The Derivation of a hyperText Widget Class from the Athena Text Widget

J. E. Drufke, Jr.
John J. Leggett
David L. Hicks
John L. Schnase

Hypermedia Research Lab

APRIL 1991

TAMU-HRL 91-002

Table of Contents

1. INTRODUCTION 1
1.1 Monolithic versus Non-monolithic Hypermedia Systems 1
1.2 Hypertext Conceptual Model 2
1.3 The HRL Prototype 3
1.4 A hyperText Widget for Xt Applications 3

2. IMPLEMENTATION 9
2.1 Brief Overview of the Athena Text Widget 9
2.2 Methods, Procedures, and Design Objectives 11
2.3 Modifications Made to the Athena Text Widget 12
2.4 Integration with Xlt 17

3. REQUIREMENTS SUMMARY 19

APPENDIX A 21

APPENDIX B 24

APPENDIX C 28

APPENDIX D 33

APPENDIX E 35

APPENDIX F 39

GLOSSARY 42

REFERENCES 43

The Derivation of a hyperText Widget Class from the Athena Text Widget

J. E. Drufke, Jr.
John J. Leggett
David L. Hicks
John L. Schnase

1. INTRODUCTION

The Hypertext Research Lab (HRL) in the Department of Computer Science at Texas A&M is engaged in the design and prototypic implementation of a next generation *hypermedia* system architecture. Since January of 1987, the HRL has contributed to furthering the state of the art through the participation in conferences, standards committees, and research into the formal modeling and specification of hypermedia systems.

The discussion that follows makes use of some concepts that are specific to the hypertext model developed within the HRL. Familiarity with terminology is assumed here, and the reader should refer to the glossary at the end of the document for clarification of terms.

1.1 Monolithic versus Non-monolithic Hypermedia Systems

Hypermedia systems of the past have typically been *monolithic* with respect to the rest of a user's computing environment [KAC]. For monolithic systems, one is confined to the system as delivered. The interchange of information to and from applications outside the domain of a monolithic system is usually limited to vendor provided import and export operations only. In HRL's development of next generation hypermedia systems, this monolithic approach is replaced by a *distributed*, *seamless* architecture that allows the extension of its services to all applications that choose to participate.

The goal of the ongoing research and development work in HRL is to provide hypermedia architectures and supporting mechanisms for *interapplication linking* among *heterogeneous* applications. Figure 1-1 shows a typical environment of a users workstation with multiple heterogeneous applications including text, image, audio, and animation.

1.2 Hypertext Conceptual Model

A conceptual model of hypertext that demonstrates interapplication linking is shown in Figure 1-2. Interapplication linking is accomplished by having minimal support functionality reside within an application which maintains persistent selections and communicates with the hypermedia layer. In the conceptual model shown, the hypermedia layer includes anchors, links, and operations on anchors and links. These operations include the ability to create associations between applications (authoring) and to navigate through existing associations (browsing).

Figure 1-3 shows the previous windowing environment after an association has been created. The application is responsible for providing a mechanism to create persistent selections and a mechanism to communicate with the hypermedia layer using some agreed upon protocol. This protocol will allow the hypermedia layer to maintain associations on the application's behalf. The application need only understand persistent selections, state changes among these selections (i.e., having or not having anchors with or without links), and the protocol for authoring and browsing.

Figure 1-4 shows, in general, a non-monolithic, distributed hypermedia system architecture. Communication channels between applications and link services are used to invoke hypertext functionality. Link services encapsulates precisely the mechanisms necessary for creating and following hypertext links (associations). Client applications may access this functionality through the specified protocol. As a consequence of this distributed architecture, clients using link services are free to assume their own *link semantics* by building on top of link services.

1.3 The HRL Prototype

The HRL system prototype (SP-1) architecture is illustrated in Figure 1-5. The hypermedia layer as defined by this prototype is composed of three subsystems. The Link Services Manager (LSM) communicates with applications in the process of creating anchors and links, following links, and toggling the state (on/off) for link and anchor markers. The Semantic Database Manager (SDM) is responsible for storing and retrieving hypertext components used by applications participating in link services. These components are defined entirely in the application domain and exist as a continuous stream of bytes within the database. The Association Set Manager (ASM) is responsible for maintaining associations created by the LSM. Both the SDM and the ASM access an object database for saving information.

1.4 A hyperText Widget for Xt Applications

This document describes modifications made to the Athena *text* widget to allow participation in the interapplication linking capabilities made available through the X link services toolkit (Xlt) of the prototype architecture SP-1 being developed within the Hypertext Research Lab. The Athena *text* widget running under X windows and the X toolkit intrinsics includes functionality common to most text editors. **Xedit**, an application written using this widget along with Xlib/ Xt libraries and several other widgets, was used for the purpose of developing and testing the modifications made in this project.

The goal of this project is to extend the functionality of the Athena *text* widget in ways that provide the user direct access to an interactive *hypertext* authoring and browsing system. The capabilities that are made available by the X link services toolkit include the support necessary for building hypertext documents (i.e., interapplication links). However, this toolkit is strictly a procedural interface. The main objective of this project was to make use of this procedural interface by providing a user interface to link services. The Athena *text* widget was a prime candidate for accomplishing this objective due to its availability and existing functionality.

Parental investment may be defined as "any investment by the parent of an individual offspring that increases the offspring's chance of surviving at the cost of the parent's ability to invest in other offspring (Trivers, 1972). Although it is difficult to distill a single metric to represent the time, energy, and risk associated with **parental investment**, a rough estimate can be obtained by considering the combined qualitative and energetic contributions of parents to the reproductive effort (Biedenweg, 1983).

In Cassin's Sparrow, territorial defense is the responsibility of the male and primarily involves the allocation of energy resources with little associated risk. Egg production, requiring energy, and incubation, requiring time, are done entirely by the female. Feeding of offspring demands time and energy and appears to be skewed toward the female during both the nestling and fledgling care phases. Protection of the female and offspring, on the other hand, is a male-biased activity involving time, energy, and risk.

Figure 1. Cassin's Sparrow fledgling foraging in a Honey mesquite tree (*Prosopis glandulosa*).

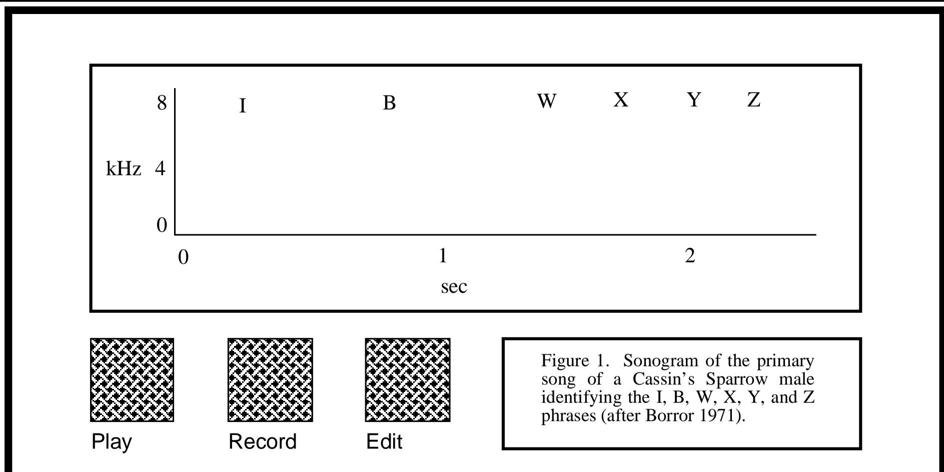

Figure 1. Sonogram of the primary song of a Cassin's Sparrow male identifying the I, B, W, X, Y, and Z phrases (after Borror 1971).

Summary. The role of bi-parental care in maximizing male and female fitness is an important assumption of the evolution of avian monogamy (Wittenberger and Tilson, 1980). In Cassin's Sparrow, there appears to be a qualitative balance between parents in parental investment, i.e. male and female investments are both required throughout the cycle if offspring are to be raised. Males, however, have a nearly two-fold greater energetic investment in offspring than females.

Taken together, these results suggest that the monogamous mating system in Cassin's Sparrow results from the necessity of 2 parents for reproductive success and the capacity of males to maximize fitness by preserving territories and mates over potentially long breeding seasons.

