The Multivalent Browser: A Platform for New Ideas

Thomas A. Phelps and Robert Wilensky
University of California, Berkeley
phelps@cs.berkeley.edu, wilensky@cs.berkeley.edu
Web site: http://www.cs.berkeley.edu/~phelps/Multivalent/

ABSTRACT

The Multivalent Browser is built on an architecture that separates functionality from concrete document format. Almost all functionality is made available via relatively small modules of code called behaviors that programmers can write to extend the core system. Behaviors can be as significant and powerful as parser-renderers for scanned paper, HTML, or TeX DVI; as fine-grained as hyperlinks, cookies, and the disabling of menu items; and as innovative or uncommon as in situ annotations, "lenses", collapsible outline displays, new GUI widgets, and Robust Hyperlink support. Behaviors can be combined in arbitrary groups for each individual document, in effect spontaneously creating a custom browser for every one. Common aspects of document functionality can be shared, so that, for example, the same behavior that handles multipage support for scanned paper documents also provides such support for DVI and PDF; similarly, the behaviors that support fine-grain annotation of HTML also support identical annotation on scanned paper, UNIX manual pages, DVI, and PDF.

We have designed and implemented this architecture, and implemented behaviors that support all of the above functionality and more. Here we describe the architecture that allows such power and fine-grained access, yet composes disparate behaviors and resolves their mutual conflicts.

Keywords

digital document architecture annotation scanned paper Multivalent behavior

Lexical signature

common lucene openeddocument decyphered multifont decypher xinclude annotatable carry

1 INTRODUCTION

Documents are idiosyncratic. Yet modern document formats are so complex (PDF, Microsoft Word, QuickTime) that picking a format is often tantamount to choosing a browser/editor/viewer with its packaged bundle of features and limitations. While most such systems are extensible, it is rare that experimenters can accomplish something surprising — in a web browser, everything reduces to some permutation of HTML. The enormous effort required to write a competitive viewer, even for an open standard such as HTML, makes it difficult for researchers to implement radically new ideas and more dangerously starts to restrict their thinking. End users have to hope that outstandingly useful ideas will be recognized by the large companies that control the viewers, and that these ideas eventually propagate across their many digital document systems: web browsing, word processing, PDF/PostScript viewer, email, and so on.

It is a primary goal of the Multivalent project to provide inventors with a system of sufficient power and fine-grained control that they find it an inviting platform for working out new ideas. Another goal, equally important, is to enable distribution of new ideas by not requiring source code changes to the core and by coordinating possibly conflicting behaviors from diverse parties.

To this end, we have developed an architecture with the following key features:

- a *document tree* core data structure sufficiently flexible to support scanned paper, HTML, UNIX manual pages, TeX DVI, PDF, and potentially any other concrete digital document format — as well as the system's graphical user interface widgets. Cross format support is crucial, for most of us regularly deal with many different document formats, such as HTML, PDF, Microsoft Word, PowerPoint, email, DVI, and so on, and any work on a system that supports just one neglects the other 75%.
- a well defined extension mechanism called *behaviors*, which can implement functionality as powerful as a new document format or as fine-grained as disabling menu items, all without modification of or special case support from the core system
- a behavior management scheme called *hub documents*, which list behaviors applicable to the system as a whole, various document genres, or specific documents, and whose manifestation in effect gives every document a custom browser
- a framework for open participation by behaviors in high-performance *low-level communication protocols*, which address all aspects of what we term the fundamental document lifecycle: restore from disk/network, build data structure, format document tree, paint document tree, process keyboard and mouse events. Conformance to protocols distinguishes the behavior from other object-oriented classes. To a remarkable degree this conformances suffices to compose behaviors without conflict.
- a framework for *high-level semantic events* so that logical activities of the document are open to participation by arbitrary behaviors, same as for low-level events
- a set of ad hoc communication mechanisms for highly specialized needs not addressed above

