

A GRAPHICAL HIERARCHY BROWSER FOR SSC LATTICE CONFIGURATION DATABASE

Jiasheng Zhou, Rao Bhogavalli

December of 1992

Lattice Database Operation, PMO
Superconducting Super Collider Laboratory†
2550 Beckleymeade Avenue, MS 4011
Dallas, Texas 75237

Table of Contents

Abstract	1
1.Introduction and Motivation	1
2.Basic Design Issues	2
2.1. Class Browser	3
2.2. Class Node	4
2.3. Tree Displaying Algorithm	6
3.Future Research and Conclusion	10
Acknowledgments	11
4.User Guide for Using the Browser	12
4.1. Executing the Program	12
4.2. Where to Find the Source	12
4.3. Interface Description	12
4.4. Program Behavior	14
4.5. Menus	14
4.6. Tools	15
4.7. Property Browsers	16
4.8. Editors	16

List of Figures and Tables

Figure 1: A tight tree (left) and a non-tight tree (right)	8
Figure 2: Directory structure of code for browser.	12
Table 1: Command line options for browser	12
Figure 3: User interface for the hierarchical browser.	13

Abstract

In this paper we will describe a graphical user interface that can be used to conveniently browse through the objects of an object-oriented database. The application is limited to databases whose objects share parent-child relationships. Further, the relationship between the objects resembles a tree structure (i.e. each object has exactly one parent-except the root which has no parent -and zero or more children). The current implementation is limited to databases that have been developed using ObjectStore, an OODBMS. The software (we call the application **browser**) allows user to view all the objects in the database in a hierarchical structure along with their relationships, attributes, identities, and constraints. But the design principle we used can be applied to build a more general object browser. The interface is developed in C++ programming language with InterViews [Linton92] graphical interface toolkit, ObjectStore's OSCC compiler and its Data Manipulation Language (DML) [OS92].

1. Introduction and Motivation

Browser applications have proved very useful in many different areas. Some examples that can be cited immediately are 'class' browsers in object-oriented programming, the data browser in OODBMS. If we take a C++ application example, what the browser does is to take a set of header files that together constitute a C++ program and process them to get all the information related to class hierarchies. This information is then presented in a comprehensive manner which makes it easy to understand. Browsers are also usually interactive i.e. the user can make them focus on a particular part of the information, adding another level of convenience.

Database browsers are particularly useful. This is because the only way to get information from a database is by performing queries on it. A program which is an *graphical interface* to performing queries would conceivably let a person choose from a set of general queries by for instance, clicking a button. The interface program would complete the query and present the results. Presenting the information graphically where necessary and with the use of scrolled windows for large textual data, would surely add another level of convenience.

These ideas gave us motivation for a **Graphical Interactive Browser** to the existing SSC Lattice Configuration Database in ObjectStore. This database is a collection of a large number of objects that share parent-child relationships resembling a tree structure (more precisely an acyclic

graph).

The **browser** allows convenient viewing of all objects in the database graphically. Beginning with the root users can view any particular part of the hierarchy. Nodes in the hierarchy represent objects in the Database. Attributes and properties of the objects are shown in scrolled windows that display text. Shared attributes usually have multiview connecting to the same subject. User can edit most of the attributes, identities and constraints of an object and make these changes as a transaction back to the database with authorization. We have also developed an algorithm to efficiently display the graphical hierarchy structure with consideration of some ergonomic features.

There are several object browsers on the market that is worth mentioning. ObjectStore's *osbrowser* is a text object browser. It can be used to traverse any object navigation hierarchy built on ObjectStore. But only one branch can be traversed each time. InterViews' *iclass* is quite similar to *osbrowser*. Energizer's class browser provides a graphical two dimensional interface. It can be used to browse any program elements in the calling tree. Program element can be modified and sent back to the database. But Energizer's class browser is not very flexible to be used in any object hierarchy.

The rest of the paper is a discussion of the design and implementation of the **browser**. The Appendices contain information required to use the browser.