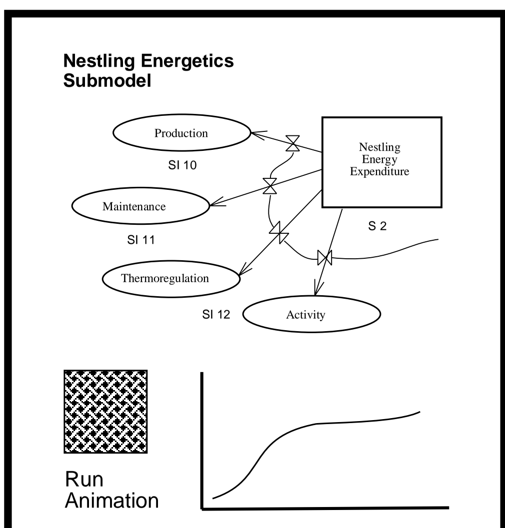

Figure 1-1. Typical Workstation Display

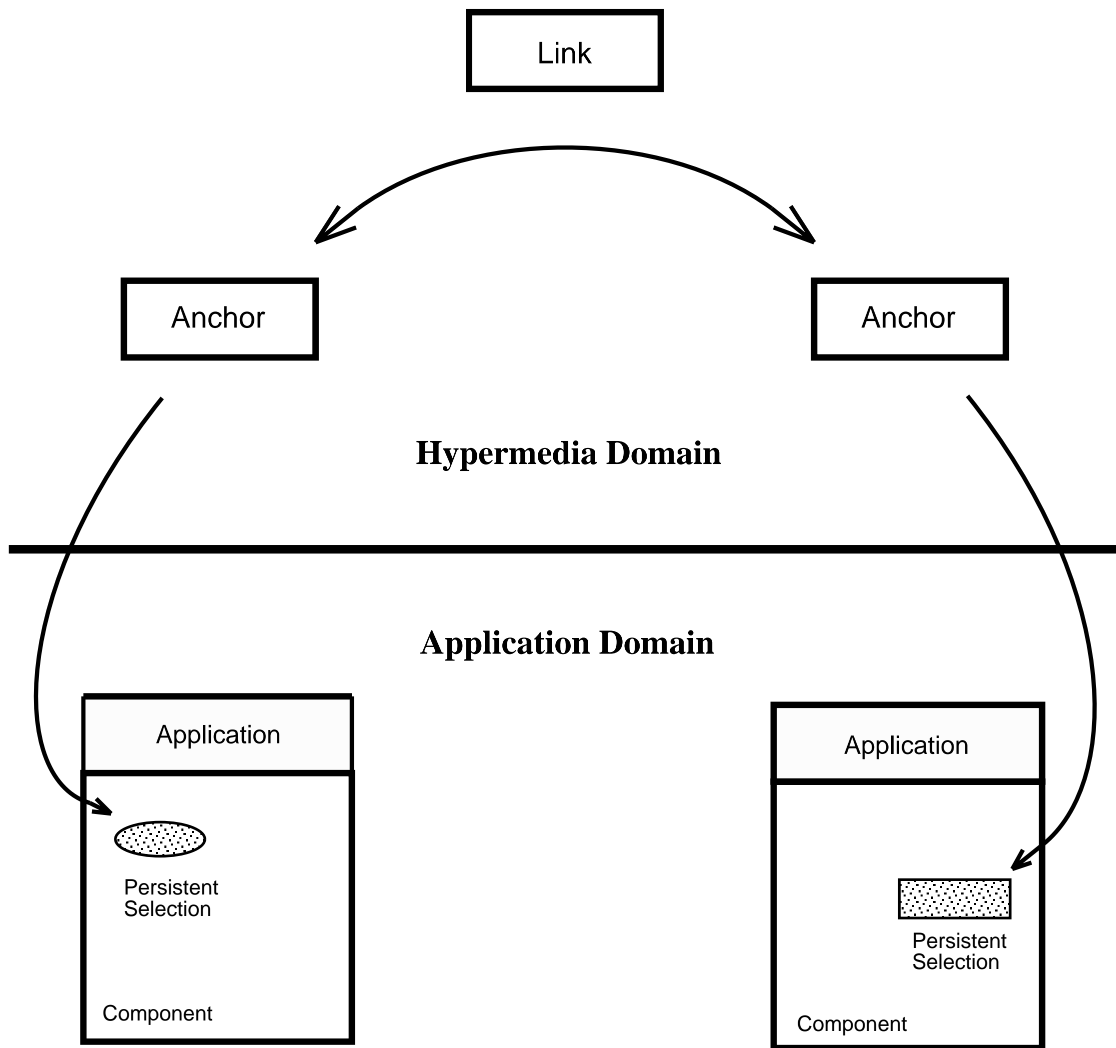

Figure 1-2. Conceptual Model of Hypertext

LinkID 6

AnchorID 5

AnchorID 5

Parental investment may be defined as "any investment by the parent of an individual offspring that increases the offspring's chance of surviving at the cost of the parent's ability to invest in other offspring (Trivers, 1972). Although it is difficult to distill a single metric to represent the time, energy, and risk associated with **parental investment**, a rough estimate can be obtained by considering the combined qualitative and energetic contributions of parents to the reproductive effort (Biedenweg, 1983).

In Cassin's Sparrow, territorial defense is the responsibility of the male and primarily involves the allocation of energy resources with little associated risk. Egg production, requiring energy, and incubation, requiring time, are done entirely by the female. Feeding of offspring demands time and energy and appears to be skewed toward the female during both the nestling and fledgling care phases. Protection of the female and offspring, on the other hand, is a male-biased activity involving time, energy, and risk.

Figure 1. Cassin's Sparrow fledgling foraging in a Honey mesquite tree (*Prosopis glandulosa*).

PSID=A CompID=1 AppID 2

kHz 8 4 0

I B W X Y Z

0 1 2

sec

Play Record Edit

Figure 1. Sonogram of the primary song of a Cassin's Sparrow male identifying the I, B, W, X, Y, and Z phrases (after Borror 1971).

Summary. The role of bi-parental care in maximizing male and female fitness is an important assumption of the evolution of avian monogamy (Wittenberger and Tilson, 1980). In Cassin's Sparrow, there appears to be a qualitative balance between parents in parental investment, i.e. male and female investments are both required throughout the cycle if offspring are to be raised. Males, however, have a nearly two-fold greater energetic investment in offspring than females.

Taken together, these results suggest that the monogamous mating system in Cassin's Sparrow results from the necessity of 2 parents for reproductive success and the capacity of males to maximize fitness by preserving territories and mates over potentially long breeding seasons.