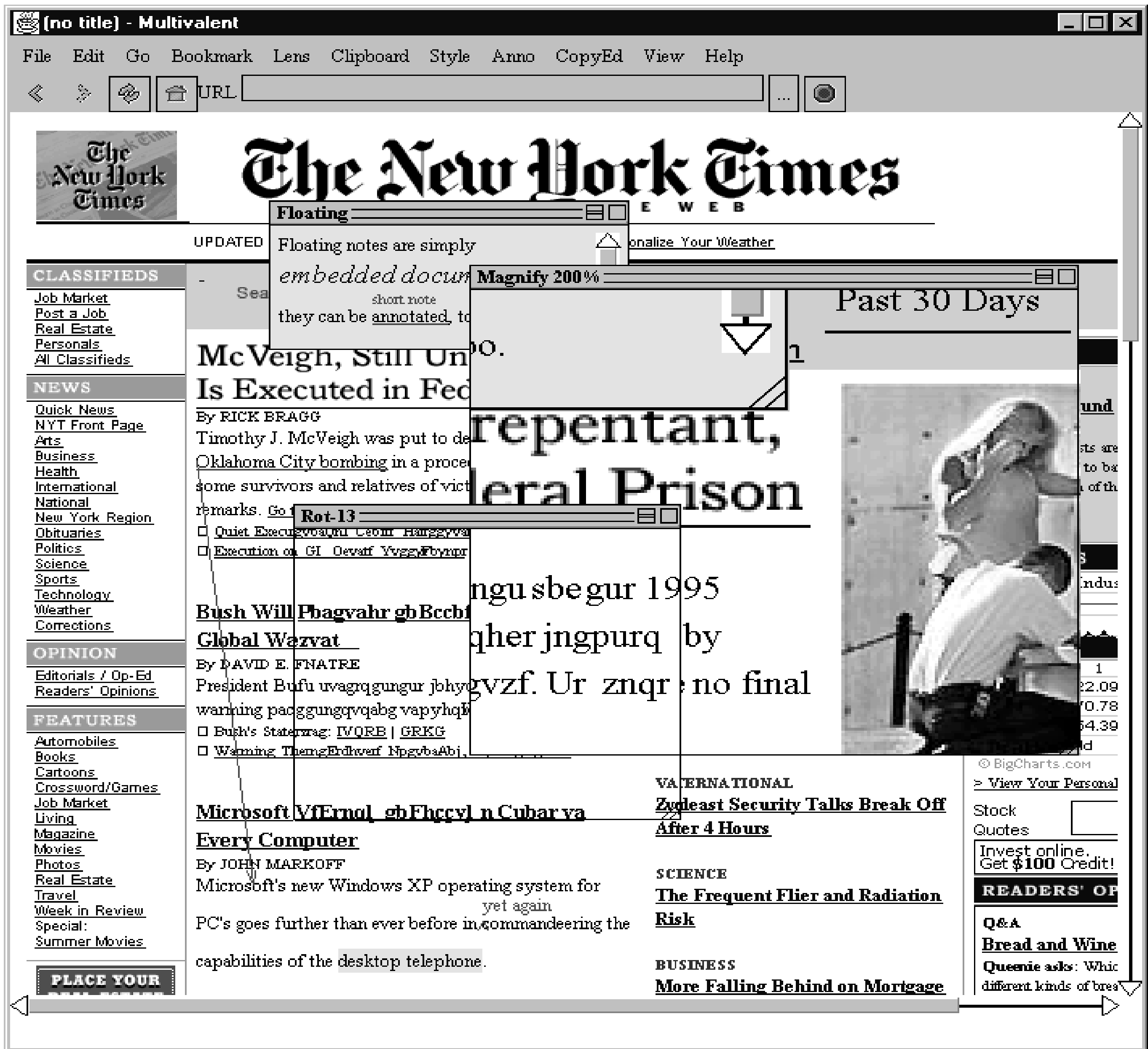
The architecture itself should appear simple, general: embodying the best practices of decades of research in digital documents with a clean implementation, but not introducing anything unproven. Indeed, a guiding principle is to condemn special cases, for if some feature requires special support from the core of the system and cannot be accomplished within the general architecture, then that is a sure sign that the architecture would fail to embrace some future new idea.

What's new is the extreme extent to which simple ideas are stressed, and the versatility of the gestalt. Results can be surprising. For example, on top of an familiar structural document tree with no special accommodation in the core, one can introduce lenses [2] — a generalization of the magnifying lens from paint programs, that can arbitrarily transform its content — and furthermore manage to compose the effects of overlapped lenses.

We argue for our architecture with a fully functional browser built according to its principles, which is described in the next section. The subsequent section documents the architecture, using features from the browser to show how simple, general architectural constructs can be exploited to varied, useful, and sometimes surprising ends. Not infrequently the architecture empowers novel, practical features not found anywhere else.

2 THE MULTIVALENT BROWSER

The screen dump below shows the running Multivalent Browser, a 700KB Java application. It happens to display an HTML page, but scanned paper, UNIX manual pages, TeX DVI, and PDF are also natively supported.



