOf specification is negation(position.IsKindOf(class)), (e.g., !tail.IsKindOf(Node)). The *TlHdFunc* element allows the user to specify a boolean function to be applied to the prospective tail or head. For a potential tail/head pair to be legal they must satisfy each of the subtrees defined in the view. The specification shown in Figure 24 defines that the tail of the Connection must be derived from the BaseElement class. The head of the relation must be derived from the Gate class. Or:

```
(tail->IsKindOf(BaseElement))&&(head->IsKindOf(Gate))
```

Event	Functionality
Die	When the element dies do others die.
Die hint	When the element dies others also die if the flag, gDoDieHints, is true.
Copy	When an element is copied(cloned) are others copied.
Copy hint	When an element is copied(cloned) others are copied if the flag, gDoCopyHints, is true.
MoveBy	When an element is moved are others moved.
MoveBy hint	When an element is moved others are moved if the flag, gDoMoveHints, is true. (See GfxView in GrandView).
Reshape	When an element is reshaped are others reshaped.
Scale	When an element is scales are others scaled.
Scale hint	When an element is scaled others are scaled if the flag, gDoScaleHints, is true.
Add Element	When an element is added to this element through the AddElement method is that element added to other elements.
Draw	Does an element draw other elements when drawn.
Signal Image Change	When the image or position of an element changes are other elements notified.
Moved	When an element moves are other elements told of the movement.
IBBox Reference	What elements does this element's reference bounding box contain. (See Section 3.2.9) (Note: One should also set the Moved event in the reverse direction the the IBBox Reference event is set).
End Point Reference	When an element's visibility is turned off where do the incident relations of the element connect to (if at all). (e.g., Section 4.8)
Shown Dependency	When an element is turned off does this force other elements to be turned off.
Shown Flag	This is not an event propagation. Rather it is a way to control the visibility of elements. e.g. if tail to head (or rel to head) of the shown flag is false then the head of the relation is not shown. (e.g., Section 4.8)
Event Set Visible	When an element is turned off are other elements also turned off.

Figure 22: Events

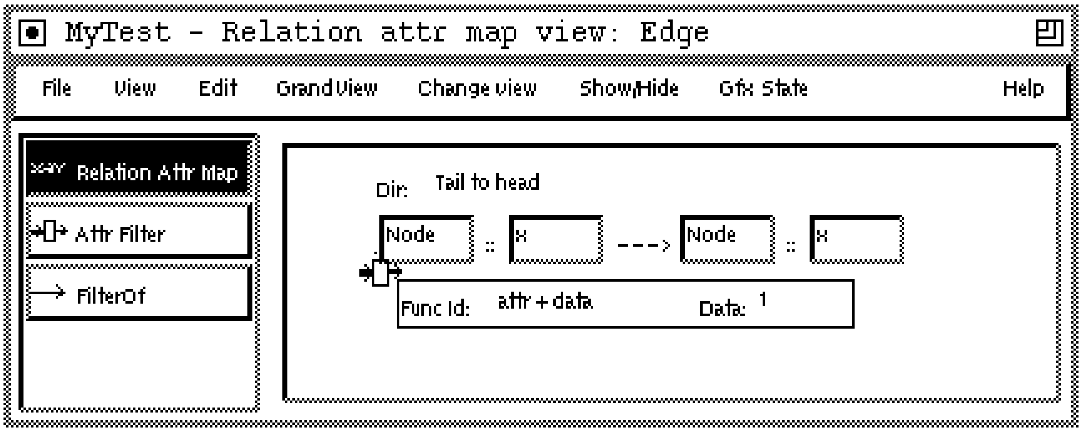


Figure 23: Relation Attr Map View

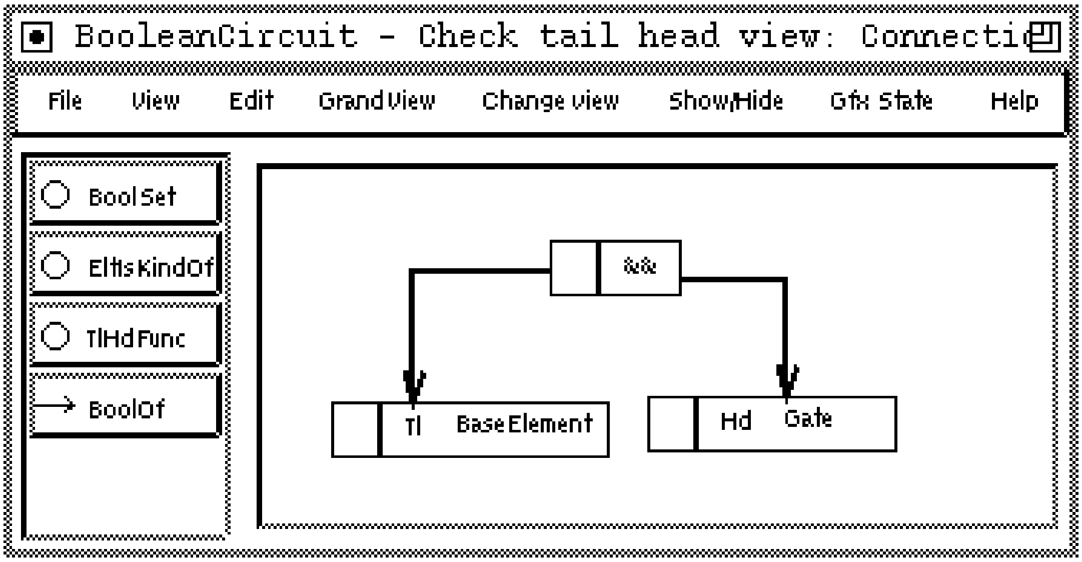


Figure 24: Check Tail/Head View

3.2.8 $S \times V$ Attribute Map View

The $S \times V$ *Attribute Map View* allows the user to define attribute mappings between structural and visual elements. This feature allows attribute mappings between the elements that make up multiple language representations. Figure 25 shows two attribute map specifications. A specification consists of a direction; visual class and attribute name; and structural class and attribute name. In the figure, the first specification defines a mapping from the SN::y attribute of the structural element to the VN::x attribute of the visual element. Also there is a bidirectional mapping defined between attributes SN::z and VN::z. If this view was for a structural class, than the SN::y to VN::x mapping would be added between the structural element and *all* visual elements associated with the structural element. If this view were for a visual class then the mapping would be added to the set of maps of the structural element by the visual element. This mapping would *only* be between the current visual element and its corresponding structural element. See Sections 4.5 and 4.7 for further examples.

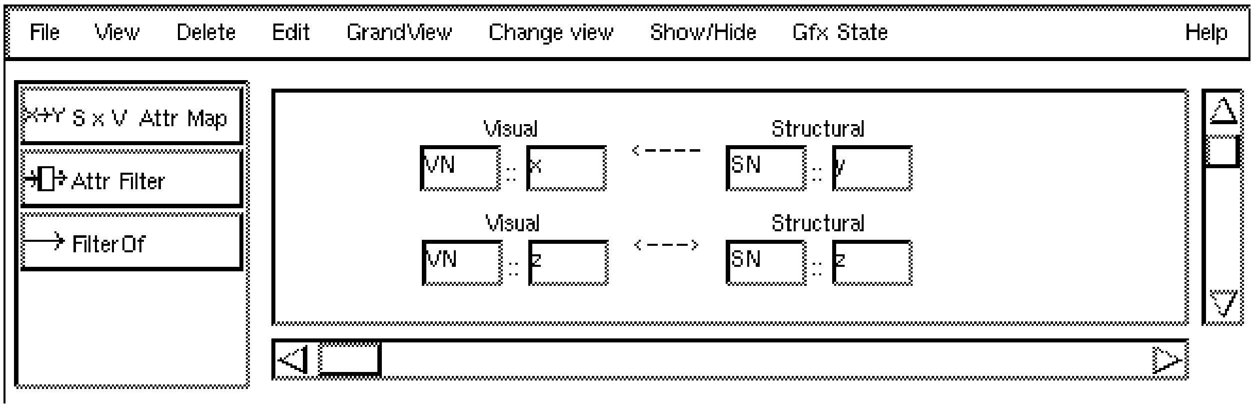


Figure 25: $S \times V$ Attribute Map

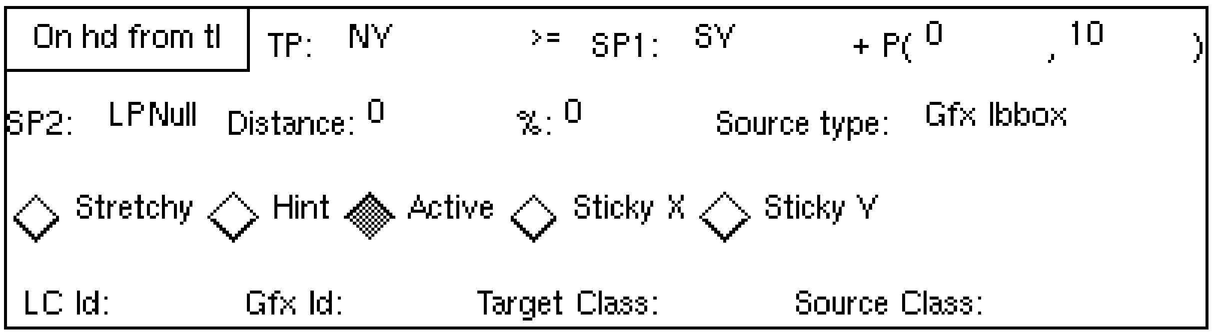
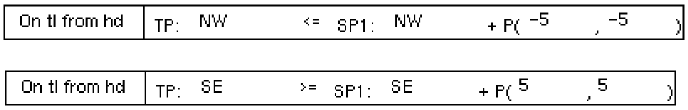
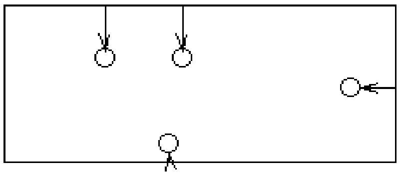


Figure 26: Location Constraint Example



(a)



(b)

Figure 27: Containment Example

3.2.9 Location Constraint View

The *Location Constraint View* allows for the definition of spatial constraints within a visual relation class (see 2.1.3). Figure 26 shows an example specification of a location constraint. The top line of this images defines that the Y coordinate of the north point (Tp: NY) of the head of the relation (On hd from tl) is greater than (>=) the Y coordinate of the south point (SP1: SY) of the tail of the relation plus 10 (P(0,10)).

The location constraint specification element is made up of a set of menu, field and button widgets. Referring to Figure 26, one can define which element is affected by which element (e.g, On hd from tl, On tl from rel) using the menu in the upper left of the element. The menu labeled TP: allows one to define the point on the affected element that is being constrained. One can then define the operator (e.g., $>, <, =$). The menu labeled SP1: is the point on the affecting element the target point is constrained around. The fields P(0, 0) lets one define an offset. SP2, Distance, and % lets one have finer control on defining a source point. Defining a source point is much like defining a location point (e.g. see 3.2.4). The source point is:

```
if(SP2 != LPNull){  //If SP2 has been specified
     if (percent  != 0.0)
source point = percent along line (SP1,SP2)
     else
source point = distance, D, along line (SP1,SP2)
}
else   //SP2 has not been specified
     result point = SP1 + Point(x,y);
```

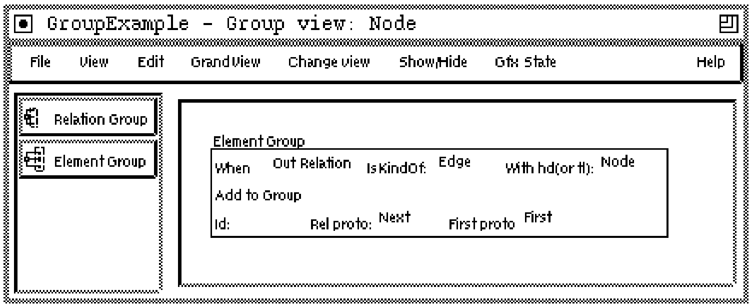
The Source type field lets one define what rectangle is to be used as the source point. "Gfx ibbox" is the bounding rectangle of the image of the source element. "Element rectangle" is the bounding rectangle of the points P1 and P2 of the source element. "Gfx box and ref rect" is the merger of the bounding rectangle of the image and the reference rectangle. "Just the reference rect" uses just the reference rectangle. The reference rectangle refers to *refrect*, an attribute of VGraphElement. The refrect for an element is set to the sum of the bounding boxes of the connected elements where the incident relation has its IBBox Reference event flag set (see Section 3.2.5, Figure 22).

When the Stretchy button is false other points of the target element are translated by the translation of the affected point (this maintains the size of the target element). When the Stretchy button is true only the affected point is moved (if possible).

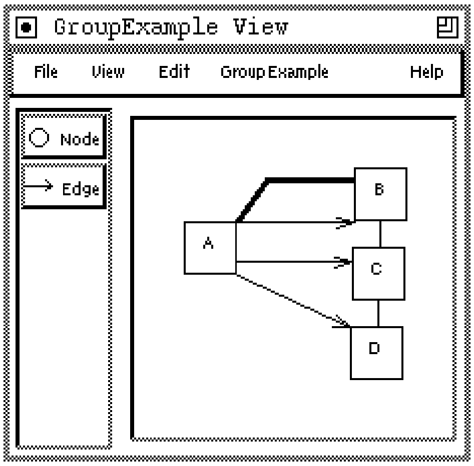
The Hint button allows one to control when this location constraint is active using the "LC Hints ON" menu entry in the View/Flags menu. The Active button lets one control whether this constraint is initially active or inactive. One can directly access the location constraints of a relation through the VRelation method *ObjList * GetLCList(int affect_what)*. This returns a list of the constraints which affect is defined by the affect_what argument, one of ONHDIX, ONTLIX, FROMHDIX, FROMTLIX. See Section 4.8 for an example of using this method.

When the StickyX and StickyY buttons are true, the affected points are moved to the closest affecting points. The LC Id: field lets one give an id to this constraint. One can access particular location constraints with the LC Id from the VRelation::lcs attribute. The Gfx Id: field lets you specify a Gfx object in the source element whose bounding box is used as the source rectangle. The Target Class: and Source Class: fields let one define that only certain classes are affected by this constraint.

Figure 27a shows how to define containment of the head of a relation by its tail (non essential details have been elided) with Figure 27b showing the result of this specification. If these constraints had had their StickyX and StickyY flags set, the affected element would only be as large as the size of the bounding rectangle of the affecting elements. See Sections 4.1 and 4.7 for other examples of defining Location Constraints.



(a)



(b)

Figure 28: Grouping Specification and Example

3.2.10 Group View

An element can have any number of incident relations and connected elements. One can group a set of incident relations or connected elements based on relation and element type. We term these *relation groupings* and *element groupings*. In its simplest form an element or relation group is simply a list, contained by the grouping element, of the grouped elements. One can also define a relation to link between the grouping element and the first element in a group. One can also define a relation that is used to connect the consecutive elements of a group. One can use the grouping mechanism to quickly access a set of incident relations or connected elements. One can use the relations being added to the group members to define a layout ordering using the location constraint.

The *Group View* of GrandView allows one to define element and relation groups for any graph element class. Figure 28a shows an example group specification for an element *Node*. Figure28b shows the result application constructed from the specification. In Figure 28a the grouping element is the entity labeled A. The entities labeled B,C and D are the members of the group.

A group specification consists of a direction menu, relation type field, element type field, group id field, relation prototype field, and a first prototype field. The direction can be either out or in. The relation type field is the name of a relation class (or null). The element type field is also a (possibly null) class name. One can give an identifier to a group. The Rel proto: and First proto: fields are (potentially null) relation class names.