The **browser** is implemented in AT&T cfront 3.0, InterViews 3.1beta [Linton92], Unidraw library [Vliss90] and the Actor Model [Zhouj92] library developed on top of the ObjectStore [OS92] database management system.

2. Basic Design Issues

browser is mainly composed of two classes: *Browser* and *Node*. *Browser* provides a window view and functions for object (node) manipulation. *Node* is a graphical representation for the object node to be browsed. The design is inspired by the *punidraw* and *schem* [Vliss91] prototypes provided with the InterViews 3.1 distribution.

Class *Browser* is a subclass of Unidraw's *Editor* [Linton92], a base class for top-level windows in an application. *Browser* provides a complete user interface for editing a *component's subject* [Linton92]. It unites one or more viewers with *commands* and *tools* that act upon the *components* and its subcomponents. Class *Node* is a subclass of Unidraw's *GraphicComps*, a base class for

graphical component subjects. *Node* provides a graphical representation and manipulation to the actual object to be browsed. Class *Actor* provides an object model for the entity to be browsed. For detailed information on Actor library, the reader should refer to [Zhouj92].

The on-going coding is done in the subdirectory `~zhouj/iv-examples/examples/browser.new` on machine `poplar.ssc.gov` (143.202.76.23).

2.1. Class Browser

Class *Browser* is derived from Unidraw's *Editor*, the base class for top-level windows in an application. *Browser* provides a complete user interface for browsing a component subject. It unites one or more viewers with Unidraw's commands and tools that act upon the component and its subcomponents.

Browser provides a list of tools such as Select a node, Expand a node into its children, Unexpand a subtree to its root, Display all properties of a selected node in a list of text editors, Move and Magnify, etc. *Browser* also has two pulldown menus for file selection and graphical editing such as redo, undo, and create multiview.

Typically the *Browser* is initialized with one *actor*, which is the root of the whole hierarchy to be browsed. A selected actor will be highlighted with eight handlers around it. The viewing area of the *Browser* can be zoomed and scrolled in two dimensions.

Class definition of *Browser* is shown below:

```
persistent<db> Actor *ssc = 0;

class Browser : public Editor {
public:
    Browser(GraphicComp* = nil);
    virtual ~Browser();

    virtual Component* GetComponent(); //get Component in the browser
    virtual Viewer* GetViewer(int = 0); //get GraphicView in the browser
    virtual KeyMap* GetKeyMap();
    virtual Tool* GetCurTool(); //get current engaged tool
    virtual Selection* GetSelection();
    void ShowIdentities(char**, int);
    void ShowAttributes(char**, int);
    void ShowConstraints(char**, int);
    void ShowRelations(char**, int);

    virtual void SetComponent(Component*);
    virtual void SetViewer(Viewer*, int = 0);
    virtual void SetKeyMap(KeyMap*);
    virtual void SetSelection(Selection*);
private:
    Interactor* Interior();
    void InitViewer();
```

```

void initStateVars();

Interactor* Commands(); //build a command for menu
Interactor* Tools(); //build a tool for selection
void Include(class Command*, class PulldownMenu*);
void Include(Tool*, class Box*);
Interactor* AddScroller(Interactor*);

PulldownMenu* FileMenu();
PulldownMenu* EditMenu();

void CreateDataDisplay();
GraphicComp* _comp;
KeyMap* _keymap;
class ControlState* _curCtrl;
Viewer* _viewer;
Selection* _selection;
ColorVar* _color;

DataBrowser* _idntsBrw;
DataBrowser* _attrsBrw;
DataBrowser* _cnstsBrw;
DataBrowser* _rlatsBrw;
Interactor* IdntsDisplay();
Interactor* AttrsDisplay();
Interactor* CnstsDisplay();
Interactor* RlatsDisplay();
Interactor* DataDisplay();

DataEditor* _idntsEdt;
DataEditor* _attrsEdt;
DataEditor* _cnstsEdt;
DataEditor* _rlatsEdt;
};

```

2.2. Class Node

Class *Node* is currently derived from Unidraw's *GraphicComps*, the base class for graphical component subjects. *Node* provides a graphical representation for the actual persistent object in ObjectStore to be browsed. There are several ways to make a graphical composition in Unidraw. In general there are three levels to choose from when defining a new component. The main trade-off between the levels is ease of specification versus efficiency.