Various annotations have been made. The arrow from "Oklahoma" to "Microsoft" is a "move text" annotation; it is executable in that clicking on the source text will delete the underlined source text and reinsert it at the endpoint of the arrow. A highlight can be seen at the bottom of the page on "desktop telephone"; its color can be changed by alt-clicking on it and choosing a color from a pop-up menu. The floating note is in fact an embedded document, so it can be annotated itself. A short text string, seen at the bottom of the page and on the note, affects formatting of the document, opening up space between lines for the message. Annotations are robustly anchored to the page so that if the source of the page changes but the anchor points remain, they will still reattach correctly [15]. Annotations work equally well across the document format listed above.

Two lenses are active. The Magnify lens covers text, image, and part of the note. The Decypher lens, set to its Rot-13 setting, covers other text. Where Magnify and Decypher overlap, we obtain magnified decyphered text.

As a test of the architecture, all user-level features have been implemented as behaviors, with no privileges over those accorded to third party extensions. For example, the following behaviors and tree node types are some of the more than 150 packaged with the base system:

- Media adaptors: scanned paper (two kinds: XDOC and PDA), HTML 3.2 with most of CSS1, PDF 1.3, UNIX manual pages (with volume listings), TeX DVI, ASCII, Zip files (with extraction), local directories.
- Annotations: highlight, hyperlink, floating note, short comment, bold/italic/underline/..., font face, font size, foreground/background color. Executable Annotations: move text, replace with, bold/italic/ underline/..., delete, uppercase/ lowercase/ capitalize.
- Lenses: magnify, show OCR/image, decypher (rot-13, Pig Latin, ...), plain view, ruler, bounding boxes.
- Tools: table sorting, speed reading, send selection to a web server (as for dictionary definition or language translation), telephone touch tone of selection, scrollbar search result visualization, slide show, and a novel focus+context visualization called Notemarks [14].
- Multivalent-native GUI widgets: button, checkbutton, radiobutton, menu, scrollbar, single-line type-in box, multi-line type-in box, frame, dialog. Widgets compose, so rather than needing a parallel set of widget buttons to function within menus as other widget sets do, the standard button types can be used. Likewise, menus can be cascaded and function as both pulldown and popup.
- Debugging: Live document tree data structure display.

The browser, source code, and documentation are available at http://www.cs.berkeley.edu/~phelps/Multivalent/. It is Open Source.

3 MULTIVALENT ARCHITECTURE

This section documents the architectural features that enable the above demonstration. Whereas in the past editors needed to relentlessly bitblt already drawn portions of the screen (as in Lilac [3]) and share memory between screen display and document content (as in Bravo on an Alto [7]), today's machines are fast enough that the Multivalent architecture can heavily favor simplicity and flexibility over absolute performance.

3.1 Central Data Structure: The Document Tree

The central data structure is the document tree. It is the same data structure commonly found in digital document systems, with the standard set of navigation and tree management functions (adding, removing, querying children, and so on). Moreso than most such data structures, the Multivalent document tree was designed to remain simple and lightweight, as free as possible of special cases for efficiency or particular document formats.

Similar to other systems but perhaps with more militancy, the document tree directly represents the structure of a document. In the canonical example of a structured book, the tree root is labeled "book" and has children labeled "chapter", chapters are divided into sections, which in turn are divided into subsections, and so on. This is a straight parse tree for XML, a normalized parse tree for SGML, and a corrected parse tree for HTML (which adheres to the DTD, if possible given the errors in the page). For scanned paper, the representation might be physically based, with the page divided into regions (of type text or image), with text regions divided into paragraphs, paragraphs into lines, and lines into words (to the degree analysis software can discern such structures).

Internal nodes, which have child nodes, capture structural hierarchy. The usual root of the tree is an internal node type called `Document`, which has a URL, a style sheet, scrollable content, and holds the list of document-specific behaviors, if any. Medium-specific qualities of a document format are encapsulated in leaves: depending on its subtype a leaf can paint an image, a word from a manual page, or a scanned paper word (clipped image). (Presently, each word of text is given its own leaf node. This simplifies linebreaking and aligning words of a line flush to both margins, and aids hit detection; however, very long unpaginated documents use a lot of memory.) The division of document content into medium-independent internal nodes and medium-dependent leaves allows developers to write behaviors against an idealized abstract document tree and have the behavior operate on any concrete document format.

All nodes have the following properties, as summarized at right: name (the tag name in XML), baseline, horizontal alignment (top/ bottom/ middle/ none), vertical alignment (left/ right/ center/ none), and float side (left/ right/ none). Nodes carry a list of "observer" behaviors, which as described in the sections on high- and low-level communications notify those behaviors of any activity in the subtree. Two exceptions to the policy of no special cases are made for the sake of efficiency, the formatting dirty bit and the span summary.