The Relation Group acts similar to the Element Group except that the elements in the group are the incident relations. One need not specify any or all of the fields. The default is any relation with any head (tail) are the criteria for group membership. If there is no Rel proto or First proto relation specified then there is no relation connection made between the members of the group.

The specification in Figure 28a states that when there is an outgoing relation of type Edge with a head of type Node then add the head to the group. Add a relation of type First between the initial element and the first element in the group. Add a relation of type Next between successive members of the group. The First relation is the thick, jointed relation from A to B. The Edge relation is the directed edge from A to B, A to C and A to D. The Next relation is the undirected edge from B to C and from C to D. The Next relation defines a Location Constraint on the head of the relation. This constraint enforces the layout of B, C and D. See Sections 4.4 and 4.7 for examples of the use of the grouping mechanism.

3.2.11 Define Menu View

The *Define Menu View* allows the user to define menus and associated actions to be added to the generated editor. The specification of a menu action is realized by a set of generated .h files. These files are produced by the Build Everything command or by the Write out menu command. One can use the generated files as starting points in defining application-specific functionality. The developer is free to edit the .h files (in the indicated areas) to achieve the desired functionality. (Caution: Changes to these files may be overwritten by later file generation. It is best to merge the generated code from the .h file directly into the target source code.).

A menu action specification, as shown in Figure 29, consists of an action name, a need, a type, and a function name. The action name is used to define an enumerated variable and in defining the menu entry. The need defines what is needed by this action, e.g., an element, a point, nothing. If the need is set to Element, Entity or Relation then one can also define what type of element is to be picked using the Of type: field. The function definition (i.e., Or using func: field) is used to define a function that determines what type of element should be picked. This function has the signature bool func(VGraphElement*).

The result of the specification in Figure 29 would be a menu entry named "Test"; activating that entry would set the state of the View so that on a left or middle button press, an element of type Node would be searched for. The code that is produced and placed in the ActionDoLeft.h method is as follows:

```
if(action == Test){
      Command *  c = SatisfyNeeds(p,pickedVGraphElement, pickedVEntity,
                                  pickedVRelation, pickedGfx);
      if( c != gNoChanges) return c;  //If nothing was found then return
      if(pickedVGraphElement){
            Node * ptr = (Node*)pickedVGraphElement;
            /**Put your action here**/
      } return gNoChanges; }
```

If the need was set to Nothing then the following code would be produced in the ActionSetAction2.h file:

```
case Test:
       /***Put your action here **/
       break;
```

Figure 29: Menu Specification

Section 4.1 provides a detailed example of the use of the Define Menu View.

3.2.12 Default Relations View

The *Default Relations view* allows one to define the default creation of relations between elements in the target application based on element type and spatial positioning. Specifications in this view result in generated code that is included in the target editor methods *editor*View::DoneAddingElements and *editor*View::DoneMovingElements. The DoneAddingElements method is called when new elements have been added to a view. The DoneMovingElements method is called after some elements have been moved. The generated code creates new relations between previous elements and the new (or moved) elements based on the criteria specified in the Default Relations View. The code is written out from the *Write out dflt rels* entry in the GrandView menu and also from the Build Everything command.

Figure 30 shows this view and two example specifications. The first specification defines that a relation of type R2 is added between a new element of type N1 and all elements of type N2 that are spatially contained by the new element. There is further criteria in this specification that the old elements (i.e., the N2 elements) do not already have an incident relation of the type about to be added.

The set of available spatial relationships are: contains, contained by, near, under, over, left of, and right of. The direction menu (e.g., from old to new) allows one to specify the tail and head of the relation. The Func: field allows one to specify a function that allows for other selection criteria. This function has the signature:

```
bool func(VGraphElement* newelt, VGraphElement* oldelt)
```

When true, the Unique new? button defines that the relation will not be added if there already exists such a relation between the new element and another element of the old type. When true the Unique old? button defines that the relation will not be added if the old element has such a relation between it and an element of the new (or moved) type. The How many: menu specifies how many relations, 1 or N, are added. If only one is to be added, then the closest old element that satisfies the criteria is used. If N are to be added, then all old elements that satisfies the criteria are used. The X: and Y: field determine the absolute minimum X and Y distance between the closest points of the new element and the old elements. If the distance is within this range then the old element is considered for the relation addition. This criteria does not affect the contains and contained spatial relations.

The second specification shown in the figure defines the addition of a relation of type ChildOf between a Child element and the closest Parent element that satisfies the given criteria. The criteria is the Child is under the Parent, the Child does not have a predecessor of type Parent and the distance between the Child element and the Parent element is less than(50,50).

3.3 Generating an Application

Once a specification has been created it should be saved (using the *File* menu). This saves the actual Grand specification that the user has created. From the Class View, the GrandView menu allows one to *Write out .C/.h* or *Write out all .C/.h*. The latter produces the .C and .h files for each class and the former produces these files for a specific class (chosen by selecting the class). In general, generating an application after changing a specification requires only that the specification for any changed classes be written out. Note that it is important not

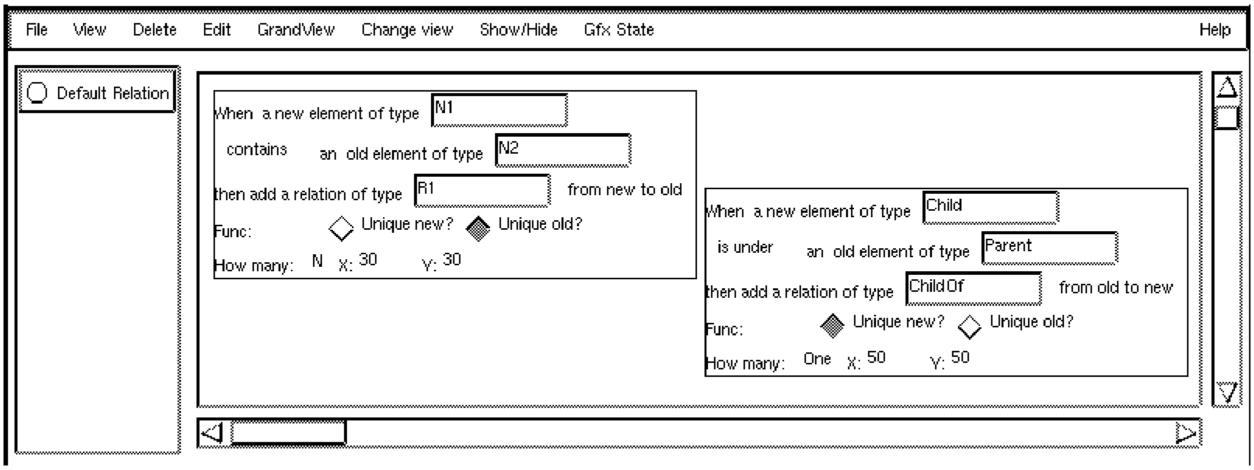


Figure 30: Default Relation Specification

to confuse the notion of saving a specification with that of *writing out* (or generating) a specification.

The first time specifications are written out, the entire application has to be generated. This is accomplished with the *Build Everything* option from the GrandView menu. The Build Everything command writes out all of the specifications and runs a set of external scripts that produce the generated editor template code. Once this process is done, an executable for the application can be produced using the **Makefile** provided for this purpose.

3.3.1 A Guide to the Generated Application Code

The previous section explains how to use GrandView to produce a visual application. In this section, we discuss the code generated by GrandView that comprises the application. We begin by discussing the various generated files and then discuss the contents of the files in detail.

The table in Figure 31 shows the files produced by GrandView. All files are produced during the Build Everything phase. The *Class Code* files can also be produced with the *Write out .C/.h* and *Write out all .C/.h* menu entries. The Action files can be produced with the *Write out menu* menu entry. Except for the Class Code files the names of all files produced are prefixed with *editor* (e.g., *editor*.C, *editor*HSEnable.h, or *editor*ActionDoLeft.h) with the exception of All*editor*.h and All*editor*.C (where *editor* is the name specified in the ETRC file or through the *Change Target Name* menu entry). The Action, HS, Ids, and MakeGraph files are incorporated into the *editor*.h and *editor*.C files through #include directives.

The generated code includes a Makefile that can be used to produce an executable for the application. The classes that realize a specific application are defined in the files *editor*.h and *editor*.C. The files *class*.h and *class*.C are the definition and implementation for each language construct class (specified in the *Class View*). The files ActionDoLeft.h, ActionDoMiddle.h, ActionEnableMenu.h, ActionEnum.h, ActionMakeMenu.h, and ActionSe-

Function	**FileName**	**Contents**
Build system	Makefile	
Editor	*editor*.h	
	editor.C	editor engine and definitions
Class code	*class*.C	class implementation
(one for each class)	*class*.h	class definition
Menu and mouse actions	ActionDoLeft.h	left mouse button
	ActionDoMiddle.h	middle mouse button
	ActionEnableMenu.h	
	ActionEnum.h	
	ActionMakeMenu.h	
	ActionSetAction2.h	editor actions
Hide/Show menu	HSEnable.h	enable HS menu items
	HSEnum.h	menu constants
	HSList.h	menu item names
	HSSetAction2.h	HS menu actions
Miscellaneous	Ids.h	Generated variables for Gfx ids
	MakeGraph.h	construct prototype graph
	All*editor*.h	include for all classes .h
	All*editor*.h	include for all classes .C
	DfltRelation.h	default relation code

Figure 31: Application Files

tAction2.h are produced from the *Define menu view*. The files HSEnable.h, HSEnum.h, HSList.h, and HSSetAction2.h implement the Show/Hide menu (produced as part of the standard application architecture). The file MakeGraph.h creates the initial set of prototype elements that appear in the palette of the default application. The default is that all visual class specifications in the Class View appear in the palette of the generated application. The file Ids.h contain integer declarations for Gfx and Location Constraint ids. The files All*editor*.h and All*editor*.C group the class definitions and implementations for inclusion into editor.C. The file *editor*DfltRelation.h contains the code generated from the Default Relation view.

Each *class*.h contains the C++ class definition, including any specified attributes or member functions (via the *Attribute View*). Each *class*.C contains the implementation of the member functions for its corresponding *class*.h. Each class definition includes the following member functions:

```
MetaDef(Node);
Node(){}
void Init();
void InitClone();
void ChangeAttribute(int attrid,void * attr,DataType type);
void* GetAttribute(int attrid,DataType & type);
OStream& PrintOn(OStream&);
IStream& ReadFrom(IStream&);
```

MetaDef(Node) is a preprocessor macro provided by ET++ (see Section 3.1). The function Init is used for object initialization for all objects of this type. For objects created via cloning, the functions InitClone and InitAfterClone are used for initialization. See Section 3.4.1 for further discussion on these methods.

The functions ChangeAttribute and GetAttribute are used for attribute mapping purposes. The functions PrintOn and ReadFrom are used for object output and input respectively. Additional member functions include the following functions for accessing individual attributes, one for each attribute (denoted by *attrname*):

```
void Set_attrname(attr_type input_arg);
attr_type  Get_attrname();
```

The file *class*.C contains the implementation for these methods (as well as other methods). As previously mentioned, all class files are included into the application code via the files All*editor*.C and All*editor*.h.

The generated editor classes contain a large number of method definitions and descriptions of those methods. The majority of these methods are commented out; they exist as hooks into the general control flow of the base editor classes. We detail the most important of these methods. For more detail on these and other methods see the generated editor code. The following methods are used in the generated editor classes to realize the default behavior of the application.

```
void         <editor>::MakeGraph(SGraphElement* & sg, ObjList* & vgraphs)
bool         <editor>View::SetAction2(Actions a,void * data)
Command*     <editor>View::DoLeftButtonDownCommand(Point,Token,int clicks)
Command*     <editor>View::DoMiddleButtonDownCommand(Point, Token, int)
void         <editor>View::DoSetupMenu(Menu*)
static Menu*<editor>View::MakeMenu(int menuId)
```

The method *editor*MakeGraph sets up the initial system architecture. This method adds a set of VGraphElements to the vgraphs argument. For each VGraphElement placed in the vgraphs list, an *editor*View is created. The vgraph attribute (see Section 2.2) of these views is set to the respective entries in the vgraphs list. By default only one VGraphElement is added to this list. The file *editor*MakeGraph.h is included in this method. In this include file are entries that create language element prototypes and add them to the VGraphElement::protos list of the elements placed in the vgraphs list. For each language element prototype in the protos list of the *editor*View::vgraph attribute an entry is created in the palette for that view. Setting up various application architectures is discussed in more detail in Sections 4.5, 4.7 and 4.1.

The method MakeMenu is called to create new menus based on the menu id parameter. DoSetupMenu is called when a menu is activated. In this method one can enable or disable menu entries and make changes to the state of menu entries. Examples of these methods are in the generated *editor*.C file.

Selection of a menu item causes control to be transfered to SetAction2 where the action is handled. Several sorts of actions can occur: first, an action may require immediate handling (e.g., hiding and showing parts of the graph, etc.); an action may require system modes to be set (e.g., delete mode); finally, an action may require further input (e.g., a point, an element). This can be set with the *EscalanteView::SetNeed()* method. This is a polymorphic method that takes the form:

```
void SetNeed(Needs n, Class * neededClass=0)
void SetNeed(Needs n,class Class * cl1,Class* cl2)
void SetNeed(Needs n,ElementOkFunc f)
```

Needs is an enumerated variable with one of the values NEED_NONE, NEED_ENTITY, NEED_RELATION, NEED_ELEMENT, NEED_E1E2, NEED_POINT and NEED_GFX. The current need is used in the DoLeftButtonDownCommand and DoMiddleButtonDownCommand methods to determine what kind of things are needed (e.g., relation, gfx, point). NEED_E1E2 is used when two elements are needed. The neededClass arguments are used to set the element class that is needed. (e.g., SetNeed(NEED_ENTITY, Meta(Node))). The cl1 and cl2 arguments are used for the NEED_E1E2. One can also define a function that is called to determine the type of element being picked.

DoLeftButtonDownCommand and DoMiddleButtonDownCommand are called when the respective mouse buttons are depressed. Like SetAction2, these functions use the current state/action to determine what action to take for a button click.

3.3.2 Creating Grid Base Applications

Escalante supports the creation of applications whose elements are based on a grid. This grid structure allows for the regular layout of the elements and for the direct access to elements based on absolute or relative positions. In this section we discuss the facilities that Escalante provides for creating grid based applications. In Sections 4.2 and 4.3 we give specific examples of grid based applications.

Grid based applications are created using the *ObjectGrid* class. This is a utility class in the src/util directory. The ObjectGrid class maintains a mapping between the position of an element and the element. Elements that are part of a grid need to be derived from the *GridElement* class, a predefined class that registers its instance elements within an instance of an ObjectGrid. This registration is based on the position of the element. The GridElement defines a virtual method, *ObjectGrid* GetGrid()*, to obtain the instance of the ObjectGrid in which it is to register. By default this method returns 0. It is up to derived classes to return the appropriate ObjectGrid pointer. The generated editor template contains code that allows one to create applications that make use of the ObjectGrid and GridElement classes. There is a global boolean variable, gDoGrid, that is defined in the generated editor module. Setting this to TRUE causes the creation of gGrid, a globally GridObject. One can then add the GetGrid method to the appropriate language classes. The default behavior of the base EscalanteView is to position all elements on a spatial grid when the gGrid has been created. To have unconstrained placement and movement of elements not registered in an ObjectGrid one can overwrite the method *bool VGraphElement::AcceptGrid()* to return FALSE.