- Level 1 (Easiest/least efficient): Derive from *GraphicComps*/*GraphicViews* and define the component's appearance by composing other *GraphicComp* subclasses (*LineComp*, *RectComp*, etc.). You can forego deriving from *GraphicViews* altogether if you don't need a special view semantics (e.g., for direct manipulation): simply specify a class id that makes the corresponding *COMPONENT_VIEW* of your *GraphicComps* subclass a *GRAPHIC_VIEWS*. For example, you might derive *InverterComp* from *GraphicComps*. The constructor simply inserts a *PolygonComp*, a *CircleComp*, etc. There's no need to redefine *Read*, *Write*, etc., since

GraphicComps already knows how to deal with its children. This approach is relatively inefficient because each graphical element is a full blown graphical component, complete with a subject and a view.

- Level 2 (Harder/more efficient): Derive from GraphicComp/GraphicView and define the component's appearance in terms of predefined structured graphics object(s), much like LineComp and EllipseComp are implemented. You'll probably use a Picture and insert other predefined graphics into it. For example, a GraphicComp-based InverterComp would contain a Picture with SF_Polygon, SF_Ellipse, etc. objects in it. You'll have to do a bit more work than you would at LEVEL 1, because you must derive a GraphicView subclass and define its Update operation. However, you gain efficiency because each graphic element doesn't require both a subject and a view.

- Level 3 (Hardest/most efficient): Derive from GraphicComp/GraphicView and define the component's appearance in terms of a custom structure graphics object. For example, you might derive an InverterGraphic from Graphic and use that as the graphic in your GraphicComp-based InverterComp. You're essentially defining the component's appearance using immediate-mode (i.e., Painter) graphics. You gain the most efficiency since you no longer dedicate memory to structured graphics objects, but you're also working the hardest.

For convenient and fast prototyping, we selected the first approach. Class definition of *Node* is shown below:

```
class Node : public GraphicComps
{
public:
Node(Browser*, reference<Actor>, Node* = nil, float x0=0, float y0=0);
virtual ~Node();

void MoveTo(float, float);
virtual void Interpret(Command*); //execute a command
virtual void Uninterpret(Command*); //undo a command

void Expand();
void UnExpand();
void Display(char**, char**, char**, char**); //display properties of a node

char* Name() { return name; }
Node* Parent() { return parent; }

float Width() { return width; }
float TreeWidth() { return treeWidth; }
void TreeWidth(float w);
float Height() { return height; }
void Center (float x, float y) { xc = x; yc = y; }
void Level(int l) { myLevel = l; }
```

```

int Level() { return myLevel; }

static Node* root; //class variable always pointing to the root of the hierarchy

private:
char *name; //node name for displaying

reference<Actor> myActor; //pointer to Actor in the ObjectStore

Node* parent;
float xc, yc; //upper-middle point of the node rectangle for position
float height, width, treeWidth;

Browser* _browser;
int nOfChild, myLevel;
Node** children; //array of pointers to children
LineComp** lineComp; //array of pointers to connections

boolean HasSpace();
void Rearrange(); // rearrange children.

static int depth; //class variable for level counting in recursive program
};

```

2.3. Tree Displaying Algorithm

Since **browser** displays a tree structure, we had to devise an algorithm to display the expanding branches of the tree within the available view space/view window. We have developed two algorithms that can dynamically allocate space to the expanding branches of the tree structure and deallocate space when a subtree is unexpanded (deleted). The algorithm is discussed below.

The **browser** essentially displays a tree structure, i.e., the database to which this **browser** forms an interface has objects that share a *hierarchical (parent, child)* relationship.