Node Properties

name (tag name in XML)

links to parent and
(internal node only) children

bounding box, baseline

horizontal and
vertical alignment

float side (left/right/none)

observers

Special Cases for Efficiency

formatting dirty bit

span summary

In addition to representing the document's structure, tree nodes contain physical layout information, namely the bounding box of their contents. This allows behaviors to easily move between structural and physical representations. A search behavior could, say, exploit structure to examine only captions, and then display the results physically, scrolling and highlighting the hits. Another behavior could convert from a format expressed in semantic markup, such as XML, into a physically-based format, such as PDF.

Thus, in many respects the document tree is quite ordinary. But as the examples below demonstrate it is powerful, and since developers can introduce new node types as easily as behaviors, which is a level of extension seldom found in scripting languages, this versatility can be expanded.

3.1.1 Example: GUI.

A typical editor's graphical user interface, whether for text or CAD or other document type, has a menubar, a set of buttons on various toolbars and palettes, various other controls, and finally a large "canvas" area for display and direct manipulation of the document. This canvas is highly specialized, similar to other widgets only in that it is a rectangle that can be drawn upon.

The Multivalent Browser reverses this relationship. All widgets are simply specialized nodes, and everything is displayed in a master document tree — the graphical user interface (GUI) as well as the content of documents per se — both descending from a common absolute root node. Some "words" happen to have specialized behavior: click on some words and they display a different document (that's a hyperlink), click on other words and they fire a semantic event (that's a button). Rather than a parallel set of widget layout functions, the GUI uses the same layout already developed for documents, such as HTML's table. Because GUI widgets are ordinary nodes, they are controlled by style sheets.

We have implemented as node types a set of essentials widgets: button, checkbutton, radiobutton, scrollbar, scrolled pane, menu, type-in box, and dialog box. For HTML forms, where content and user interface are mixed, this approach results in an especially clean representation.

3.1.2 Example: embedded documents.

Since document content and widgets use the same uniform model, one type can easily embed the other. In other widget toolkits, button labels and dialog box text is usually displayed by a simple text layout, probably with an images allowed in fixed places, but it is limited.

Multivalent widgets can embed full HTML documents (or manual pages or scanned paper, should that ever prove useful), giving multifont text with images, even video and annotations. This same embedding is used in editable notes, which by simple reuse of existing node types are scrolled, multifont, annotatable. Likewise, dialog boxes that ask for input are HTML forms, whose values are processed internally rather than being sent to a server. Preferences setting will be implemented analogously. Furthermore, all buttons can be embedded in menus as opposed to requiring a parallel set of menu-specific versions as in other GUI toolkits. Through simple node reuse, long menus are scrollable.

3.1.3 Example: visual layers.

Internal nodes control the display of their subtrees. Ordinarily, nodes simply transform the coordinate space to be relative to itself. The scrolled pane node further adjusts for the current settings of vertical and horizontal scrollbars.

Visual layers are implemented simply by adding an internal node under a document's content root, and all of its children are drawn on top of document content. Dialog boxes and menus use this method. The simplest visual layer makes no further coordinate transformations, leaving its children at absolute locations on the document. Another type of node maintains its children at a fixed location on the screen by reversing the scrollbar settings. Portals and zooming could be implemented similarly.

3.2 Functionality: Behaviors

Whereas many document systems support some form of extensibility, the Multivalent system pushes this idea to the extreme. Almost everything that is not a tree node is an extension called a *behavior*. Behaviors are Java classes that participate in the communication protocols detailed in subsequent sections. Programmatically, this means that behaviors subclass the class `Behavior` and override methods corresponding to those protocols. Some behaviors happen to be packaged with the basic system, but they have no privileges over third-party behavior extensions. All user-level functionality is implemented by behaviors. The extension language is the implementation language.

Behaviors can be categorized according to primary function, although a single behavior may participate in several.

Media adaptors

> Behaviors that primarily bridge some concrete document format into the runtime document tree are known as media adaptors. For example, the UNIX manual page media adaptor reads roff source, the HTML media adaptor goes to great pains to correct the files of random bytes found on the Web into a structurally reasonable tree, and the scanned paper adaptor builds a document tree hierarchy of region, paragraph, line, word. In addition to viewing documents, media adaptors can be used for general purpose access; for example, a full-text indexer could use media adaptors to decode PDF and DVI

uniformly to and as easily as ASCII and HTML. Once bridged into the tree, the document of whatever source format enjoys the array of existing functionality; this contrasts with single-format viewers, such as the excellent IDVI [6] for TeX DVI, Ghostscript [5] for PostScript, xpdf [12] for PDF, which must recapitulate a large and growing amount of standard functionality.