Figure 32 shows some of the methods defined by the ObjectGrid class. These methods allow for the access of elements in various directions from a point based on the type of the element. If the Class*c argument is non-null then the method returns an element of type c (if any) at the given grid position. If Class*c is null then the first element at the given grid position is returned. The GetFirst... methods returns the first element encountered in the direction specified. Grid points that are some direction from a given point can be retrieved using the Offset... methods. The OffsetByDir method uses a set of predefined directions (e.g., DIR_RIGHT, DIR_DOWN_RIGHT, DIR_UP) to calculate the grid point with respect to a given point. The GetElementByDir method allows one to retrieve an element using the predefined directions.

3.3.3 Creating Dynamic Applications

To achieve dynamic behavior within an application (e.g., simulation) an *editor*Timer class is provided in the editor template code. This class is derived from the *TimerObject* class which registers itself to be called back every n time units. The virtual method *TimerObject::Tick()* is used within the derived class to implement application specific functionality. For an element to register itself with a timer the method, *TimerObject* GetTimer()*, is overwritten by the element, returning a pointer to a TimerObject. The generated editor template contains the code and class definitions that allow the developer to quickly create applications that

```
VGraphElement* Get[Above,Below,Right,Left...](Point p, Class*c =0)
VGraphElement* GetFirst[Above,Below,Right,Left](Point p)

Point Offset[Up,Down,Right,Left](Point o)
Point Offset[UpRight,UpLeft,DownRight,DownLeft](Point o)

Point OffsetByDir(int dir, Point p)
VGraphElement * GetElementByDir(int dir, Point origin, Class * c =0)
```

Figure 32: Object Grid Methods

make use of the TimerObject. There is a global boolean variable, gDoTimer, defined in the generated editor module. Setting this to TRUE causes the creation of g*editor*Timer, an instance of a *editor*Timer. Template code is provided in the generated module that can be changed to suit the intended purposes. Menu entries are added to the *editor* menu that allow for the starting, stopping and stepping of the timer as well as controlling the speed of the timer.

3.4 Finishing Touches

In this subsection we discuss how one can extend an application that is produced using Escalante to include application- specific functionality. There are two ways to add code to an application: first, functionality can be inserted directly into the generated language classes using the Attribute View TextInclude element. These methods may be language specific or may be instances of virtual methods defined in the base classes of the hierarchy. Second, the language developer can (and should be prepared to) modify the application code directly. The generated editor module code (*editor*.[Ch]) is specifically meant to be modified by the language developer.

In the next two subsections we discuss both of these methods for adding functionality to an editor. In the second section we discuss commonly used hooks and methods for adding code directly to an editor.

3.4.1 Support for Additional Code

For class methods generated by GrandView, there are a set of corresponding methods defined in the hierarchy that get called that have the prefix MS_. One can also use the virtual methods that are not generated to tap into the overall control flow (e.g., AddInRelation, NewHead).

This first set of member functions are for initialization. Initialization can occur at one of three times. The Init and MS_Init methods are called on the creation of a new element through the *new* method. InitClone and MS_InitClone are called on the cloning (or copying) of the element. Most of the creation of new elements within Escalante is accomplished through the cloning of previously created elements. InitAfterClone and MS_InitAfterClone are called after all elements of a set have been cloned and have had their InitClone methods called. InitAfterReading and MS_InitAfterReading are called after an element has been read in from disk.

```
void MS_Init();               Called when an object is created
void MS_InitClone();          Called when an  object is cloned
void MS_InitAfterClone();     Called after the object has been cloned
void MS_InitAfterReading();  Called after the object has been read in
```

If one adds their own attributes to a class (i.e., outside of GrandView) the attribute mapping functionality can still be used through the MS_GetAttribute and MS_ChangeAttribute methods.

```
void *  MS_GetAttribute(int attrId,DataType & type);
void    MS_ChangeAttribute(int attrId,void * attr,DataType type);
```

ResolvePtrNeeds and MS_ResolvePtrNeeds are called after all objects have been cloned and allows for the updating of any pointers to objects defined that were also cloned. Before calling this function, call the function NeedPtr(this) in MS_InitClone(). This registers the newly cloned object to be called back later.

```
void MS_ResolvePtrNeeds();
```

These member functions support input and output of user defined attributes.

```
OStream& MS_PrintOn(OStream&o);
IStream&  MS_ReadFrom(IStream&o);
```

MS_ElementsOk has two forms and is called (in VRelation and SRelation) whenever the tail or head of a relation is being set. If FALSE is returned then the action is not taken. These methods are also called when picking possible elements on the screen when setting the tail or head of a relation. MS_ElementsOk has the forms:

```
bool MS_ElementsOk(SGraphElement*tl,SGraphElement*hd)
bool MS_ElementsOk(VGraphElement*tl,VGraphElement*hd)
```

MS_OkToAddElement is called when there are multiple VGraphElements associated with a single SGraphElement. When an element is added to one vgraph that is propagated to the sgraph and a corresponding SGraphElement is added. The addition of that element is propagated to the other vgraphs. This method is called to check to see if it is okay to add a VGraphElement that corresponds to the SGraphElement. MS_OkToAddElement has the form:

```
bool MS_OkToAddElement(SGraphElement * sg)
```

There also exist a large number of virtual methods defined and used in the base classes that are not defined or used in the generated classes. One can overwrite these methods in the generated classes to accomplish some language specific functionality. Some of these methods are shown in Figure 33. (Note - the P[V,S] refers PV and PS. PVGraphElement and PSGraphElement are template based classes which the VGraphElement and SGraphElement classes are derived from.)

```
//Used by EscalanteView when picking the tail or head of a relation.
bool OkToAdd[In,Out]Relation(P[V,S]Relation * r);

//Called when adding or removing in and out relations
void Add[In,Out]Relation(P[V,S]Relation *r);
void Remove[In,Out]Relation(P[V,S]Relation *r);

//Called when the SetDead method  has been called in a incident relation
void [In,Out]RelationDead(P[V,S]Relation *, bool);

//Called when a there has been a change to a connected element
void New[Tail,Head](P[V,S]Relation*r,
                    P[V,S]GraphElement*newtl=0,
                    P[V,S]GraphElement*oldtl=0
                   );

//Called in a Relation when setting the tl or hd
bool Relation::Set[Head,Tail](P[V,S]GraphElement * h= 0);
bool Relation::SetTail(P[V,S]GraphElement * t=0);
```

Figure 33: Element and Relation Virtual Methods

3.4.2 Commonly Used Hooks

In this subsection, we provide detailed descriptions of commonly used hooks in the generated software, and also provide a few answers to commonly asked questions about adding functionality.

Iterating Through the Graph. Many times one needs to iterate through the elements which make up a graph. In Escalante there is no special graph data structure; rather, the graph is a VGraphElement that has incident relations connecting it to the members of the graph. The member attribute of the view, vgraph, is a pointer to the element that is the "graph." So the question should be how to iterate through the connected relations/elements of an element.

There are macros defined in the system that support iterating through the incident relations of an element. If you want to iterate through the elements of the graph then use the ITERATE_OUT macro on the vgraph.

```
VRelation * r; VGraphElement * elt;
ITERATE_OUT(vgraph,r,VRelation,elt,VGraphElement)
//Use elt here
//elt will be set to the head of the outgoing relation, r.
END_ITERATE_OUT
```

You can change the class types, VRelation, VGraphElement, if you want to limit your search to a particular type. e.g.

```
VRelation * r;
SomeParticularEltType * elt2;
ITERATE_OUT(vgraph,r,VRelation,elt2,SomeParticularEltType)
//only elements of type  SomeParticularEltType will get to here.
END_ITERATE_OUT
```

The ITERATE_IN and END_ITERATE_IN macros allow one to iterate through the incoming relations of an element. The ITERATE_IN_OUT and END_ITERATE_IN_OUT macros allow one to iterate through both the incoming and outgoing relations of an element.

User Defined Attribute Filters. One may want to use some function as an attribute filter that is not predefined. To do this, set the global variable gExtraFilterFunc to a function defined as follows.

```
bool func(  int funcId,
            void * & data,      DataType & type,
            Object * from,      Object * to,
            int fromid = -1,     int toid = -1
       );
```

Now you can use the set of predefined filter ids, cUserDefined[1-10], to do your own mapping. The *funcId* parameter is one of the cUserDefined ids. The parameter *data* is the incoming attribute value and should be set to the new value. The parameter *type* is the type of the incoming value and should be set to the type of the new value (e.g. eInt, eChar, etc.). The parameters *fromid* and *toid* are the attribute ids of the source and target attributes. One can use the following methods (defined in src/util/AttrMap.h) to change the incoming void*data to the *newdata* argument (e.g., data to int, data to bool).

```
bool ChangeData(void * data,DataType fromtype, int   & newdata);
bool ChangeData(void * data,DataType fromtype, bool  & newdata);
bool ChangeData(void * data,DataType fromtype, float & newdata);
bool ChangeData(void * data,DataType fromtype, double& newdata);
bool ChangeData(void * data,DataType fromtype, char* & newdata);
```

One can use the following methods to return pointers to the input data and to set the type parameter.

```
void * GetValue(double d, DataType & type);
void * GetValue(char*  d, DataType & type);
void * GetValue(int    d, DataType & type);
void * GetValue(bool   d, DataType & type);
void * GetValue(float  d, DataType & type);
e.g.:
bool b; ChangeData(data,type,b); return GetValue(b,type);
```

Mapping Values to Colors. The system supports mapping a numeric value to a color. However this mapping is fixed, i.e. the incoming value is transformed to in integer, this is taken as one of the fixed enumerated color types, (see src/gfx/CommonGfx.h). If you want to define your own mapping there is a global array, gColorMap, which is an array of enumerated color types. You can set the elements of the array to the desired colors. The value mapped to the Gfx attributes [pen, fill, text]color_map is taken to be an index into this array. See Section 4.2 for an example.

Finding Connected Elements and Incident Relations. There are a set of methods that allow one to find predecessor (tails of in relations), successor (heads of out relations) elements and incident relations based on type. They are as follows.

```
[PS,PV]GraphElement * Pred(Class *relClass=0, Class * eltClass=0)
[PS,PV]GraphElement * Succ(Class *relClass=0, Class * eltClass=0)
[PS,PV]Relation * RelPred(Class *relClass=0, Class * eltClass=0)
[PS,PV]Relation * RelSucc(Class *relClass=0, Class * eltClass=0)
```

To find an incoming relation of type E for elt one would use rel = elt.RelPred(Meta(E)). To find the head of type N of an outgoing relation of type E one would use head = elt.Succ(Meta(E),Meta(N)).

Unique Attribute Values It is often the case that an element requires some unique value for an attribute (e.g., an identifier). To accomplish this one needs to overwrite the methods MS_InitAfterClone and MS_InitAfterReading to set the attribute to some unique value. These methods are called after the two ways in which a new element is created during an application(i.e., cloning and reading from disk). For example:

```
int gNodeId=0;
void Some_Class::MS_InitAfterClone(){
      Some_Class_BASE::MS_InitAfterClone();
      my_unique_id = gNodeId++;
}
```

```
void Some_Class::MS_InitAfterReading(){
      Some_Class_BASE::MS_InitAfterReading();
      my_unique_id = gNodeId++;
}
```

Dialog Boxes. There is no explicit support provided by Escalante for dialog boxes. However, there is a class, EscalanteDialog, that provides some base functionality (a bit more than deriving directly from the ET++ class Dialog). EscalanteDialog sets up an **Ok** and a **Cancel** button. You can overwrite the method Done(int id). This method is called when the Ok or cancel button is selected. The method, VObject * GetInner(), returns the body of the Dialog. There is also a TextFieldDialog class that provides a simple text field. You can then retrieve the text with the method char* GetText(). These classes can be found in src/editor/EscalanteView.h.

4 Examples

In this section we provide several examples that are intended to illustrate various aspects of how Escalante can be used to create a visual language application. Each of these examples has been constructed using Escalante; in some cases the visual application was required for another research project, while in other cases the example is contrived to illustrate an Escalante concept. Thus the examples are intended to describe the versatility of Escalante, and to provide guidance for the visual application designer in building his/her own applications using the system.

4.1 Boolean Logic Circuit

Figure 34 shows a system that supports the creation and manipulation of boolean logic circuits. The language this application is based on consists of *AndGate, OrGate, NotGate,* and *OnOff* entities and the *Connection* relation. The user constructs a circuit using these language elements. The image of the OnOff entity consists of an ImageButtonGfx and a TextFieldGfx. The user can directly manipulate the input values of the circuit through the ImageButtonGfx of the OnOff entity. These changes are propagated by the Connection relation to the gate entities. The gates apply their respective boolean operations to their input Connection relations and propagate the result value to their output Connection relations. One can label an OnOff entity using its TextFieldGfx. The value of the label is propagated by the Connection relations to the gates which use their set of input labels to construct a textual representation of their boolean operation. Through the BooleanCircuit menu the user can turn on or off the display of the labels for all elements or for a particular element. The tail point of a Connection relation is defined to be the east point of the tail of the relation. The NotGate entity is constrained to have only one incoming Connection relation. We will now describe the Grand specification and the manual coding required to implement this application.

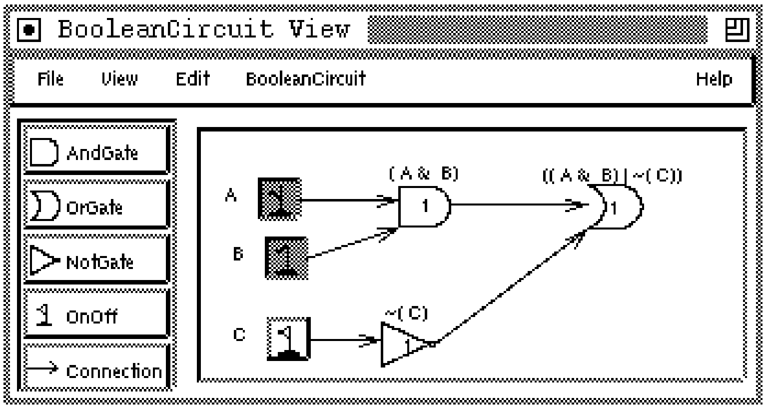


Figure 34: Circuit Application

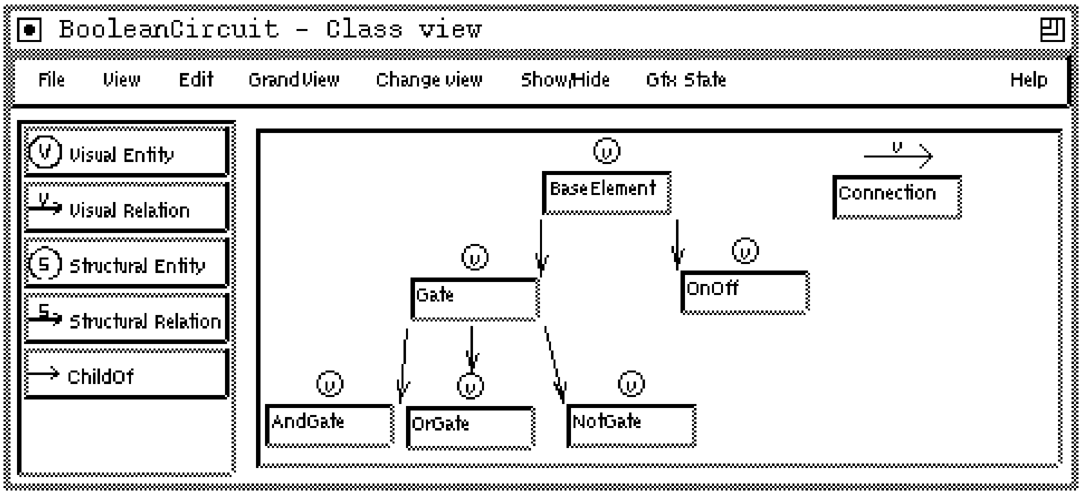


Figure 35: Circuit Specification

Figure 35 shows the Class View[7] that is used to create the BooleanCircuit application. The BaseElement class is used to encapsulate the common state and functionality of the language entities. Derived from BaseElement is the OnOff class and the Gate class. The Gate class serves to encapsulate functionality that is common to the AndGate, OrGate and NotGate classes. The Connection class is the only relation class used in this application.