The Problem:

In this program, the tree expands and takes shape at run time. In fact the Expansion of the tree is interactive and controlled by the user, i.e. the user explicitly Expands certain branches of the tree structure he/she wishes to examine. Also users can Unexpand branches of the tree that they no longer wish to examine.

One of the design requirements from the perspective of ergonomics is that the nodes/objects in the tree be displayed relatively close together. When the objects are close together, it offers to the viewer a *perspective of the relationship* between the objects. Another desirable feature is the ability to distinguish between different *levels of the tree* and identify objects of the same level in the tree.

Owing to the limited size of the viewing area and large database size, all the objects or the whole tree cannot usually be displayed at one time, i.e., we are bound to experience a conflict in space

allocation when several branches try to Expand. We need to resolve the conflict and allocate space as and when required among the branches of the tree. By space here we mean horizontal space or breadth. This is where the conflict occurs. This is because all nodes of the tree of a given level, occur at exactly the same height or vertical distance from the root. If a certain branch did not contain sufficient vertical space to expand fully, the problem would be faced by all branches that expand to the same level, i.e., such a problem would be global to the tree and not of a nature of conflict among various branches of the tree. Such a problem would have to be solved by limiting the number of levels of the tree that can be viewed at one time or increasing the view area in the height dimension. We are not concerned with such a problem in this tool as it does not occur in our specific case.

The Possible Approaches:

We could implement a solution for the preceding problem of space allocation in two ways. We could follow a *Static Space Allocation* method or a *Dynamic Space Allocation* method.

The Static Space Allocation Method:

In this method, one would allocate all the available space to the root node and divide it equally among its children. When these children expand, the new nodes would be spread out evenly among the available space of its parent and so on. In this way, the leaf nodes get the smallest allocation and the nodes at the top are allocated more space, with the root getting the maximum space. This approach has two important drawbacks.

The first drawback is that since at any level the available space is divided equally among the children, at the higher levels when there is more space, each child will be allocated a larger space. Since each node is positioned at the center of its allocation, each node would be positioned such that they are quite far from each other. This does not allow the user to view all the children of a node at one time. This is not desirable from an ergonomic design perspective.

The second major drawback: Static allocation inherently has the advantage that we need less computation at run time. But for this to be realized we should not have any conflicts or insufficiency of allocated space once we have made a static allocation. Given the size of the view window and the database we are dealing with, this is not often the case and statically-allocated space is not sufficient for several or all of the branches of the tree for their expansion up to the leaf level. When

this conflict occurs, we are forced to perform Rearrangement of the tree structure. This forces some less expanded branches to squeeze in order to accumulate more space for the heavily expanded branches, thereby losing the static allocation advantage.

The preceding listed drawbacks lead us to the conclusion that a *dynamic approach* is the better alternative in our situation.

The Design of the “Rearrange” Algorithm:

Our *dynamic approach* algorithm allocates space to expanding branches dynamically and “Rearranges” the tree each time a change has been made to positions and allocations of nodes in the tree. This dynamic algorithm always maintains a *tight tree* in the process of expanding and unexpanding.

Definition: A *tight tree* is a tree for which there is no space left between any adjacent leaves or their projection on the lowest level of the tree. Figure 1 gives an example.