PSID=B CompID=4 AppID=3

Nestling Energetics Submodel

Production

SI 10

Nestling Energy Expenditure

Maintenance

SI 11

S 2

Thermoregulation

SI 12

Activity

Run Animation

Figure 1-3. An Interapplication Link

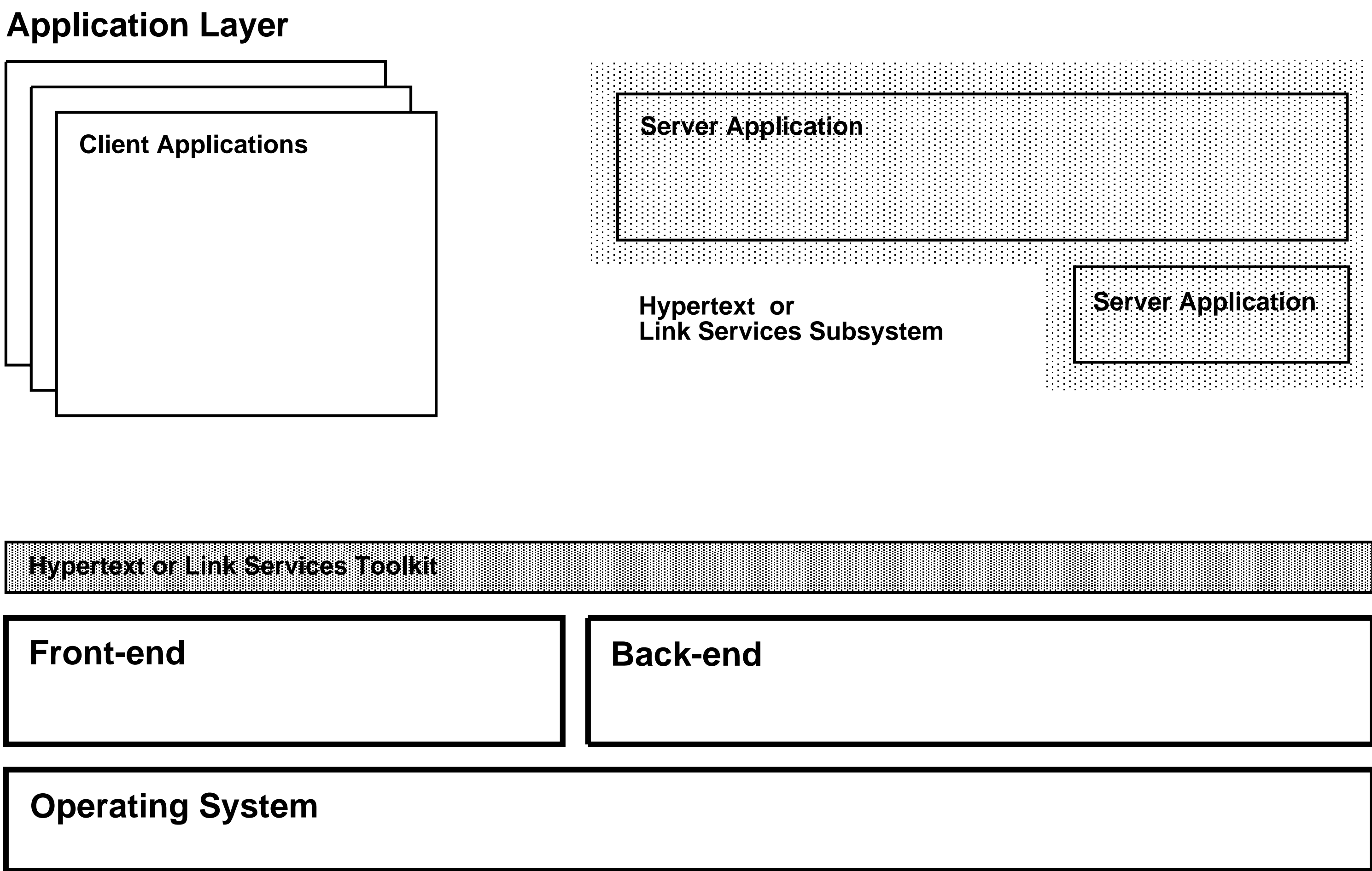

Figure 1-4. Non-Monolithic, Distributed Hypertext System Architecture

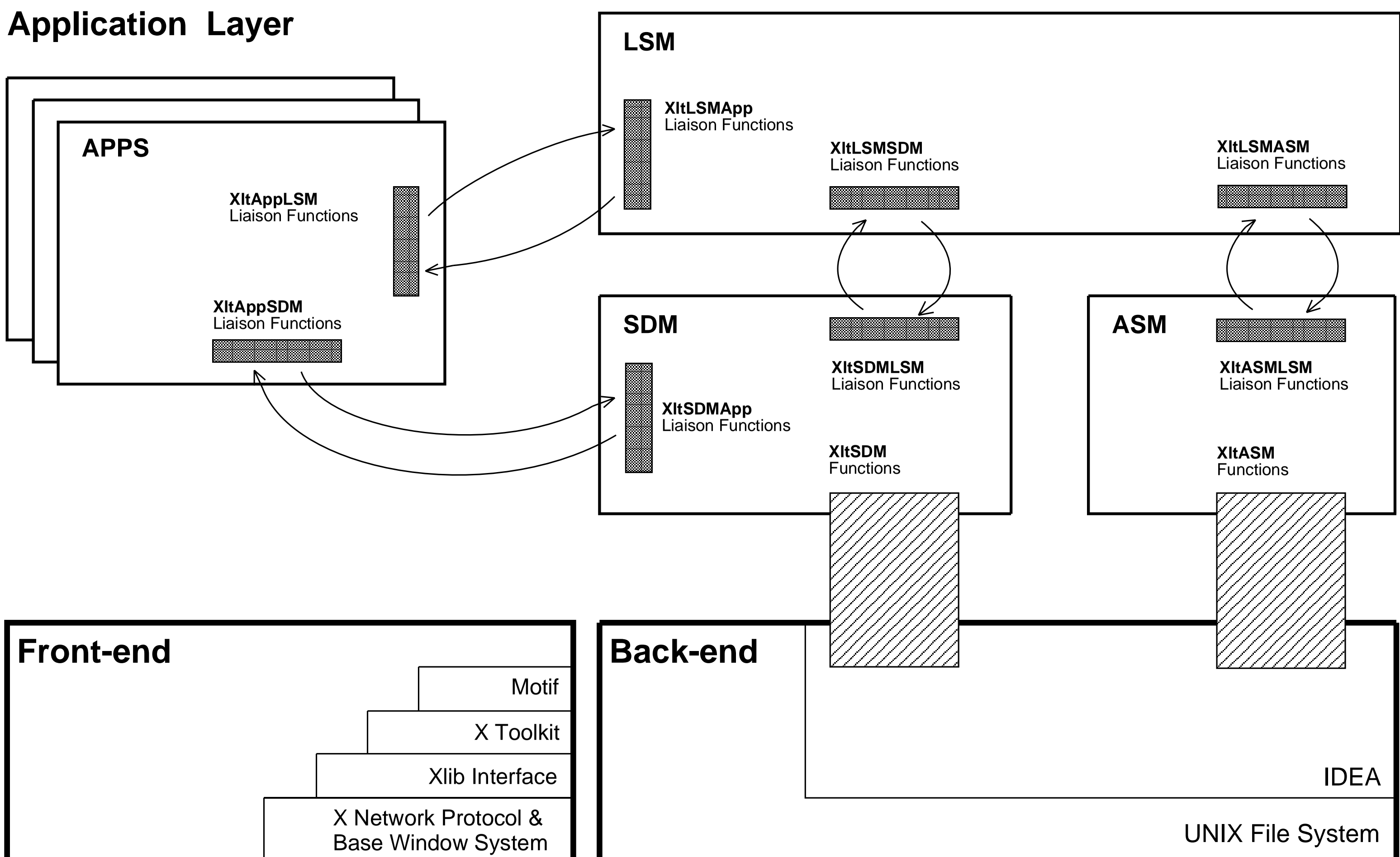

Figure 1-5. The HRL Prototype System Architecture

2. IMPLEMENTATION

The AsciiText Widget provided in the MIT distribution of X11 Release 4 was modified to incorporate the interapplication linking capabilities provided by Xlt. These modifications were, for the most part, the addition of a table of persistent selections (PSTable) and its associated methods needed to create and maintain selections in the current *hyperText* component being authored/browsed.

The ability to select text by highlighting with the pointer is among the existing capabilities in the text widget. These selections are only transient, however, used primarily in cut/copy/paste operations and only one selection can exist at a time. There is no mechanism for multiple selections to persist from one invocation of the application to the next. Persistent selection, a key requirement for building hypertext documents, is the primary requirement satisfied by the modifications made to the text widget in this project. Once text items are made persistent, they can have anchors attached and links to other associated applications can be created. Operations to follow these links may be invoked to bring up other applications for displaying components at the link's destination in a hypertext windowing environment. The integration of the text widget with Xlt demonstrates these capabilities in the *hyperText* widget.

2.1 Brief Overview of the Athena Text Widget

Some familiarity with the X toolkit intrinsics, X toolkit widgets, and the X architecture is assumed in the following discussion. For a more thorough, rigorous treatment of this subject, the reader is referred to (NYE1, NYE2, ASE, SCHEIF).

The text widget description found in NYE2 refers to X11 release 3 and a few changes were made in release 4. Specifically, the *asciiString* and *asciiDisk* widget classes have been combined into an *asciiText* widget class. The *asciiText* widget allows one to specify the resource **XtNtype** which can have values *XawAsciiString* or *XawAsciiFile*. Behavior of either of these two R3 widgets is obtainable by setting this resource value at widget creation time. Although the class structures for these R3 widgets exist in the R4 source code, they should be considered obsolete.