Structural
: Structural behaviors modify protocols over a document subtree. Such behaviors "register interest" in a particular subtree, and subsequently each protocol invokes the behavior's corresponding methods before and after passing through the subtree rooted at that node. For example, table sorting rearranges the children of the given parent to achieve sorted order, clipboard markup generates a representation of the selected text with markup tags, and one type of search visualization hooks onto the scrollbar to paint its results on top of the scrollbar every time it is painted.

Span
: This very common behavior type of behavior extends from some offset within a start leaf linearly through leaf nodes to an offset within an end leaf. Examples of span type behaviors include font change, highlight, hyperlink, and copy editor markup.

Lenses
: Lenses, such as Magnify and Decypher, control a geometric portion of the document (described with a movable, resizable window). Lenses compose effects where they overlap, so that magnify plus Show OCR yields magnified OCR, and Show OCR plus decypher yields decyphered OCR.

Managers
: Managers provide specialized coordination among behaviors beyond that provided by the usual means of communication. For example, Lens coordination of overlapping lenses is very specialized, to compose effects when lenses overlap, and yet its coordinating manager behavior has no special privileges in the system. When a lens is made, it queries the browser-level attributes for the lens manager, spontaneously creating one if it is the first, and registers its existence. During document painting, the lens manager computes intersections and invokes the individual lenses.

Writing behaviors ranges in flexibility and difficulty. An individual behavior may be customizable with attributes in its hub. The behavior that appears in the popup menu on a word and sends that word to a dictionary or language translation service takes as attributes that title to show in the menu and the URL of the service. Most span types rely on a `SpanUI` behavior to put them in a menu (this separates functionality from user interface), and other spans could be added and their organization in the user interface rearranged. The `SemanticUI` behavior sends an arbitrary semantic event in response to invoking a menu item or button on the toolbar. A number of other behaviors are probably simple variations on existing behaviors. A demonstration "FBI Redaction" behavior, which blacks out spans of text and associates a reason code and comment, was written in two hours by starting with the hyperlink annotation behavior, changing the blue underline to black foreground and background, and changing the dialog box to ask for a comment rather than a URL.

Of course, the wholly original ideas can be satisfied only by writing a new behavior from scratch, but it seems possible to accomplish interesting things in just a few hundred lines of code. Media adaptors can reuse node types already created for flowed and fixed document types, and current media adaptors range from 167 lines for ASCII, 180 for Zip, 238 for directory listing and 260 for Perl's POD among the simple formats, to about 1000 for UNIX manual pages in the mid-range. At 4000 lines, HTML is the largest media adaptor, yet this is only 5% as large(!) as the rough equivalent in Mozilla. Spans are simpler, with most under 100 lines and the most complex (hyperlink) at 250. Lens range from 50 to 100 lines. Other behaviors range from 100 to 400 lines.

3.3 Hubs, which catalog behaviors

In effect, every document is given a custom browser. This set of behaviors to use is listed in a *hub*, which is an XML document. Hubs are loaded by the system when the system starts up and when individual documents are loaded. The system's built-in hub is loaded first, then a hub, if any, from the user's home directory, which can augment or delete behaviors given in the system hub. By editing the applicable hub, behaviors can be added, removed, rearranged (to affect their order in the user interface), replaced, and specialized (by editing attributes). For example, one could swap the Emacs editing key bindings for the Microsoft Windows ones, and enjoy Emacs editing commands in everything from URL type-ins to editable notes to text fields in HTML forms. Hubs can be compared to style sheets in that, given a document with some structure, style sheets describe how to display the document, while hubs describe how one may interact with it. However, hubs go much farther and control the construction of the entire application.

```
<?xml version='1.0' ?>
<System title="System default behaviors">

<xinclude:include href="Core.hub" />
<xinclude:include href="Net.hub" />

<MenuBar behavior='multivalent.std.ui.Menubar'

titles='File|Edit|Go|Lens|Style|Anno|CopyEd|View|Help'
/>

<MenuItem Behavior='SemanticUI'
    SCRIPT="event newBrowserInstance"
    title="New Browser" parent="File" category="File"
    TYPE="Button" />

<Events Behavior='multivalent.std.ui.EmacsBindings' />

<MenuItem Behavior='SemanticUI' SCRIPT="event EXIT"
   title="Quit" parent="File" category="Quit"
TYPE="Button"
 />

</System>
```