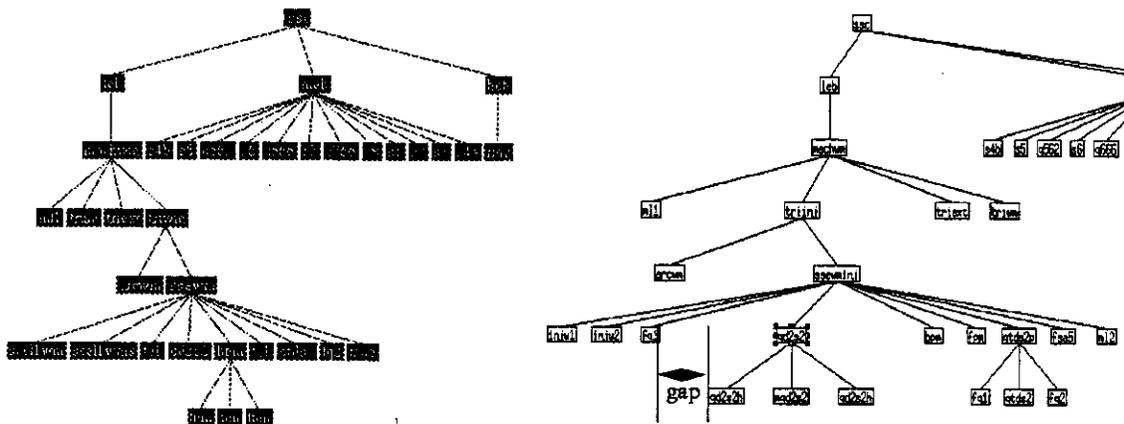


Figure 1: A tight tree (left) and a non-tight tree (right)

The algorithm is recursive. The algorithm is also very efficient in the use of display space. It allocates just sufficient space so that all nodes are clearly visible (without cluttering or intersection) while making sure that space is not wasted between nodes (a tight tree).

The algorithm is also easily modifiable with respect to certain features such as gap between nodes and height between different levels of the tree.

Finally, features of InterViews software have helped us to maintain the graphic functionality, even when the picture is zoomed in or zoomed out several times.

The Data Structures:

The main *Data Structures* of the algorithm are the data associated with each node. They are:

- 1) x_c, y_c co-ordinates of the top center of the node showing where the node is placed in the coordinate space.
- 2) An integer value `myLevel` which is obtained from the database. For leaf nodes `myLevel` will be '0'. For non-leaf nodes it indicates the number of levels of the node from the leaf nodes in its subtree. This value is present in the database and is retrieved by the program from there.
- 3) An integer value `nOfChild` which shows the number of children a node has.
- 4) If `myLevel` member variable of a node is greater than 0, i.e., the node has children: pointers are allocated dynamically to all `child` node objects and `line` (graphic) objects connecting the node between children and parent.
- 5) Name of each node and pointers to graphic objects, i.e., the rectangular box (`rect`) and text graphic (`text`) representing the name of the node.
- 6) A float value `treeWidth` which holds the width of the subtree represented by each node. If a node is a leaf node, `treeWidth` represents the width of the rectangular box representing the node. If a node has children the `treeWidth` of that node represents the sum of the `treeWidth`s of all its children.

The Algorithm:

Having discussed the important data structures, we will now be able to discuss the *Algorithm* itself including a listing of the pseudo code.

The two important operations of the tool (**browser**) which require graphic manipulations are:

- (1) The Expand operation and (2) The UnExpand operation.

The Expand operation when performed on a Node object causes the following:

If the node has children, Expand (1) obtains the number of children and their names from the database and (2) creates nodes for the child objects and displays the nodes. If the node does not have children, issue a "bell". The following is pseudo code of the function.

```
Node::Expand() {
  if (nOfChild != 0 // node already Expanded.
      OR myLevel == 0) // leaf node
    return;
  do transaction { // transaction with database.
    get nOfChild; // retrieve child information.
    get names of each child;
  }
  //create child nodes to represent the child objects;
  for(i=0; i<nOfChild; i++) children[i] = new Node(...); //with relevant parameters.
  set treeWidth of each child node;
  float totalWidth = sum of treeWidth of children + (nOfChild -1)*NODE_GAP;
```

```

TreeWidth(totalWidth); //update the treeWidth of this and all Parents in heirarchy.
if HasSpace(totalWidth)// is required space available?
  ReArrange();// then Redraw this node. Add children.
else
  root->ReArrange();//else redraw from root. i.e. Create space.
Update canvas to reflect changes.
} // end of Expand().