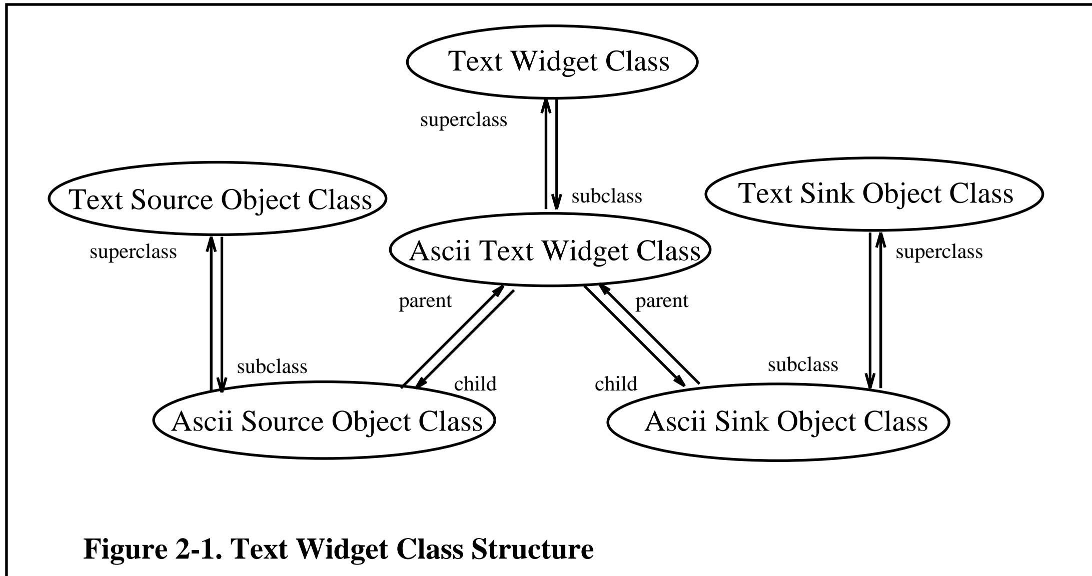

Figure 2-1. Text Widget Class Structure

Figure 2-1 shows the text widget class hierarchy. The *asciiText* widget class is a subclass of *text* widget class. In addition to inheriting behavior from this superclass, it also includes a *source* and *sink* object as its two child widgets which are created automatically via its **initialize_hook** method during widget creation. The text *source* object encapsulates all functionality required to work with the source of the text data such as the reading and writing of files and the allocation of memory for text strings. It is the source object which contains the **XtNtype** resource. The text sink object encapsulates all functionality required to maintain the display of the text in the widget's window.

As mentioned above, there are two distinct types for the text source. The **XtNtype** resource can be specified as *XawAsciiString* or *XawAsciiFile* when the widget is instantiated. If *XawAsciiString* is specified, the current string associated with the **XtNstring** resource of the *text* widget is used as the text source data. When *XawAsciiFile* is specified however, the current string associated with the **XtNstring** resource is taken as a filename whose contents are used as the text source data.

The text *sink* allows displaying of selections of text items within the *text* widget's window. This is accomplished by dragging the pointer over the desired text as is typically done in most X window

applications that manipulate text data. This selection is stored as the PRIMARY selection and only one PRIMARY selection exists for the X server.

A recursive algorithm is used to display text in the widget's window. To redraw text in the widget's window, the **DisplayText** method for the *text* widget class is first called with a start and an end text position. These are offsets into the *source* string. These values are compared to the offsets of the PRIMARY selection if it exists. When some but not all of the text is currently selected, three recursive calls are made which split the offsets into before, during, and after the text selection. If no overlap occurs or all text is selected, the **DisplayText** method of the *ascii sink* object class is called. A flag is passed to this method for highlighting the selection when the offsets lie within selection text.

2.2 Methods, Procedures, and Design Objectives

Before any changes to the source code were attempted, the following guidelines were formulated in order to achieve the best results:

1. Carefully examine the text widget source code. Gain an understanding of the methodology employed and design for consistency with this methodology.

2. Minimize the degree of modifications made to existing functionality. Hesitate before correcting any perceived bugs in the existing code.

3. Do not alter the include file dependencies in the existing text widget C files. Only newly created header files may be added and only if absolutely necessary. Design for ways to minimize this.

4. Keep key functionality specific to persistent selection apart from the existing text widget source code. These should be in newly created files for possible reuse in other sources. The same applies to integration with link services.

5. Allow the existing code and flow of control to remain in control. Modifications to the widget code should consist mainly of the insertion of hooks in only the right places.

6. Search for existing functionality within the text widget code that can be reused before writing new code. This is especially important in the display methods.

Items 2 and 3 can result in conflicting constraints when an extension in functionality is made that belongs in one source file yet requires a reference to an object with its definition found in one of

the headers not currently included. In situations like this, item 3 should be given precedence i.e., abstract away the direct reference by implementing part of the desired functionality in the other source file that contains functionality more directly related to the object whose reference is required.

2.3 Modifications Made to the Athena Text Widget

This section covers the modifications made to each part of the Athena *text* widget. What follows is the result of examing the source as it was and determining which parts of it should contain what modifications. An attempt was made to have the required extensions blend in naturally with the implementation and, at the same time, provide for the management of persistent selections and the handshaking with link services. The result was a small number of new source files each involving the encapsulation of a new object or a set of interface routines between two objects. Appendices A through E document these new sources. Changes made to existing sources is documented in appendix F.

2.3.1 Extensions to the Text Widget

The **DisplayText** method mentioned earlier was modified to check for persistent selections. The recursive algorithm found for handling the PRIMARY selection text was employed here. The **DisplayText** method was renamed to **PSDisplayText** and a new recursive **DisplayText** method was added to check for persistent selections and recursively split the calls into before, during, and after for a run of text containing a persistent selection. When start and end position parameters given to the new **DisplayText** are found not to include or to only include a persistent selection, **DisplayText** calls **PSDisplayText** and a flag is passed to indicate the selection status (LINKED, ANCHORED, SELECTED, PERSISTENT, or NORMAL TEXT). This is the same flag passed on to the **DisplayText** method of the *ascii sink* as before, however, it now may contain these new values in addition to the values for highlighting the PRIMARY selection. The selection status is explained in the section of this document covering Xlt integration.

The preexisting recursion employed within **PSDisplayText** remains as the mechanism to handle displaying the PRIMARY selection. Text containing the PRIMARY selection will be broken up into three recursive calls as is done in the original text widget. When the text to be displayed is found to be entirely within the PRIMARY selection, **PSDisplayText** passes its own value for the flag passed to the **DisplayText** method of the *ascii sink*, thus overriding the flag value passed to it from the new **DisplayText** method of the *text* widget mentioned above. Otherwise, this value is used. In this way, displaying the PRIMARY selection is unaffected by the modifications made for handling persistent selection.

2.3.2 Translations

Four translations were added to the *text* widget to allow interactive authoring and browsing of *hyperText* components. The names given to these translations with their default bindings are listed in table 2-1. The bindings shown may be modified in the usual ways for widgets by adding statements to the .Xdefaults file or to the application defaults file of any application that uses the *hyperText* widget. Examples of this can be found in (NYE1).

Default Binding	Function
<Key>F10	add-PS-selection
Ctrl<Key>F10	del-PS-selection
<Key>F9	select-PS
Ctrl<Btn1Down>	follow-link

Table 2-1. Translations added to the Text Widget

The translation function *add-PS-selection* can be used to create a persistent selection. This is done by highlighting text with the mouse followed by pressing the F10 function key. This translation takes the PRIMARY selection (previously set by highlighting) and makes an entry in the PSTable using its start and end text positions.

The translation function *del-PS-selection* can be used to remove a persistent selection. This is done by placing the text insert cursor within the region of a persistent selection followed by holding down the control key while pressing the F10 function key.

Certain interactions with the LSM require specific persistent selections to be designated as the target for an LSM operation (e.g., attaching anchors). This can be accomplished by the *select-PS* translation function. By placing the insert cursor within a persistent selection and pressing the F9 function key, the persistent selection's SELECTED status can be toggled on or off depending on its previous state. More than one persistent selection can be selected this way before an LSM operation is initiated.