Above is an excerpt from a hub. At runtime a hub is converted into a *layer* consisting of a list of instantiated behaviors and a list of non-behavior data subtrees. Attributes in the hub become attributes in the runtime behaviors and similarly with the data subtrees. Thus layers can be converted between XML file and runtime representations with no loss of data. The name of the layer is given by the root of the XML parse tree. Since hubs are

written in XML, they can include by reference other hubs with `xinclude`. Behaviors are identified as such by an attribute named "behavior"; in the absence of such a tag, that subtree is added to the list of data subtrees. The name of the behavior can be given as a fully qualified classnames, such as `multivalent.std.ui.Menubar`, or as `Hyperlink`, in which case it is mapped to a full classname via a table; the map enables all instances of some popular behavior type to be conveniently updated to a new, more powerful implementation of that same class of functionality. Hierarchy within a behavior tag is left to that behavior to interpret. It can be content, as in Post-it notes where the text of the note is stored inline; or it can be nested behaviors, as in span annotations which nest a pair of Location behaviors to robustly anchor their endpoints to survive edits to the document.

Since many documents share functionality and other documents have no associated hub, the following types hubs are cascaded to produce the full set of behaviors active on a given document.

System hub
: The system hub lists behaviors applicable to all documents. It includes the basic parts of the File, Edit and Help menus; searching and search visualization; document popup menu and entries for word lookup in dictionary, language translation and other sites; key bindings; and others.

Genre hubs
: A genre hub can be based essentially on a document's MIME type, but more specific categories can be made, as for instance "UNIX manual page" is a genre type, which is more specific that its MIME type of "roff document". The genre hub for scanned paper documents includes the "Show OCR" lens, the control to change the view to OCR text-only or image-only, and the heuristic link identification behavior. Hubs can contain by reference other hub documents, and the Scansoft XDOC OCR format includes this general OCR hub that is shared with the Caere PDA OCR format.

Document-specific hubs
: Finally, a document-specific hub holds behaviors that apply to that document only. Few documents are so special that behaviors are written just for them, but document-specific hubs are used to hold annotations, for which the behavior code is common, but the application instances to the particular document as specified in attributes is unique.

The separation of document function in hubs and document data in existing concrete formats gives a number of advantages. Simple document formats can be enlivened with modern ideas (hyperlinks for ASCII), and more modern formats can be enlivened with ideas that are too new to be included, too specialized or esoteric for a general use, or simply too complex to be included in a specification meant to be implemented by a number parties of a variety of devices. Moreover, since hubs can be stored separately from content, they can be applied to servers and media that are not cooperative. For example, one can annotate a scanned page image on a web site, which itself supports just simple image display.

3.4 Low-level Communication: Restore, Build, Format, Paint, Events Protocols

In general, behaviors do not directly invoke one another's methods as this would lock out changes by other behaviors. That is, closed communication patterns would lock out behaviors with new ideas about how the system should operate. Instead, the general template for all communication is the following:

1. system or behavior initiates or requests some action,
2. all behaviors potentially interested in this action have a chance to modify or even cancel (or just record or ignore) the action, and
3. the modified action is executed.

All digital document systems share a fundamental document lifecycle: the application is loaded, the document is read in and internal data structures are built for it, the document is formatted, then it is painted on the screen, at which point the system waits for the user do something. To open this to arbitrary new behaviors, the process is reified into sharply segregated protocols: restore behaviors, build document tree, format document tree, paint document tree, low-level events (such as from the mouse and keyboard), semantic events (described next section). During the build protocol, say, all interested behaviors have the opportunity to build on the document tree data structure, modifying the work of other behaviors.

The system is a *framework* where the system is in charge of the overall flow of control. Frameworks contrast with systems in which the developer's program defines the flow of control, sometimes passing through libraries. For every protocol in the framework, the system will pass through relevant behaviors, briefly handing off the flow of control to a behavior to perform all of its work for that protocol only. Behaviors may not have any work to perform in a particular protocol, but while in one protocol it is illegal to work on another protocol.

Most protocols have *before and after phases*: the before phase of the protocol is executed in its entirety, then the after phase. With arbitrary behaviors active at any point in the document lifecycle and with potential dependencies on other behaviors, the division of protocols into phases helps sequence behaviors. The rule of thumb is that behaviors that build structures do so in *before*, so that all such activity is known to be completed by *after*, which can modify it or cancel it. For example, during build before, one behavior can load in the main body of a document, so that it will be available for annotations to hook into during build after. A behavior can short-circuit from *before* to *after* phases thus bypassing lower priority behaviors, or from *after* to end the protocol.