The Attribute View for the BaseElement is shown in Figure 36. The *value* attribute is a boolean attribute with default value of FALSE. This is the circuit value of the element. A precondition, CheckValue, is defined for this attribute. CheckValue is a virtual method defined in the Text Include element that, by default, returns TRUE. This method is overwritten by the OrGate, AndGate and NotGate classes in order to apply their respective boolean operations on their set of input values.

Figure 37 shows the CheckValue method for the AndGate and NotGate elements. In AndGate::CheckValue the newvalue argument is an aliased boolean variable which is the input to the method BaseElement::Set_value(). The newvalue is initially set to TRUE. The in relations of type Connection with tail of type BaseElement are iterated through for the AndGate using the macro ITERATE_IN. If any of these values are FALSE then the newvalue argument is set to FALSE. This method returns whether the newvalue is not equal to the previous value because there is no need to change BaseElement::value if newvalue is the same. Since there can only be one input Connection relation to a NotGate the NotGate::CheckValue method uses the Pred method to find any predecessor BaseElement. It then sets the newvalue to the inverse of the predecessors value attribute.

The BaseElement class also contains a character attribute, *label*. This attribute can be changed through the TextFieldGfx of the OnOff entity. CheckLabel is a precondition defined for the label attribute. This method returns TRUE by default but is overwritten by the Gate class in order to create the textual representation of the boolean expression. This method is

[7]This specification is included in the release of Escalante2.3.

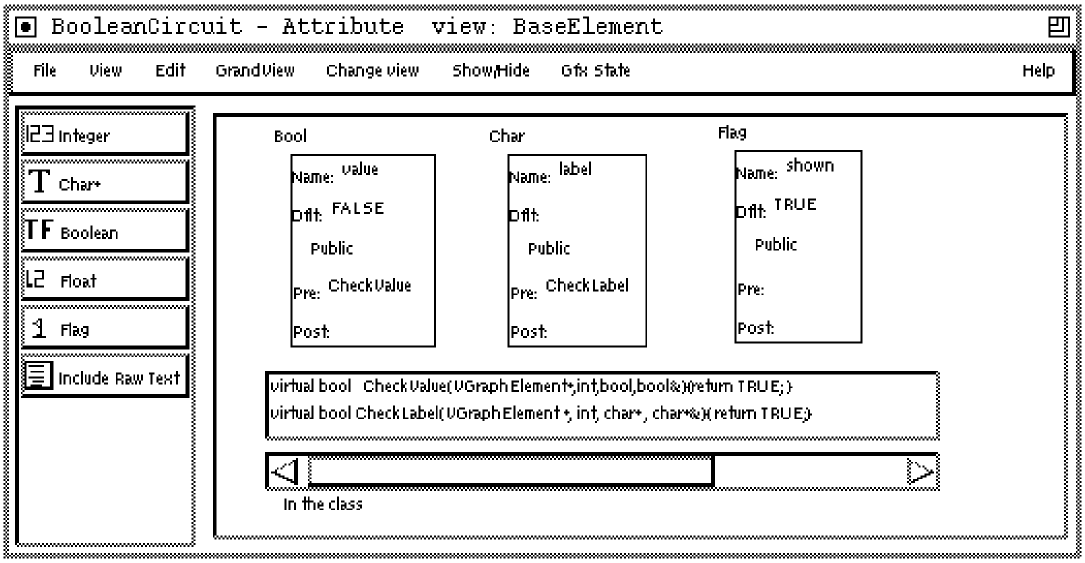


Figure 36: BaseElement Attribute View

similar to the CheckValue methods described above. In the Gate::CheckLabel method the input elements are iterated through adding their label attributes to the label attribute of the current Gate element. The *shown* attribute is a flag attribute used to turn on/off the visibility of the label of an element. When BaseElement::shown is TRUE the label is shown, when FALSE the label is not shown.

Figure 38 shows the GfxView for the Gate element. The image of a Gate element consists of a BitmapGfx and two TextGfx. The filename field of the BitmapGfx specification is normally taken as raw text which is quoted in the generated code. However, in GrandView when any field that is normally taken as raw text is prefixed with a \ then the remainder of the field is dumped out verbatim in the generated code without quoting. In the case of Figure 38 the value of the filename field is \GetIconName(). GetIconName is a virtual method defined in VGraphElement that is overwritten by the AndGate, OrGate and NotGate classes to return the name of their respective bitmaps. The left most TextGfx specification in Figure 38 is used to display the value attribute in the center of the Gate element. Note the GfxAttrMap attached to this TextGfx. This maps the BaseElement::value attribute to the TextGfx::value attribute. The TRANSLP Location Point is used to define that the TextGfx is translated to the center + (0,-1) of the bounding rectangle formed by P1 and P2. The Trans: field of the GfxBase image (not shown) for the TextGfx is set to CTR. The other TextGfx specification is used to display the label attribute at the top of the element. This is accomplished using the TRANSLP Location Point set to N. The Trans: field of the GfxBase image is set to S. For this TextGfx two attribute mappings are used. The first GfxAttrMap is used to map the BaseElement::label attribute to the value of the TextGfx. The second GfxAttrMap defines a mapping from the BaseElement::shown attribute to the Gfx::Shown

```
BaseElement * baseelt; Connection * conn;
bool AndGate::CheckValue(VGraphElement *,int,bool, bool&newvalue){
     newvalue = TRUE;
     ITERATE_IN(this,conn,Connection,baseelt,BaseElement)
          if(!baseelt->Get_value()) {newvalue = FALSE;break;}
     END_ITERATE_IN
     return (Get_value() != newvalue);}
bool NotGate::CheckValue(VGraphElement *,int,bool, bool&newvalue){
     newvalue = FALSE;
     baseelt =(BaseElement*) Pred(Meta(Connection), Meta(BaseElement));
     if(baseelt) newvalue = !baseelt->Get_value();
     return (Get_value() != newvalue); }
```

Figure 37: CheckValue Methods

attribute. The BaseElement::shown attribute is used to turn on or off the display of the label.

As shown in Figure 39, the relation attribute map mechanism is used within the Connection relation to propagate the BaseElement::value and the BaseElement::label attributes from the tail of the relation to the head of the relation. Note the mapping of the attribute GraphObject::existence to the BaseElement::label attribute from the relation to the head. The GraphObject::existence attribute represents whether a GraphObject is deleted or not. On deletion of a Connection relation the change to its existence is propagated to its head element. This triggers a call to the Gate::CheckLabel method which updates the textual representation of the boolean expression, taking into account the existential state of its input relations.

Figure 40a shows the Check Tail Head View for the Connection relation. This specification states that the tail can only be of type BaseElement and the head can only be of type Gate (i.e., one cannot have an input Connection relation to an OnOff element).

Figure 40b shows the Location Constraint view for the Connection relation. This defines that the tail point of the relation is equal to the east point of the element rectangle of the relation's tail. The Source type: field is set to Element rectangle (i.e., rectangle formed by P1 and P2) instead of the Gfx ibbox because the image of the the label attribute of the Gate and OnOff elements affects the Gfx ibbox. We want the relation to attach to the east point of the BitmapGfx which is the east point of the Element rectangle. Also, note that the Stretchy button has been set to TRUE. We do not want this constraint to affect the other points of the relation, only the tail point.

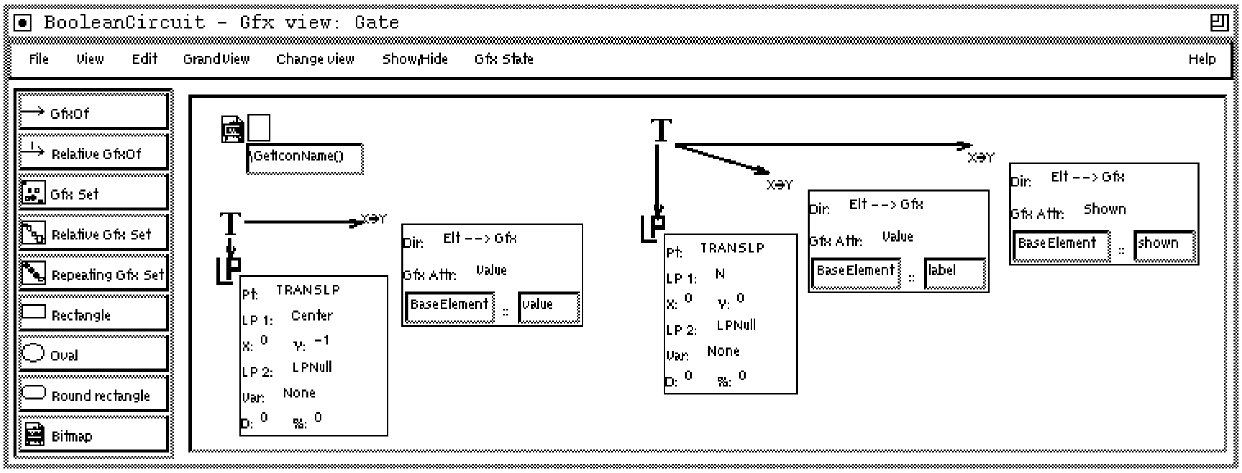


Figure 38: Gate GfxView

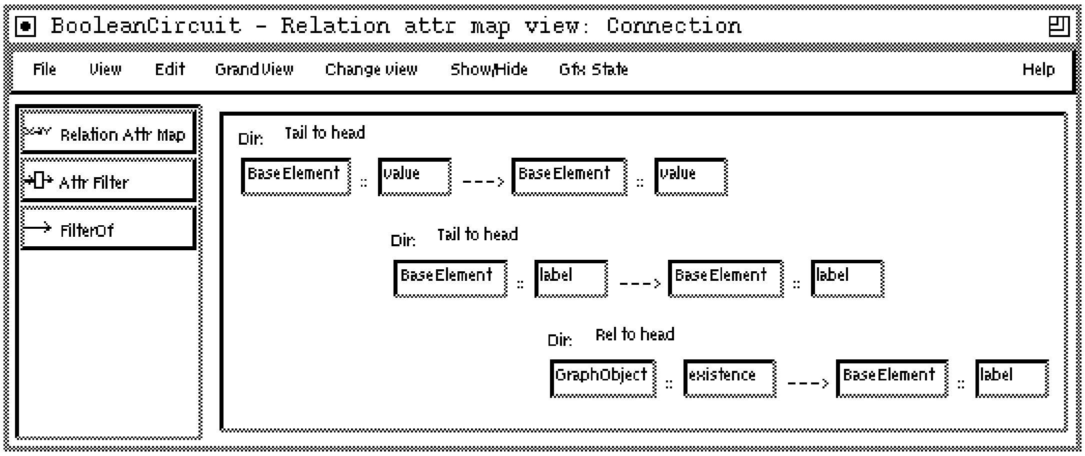
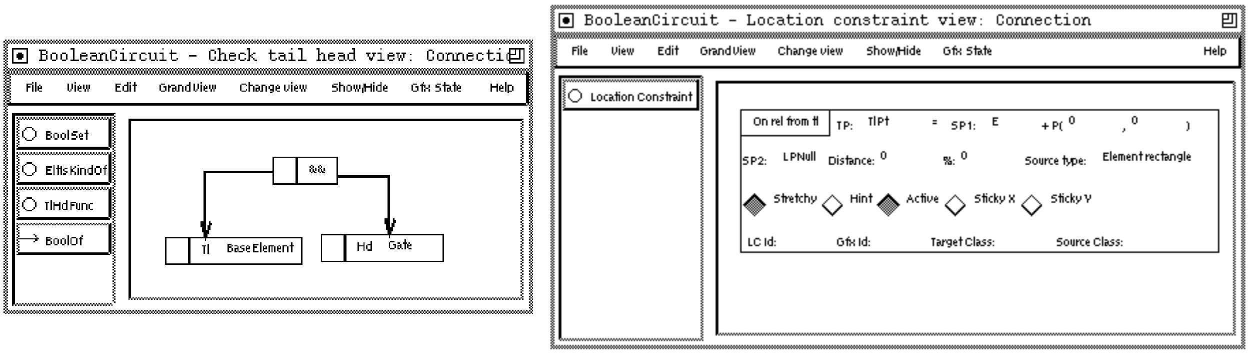


Figure 39: Connection Relation Attribute Map View

4.1.1 Modifications to BooleanCircuitView

Other than the CheckValue and CheckLabel methods described above the only other manual coding required to implement the BooleanCircuit application was to set up the initial system architecture in MakeGraph, to enforce the single input relation rule for the NotGate and to allow the user to turn on or off the label of a particular element and the labels of all elements.

The following is the manually written code in BooleanCircuit::MakeGraph that is used to overwrite the generated code in the BooleanCircuitMakeGraph.h file. The include directive has been commented out and the appropriate element prototypes have been added to the *protos* list of vgraph1 using the ADDV macro. This macro creates an instance of the first argument and adds it to the *protos* list attribute of the second argument.



(a) (b)

Figure 40: Connection Views

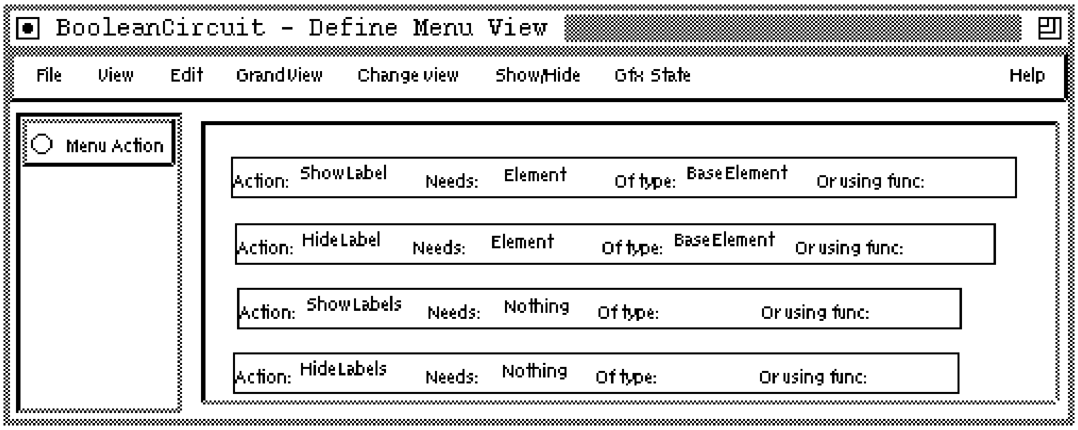
```
//#include "BooleanCircuitMakeGraph.h"
ADDV(AndGate,vgraph1)
ADDV(OrGate,vgraph1)
ADDV(NotGate,vgraph1)
ADDV(OnOff,vgraph1)
ADDV(Connection,vgraph1)
```

To enforce that a NotGate only have one incoming Connection relation we have overwritten the EscalanteView::NewHeadOk method as shown below. This method is called when a relation is being added and the head of the relation is being picked. If FALSE is returned then the candidate new head element is not chose. In this method we check if the new relation, nr, is of type Connection and the new head, newhd, is of type NotGate. If so we check if the newhd already has an incoming relation of type Connection.

```
bool BooleanCircuitView::NewHeadOk(VRelation* nr,VGraphElement *newhd){
if(nr && newhd && newhd->IsKindOf(NotGate) && nr -> IsKindOf(Connection))
        return  !(newhd->RelPred(Meta(Connection)));
return BooleanCircuitView_BASE::NewHeadOk(nr,newhd);
}
```

The Define Menu View was used to create the framework to turn on and off the display of labels of BaseElement elements. Figure 41a shows the Define Menu View specification used. We have defined four menu entries - ShowLabel, HideLabel, ShowLabels, HideLabels. The ShowLabel and HideLabel commands require an element of type BaseElement. The HideLabels and ShowLabels commands require nothing. These commands act on all of the elements of the graph.