Following hypertext links is accomplished by the follow-link translation function. This is done by placing the insert cursor within the persistent selection whose link is to be followed and, afterwards, holding down the control key while pressing mouse button 1 (usually the leftmost button).

2.3.3 Extensions to the Text Sink Object

The mechanism for displaying selections was extended to persistent selections. For the *sink* object, this involved the addition of color resources to specify how these selections are displayed. For participating applications, persistent selections may have anchors attached to them and links may be attached to anchors. It is desired to be able to distinguish between these states for a given selection. Table 2-2 shows the color resources that have been added to the *text sink* object. The resources correspond to foreground and background colors for the four different states of a persistent selection (PERSISTENT, SELECTED, ANCHORED, and LINKED) to be explained in the section on Xlt integration.

Instance Name	Class Name
XltNpsForeground	XltCPsForeground
XltNpsBackground	XltCPsBackground
XltNselectForeground	XltCSelectForeground
XltNselectBackground	XltCSelectBackground
XltNanchorForeground	XltCAnchorForeground
XltNanchorBackground	XltCAnchorBackground
XltNlinkForeground	XltCLinkForeground
XltNlinkBackground	XltCLinkBackground

Table 2-2. Color Resources for Persistent Selections

These resources can be specified in the usual ways for widgets. The application defaults file for any application using the text widget or the users .Xdefaults file may specify values by omiting the XltN and XltC prefixes from the names above. Values for these resources can also be set with **XtSetValues** or at widget creation time. It is up to the user or application developer to determine the color combinations that are desired to show the different states for a persistent selection. A monochrome display will show all selections in inverse-video as is done for the PRIMARY selection.

The *TextSinkPart* structure was modified to hold the additional color resource values. However, the displaying of selected text is accomplished by using the appropriate graphics context (GC) in the **PaintText** method of the *ascii sink*. Four additional GC values were added to the *AsciiSinkPart* structure. **Initialize** and **SetValues** methods for the color resources will allocate appropriate GC's using the corresponding foreground and background values. When text is drawn or redrawn in the *text* widget's window, flags mentioned earlier that are passed to the **DisplayText** method of the *ascii sink* are used to determine which GC is passed to the **PaintText** method.

2.3.4 Extensions to the Text Source Object

A third type, *XltPSComponent*, was added to the **XtNtype** resource. When this type is specified at widget creation, the string associated with the **XtNstring** resource is taken as the component name to retrieve from the SDM. Also, specifying this type causes the *text* widget **Realize** method to attempt registration of the widget with Xlt. Likewise, the **Destroy** method will attempt unregistration.

The *AsciiSrcPart* structure was modified to include a pointer to the table of persistent selections (PSTable). When a component is retrieved from the SDM, the SDM returns an object as a string of bytes and its length. These objects contain the PSTable as the first bytes in the string followed by the text data as the remaining bytes. Figure 2-2 shows the object format for *hyperText* widget components stored in the SDM.

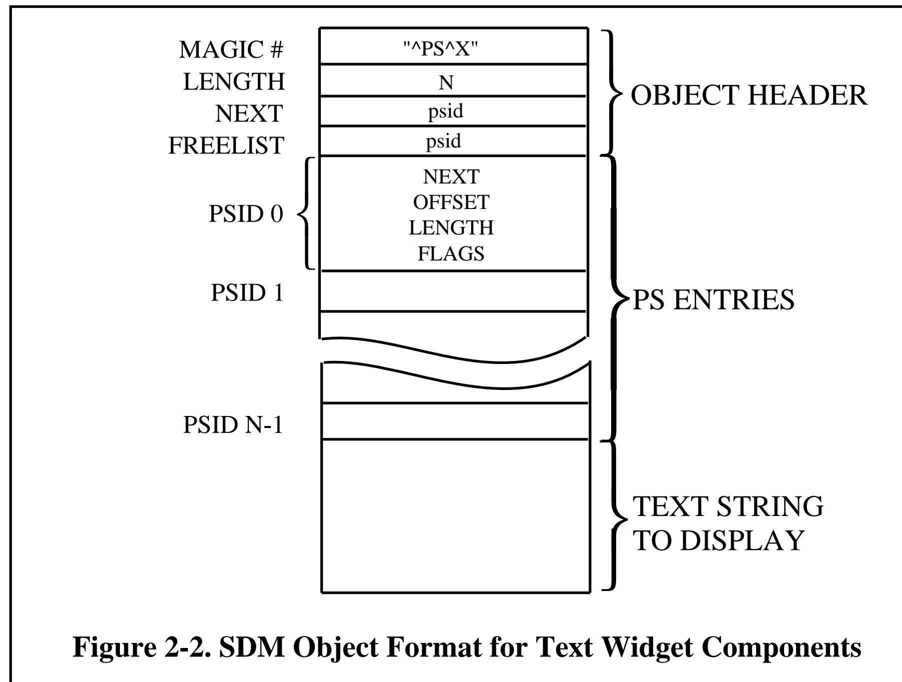

Figure 2-2. SDM Object Format for Text Widget Components

Object Serving with the SDM

A source file (PSCompónts.c) was created to handle the storing and retrieving of components from the SDM. This file includes references to the *AsciiSrcObject* and *Xlt data* types. The parent of the *ascii source* (the *text* widget) is referenced only to obtain the *text* widget's window and display connection which are used by the object server code provided by Xlt. A summary of this file's content is given in appendix A.

The first four bytes (MAGIC #) of the *hyperText* widget component are used to authenticate the object retrieved from the SDM. If <Control-P, S, Control-X, null> are not found to be the first four bytes of the object, the entire string will be imported as an ASCII file. Otherwise, upon retrieval of an object, the PSTable is copied from the object into an allocated block pointed to by the pstable member of the *AsciiSrcPart* structure while the remaining bytes are copied into the string member of the *AsciiSrcPart* structure as is done for *XawAsciiString* types. Conversely, when storing an

object, the magic number, PSTable, and text strings are pieced together according to the above format and given to the SDM.

PSTable Operations

A second source file (PSTable.c) was added for performing operations on the PSTable. This file contains references to the PSTable structure only; it is entirely independent of the *text* widget sources and has no knowledge of Xlib/Xt. Its functionality could be redesigned and rewritten without any affect to the text widget code provided that the procedural interface remained the same. Conversely, as a result of this independence, it can be reused in other sources (e.g., the Motif *text* widget). A summary this file's content is given in appendix B.

Access to the PSTable from sources having no knowledge of the private structures of the *AsciiSrcObject* was provided in a third source file (XltAsciiSrc.c). This file contains functionality needed by both the parent widget of the *AsciiSrcObject* (the *text* widget) and the LSM to be discussed later. The **DisplayText** method of the *text* widget call functionality in XltAsciiSrc.c to get lengths and offsets of persistent selections; the translations added to *text* widget rely on functionality to access and modify the PSTable; operations initiated by LSM messages call functionality in this file to handle the details. A summary of this file's contents is given in appendix C.

2.4 Integration with Xlt

Three status bits have been associated with a persistent selection. They are SELECTED, ANCHORED, and LINKED. A persistent selection with none of these status bits set is said to have PERSISTENT status. LINKED status is used to indicate that a persistent selection has a hypertext link associated with it. LINKED status implies ANCHORED status. ANCHORED status is used to indicate that a persistent selection has a hypertext anchor associated with it. ~ANCHORED status implies ~LINKED status. SELECTED status is used to allow the user to pick selections for specific LSM operations. SELECTED status works independently of anchors and links although a persistent selection will have this status bit cleared after an LSM operation that needs SELECTED

status in order for the operation to be performed on the selection. This is done for user convenience.

Registration of **psedit** (formerly **xedit**) with the Xlt was done entirely within the text widget. A forth source file (XltRegister.c) was created which includes references to the Xt intrinsics and Xlt only. It is mainly concerned with registering and unregistering a widget's window with the Xlt and adding an event handler for the widget to receive property notify events on this window. A summary of this file's contents is given in appendix D.

The file XltAsciiSrc.c contains references to the *AsciiSrcObject* and *Xlt data* types. It provides a level of abstraction between the operations defined in XltRegister.c (which can be applied to any widget class) and operations on the PSTable's private structures. When an LSM operation (sent via property notify events) results in a modification to the PSTable, the event handler in XltRegister.c will call procedures in XltAsciiSrc.c to manage these operations on its behalf.