```

The UnExpand operation, when performed on a node, causes the following. If a node has children, (1) remove children and lines to them from graphic view, (2) delete the child objects and the lines from memory, (3) free memory, (4) update treeWidth of the node and all parents in hierarchy and (5) redraw the tree from root. If the node does not have children, issue a "bell". The following is pseudo code of the function.

```

Node::UnExpand() {
  if (nOfChild==0) { ring_bell(); return; } // cannot UnExpand.
  for (int i=0; i<nOfChild; i++) {
    children[i]->UnExpand();// Recursive call to children.
    delete children[i];
    delete lineComp[i];
  }
  TreeWidth(width);//Recursive function to update all nodes in parent heirarchy.
  root->ReArrange();//redraw the whole tree form root.
  Update canvas to reflect changes;
} /end of UnExpand.

```

The ReArrange algorithm has been used in Expand and UnExpand routines. The following is pseudo code and description of this algorithm.

```

Node::ReArrange() {
  if (nOfChild==0) return;// all nodes are arranged by their parent node.

  delete old lines connecting node to child nodes; // delete lineComp.
  allocate memory for creating lines; //new lines are drawn to locations of children
  lineComp = new LineComp*[nOfChild];

  float y = yc - height*4; // y is for child nodes. yc and height of current node.
  float x; // will hold xc for child nodes.
  x = xc - treeWidth/2;// points to left end of span of node.
  for (int i=0; i<nOfChild; i++) {
    x += children[i]->TreeWidth()/2; //x points to center of span of ith child.
    lineComp[i] = new Line(xc, yc-height, x, y);
    // new line from bottom center of node to top center of child.
    children[i]->MoveTo(x, y);//move child[i] to x, y.
    children[i]->Center(x, y);//set xc, yc of child[i] to new position.
    children[i]->ReArrange();//Recursive call to update position of this subtree.
    x += children[i]->TreeWidth()/2; // to point to end of span of ith child.
    if (i < nOfChild-1)// if i is not last child.
      x += NODE_GAP;// to create a small gap between nodes.
  }
} // end of ReArrange().

```

3. Future Research and Conclusion

There are four areas in which we plan to continue this research. They are listed as follows:

- 1) Build a more general object browser that is able to browser any objects bound with certain

configuration using Meta Object Protocol (MOP) [OStran92].

- 2) Use of Pin class of Unidraw for semantic connection between nodes to support dataflow modeling. When a node is moved, the connection will be maintained by stretching shrinking and moving as necessary, all connecting lines that belong to the node. The connection can also represent internally a relationship such as a-part-of.
- 3) Support single-subject-multiple-view paradigm. Changes made to an object will be reflected to its subject and all its views in different windows at the same time.
- 4) Incorporating editing capabilities in **browser** to support configuration manipulation.

First is the use of Pin class of Unidraw. Pin class is derived from class Connector. Pins can be used in connecting parents to children in the tree. This connection would be at a semantic level besides being represented as a line connection. Such semantic connectivity ensures that even though the tree may be manipulated by the user or nodes moved arbitrarily from each other, the parent child connecting lines will change as necessary to represent the relationship or connection. Without the use of Pin class, as we are doing now, when nodes change position we need to determine a new location, create new connecting lines to represent the relationships and delete old lines.

The facility of multiple views has proved useful in graphic applications, especially in CAD etc. With this facility, user can create a second or third view of **browser** at an arbitrary stage during run time. The additional views will be initiated with the current graphic state of browser. Additional views have the same functionality as the original and provide all operations.

Multiple views can typically be used to zoom in on a part of the figure or manipulate a part of the figure. All changes made to the additional views are reflected in the original and all other views.

We have plans to incorporate editing capabilities in the **browser**. Editing will let the users add new Nodes(objects) to the database and also define parent child relationships between nodes. Editing facilities can also be used to specify properties, attributes and constraints of an object.

Acknowledgments

We greatly appreciate the valuable guidance and comments from Dr. Garry Trahern of Lattice Database Operation at PMO. We also thank Dr. Mike Allen of Control Department for his encouragement to this project.

Appendices

4. User Guide for Using the Browser

The following is a User Guide for using the Interactive Graphical Browser.