Figure 41b shows the code from the BooleanCircuitActionSetAction2.h file that realizes the ShowLabels command. First the EscalanteView attribute vgraph is iterated through, setting to TRUE all of the BaseElement::shown attributes (causing the Gfx to be turned on).



(a) Define Menu View

```
case eShowLabels: {
      BaseElement * elt; VRelation * rel;
      ITERATE_OUT (vgraph, rel, VRelation, elt, BaseElement)
            if(!elt->Get_shown())       elt->Set_shown(TRUE);
      END_ITERATE_OUT
      Iter next(vgraph->protos);      //Check the protos list
      while(elt = (BaseElement*)next())
            if(elt->IsKindOf(BaseElement)) elt->Set_shown(TRUE);
}        break;
```

(b) BooleanCircuitActionSetAction2.h code

Figure 41: Defining BooleanCircuit Menus

Next the protos list of the vgraph is iterated through, changing the value of the BaseElement::shown attribute for any prototypes derived from BaseElement. The implementation of the HideLabel and ShowLabel commands consists of inserting one line of code for each command in the BooleanCircuitActionDoLeft.h file (e.g., ptr->Set_shown(TRUE or FALSE);).

4.2 WaterWorks

The WaterWorks system, shown in figure 42, allows the user to create a dynamic system of water sources, pipes, sinks, water drops and water vapor. All of the elements have an amount of water that can be set directly. The amount of water in a pipe is mapped to the color of the pipe. The amount of water in a droplet or in vapor is shown directly as numeric text. Pipes and buckets also have a capacity. When the amount of water in a pipe reaches the pipe's capacity the color of the pipe changes to red and water begins backing up in the upstream pipes. When a bucket reaches capacity water flows out of the bucket.

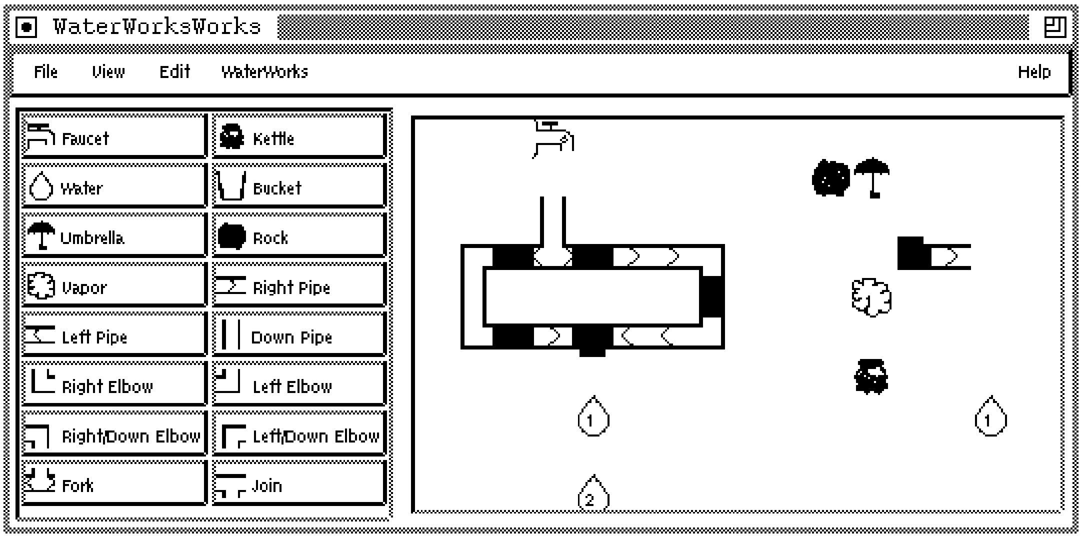


Figure 42: WaterWorks Example

When animating a WaterWorks system water drops fall until they hit some other element. If that element can accept water then the drop flows into the element. If the element cannot accept water then the water drop turns into water vapor. Water vapor behaves like a water drop except water vapor rises. On encountering an obstacle vapor condenses and turns into a drop. Faucets produce water drops with a certain amount at a certain rate. Kettles produce water vapor. An umbrella catches both water drops and water vapor and splashes them to either side. Rocks do not accept any water.

WaterWorks is a grid based system as discussed in Section 3.3.2. Figure 43 shows the specification of the classes which make up the WaterWorks system. The elements of a WaterWorks program are all derived from the GridElement class. The *BaseElement* class overwrites the GetGrid and GetTimer methods to return the globally defined pointers, *gGrid*

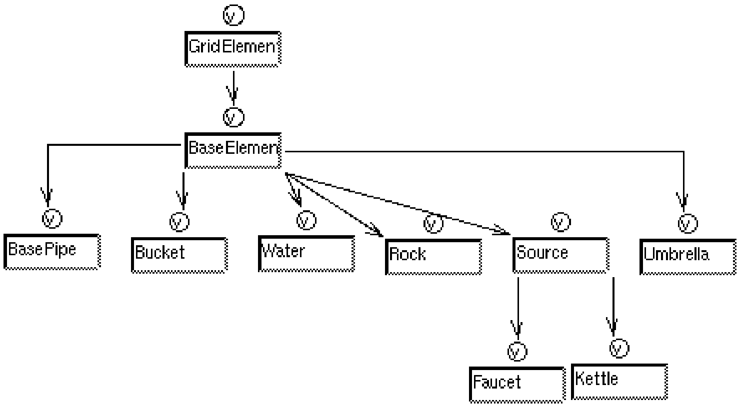


Figure 43: WaterWorks Class Specification

```
class RightPipe :public BasePipe{
int           AcceptFromLeft();
char*         GetOutlineIcon(){return "rpipe.im";}
char*         GetFillIcon(){return "rpipe2.im";}
PipeDirection GetDirection(){return eDirRight;}};
class DownPipe :public BasePipe{
int      AcceptFromTop();
char*    GetOutlineIcon(){return "dpipe.im";}
char*    GetFillIcon(){return "dpipe2.im";}
PipeDirection GetDirection(){return eDirDown;}};
```

Figure 44: Derived Pipe Classes

```
gColorMap[1] = eCyan;          gColorMap[2]  = eSkyBlueLight;
gColorMap[3] = eSkyBlueDeep;   gColorMap[4]  = eCornflower;
gColorMap[5] = eBlue;          gColorMap[6]  = eSlateBlueLight;
gColorMap[7] = eSlateBlue;     gColorMap[8]  = eCobalt;
gColorMap[9] = eMidnightBlue;  gColorMap[10] = eUltramarineViolet;
```

Figure 45: WaterWorks Color Map

and *gWaterWorksTimer* as described in Sections 3.3.3 and 3.3.2. The gWaterWorks timer drives the animation of a WaterWorks system.

The *Water* element has a boolean attribute, *liquid*, that determines the state of the water (i.e., vapor or drop). The Water::liquid attribute is used to display the appropriate bitmap and to determine the behavior of the element during animation.

The *BasePipe* class is the parent class of a set of derived pipe elements (e.g., RightPipe, DownPipe). The derived pipe element classes were manually coded. Figure 44 shows the manually coded class definitions for the RightPipe and the DownPipe classes. The images of the derived pipe elements are defined using two BitmapGfx in the BasePipe class. These BitmapGfx show the outline of the pipe and the water in the pipe. The bitmap filenames for these two BitmapGfx are obtained through the virtual methods GetOutlineIcon and GetFillIcon as shown in Figure 44. The amount of water in a pipe is an integer attribute and is mapped to the shown attribute of the BitmapGfx that represents the filled image of a pipe (i.e. amount != 0 implies show the filled bitmap). The amount attribute is also mapped to the Fill color_map attribute of this BitmapGfx. Figure 45 shows how the gColorMap was set. Higher index numbers are mapped to successively darker shades of blue. The animation behavior for most of the derived pipe elements is defined in the BasePipe class. This class uses the method GetDirection to determine in which direction the water flows out of the pipe.

The methods AcceptFrom[Left,Right,Bottom,Top] are virtual methods that are defined in the BaseElement class that return how much water can be accepted by an element from the direction specified.

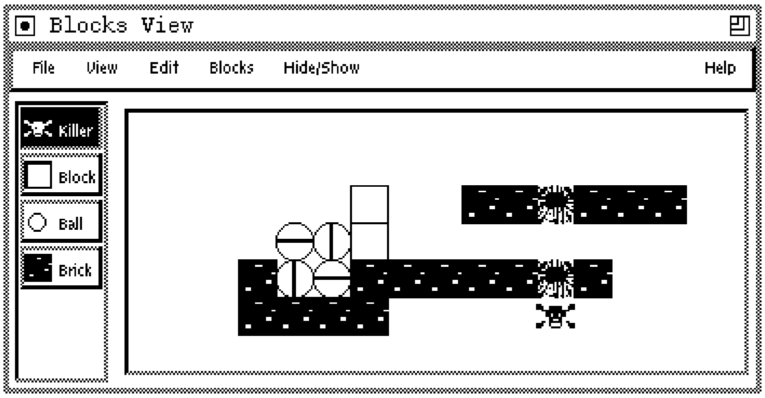


Figure 46: Blocks

4.3 Blocks

The Blocks application, shown in Figure 46, consists of a set of elements derived from the GridElement class described in Section 3.3.2. The *Killer, Block* and *Ball* elements are registered with a TimerObject. The *Brick* element is not registered with the timer. In this application the elements registered with the timer drop 1 level down each timer tick. Blocks will stop dropping when there is another element under the Block. Balls will attempt to roll to the right or left if there is an element below the Ball and if there is no element to the right or left. The Killer elements destroy whatever is in their path, leaving a Splat element behind.

```
void Killer::Tick(){
Point p = GetOrigin();         if(p.y > 1000)  return;
if(justEaten){
      VGraphElement * vgraph = Pred(Meta(VMemberOf));
      Splat * d =(Splat*) CloneElt(gSplatProto,vgraph,gPoint0);
      d->SetOrigin(p);                justEaten = FALSE; }
if(elt = gGrid->GetBelow(p) && elt != this){
            justEaten = TRUE;KillElement(elt);}}
SetOrigin(gGrid->OffsetDown(p)); }
```

Figure 47: Blocks Example

Figure 47 shows some example code used to implement the behavior of the Killer element. The method Tick is called by the BlocksTimer at every clock tick. In the Killer::Tick method the current position of the element is found using the GetOrigin method. Next, if the Killer element had previously killed an element then the Killer element retrieves the graph it is part of and calls the procedure CloneElt with its graph and the gSplatProto, a predefined instance of a Splat. CloneElt clones the gSplatProto and adds it to the vgraph. The Killer element then checks to see what is below it using the ObjectGrid::GetBelow method. If something is below it the Killer element deletes that element with the predefined procedure KillElement procedure. The Killer element then sets its origin to the grid point below its current position.

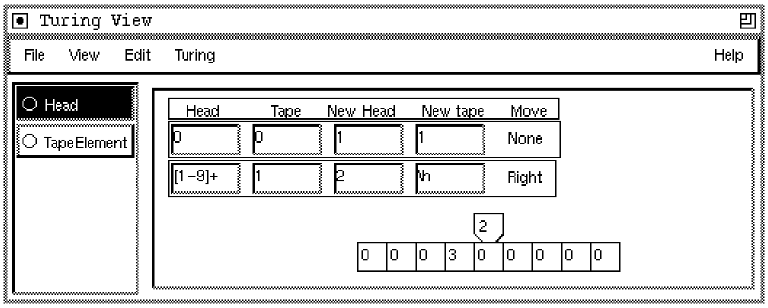


Figure 48: Table Based Turing Machine

4.4 Turing Machine

We have developed an application, shown in figure 48, that allows the user to graphically construct and run a Turing machine. One creates the machine tape with a series of *Tape* objects and places a *Head* object over the tape. Both the Tape and the Head have a text field in which one can directly change the state of the element. One can define the rules which govern the activity of the Head in two ways. The first method is a table based approach as shown in Figure 48. One can add any number of rules to the table. These rules consist of a Head and Tape state, new Head and Tape state and a movement specification. The specification of the Head and Tape state is an arbitrary text string which is used as a regular expression in the process of matching the active Head to a rule.

The second method of defining rules, using *finite state machines*, is shown in Figure 49. A set of nodes and edges are used to define the rules in this method. The value of a node represents the state of the Head. The edges between nodes define the input Tape value, the changes to the Tape and the movement of the head.

The table shown in figure 48 is built with a *TableHeader* and *TableEntry* elements. The group mechanism discussed in Section 3.2.10 is used to define the layout of the table. Figure 50 shows the group specification for the TableHeader element. This grouping states that when a relation of type *EntryOf* with head of type TableEntry is added to a TableHeader element then add the head to the group. A relation of type *NextEntry* is used to connect

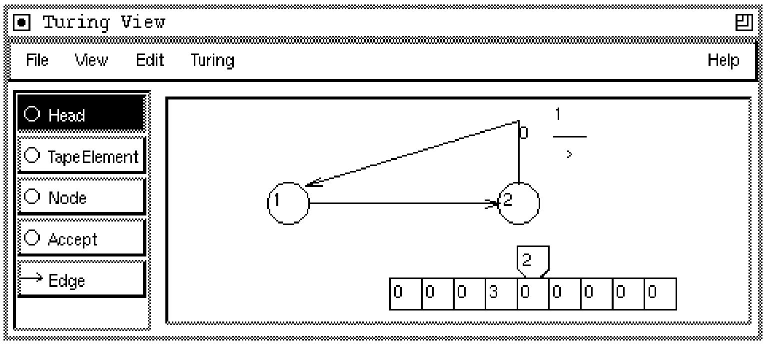


Figure 49: Graph Based FSM Turing Machine

consecutive members of the group. A relation of type *FirstEntry* is used to connect the TableHeader element to the first TableEntry. The NextEntry relation defines a location constraint that places the NW corner of the head of the relation at the SW corner of the tail. The FirstEntry relation uses a similar location constraint.