Communication between the Xlt and an application that uses the text widget is accomplished via X window properties. Xlt sends messages to the text widget by writing data to the _XLT_LSM_APP property of the widget's window. Xt takes care of dispatching property notify events to the *text* widget's event handler whenever a window property changes. These messages may (1) direct the text widget to perform operations on specified persistent selections, (2) ask the application to inform the LSM of the status changes made to SELECTED persistent selections after an LSM operation, or (3) inform the application of global state changes that take effect during a session with the Xlt. A summary of these messages is given in appendix E.

3. REQUIREMENTS SUMMARY

Of the existing text widget source files from the MIT distribution of X11 Release 4, there were 3075 lines of code including comments in files that needed no modifications. A total of 7877 lines of code made up the files that needed modifications. After modifications were made to these files, this total reached 8125. Of the newly created modules, 326 lines of code were necessary for the definition and maintenance of the *hyperText* widget's PSTable, 290 lines were required for storing/retrieving components to/from the SDM, 239 lines were necessary to perform handshaking with the LSM, and 363 lines were required to integrate the ascii source object's PSTable with Xlt and Xt, thereby, providing the capabilities for authoring and browsing hypertext components.

APPENDIX A

SDM ⇔ AsciiSrcObject
Integration

APPENDIX A

static void PSGetComponent(text,compname,object,size)

Widget *text*;
char **compname*;
char ***object*;
int **size*;

This routine takes a widget id *text* and a component name *compname* and returns a pointer to an object and its size in bytes. The object is retrieved directly from the SDM by resolving the component name with **XltAppSDMResolve** to get an object id and then calling **XltAppSDMRetrieve** with that id. **XltAppSDMRetrieve** fills in values for *object* and *size*. The widget id is used to obtain the window id and display connection needed in the Xlt calls.

void PSLoadComponent(src)

AsciiSrcObject *src*;

This routine takes an AsciiSrcObject *src* and loads the component into it from the SDM. The string member of the source object is used as the component name. The component is retrieved by calling **PSGetComponent**. The component returned is parsed to obtain the PSTable structure and string member of the *AsciiSrcObject*.

static void PSReplaceComponent(text,compid)

Widget *text*;
COMPID *compid*;

This routine takes a text widget id *text* and an SDM component id *compid* in order to perform component id to component name resolution followed by setting the **XtNstring** resource of the *text* widget to this component name. This will trigger the loading of the component into the *text* widget.

static void PSPutComponent(text,compname,object,size)

Widget *text*;
char **compname*;
char **object*;
int *size*;

This routine takes a widget id *text*, component name *compname*, pointer to an *object*, and the object *size* in bytes and stores the object into the SDM. **XltAppSDMResolve** is used to obtain an object id from the component name. If no object exists, **XltAppSDMCreate** is called to create a new object having this component name. Otherwise, **XltAppSDMStore** is called to save the component in the SDM. The widget id is needed to obtain the window id and display connection for the Xlt library calls.

int PSStoreComponent(src,compname,buf)

AsciiSrcObject *src*;
char **compname*;
char **buf*;

This routine takes an AsciiSrcObject *src*, component name *compname*, and a text string *buf* and stores the component into the SDM. The text string and PSTable structure of the source object are combined to produce a character array containing the component to save and having the SDM object format described above. The component is stored by calling **PSPutComponent**. A return value of TRUE was supplied here for consistency with the *XawAsciiFile* counterpart of this *AsciiSrcObject* method.

APPENDIX B

Persistent Selection
Data Management

APPENDIX B

PSTable

```
typedef struct {
        PSID        next;
        PSOffset    psoffset;   /* current text position in component */
        PSLength    pslength;   /* current length of persistent selection */
        PSFlags     flags;      /* persistent selection status */
} PSEntry;

typedef struct {
        PSTableLen  length,     /* length of entries in use */
                    allocated;  /* size of array pointed to by entries */
        PSID        free, next; /* indices into entries */
        PSEntry     *entries;   /* table entries */
        COMPID      compid;     /* component id currently loaded */
} PSTable;
```

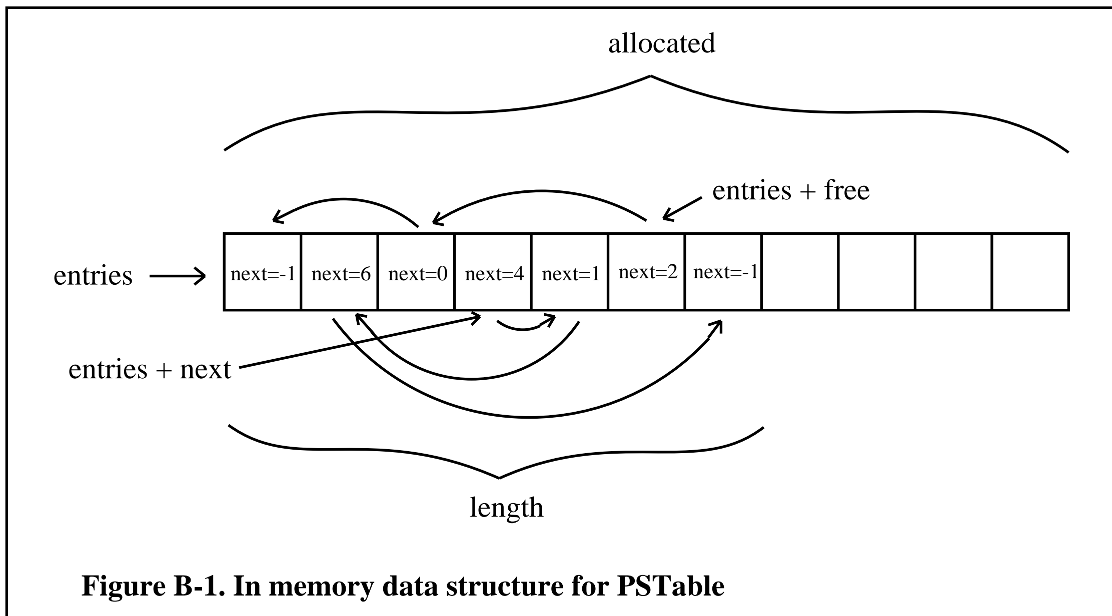

Figure B-1. In memory data structure for PSTable

Figure B-1 shows the in memory organization of the PSTable. An array of table entries is allocated and grows as necessary to accommodate the persistent selections of a component. Two internal lists are maintained in the array. The free list holds deleted persistent selections and the next list holds all currently active persistent selections. When a new persistent selection is created, the free

list is used first if it is not empty. Otherwise the length is increased by one and if necessary, the array will be reallocated to twice its current size.

void PSAddSelection(pstable,offset,length)

PSTable **pstable*;
PSOffset *offset*;
PSLength *length*;

This routine is passed a pointer to the PSTable *pstable* and the *length* and *offset* values for a new selection to be entered into the table. The flags field of the entry is initialized to zero.

static void PSDeleteSelection(pstable,psid)

PSTable **pstable*;
PSID *psid*;

This routine removes the persistent selection entry having PSID *psid* from the persistent selection table pointed to by *pstable*.

PSID PSFindSelectionAt(pstable,pos)

PSTable **pstable*;
int *pos*;

This routine searches the PSTable pointed to by *pstable* for an entry where *pos* is within its span of text. The span of text is defined as the text positions x such that offset <= x < offset + length where offset and length are the fields of the PSTable entry in question. The first such entry found is returned.

void PSDelSelectionAt(pstable,pos,PSstart,PSend)

PSTable **pstable*;
int *pos*,**PSstart*,**PSend*;

This routine searches the PSTable *pstable* as in PSFindSelectionAt and removes the first entry found that contains position *pos*. The start and end text positions of this entry are returned via the pointers *PSstart* and *PSend* so the corresponding text can be repainted if necessary.

void PSUpdateTable(pstable,startPos,endPos,newlength)

PSTable **pstable*;
int *startPos*,*endPos*,*newlength*;

This routine modifies all PSTable entries pointed to by *pstable* according to edits made to the source string. *startPos* and *endPos* correspond to a span of text within the source string that has changed. *newlength* is the length of that span of text after it was changed. All offsets and length fields in *pstable* are adjusted accordingly.

void PSGetSelection(pstable,startPos,endPos,PSstart,PSend,flags)

PSTable **pstable*;
int *startPos*,*endPos*,**PSstart*,**PSend*;
PSFlags **flags*;

This routine searches the PSTable pointed to by *pstable* and returns the span of text for the first entry found that overlaps the span of text specified by *startPos* and *endPos*. The *flags* field is also returned so that appropriate highlighting can be determined by the **DisplayText** methods. *PSstart* and *PSend* return the span of text for the persistent selection or -1 if none found.