4.1. Executing the Program

The executable is called “**browser**”. It is located in the directory `~zhouj/research/os/browser.new/SUN4` on “poplar”. There is public access to the executable (just type `~zhouj/research/os/browser.new/SUN4/browser`).

Command line options are shown in Table 1 below:

Table 1: Command line options for **browser**

option	example
<code>-database <database name></code>	<code>browser -database /zhouj/ssc</code> (default: <code>/zhouj/ssc</code>)
<code>-db <database name></code>	<code>browser -db /zhouj/ssc</code> (default: <code>/zhouj/ssc</code>)
<code>-display <target></code>	<code>browser -db /zhouj/leb -display telperion:0.0</code>
<code>-bg</code>	<code>browser -gb yellow</code>
<code>-fg</code>	<code>browser -fg gray</code>

4.2. Where to Find the Source

The program consists of several files. The *.cc source files are located in `~zhouj/research/os/browser.new` directory. The *.h header files are in the `include` subdirectory. The Makefile and executable are in `SUN4` subdirectory, see Figure 2 below:

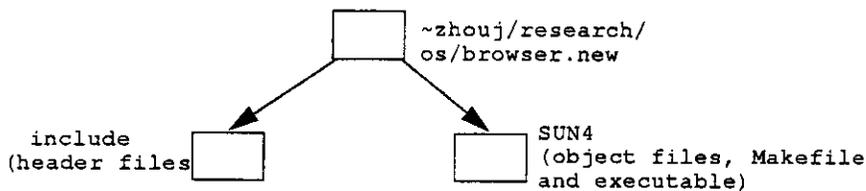


Figure 2: Directory structure of code for browser.

4.3. Interface Description

Figure 3 is a “snapshot” of the graphical user interface during execution. Across the top of the window are Command Menus and Tools to manipulate the Graphic View area. The Menu Commands and Tools are explained in detail below.

Along the right side of the window are scrollable string browsers to view the Identities,

Attributes, Constraints and Relationships [Zhouj92] of objects being viewed.

4.4. Program Behavior

Objects of the database are referred to as nodes within the program. The File Menu New option can be used to open a data base file for viewing with this browser. Once the file is opened, the root object is brought up on the screen. This object can be expanded to view its children using the Expand Tool which is present among the tools in the top right hand side of the browser. Unexpand can be used on an expanded object to remove its children from the screen in situations where the screen might have become cluttered with many objects. Display Tool on an object displays the Identities, Attributes, etc. in String browsers to the right of the view area. Double clicking the highlighted item in a string browser will move the selected item to the editor below for editing. The result of a editing will be ended by a carriage return and be reflected in the string browser above. Each of the Menu options and Command tools are explained in detail below.

The viewing area of the browser can be zoomed and scrolled using the Panner in the lower-right corner of the user interface as shown in Figure 3 or use the mouse as follows:

^mouse-middle-button a hand icon will appear for 'drag' scrolling in two dimensions.

^mouse-right-button an arrow icon will appear for continuous scrolling (speed sensitive).

4.5. Menus

The browser has two PulldownMenus on the top left hand side of the application window.

The first PulldownMenu is File Menu and the second is Edit Menu. File Menu: This Pulldown-Menu offers five operations. Options in the menu also have key bindings. The operations provided by File Menu are:

New key **^N**: (Hold control key and upper case N key at the same time) Opens a new Data File for browsing using this package.

Open key **^O**: Opens a previously-viewed file whose state has been saved. Open brings up a file-chooser dialog listing the files in the current directory.

Save **^S**: To save a file being viewed. Save also brings up file-chooser dialog to specify the name of a new file or select an existing file.

Print **^P**: Option to print a file.

Quit **^Q**: Quits the application.

Edit Menu: Edit performs the operations to a selected object or a selected area. The operations provided by Edit are:

Undo key **U**: Performs several layers of Undo.

Redo **R**: To repeat an operation.

Delete **^D**: Deletes a selected object from view area.

Duplicate **d**: Duplicates a selected object.