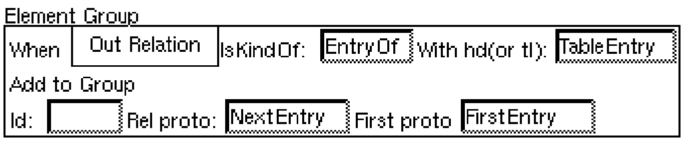


Figure 50: Turing Machine Table Specification

```
case  Switch:
gDoGraph = !gDoGraph; drawfunc = TuringFilter; MakeToolList();
action = ACTION_NONE; ForceRedraw(); break;
```

(a) TuringView SetAction2

```
bool TuringFilter(VGraphElement * elt){
if(gDoGraph) return (elt->IsKindOf(Node) || elt->IsKindOf(Edge) ||
                    elt->IsKindOf(TapeElement) ||   elt->IsKindOf(Head));
return (elt->IsKindOf(BaseRule) || elt->IsKindOf(TableHeader) ||
        elt->IsKindOf(TapeEntry) ||   elt->IsKindOf(Head)); }
```

(b) TuringFilter

Figure 51: Turing Machine Switch Mode Code

The ability to switch between the tabular view and the FSM view is accomplished using the Switch Mode entry in the Turing menu. As shown in Figure 51a when this command has been called the global variable that defines which type of representation is used is toggled. The method EscalanteView::MakeToolList is then called which causes the base view to rebuild the palette using the list of prototypes returned from a call to GetProtoList, a virtual method defined in EscalanteView. The TuringView class overwrites the GetProtoList method to return one of two prototype lists, depending on which view is active. The first list contains prototypes for the tabular view. The second list contains prototypes for the FSM view. To filter out the other elements from display the EscalanteView::drawfunc attribute is set to the procedure TuringFilter shown in Figure 51b.

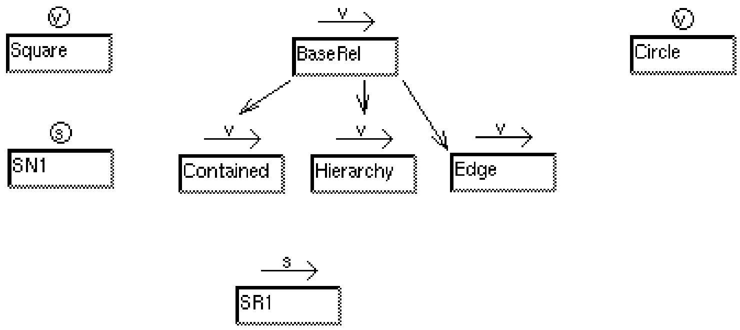


Figure 52: MView Class Specification

4.5 A Multi-Representation Application

In Section 2.1.5 we discussed the basic uses of the structural graph element classes. In this section we will provide further details as to the specification and construction of the application shown in Figure 7.

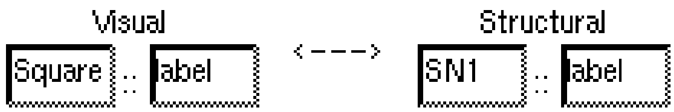


Figure 53: SxV Attribute Mapping

Figure 52 shows the class specification of the MView application discussed in Section 2.1.5. The Square entity has a character attribute, label. The graphical representation of a Square is a rounded rectangle and a text field. There is a bidirectional attribute mapping between the Square::label attribute and the SN1::label attribute as shown in Figure 53. The Contained relation uses the location constraint mechanism to define containment of the head of the relation by the tail. The Hierarchy relation uses location constraints to define that

```
void MView::MakeGraph(SGraphElement* & sg, ObjList* & vgs){
gVGraph1 = new VGraphElement();    gVGraph1->Init();
gVGraph2 = new VGraphElement();    gVGraph2->Init();
sg = new SGraphElement();          sg->Init();
sg->AddVGelt(gVGraph1);            sg->AddVGelt(gVGraph2);
gVGraph1->SetSGelt(sg);            gVGraph2->SetSGelt(sg);
vgs->Add(gVGraph1);                vgs->Add(gVGraph2);
ADDSV(SN1,Square,gVGraph1)         ADDV(Circle,gVGraph1)
ADDSV(SR1,Contained,gVGraph1)      ADDV(Edge,gVGraph1)
ADDSV(SN1,Square,gVGraph2)         ADDSV(SR1,Hierarchy,gVGraph2)
```

Figure 54: MView::MakeGraph

the tail of the relation horizontally spans the head of the relation. The Y coordinate of the top of head of the relation equals the 15 + the Y coordinate of the bottom of the relations tail.

Figure 54 is the code in the MView::MakeGraph method that realizes this particular system configuration. Two VGraphElements are created, gVGraph1 and gVGraph2. These are placed in the vgs list resulting in the initial creation of two windows. A structural graph element is created and serves to bind the two visual graphs together. The structural and visual elements are related with SGraphElement::AddVGelt and VGraphElement::SetSGelt methods. The macro ADDSV(sclass,vclass, vgraph) creates an instance of sclass and vclass, relates them to on another and places the vclass instance in the protos list of vgraph. The macro ADDV(vclass, vgraph) creates an instance of vclass and places it in the protos list of vgraph.

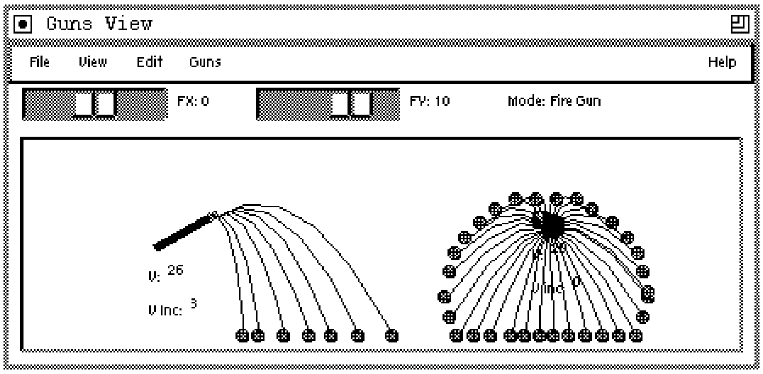


Figure 55: Guns and Bombs Application

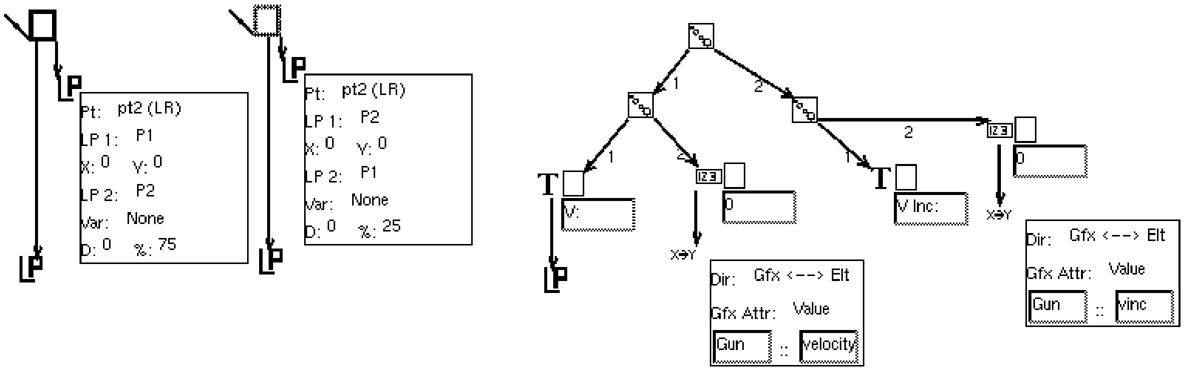


Figure 56: Gun Specification

```
VObject * GunsView::GetTopVObj(){
//double the range     and offset the actual value
s1 = new Slider(cIdAX,eHor,TRUE);   s1->SetMax(2*MAXA,0);
s1->SetVal(Point(MAXA+gAx,0));      s1->SetFlag(eVObjVFixed,TRUE);
... Do the same for Slider s2.
Filler*fill1=new Filler(Point(8,0));
fill1->SetFlag(eVObjHFixed|eVObjVFixed,TRUE);
...Do the same for Filler fill2
return new HExpander(Point(0,0),fill1,s1,f1,s2,f2,fill2,info,0);
}
```

Figure 57: GunsView GetTopVObj Method

4.6 Guns and Bombs

The Guns and Bombs application is shown in Figure 55. This system lets the user add a number of *Gun* and *Bomb* elements. Both of these elements have a velocity and a velocity increment field. On the *firing* of a Gun a *Bullet* element is created and given as initial velocity the velocity of the Gun. The velocity of the gun is incremented by the velocity increment on every firing. The initial velocity of a Bullet has an X and Y component which is derived from the angle of the Gun. Bullet elements are registered with a timer object. For every tick of the timer a Bullet recalculates its position based on its initial velocity and a global X and Y force. A *Trail* relation is used to draw the trail the Bullet makes between the Gun and the Bullet. On firing a Bomb a set of Bullets are created in a circular pattern with velocities based on the initial velocity of the Bomb and the initial trajectory of the Bullets. This creates a fireworks pattern of Bullets about the Bomb element.

```
void GunsManager::Control(int id, int part, void *val){
if(part == eSliderThumb && id == cIdAX){
// Get the value and adjust from 0->2*MAXA to -MAXA -> MAXA
            gAx = (float)*((int*)val) - MAXA;
            ForceChanged(); //Update the views
... Do the same for the Y component Slider
GunsManager_BASE::Control(id,part,val); }
void GunsManager::ForceChanged(){
      Iter next(MakeIterator()); //Iterate through all of the documents
      GunsDocument * d;      GunsView * gv;
      while(d = (GunsDocument*) next())
            if(d->IsKindOf(GunsDocument) && d->views){
                  Iter next2(d->views); //Iterate through all of the views
                  while(gv = (GunsView*)next2())
                        if(gv->IsKindOf(GunsView))  gv->ChangeForce();
}}
void GunsView::ChangeForce(){
      if(f1) f1->SetString(form(" FX: %d ",(int)gAx),TRUE);
      if(s1) s1->SetVal(Point((int)(MAXA + gAx),0),TRUE);
...Do the same for f2 and s2
```

Figure 58: GunsManager Control Method

The specification of the image of a Gun is shown in Figure 56. The barrel of the gun is defined with two LineGfx one of which denotes the end of the barrel with a different color. The Location Points used are as follows:

```
Line1:        pt1 = P1; pt2 = 75%(P1->P2)
Line2:        pt1 = P2; pt2 = 25%(P2->P1)
```

The GunsView is built without the default element palette. This is accomplished by setting to false the makePalette argument of the GunsView constructor. The two sliders at the top of the GunsView are created by overriding the EscalanteView::GetTopVObj() method as shown in Figure 57. Two Sliders are created with ids cIdAX and cIdAY. These sliders, s1 and s2, are attributes of the GunsView class. These sliders control the x and y components of the global acceleration force. Because the range of values for a Slider are positive we double the range and then offset the actual values (which may be negative). The variables f1, f2 are attributes of the GunsView class and are used to textually display the values of the x and y components of the global acceleration force. The variable info is an attribute of EscalanteView class and is used to give command feedback during editing.

Changes to the slider are caught in the GunsManager::Control method as shown in Figure 58. The change to a slider in a view is passed to the GunsDocument::Control method which

```
void Gun::Fire{
//Clone the bullet and add it to  my graph
      VGraphElement * mygraph = Pred(Meta(VMemberOf));
      Bullet * b = (Bullet*) CloneElt(gBulletProto,mygraph,gPoint0);
      if(gDoTrails){
            Trail * t = (Trail*) CloneElt(gTrailProto,mygraph,gPoint0);
            t->SetTailHead(this,b); }
...Set bullet's velocity, etc. }
```

Figure 59: Gun::Fire Method

passes it on to the GunsManager::Control method. In this method it is determined which slider caused the activity and the global force variable is set to the new value of the slider. The method GunsManager::ForceChanged is called. This method iterates through all of the GunsView objects contained by all of the GunsDocument objects in the application. The method GunsView::ChangeForce is called which updates the textual and slider representation of the global forces.

Guns and Bombs are added through a menu command. The following is the code from GunsView::SetAction2 that allows for the creation of Bombs through a menu command (Same functionality for a Gun).

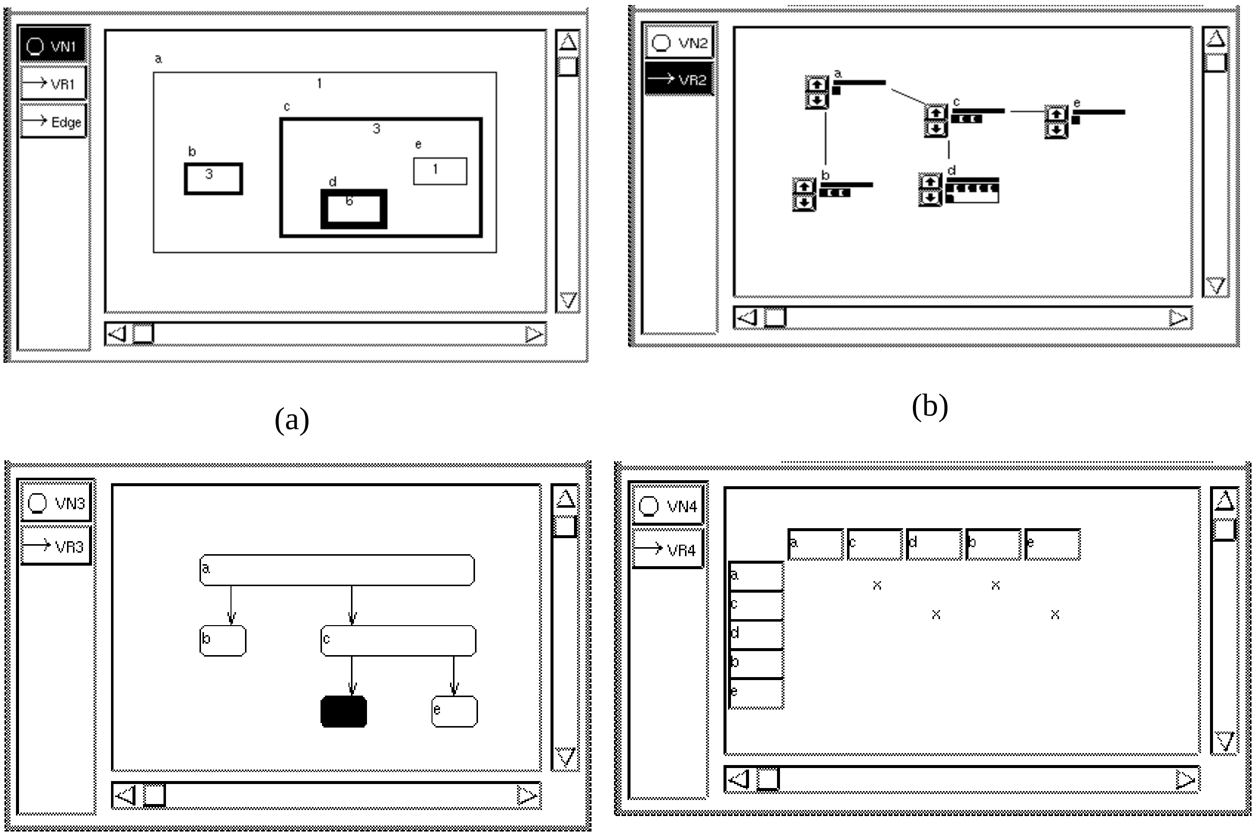
```
case AddBomb:
if(gBombProto == 0){
//Don't add the proto to the global set of elements
Set dummy; SetCurrentVGraphElementSet(&dummy);
gBombProto = new Bomb(); gBombProto->Init();
SetCurrentVGraphElementSet(0);
}
SetTool(gBombProto);
break;
```

Figure 59 shows the Gun::Fire method. When a Gun fires it creates a Bullet and a Trail and adds them to its graph as shown in the Figure.

4.7 Another Multi-Representation Application

Figure 60 shows an example application that makes use of the multiple representation functionality discussed in Section 2.1.5. This example application is made up of a set of four groups of visual elements related to a group of structural elements. Figure 61 shows the Class View of the specification for this application.