APPENDIX C

Xt Widget, Xlt ⇔ AsciiSrcObject
Integration

APPENDIX C

void GetPersistentSelection(src,pos1,pos2,PSstart,PSend,hilite)

AsciiSrcObject *src*;
XawTextPosition *pos1*,*pos2*,**PSstart*,**PSend*;
int **hilite*;

This routine searches the PSTable in AsciiSrcObject *src* for the first persistent selection that overlaps text position *pos1* to *pos2*. *PSstart* and *PSend* are set to the region of text occupied by the persistent selection found. *hilite* is set to indicate the status of the persistent selection for painting purposes.

PSID FindPersistentSelection(src,pos)

AsciiSrcObject *src*;
XawTextPosition *pos*;

This routine searches the PSTable in AsciiSrcObject *src* for the first entry whose region of text overlaps with text position *pos*. The corresponding PSID is returned or -1 when none found.

XawTextPosition PersistentSelectionTextPosition(src,psid)

AsciiSrcObject *src*;
PSID *psid*;

This routine returns the text offset of PSTable entry *psid* in AsciiSrcObject *src* . Used to allow scrolling to the persistent selection at the destination of a *follow-link* operation.

COMPID PersistentSelectionComponent(src)

AsciiSrcObject *src*;

This routine returns the component ID for the current component loaded in AsciiSrcObject *src* . Used when processing a list of *triples*.

void SelectPersistentSelection(src,pos)

AsciiSrcObject *src*;
XawTextPosition *pos*;

This routine toggles the SELECT_BIT of the first persistent selection found in AsciiSrcObject *src* to overlap text position *pos*. UpdatePSText is called to cause repainting. Used by *select-PS* translation.

void DelPersistentSelection(src,pos)

AsciiSrcObject *src*;
XawTextPosition *pos*;

This routine deletes the first persistent selection found in AsciiSrcObject *src* to overlap text position *pos*. Activates widget's **sourceChanged** callback and repaints if necessary. Used by *del-PS-selection* translation.

void AddPersistentSelection(src,psoffset,pslength)

AsciiSrcObject *src*;
PSOffset *psoffset*;
PSLength *pslength*;

This routine adds a new persistent selection to the PSTable in AsciiSrcObject *src* having *psoffset* and *pslength*. Activates widget's **sourceChanged** callback. Used by *add-PS-selection* translation.

void CheckXltRegistration(src)

AsciiSrcObject *src*;

This routine registers an *XltPSComponent* type *text* widget with the Xlt. Used by the *text* widget's **Realize** method. The **XtNtype** resource in AsciiSrcObject *src* is checked before registration is attempted.

void CheckXltUnregistration(src)

AsciiSrcObject *src*;

This routine unregisters an *XltPSComponent* type *text* widget with the Xlt. Used by the *text* widget's **Destroy** method. The **XtNtype** resource in AsciiSrcObject *src* is checked before unregistration is attempted.

void anchors_set_clear(src,triples,i,set,all)

AsciiSrcObject *src*;
XltLSTripleType **triples*;
int *i*; /* size of triples array */
Boolean *set*,*all*;

This routine sets or clears (depending on the value of *set*) the ANCHOR_BIT for all PSTable entries of the *triples* list having the component ID matching the current component loaded. If *all* is true, all persistent selections of the PSTable in AsciiSrcObject *src* are affected (i.e., the *triples* list is ignored).

Note: Clearing the ANCHOR_BIT implies clearing the LINK_BIT.

int anchor_bits(src,triples,set)

AsciiSrcObject *src*;
XltLSTripleType ***triples*;
Boolean *set*;

This routine attaches or detaches (depending on the value of *set*) anchors to all persistent selections of the PSTable in AsciiSrcObject *src* that have SELECTED status. A list of *triples* is constructed to allow reporting the new anchors to the LSM. The return value is the size of the array pointed to by *triples*. The PSTable entries' SELECTED status are automatically cleared.

void links_set_clear(src,triples,i,set,all)

AsciiSrcObject *src*;
XltLSTripleType **triples*;
int *i*; /* size of triples array */
Boolean *set*,*all*;

This routine sets or clears (depending on the value of *set*) the LINK_BIT for all PSTable entries in the *triples* list having the component ID matching the current component loaded. If *all* is true, all persistent selections in AsciiSrcObject *src* are affected (i.e., the *triples* list is ignored).

Note: Setting the LINK_BIT implies setting the ANCHOR_BIT.

int link_bits(src,triples,set)

AsciiSrcObject *src*;
XltLSTripleType ***triples*;
Boolean *set*;

This routine attaches links to all persistent selections that have anchors without links or detaches links for all persistent selections in AsciiSrcObject *src* having SELECTED status. The value of *set* determines which operation to perform. A list of *triples* is constructed when detaching links to allow reporting the removed links to the LSM. The return value is the size of the array pointed to by *triples*. The PSTable entries' SELECTED status are automatically cleared.

void XltFollowLink(src,psid)

AsciiSrcObject *src*;
PSID *psid*;

This routine constructs a *triple* (application ID, component ID, psid) to give the LSM in a follow link message. The Xlt library function to follow a link is called with this triple. The application ID is an agreed upon value determined for applications using the text widget; the component ID is that of the current component loaded in AsciiSrcObject *src*; *psid* is passed from the *follow-link* translation function that calls this routine.

APPENDIX D

LSM ⇔ Xt Widget Integration

APPENDIX D

void PSpropertyChange(w,data,event)

Widget *w*;
caddr_t *data*;
XEvent **event*;

This routine is the property notify *event* handler for the text widget *w*. All ongoing communication between the *text* widget and the Link Services Manager is done with this routine. The LSM changes properties on the *text* widget's window. The X server notifies the application of these changes via the display connection managed by the Xt event handler. An XAtom _XLT_LSM_APP is defined to be used as the window property name that the LSM uses to send messages to the *text* widget. A summary of Xlt to application handshaking is given in appendix E.

Note: *data* is not used but must be present to conform Xt's event handler procedural interface.

void PSRegister(widget)

Widget *widget*;

This routine registers a widget *w* with the Xlt. Its main concern is adding the property notify event handler, initializing communication with Xlt, and opening a session with the SDM. The widget's window is used to perform registration.

void PSUnregister(widget)

Widget *widget*;

This routine performs unregistration of a widget *w* with the Xlt. The session with the SDM is closed and the property notify event handler is removed from the widget.

APPENDIX E

LSM ⇔ Application
Session Management

APPENDIX E

XltMsgLSMAppRegRegisterAcknowledge

This message is sent after the LSM has registered the application for link services. A flag is set upon receipt of this message.

XltMsgLSMAppRegShutdown

This message is sent when the LSM exits. A flag is cleared upon receipt of this message.

XltMsgLSMAppLSAttachAnchor

This message is sent to inform the application that it may attach anchors to selections. The *text* widget informs the LSM of the anchors created from this action by sending *triples* back to the LSM. Each *triple* in the array contains the application ID, component ID, and persistent selection ID.

A persistent selection must be picked to receive an anchor. This is accomplished by *select-PS*. When the XltMsgLSMAppLSAttachAnchor message is received, all such persistent selections will be given anchors and the array of *triples* will be constructed in order to inform the LSM of the newly created anchors. These persistent selections will have their status set to ANCHORED and ~SELECTED.

XltMsgLSMAppLSSetAnchorBits

In this message, the LSM sends a list of *triples* indicating which persistent selections are to have their status set to ANCHORED. The LSM uses this to inform the widget of anchor positions after a component is loaded.