4.6. Tools

The Tool palette on the top left-hand side of the screen offers tools which provide the functionality of the browser. For convenience, all the tools have a corresponding key with which they can be selected. Each tool is described in detail below.

Select s: (Click on **Select** tool or just type the **s** key) Select Tool lets the user change the current selection by a single mouse click upon any object. Multiple objects can be selected by drawing a box that completely includes the objects. Selection determines the object on which consequent operations may be performed.

Expand e: Expand operation on an object on the screen brings up its children from the Data Base if it is not the leaf node in the hierarchy. To perform Expand on an object, select Expand from the tool palette. Click on the object you wish to expand. This will result in the children of this object being brought up on screen if this is not the leaf object.

UnExpand u: UnExpand on a node removes the children of that node and their children (recursively until the end) from view. This may be used to clear up the view area or when objects are no longer needed.

Display d: Performing Display on an object on the screen brings up several properties of the object from the Data Field for viewing. The properties shown are: Identities, Attributes, Constraints, and Relationships. Each is shown in a separate String browser. The use of this string browser is explained below.

Move m: Move lets an object on the screen be moved around. This tool is not as much of use to the user as it is to the program which uses this to rearrange objects on the screen when it gets cluttered.

Magnify z: Magnify zooms on an area selected by boxing the area.

Connect C: This tool allows separate objects in view to be connected in a data flow manner. This may be used to indicate a relationship between different objects that are being built.

Line l: This and the next two tools in the palette (Rectangle r and Ellipse o), can be used to create objects on the screen.

4.7. Property Browsers

There are four string browsers that display different properties of the objects. They are placed vertically along the right side of the application window. The four browsers display Identities, Attributes, Constraints and Relationships, respectively. All the string browsers are scrollable. At the bottom of each string browser is a string editor. String editor can be used by double-clicking on a string. This will place the string in the editor and will update the string at the end of editing. Key bindings exist for all the functions of the string browser. They are:

g Go to the first string.

G Go to the last string.

a select all.

DEL/BS unselect all.

p select previous string.

n select next string.

< select topmost-visible string.

> select bottommost visible string

j scroll down one string.

k scroll up one string.

SPACE scroll down one screen full.

b scroll up one screen full.

d scroll down one-half screen full.

u scroll up one-half screen full.

^mouse-middle-button a hand icon will appear for scrolling vertically.

^mouse-right-button a arrow icon will appear for continuous scrolling (speed sensitive).

4.8. Editors

Associated with each string browser is a string Editor at the bottom. This can be used to edit

properties of the object on which “Display” tool is used. After each edit operation, the value is updated by the browser which contains the string. The key bindings for the Editor are as follows.

Key Operation

^B CharacterLeft

^F CharacterRight

^A BeginningOfText

^E EndOfText

^M Move the selection one character position to the left, or right, or to the beginning or end of text.

^H, DEL Erase the text of current selection. If at blank, delete previous character.

^D Delete the text of current selection. If at blank delete next character.

^U Select all.

^W SelectWord. Select the entire text, or extend the selection to the left by one whole word.

These commands enable common editing operations to be performed without using the mouse. For example, to replace the previous word in the text, do a SelectWord, DEL and type the new text.

References

[Linton92] Linton, M.; Calder, P.R.; Interrante, J.A.; Tang, S.; Vlissides, J.: “InterViews Reference Manual Version 3.1-Beta”, June 26, 1992.

[OS92] ObjectStore 1.2 Reference Manual.

[OStran92] ObjectStore2.0 Announcement Seminar Lecture Notes.

[Vliss90] Vlissides, John: “Generalized Graphical Object Editing”, Technical Report: CSL-TR-90-427, Stanford University, June, 1990.

[Vliss91] Vlissides, John: “Unidraw Tutorial I: A Simple Drawing Editor”, Stanford University, July, 12, 1991.

[Zhouj92] Zhou, Jiasheng: “Object-Oriented Modeling for Dynamic Simulation”, SSC Lab Internal Report, August, 1992.