The window shown in Figure 60a is made up of visual entities of type VN1 and visual relations of type VR1 and Edge. The relation VR1 defines the spatial containment of the



(a) (b) (c) (d)

Figure 60: Multiple Representations

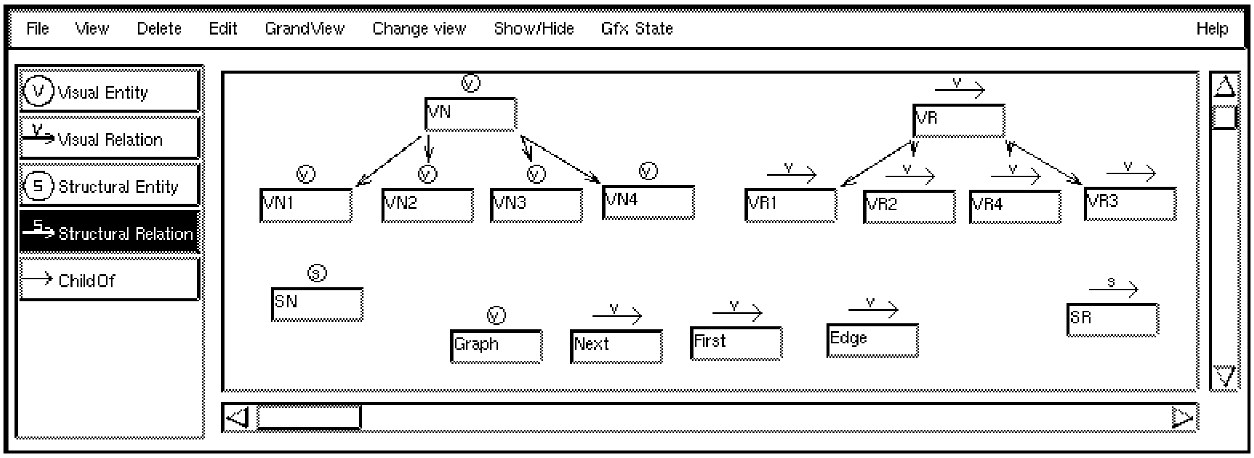


Figure 61: Multiple Representations Class View

head of the relation by the tail. In Figure 60b there are VN2 and VR2 elements. The VR2 relations are shown as an edge. Figure 60c contains VN3 and VR3 elements. The VR3 relations use location constraints to define a hierarchical layout of the VN3 entities.

Figure 60d contains VN4 and VR4 elements. The VN4 entities are displayed using two TextFieldGfx. They are laid out using the grouping mechanism described in Section 3.2.10. We make use of the VEntity class *Graph*, defined in the specification, as the vgraph for this window. One can use any visual element type as the vgraph for a view. The code that realizes this architecture is shown in Figure 62.

The Graph class uses the grouping mechanism described in Section 3.2.10 to define the layout of the VN4 entities. The specification used is shown in Figure 63. The *First* relation defines the position of the initial VN4 element. The *Next* relation defines the position of the successive elements. When a VN4 element is added to the graph First and Next relations are added as defined in the group specification. The VR4 class uses the location constraint specifications shown in Figure 64 to position its point P1. An “x” is drawn at the point P1.

```
void MView2::MakeGraph(SGraphElement* & sg, ObjList* & vgs){
... Create gVGraph1, gVGraph2 and gVGraph3
//Here we use the Graph class as the vgraph
gVGraph4 = new Graph();         gVGraph4->Init();
...Connect the vgraphs to the sgraph and
...Add the vgraphs to the vgs list and create the prototypes
ADDSV(SN,VN1,gVGraph1) ADDSV(SR,VR1,gVGraph1)
ADDSV(SN,VN2,gVGraph2) ADDSV(SR,VR2,gVGraph2)
ADDSV(SN,VN3,gVGraph3) ADDSV(SR,VR3,gVGraph3)
ADDSV(SN,VN4,gVGraph4) ADDSV(SR,VR4,gVGraph4)
ADDV(Edge,gVGraph1)
```

Figure 62: MView2::MakeGraph

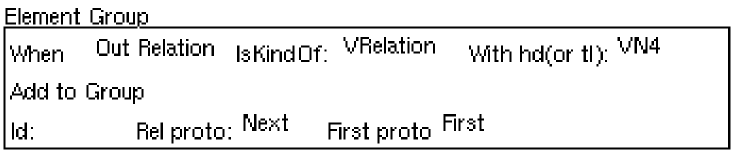


Figure 63: Graph Grouping

Both the VN and the SN classes have an integer attribute x. The attribute mapping mechanism is used to bidirectionally map the VN:x and the SN:x attributes. Changes to VN:x in a visual element are mapped to the corresponding structural elements SN:x attribute.

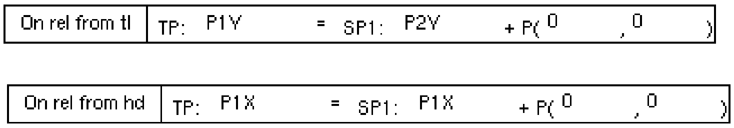
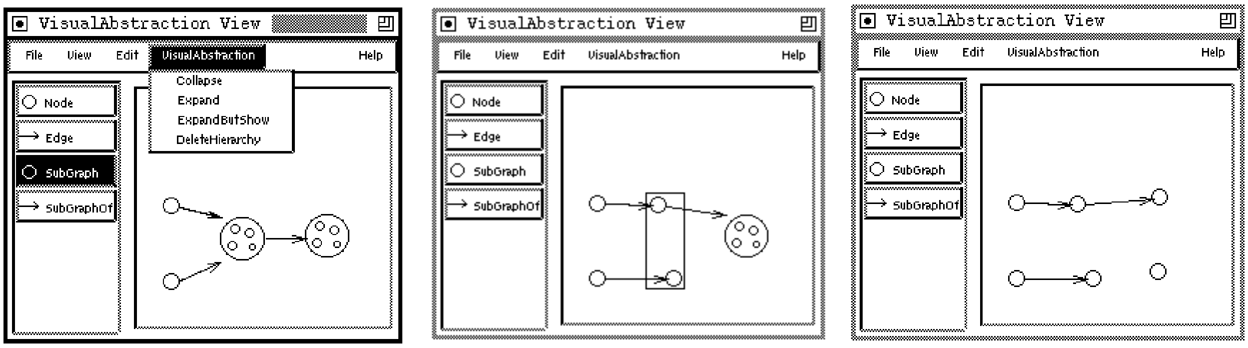


Figure 64: VR4 Location Constraints

This change is then mapped to the set of other visual elements. Each of the visual element classes, VN1,VN2 and VN3 map the value of the attribute VN::x to various aspects of the graphics which define the representation of the element. In the case of VN1, VN::x is accessed through an IntFieldGfx. The value of x is also mapped to the pen width of the rectangle. For the VN2 class VN::x is mapped to the number of OvalGfx displayed. In VN2 the attribute is manipulated through the IncDecGfx button. The VN3 class maps VN::x to the filled state of the rounded rectangle Gfx. If VN::x is greater than 5 then the Gfx is filled. VN3 does not have any way of directly manipulating VN:x. There is also a character attribute, label, in the VN and SN classes. This is accessed in each of the derived VN classes through a TextFieldGfx and is bidirectionally mapped between the VN and SN elements.



(a) (b) (c)

Figure 65: Visual Abstraction Example

4.8 A Visual Abstraction Hierarchy

Figure 65 shows a series of three screen dumps of an example application that implements a visual abstraction hierarchy. Each of these images displays a different abstraction state of the same set of elements. In this application there are two entity classes, *Node* and *SubGraph* and two relation classes *Edge* and *SubGraphOf*. The SubGraph and SubGraphOf elements implement the actual visual abstraction hierarchy. An area of the screen is swept out when adding a SubGraph. Elements contained in that area that are not part of a SubGraph (i.e., do not have an incoming SubGraphOf Relation) are added to the newly created SubGraph element using the SubGraphOf relation. The Shown Flag event (see Figure 22) is set to

```
SubGraphOf * sof; VGraphElement * elt; ... Find element ptr
if(action == eCollapse){
      SubGraph * sg = (SubGraph*)ptr->Pred(Meta(SubGraphOf));
      sg->Set_rectShown(FALSE);                            //Show bitmap
      CollapseSubgraph(sg);
      Point ctr = sg->AsRect().Center();                   //Get center
      sg->SetP1P2(ctr - Point(15,15), ctr + Point(15,15)); //Shrink subgraph
}
if(action == eExpand) {
      ITERATE_OUT(ptr,sof,SubGraphOf,elt, VGraphElement)
            //Turn off the SubGraph and turn on the element
            sof->SetEventDep(eShownFlag,eHdToTl,FALSE);
            sof->SetEventDep(eShownFlag,eTlToHd,TRUE);
      END_ITERATE_OUT   }
if(action == eExpandButShow){
      ITERATE_OUT(ptr,sof,SubGraphOf,elt, VGraphElement)
            SetLocConstraint(sof,TRUE);                //Turn on lcs
            ptr->Set_rectShown(TRUE);                //Show rectangle
            sof->SetEventDep(eShownFlag,eTlToHd,TRUE); //Turn on element
      END_ITERATE_OUT }
if(action == eDeleteHierarchy){
      action = DELETE_ENTITY;  SetNeed(NEED_ENTITY,Meta(SubGraph));
      gDoDieHints = TRUE;    //Propagate the  die hint
      Command *c = VisualAbstractionView_BASE::
                   DoLeftButtonDownCommand(p,t,clicks);
      gDoDieHints = FALSE;       return c;}
```

Figure 66: Implementation of Expand, Collapse, etc.

FALSE in the SubGraphOf relation to control the visibility of the elements of a SubGraph so that initially only one "level" of the abstraction hierarchy is visible. The sweeping out of an area on the creation of the SubGraph is caused by setting a flag through the method call VGraphElement::NeedJoints(TRUE). When set, this causes the interface to behave similar to when adding a relation. The Default Relations View of GrandView was used to define that when adding the SubGraph any contained elements are added to the SubGraph.

The SubGraph element has two images, a rectangle and bitmap. The visibility of these images is controlled through an attribute, *rectShown*. When initially adding a SubGraph the rectangle is shown. After adding the element the rectangle is turned off and the bitmap is turned on.

There are various ways to view the hierarchy the SubGraph and SubGraphOf elements form. The first is to elide from view the elements of a SubGraph and show the SubGraph's

```
void CollapseSubgraph(SubGraph * sg, Set * s =0){
      ITERATE_OUT(sg,sof,SubGraphOf,elt, VGraphElement)
            SetLocConstraint(sof,FALSE);
            sof->SetEventDep(eShownFlag,eTlToHd,FALSE);
            sof->SetEventDep(eShownFlag,eHdToTl,TRUE);
            if(elt->IsKindOf(SubGraph) && !s->Contains(elt)){
                  s->Add(elt);
                  CollapseSubgraph((SubGraph*)elt,s);}
      END_ITERATE_OUT }
void SetLocConstraint(VRelation * rel, bool state){
      LocConstraint * lc; ObjList *lclist = rel->GetLCList(ONTLIX);
      if (lclist){
            Iter lciter(lclist);
            while(lc = (LocConstraint*)lciter()) lc->SetActive(state);
}}
```

Figure 67: Other Procedures

bitmap image as shown in Figure 65a. The incident Edge relations of an element (shown as a directed edge) that has been elided from view connect up to the first shown parent SubGraph of the elided element. Another way to display the hierarchy is shown in Figure 65b. In this method both the SubGraph and the elements of the SubGraph are shown. The SubGraph's rectangle image is shown and the SubGraph contains the members of the SubGraph. One can also elide from view the SubGraph and only show the elements of the SubGraph as shown in Figure 65c.

Four commands have been defined using the Define Menu View of GrandView that implement functionality pertaining to the abstraction hierarchy. The VisualAbstraction menu that contains these commands is shown in Figure 65a. The implementation of these commands (from the ActionDoLeft.h file) s shown in Figures 66 and 67. The *Collapse* command takes an element and finds its parent SubGraph (if any). The rectangle image of the Subgraph is turned off and the visibility of the elements of the SubGraph is turned off with the *CollapseSubgraph* procedure. The implementation of this procedure is shown in Figure 67. This procedure recursively turns off the visibility of the elements of the SubGraph and the active flag of the Location Constraints of the SubGraphOf relations. The SubGraph is then reduced in size with the SetP1P2 method. The result of the *Expand* command is to elide from view the SubGraph and show the elements of the SubGraph. The *ExpandButShow* command turns on the visibility of the elements of the SubGraph and shows the rectangle image of the SubGraph. The Location Constraints in the SubGraphOf relations are turned on with the *SetLocConstraint* procedure. This procedure, shown in Figure 67, gets the list of Location Constraints from the relation using the GetLCList method. It then iterates through the list turning on or off the active flag of the constraint. The *DeleteHierarchy*

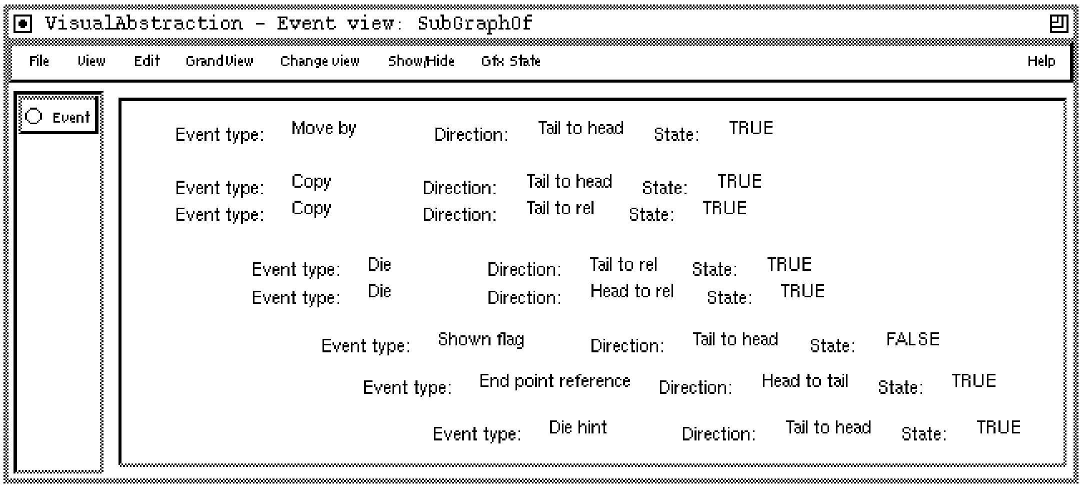


Figure 68: SubGraphOf Event View

command sets the view attribute *action* to DELETE_ENTITY and calls the method SetNeed(NEED_ENTITY, Meta(SubGraph)). The global flag, gDoDieHints, is set to TRUE to propagate the deletion of the SubGraph along the SUbGraphOf relations to the elements of the SubGraph. The functionality for deleting elements in the EscalanteView method DoLeftButtonDownCommand is used to actually delete the SubGraph.