XltMsgLSMAppLSDeleteAnchor

This message is the reverse of the XltMsgLSMAppLSAttachAnchor message. Persistent selections with anchors can be selected to have their anchors removed. Links cannot exist without anchors and this operation also removes LINKED status from the selections.

XltMsgLSMAppLSClearAnchorBits

In this message, the LSM sends a list of *triples* indicating which persistent selections are to have there status set from ANCHORED to PERSISTENT. If the selection status is SELECTED and ANCHORED, it will become SELECTED and PERSISTENT . Links cannot exist without anchors and this operation also removes LINKED status from the selections.

XltMsgLSMAppLSAttachLink

This message is used by the LSM to inform the application to set all persistent selections with ANCHORED status to LINKED status. Persistent selections with LINKED status can be used to follow hypertext links.

XltMsgLSMAppLSSetLinkBits

In this message, the LSM sends a list of triples indicating which persistent selections are to have there status set to LINKED. The LSM uses this to inform the widget of link positions after a component is loaded. All links must have anchors and therefore this operation also sets ANCHORED status.

XltMsgLSMAppLSDeleteLink

This message is the reverse of the XltMsgLSMAppLSAttachLink message. Persistent selections with links must be picked to have their links removed. This is accomplished by *select-PS*.

XltMsgLSMAppLSClearLinkBits

In this message, the LSM sends a list of triples indicating which persistent selections are to have there status set from LINKED to ANCHORED. If any of these selections having LINKED status also has SELECTED status, it will become SELECTED and ANCHORED.

XltMsgLSMAppLSCloseContext

This message is used to inform the application that the current context is being closed. This means that all anchors and links must disappear. All persistent selection states are set to ~ANCHORED and ~LINKED. SELECTED status will be left on for those persistent selection having such status.

XltMsgLSMAppLSGotoPS

This message is used by the LSM to inform the application that a new component is to be loaded. This is done usually after a follow link message has been sent by some application and the LSM has initialized the other side of the link being followed.

XltMsgLSMAppLSTurnOnBrowsing

This message is sent by the LSM to indicate that it is responsive to follow link messages.

XltMsgLSMAppLSTurnOffBrowsing

This message is sent by the LSM to indicate that it is not responsive to follow link messages.

XltMsgLSMAppLSTurnOnAnchorMarkers

This message is sent by the LSM to indicate that the displaying of anchors should be allowed. The DisplayText methods will behave accordingly.

XltMsgLSMAppLSTurnOffAnchorMarkers

This message is sent by the LSM to indicate that the displaying of anchors should be suppressed. The DisplayText methods will behave accordingly.

XltMsgLSMAppLSTurnOnLinkMarkers

This message is sent by the LSM to indicate that the displaying of links should be allowed. The DisplayText methods will behave accordingly.

XltMsgLSMAppLSTurnOffLinkMarkers

This message is sent by the LSM to indicate that the displaying of links should be suppressed. The DisplayText methods will behave accordingly.

APPENDIX F

Athena Text Widget
Source Extensions

APPENDIX F

Text.h

This file contains the definitions of resource names for the *text* widget. The color resource names discussed in section 2.3.3 are defined here.

Text.c

This file contains the **Realize**, **Destroy**, and **DisplayText** methods of the *text* widget class. Registration was added to widget realization and unregistration was added to destruction for the case when the **XtNtype** resource is *XltPSComponent*. The modifications made to the **DisplayText** method discussed in section 2.3.1 were done in this file.

A utility function **UpdatePSText** was added to repaint persistent selections after state changes.

```
void UpdatePSText(w,pos1,pos2)
Widget w;
XawTextPosition pos1,pos2;
```

TextTr.c

This file contains the default key bindings for the widget's translations. The bindings for the translations discussed in section 2.3.2 were added to this file. Also, preexisting bindings for mouse button 1 were given the ~Ctrl modifier so as not to conflict with the *follow-link* binding.

TextAction.c

This file contains the mapping from translation function names to their corresponding C function. An array of *XtActionsRec* type contains strings paired with pointers to functions to achieve this mapping. The following actions were added.

```
static void add_PS_selection(w, event)
Widget w;
XEvent event;

static void del_PS_selection(w, event)
Widget w;
XEvent event;

static void SelectPS(w, event)
Widget w;
XEvent event;

static void FollowLink(w, event)
Widget w;
XEvent event;
```

The LSM may send messages to turn on/off all link or anchor markers. An action to redraw the display **RedrawDisplay** exists here and its default binding is Ctrl<Key>L. The utility routine PSUpdateWindow was added here to call **RedrawDisplay** text whenever the window needs updating because of global state changes.

```
void PSUpdateWindow(w)
Widget w;
```

TextSinkP.h

This file contains the definitions for private data of the *text sink* object. The pixel resources for foreground and background colors discussed in section 2.3.3 were defined here.

TextSink.c

This file contains a table mapping resource names to their data structures in the *text sink* object's private data. The mapping of color resource names in Text.h to the corresponding pixel resources in TextSinkP.h were added to the table here.

AsciiSinkP.h

This file contains the definitions for private data of the *ascii sink* object. The GC resources for corresponding foreground and background pixels defined in the *text sink* object were defined here.

AsciiSink.c

This file contains the methods of the *ascii sink* object. The **DisplayText** method of the *ascii sink* is passed a flag mentioned in section 2.3.1. The flag is used here to select the appropriate GC for repainting text. The **Initialize**, **Destroy**, and **SetValues** methods allocate and deallocate these GC's.

AsciiSrc.h

This file contains resource names for the *ascii source* object. The resource name *XltPSComponent* was added here for the **XtNtype** resource.

AsciiSrcP.h

This file contains the definitions for private data of the *ascii source* object. A pointer to the PSTable was added to this object.

AsciiSrc.c

This file contains the methods of the *ascii source* object. The type converter that handles the **XtNtype** resource setting was modified to include *XltPSComponent*. The **Initialize** method was modified to initialize the PSTable. The **ReplaceText** method was modified to update the PSTable after edits. The methods for handling text files were modified to perform object serving with the SDM when the **XtNtype** resource is *XltPSComponent*.

GLOSSARY

anchor

An attachment given to a persistent selection indicating that it is primed to receive a link. The Xlt uses the concept of an anchor in order for an application to change a persistent selection's status so as to make a distinction between persistent selections that are targeted by the Xlt to receive links and those that are not when the next attach link operation is executed.

application

An executable program that is invoked from a computer display. The program is used as a tool to perform operations on one or more components, e.g. a text editor can be used to perform editing operations on text files.

component

An object for storing information. Components are created and modified by applications. The system employed here uses an object database for storing and retrieving an application's components.

link

A pathway from one or more components to one or more other components. Links are created within a hypertext system by established mechanisms. The architecture employed in the Xlt uses persistent selections as places to attach anchors in components. A set of anchors can be linked to another set of anchors. A link is said to have been "followed" when the user picks the persistent selection (by whatever mechanism the application provides) having the anchor attached with the link. When a link is followed the Xlt will invoke all applications for the set of anchors on the other side of the link. Xlt will cause these applications to be initialized with the corresponding components that contain the anchors of the other side.

persistent selection

A region within a component made visibly distinct from its surroundings; these are used in hypertext systems to provide the necessary support for the interactive, on-line passage from a component in one application to one or more components in other applications.

triple

A data structure containing an application ID, a component ID, and a persistent selection ID. Triples are used to specify the sources and destinations of hypertext links in the Xlt.

REFERENCES

[KAC] Kacmar, C. J. 1990. *PROXHY: A Process-Oriented Extensible Hypertext Architecture*. Ph.D. dissertation. Dept. of Computer Science, Texas A&M University, College Station, Texas.

[NYE1] Nye, A. and O'Reilly, T. *X Toolkit Intrinsics Programming Manual*, O'Reilly & Associates Inc., Sebastopol CA, 1990

[NYE2] Nye, A. and O'Reilly, T. *X Toolkit Intrinsics Reference Manual*, O'Reilly & Associates Inc., Sebastopol CA, 1990

[SCHEIF] Scheifler, R.W. Gettys, J. Newman, R. *X Window System C Library and Protocol Reference*, Digital Press, Bedford MA, 1988

[ASE] Asente, P.J. Swick, R.R. *X Window System Toolkit: The Complete Programmer's Guide and Specification,* Digital Press, Bedford, MA 1990