As appropriate to their intrinsic nature, protocols are either *round robin* or *tree based*. Round robin protocols flow through the before phase of all active behaviors from highest priority to lowest, then the after phase in reverse order. Thus the highest priority behavior is called first and last, getting the first word and the last say, as it were. The round robin protocols are Restore, Build (at the start of which the tree does not exist), Semantic Events (such as `openDocument` and `newBrowserInstance`), and Save.

Tree-based protocols proceed through a depth-first tree walk, during which tree nodes can also affect control flow. Behaviors interested in a structural portion of the tree register interest (programmatically, add themselves as observers) to the node at the head of the subtree, and during the tree walk the *before* methods of the observers are called before the node and its children are traversed, and the after methods of the observers are called after the node is done. Behaviors can short-circuit from before to after, thus bypassing that subtree; and *after* can short-circuit to cancel the remainder of the tree walk. The tree-based protocols are Format, Paint, and Low-level Events.

Below are described the low-level protocols: those performance-intensive protocols concerning construction and display of the document, and system input.

3.4.1 Restore

When a behavior instance is created, it is restored, during which it can perform initialization. Behaviors inherit the attributes from their hubs, which provides an easy means of end user customization. The menubar behavior takes the list and order of its menus from an attribute.

3.4.2 Build

The build protocol iterates through the build before methods of all behaviors, then the build after of each in reverse order. Media adaptors execute most of their work during this protocol, reading a concrete format and building its runtime manifestation as a document tree. With media adaptors working in the build before phase, annotations, which are stored separately from documents and can annotate any document format, attach themselves in the after phase, at which time the basic document tree is known to have been constructed.

3.4.3 Format

Formatting is largely left to document format-specific nodes to carry out. Generally, nodes determine how large they would like to be (width and height dimensions), and parents set the locations (x,y) of their children. Formatting occurs during a walk of the document tree, top down propagating maximum dimension constraints and property settings from style sheets, then bottom up propagating requested dimensions to be positioned.

All nodes, whether heavily medium-dependent leaves or largely medium-independent internal nodes, must respect a core set of properties during Format and Paint protocols, the full set of which is given in the table at right. Behaviors rely on a this guaranteed level of functionality across media types. For instance, the selection and the highlighting annotation rely on setting the background of a word to a given color. Most media types can simply draw the word over the colored background, but scanned paper must identify the white pixel in its image and convert it to transparent before drawing the image of that word.

Core Format and Paint Properties

foreground and background colors

font family, style, size

underline, overstrike

elide

justify

spaceabove/below

horizontal and vertical alignment

float side

margins, padding, border

signals

3.4.3.1 Example: formatting scanned paper images.

Two core properties that must be respected are space above and below the baseline, which is used to open up space between lines for short text annotations. For flowed document formats such as HTML and manual pages, this is simply another variable to consider during formatting. For scanned paper, repecting this property requires reformatting a fixed image, pushing areas that would be overlapped farther down or to the right.

3.4.4 Paint

The Paint protocol renders a formatted data structure to the screen, printer, or other device. It is very similar to Format in the management of core properties. The *before* phase can be used to draw backgrounds or set graphics transformations, and *after* can draw on top of the corresponding subtree.

All painting is performed under a *clipping region*, which at largest is the size of the visible screen. During the tree walk, only nodes that lie within the clipping region are pursued; thus, when an internal node lies outside of the clipping region, its entire subtree is ignored, and the system is able to quickly paint, with just a small amount of wasted effort, that part of a potentially long document that is visible on the screen. Lenses operate by setting the clip to just their bounds, or their intersected bounds, and repainting the entire document with certain properties set. This process is so rapid that the entire screen can be repainted in full continuously during scrolling. This is as opposed to a "bitblt" which copies that part of the document that remains onscreen but at a different location; bitblt-based scrolling is problematic in Multivalent because notes and lenses can be fixed to a point on the screen and so don't scroll with the rest of the document.

3.4.4.1 Example: Move-To annotation.

As shown in the demonstration of the Multivalent Browser, the Move-To annotation type draws an arrow from its source span to its destination point. This arrow must be drawn the document's coordinate space, which may be scrolled; it should be fast to

redraw, since the destination point is interactively chosen; and it should be efficient so that we can have 1000s of annotations that draw on a document 100s of pages long, and so only those arrows that will appear on the screen should be drawn.