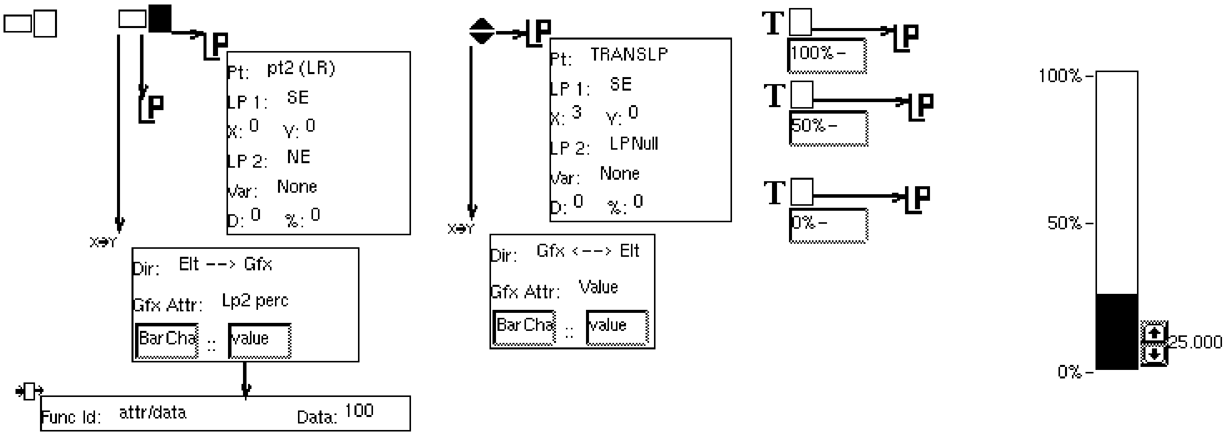
Extensive use is made of the event propagation mechanism as shown in Figure 68, the Event View for the SubGraphOf relation. From the top we have a MoveBy event defined from the tail to the head. When a SubGraph element is moved this event specification causes the members of the SubGraph to also be moved. Next, two Copy events are set. When a SubGraph is copied the contents of the SubGraph are also copied. Two Die events are used to delete the SubGraphOf relation when the the tail or the head are deleted. The Shown Flag from the tail to the head is initially FALSE. This causes the initial elision from view of the member of the SubGraph. The End Point Reference event from head to tail is set to TRUE. This causes the incident Edge relations of an element to link up to the first shown parent SubGraph element when the initial element is not shown. The Die Hint event from tail to head is used with the DeleteHierarchy command.

4.9 Example Gfx

We will now show and discuss a set of example images defined using GrandView. These examples show the use of the different Gfx types available and various aspects of the process of specifying the image of an element.

4.9.1 Bar Chart

Figure 69 shows a specification and the result of the specification of an image for a *BarChart* element. The BarChart element has an integer attribute *value* which is manipulated and



(a) (b)

Figure 69: Gfx Example 1

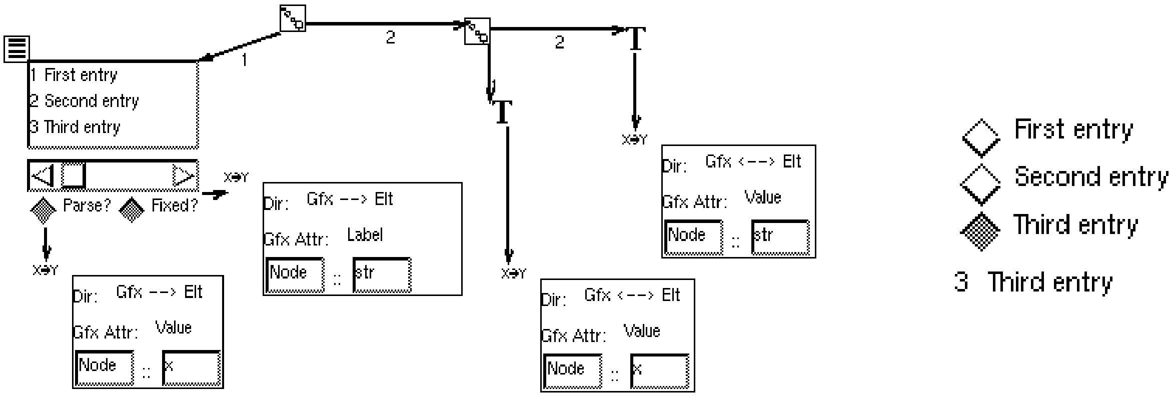
displayed through the image of the element. This image consists of an outer and inner rectangle, an IncDecGfx and a set of TextGfx that provide the % labels. The value of the IncDecGfx is mapped to the BarChart::value attribute. The location of the inner rectangle is given as:

```
pt1 = SW;
pt2 = %(SE->NE)
```

The BarChart::value attribute is divided by 100 and mapped to the *LP2 Percent* attribute of the inner rectangle. This attribute is the percent used in the Pt2 Location Point. Changes to the BarChart::value attribute occur through the IncDecGfx. These changes are propagated through the divide by 100 filter to the LP2 Percent attribute of the inner rectangle. The layout of the % labels is accomplished using the TRANSLP Location Point. This is set to SW, W and NW, for the three TextGfx: "0%-", "50%-" and "100%-". The Trans point for each of the TextGfx is set to E.

4.9.2 OneOfListGfx Example

The second example is shown in Figure 70. The image is made up of a OneOfListGfx and two TextGfx. A OneOfListGfx is a set of labeled buttons, one of which is on. There are two forms of this Gfx: in the fixed form, the size of the Gfx is determined by the size of the set of buttons. If the Gfx is not fixed then the size is solely determined by the two Location Points. The TextViewGfx component of the OneOfListGfx specification is used to define the set of buttons and their labels. This specification takes the form of: id label. The first set of characters in each line is taken as an integer id. The following set of characters up to the end of the line is taken as the label. (The MenuGfx specifications use the same format).



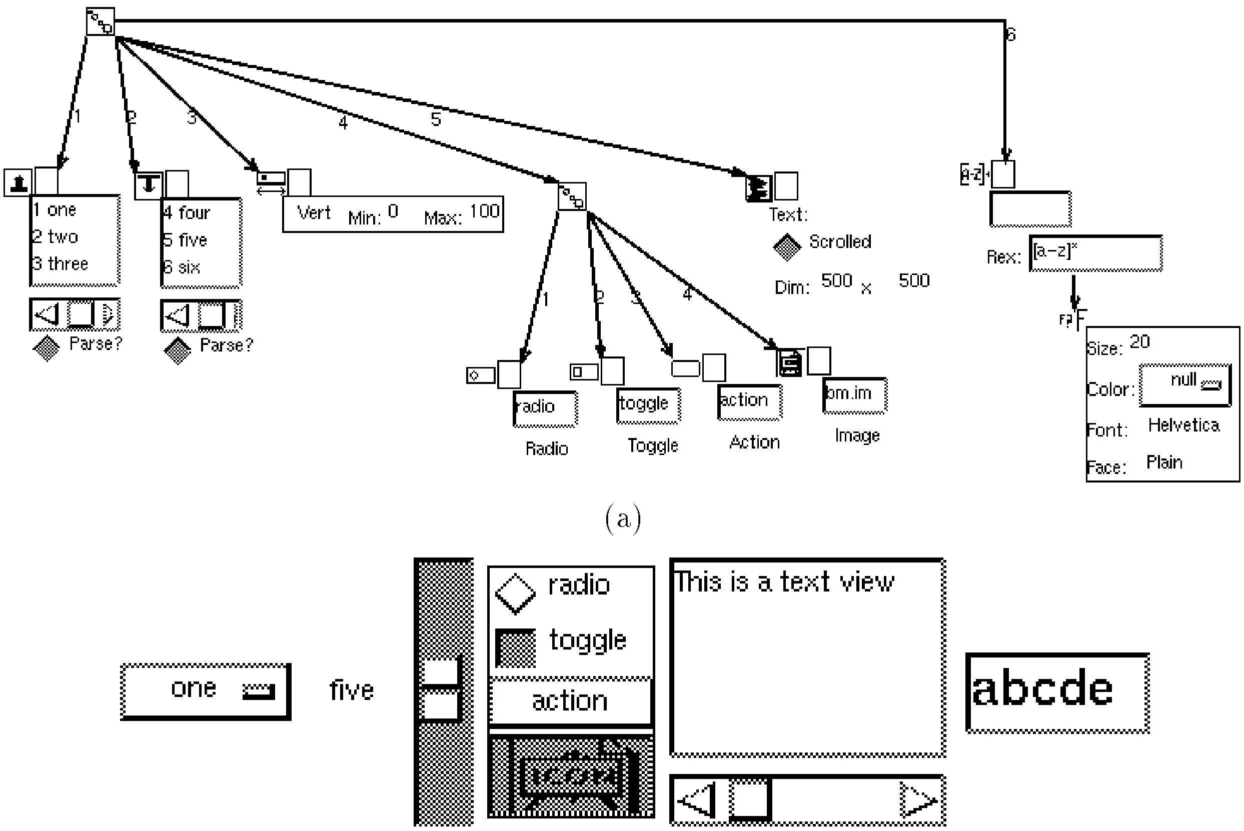
(a) (b)

Figure 70: Gfx Example 2

The value attribute of the OneOfGfx is the id of the currently selected button. The label attribute of the OneOfGfx is the label of the currently selected button. In the example, the value and label attributes of the OneOfListGfx is mapped to an x and str attribute of the element. These, in turn, are mapped to the TextGfx objects. The Parse? button controls whether the list of ids and labels is parsed or not. If the list is not parsed, the string in the TextView is written verbatim in the generated code. This allows for the inclusion of complex or lengthy lists (See apps/GrandView/SRCS/LPSpec.C).

4.9.3 Widget Gfx Example

Figure 71a shows a specification of a collection of widget Gfx. Figure 71b shows the result of this specification. From the left there is a popup and pull down menu. These are defined similarly as the OneOfListGfx described above. The SliderGfx specification consists of a direction and min and max fields. Next is a collection of different ButtonGfx. The specification of a button consists of the label and the button type. The label of an ImageButtonGfx is taken to be a bitmap file name. The value attribute of a button is the state (0 or 1) of the button. The size of an ImageButtonGfx is arbitrary. The specification of a TextViewGfx consists of initial text, dimension and whether the TextViewGfx is scrolled or not. The dimension is the size of the underlying view that is contained within the scroller. The RegExpFieldGfx lets one define a regular expression that any input to the field must satisfy. In the case of the example, the only legal input is lower case letters. A *TextState* element is connected to the RegExpFieldGfx element in the Gfx View. The TextState element allows one to to specify text size, font, color, etc.



(a)

(b)

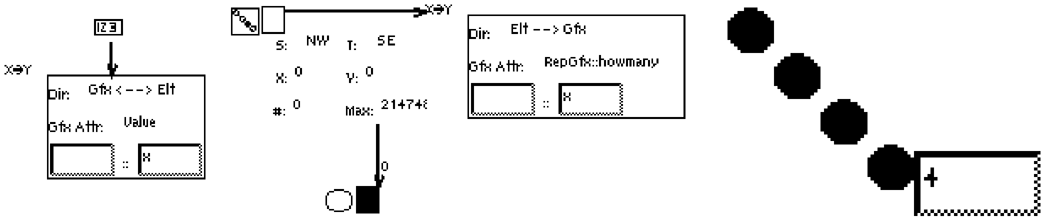
Figure 71: Gfx Example 3

4.9.4 Displaying tokens

The representation of a visual language may incorporate the display of multiple graphical images that represent some internal state (e.g., tokens). Figure 72a shows the Gfx specification of an element and Figure 72b shows the resulting image. This specification consists of a IntFieldGfx, a Repeating Gfx Set and a filled OvalGfx. The value of the IntFieldGfx is mapped to an attribute x. The attribute x is mapped to the Repgfx::howmany attribute that determines how many copies of the child Gfx are created. The Repeating Gfx Set lays out its children SE = NW. The OvalGfx is the child of the Repeating Gfx Set. As seen in Figure 72b the value of the x attribute has been set to 4 through the IntFieldGfx. This value has been propagated to the Repeating Gfx Set, causing the creation of 4 ovals, laid out according to the specification.

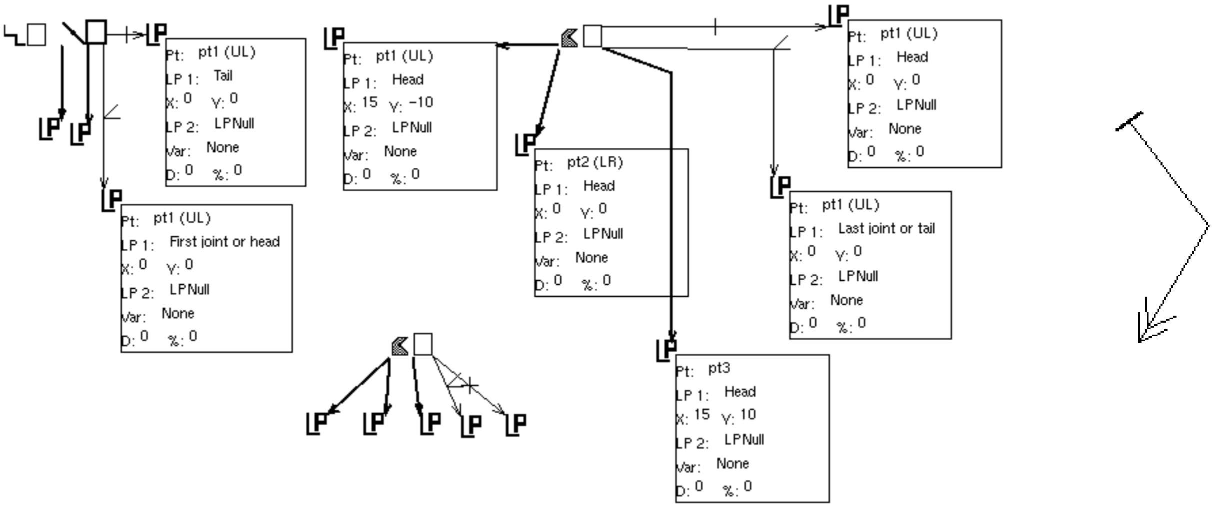
4.9.5 Using the OriginOf and AngleOf Elements

One can use the *OriginOf* and *AngleOf* elements (they are relations) to define a rotated coordinate system. The head of the relation is preset to be a Location Point element (one



(a) (b)

Figure 72: Gfx Example 4



(a) (b)

Figure 73: Defining New Coordinate System

should ignore the Pt: field). One attaches the tail of the relation to the GfxSpec element to be rotated. The head of the OriginOf relation defines the new origin of the coordinate system. The head of the AngleOf relation defines the positive X axis of the new coordinate system. One defines the position of a Gfx object with the Location Points as described above. OriginOf and AngleOf elements are used to rotate the Gfx. For example, Figure 73a shows how these elements are used to define different arrowhead styles. (Figure 73b shows the result of this specification.) The line perpendicular to the edge is defined with Location Points: Pt1 = Tail + (0,10), Pt2 = Tail + (0,-10). The point defined as the new origin is the tail. The point defined as the new angle is the first joint or the head. The double arrowhead at the head of the relation is defined using two PolyGfx. The OriginOf point for both of these PolyGfx is the head. The AngleOf point is the Last Joint or Tail. The Location Points of these PolyGfx are defined as follows:

```
PolyGfx1:  Pt1 = Head + (15,-10)
           Pt2 = Head
           Pt3 = Head + (15,10)
PolyGfx2:  Pt1 = Head + (25,-10)
           Pt2 = Head + (10,0)
           Pt3 = Head + (25,10)
```

5 Acknowledgments

Escalante is a realization of the Ph.D. research conducted by author McWhirter; he has been supported in this work by a grant from US West Advanced Technologies. Eckert has used Escalante to build several visual applications, some of which are described in Section 4, and has been supported by a grant from NCR. Nutt has been supported by Bull Worldwide Information Systems and US West Advanced Technologies on this work.

References

[1] A. Weinand, E. Gamma, and R. Marty. ET++ - an object oriented application framework in C++. In *OOPSLA'88 Conference Proceedings*, September 1988.

Index