The Move-To behavior accomplishes this by using a tree node function to determine the lowest node in the tree common to both start and end points, and registers interest on that node. At this time it computes the coordinates of the start and end points of the arrow relative to the common node, to be used in any number of future paintings. Now for all protocols the behavior receives notification of activity on the node; in this case, after the content of the subtree has been painted, which is to say in the *after* phase, the behavior draws the arrow using the precomputed coordinates. Since the lowest common node frequently extends beyond the visible screen, parts of the arrow are at times painted uselessly, but the total amount of wasted work is contained within a relatively small portion of the document. During interactive setting of the arrow, the behavior quickly unregisters from the last lowest common node, computes the new node, registers on that, computes coordinates, and repaints the screen.

3.4.5 Low-level Events

Low-level events, such as keystrokes, mouse clicks, and OS window activity (close, iconify) are propagated through the tree, its path winnowed by the event's coordinates if any, where they can be seized by behaviors that have registered interest in that part of the tree.

3.4.5.1 Examples: spans, document popup, disabled menu items.

Span behaviors receive events by virtue of being anchored to leaves. Hyperlinks of course translate a mouse click into a request to load a new page. The document popup menu behavior seizes a button 3-down event, in the *after* phase in order to give any other behavior priority, and then creates the menu according to behaviors active at the cursor point. The disabling of menu items is not built into menus or menu items, but is added as a behavior. A structural behavior registers itself on each disabled menu item and short-circuits events on that subtree so mouse movement cannot select that menu item. (It also participates in Paint *after*, graying out the appearance of the menu item by drawing over the content spaced lines the same color as the background.) This same method could implement guards for widgets with dangerous effect: click once to remove the guard, click again to execute.

3.5 High-level Communication: Semantic Events

The low-level protocols open document manifestation and interaction with the user. They do not address higher-level logical or semantic actions, such as opening documents and sorting tables, but the spirit of the system demands that, no less than low-level actions, these high-level actions must be open to modification by any behavior.

As a high level action, semantic events are relatively infrequent and thus not performance critical. They are sent to all behaviors, with *before* and *after* phases. A semantic event consists of a *message*, such as `openDocument`, and three fields labeled *argument*, *in*, and *out*, whose exact content types depend on the message, though the rule of thumb is that argument corresponds to the most commonly needed auxiliary data, in holds the sender of the event, and out collects results from participating behaviors.

Semantic events are most often *sent* to request action and to announce state. Even when a behavior could directly invoke itself, it is often better to request the action and give other behaviors a chance to modify or cancel the request. Other semantic events announce potentially interesting state. Semantic events are most often *acted upon* by behaviors to implement a requested action, to modify the event, or to update state in response to an annoucement event.

For example, when a document has been loaded, surviving potential 404 File Not Found's and redirections, the system announces `openedDocument` (note past tense), and the forward/backward behavior adds it to history list. The search behavior announces its results in a `searchHits` semantic event, which is caught by the scrollbar visualization behavior. Media adaptors for HTML, directory listings, and Zip archive listings all send the table sort behavior, with `sortTable` as the message, and node and direction in fields. The table sorting behavior catches the event, inspects table to determine the data type of the requested column, sorts, and rearranges the document tree to reflect the sort order.

3.5.1 Example: TeX DVI.

TeX's DVI is a page description language like PostScript, but very simple: it has movement commands, registers, font changes, character drawing, and little else. Moreover, its author froze the format. Everything from hyperlinks to images to graphics drawing is implemented by means of "specials", which are embedded strings that are left to the viewer to interpret. There are probably hundreds of specials that have been defined, and different viewers implement different, usually small subsets. Supporting a new special involves hacking the source code, if the source code is available, and hoping the viewer's main author accepts the changes.

The Multivalent DVI media adaptor supports specials without modification of the parser-renderer with the same mechanism as used in the rest of the system, the behavior. The parser announces specials as semantic events, passing the special string, and geometric and logical (document tree) positions in fields. Behaviors implementing specials listen for the relevant messages, and have enough information in the fields to accomplish their work. The HyperTeX hypertext and PageSize specials are currently supported, and image and color will be.

3.5.2 Example: menu construction.

All behaviors have a chance to contribute to all menus, whether they drop down from the menu bar or pop up from document content, hyperlinks, or the selection. Menus are built on demand. At the moment the menubar behavior needs to build a menu, it sends a semantic event with message `createWidget/`*`menu-title`*, and seeds the out field with an empty menu. Arbitrary behaviors can add to the menu, and when the event returns to the menubar, the menubar formats and paints the resulting menu.

3.6 Ad Hoc Communication

The protocols described above involve all behaviors in ways that they all understand, even if that understanding results in the behavior knowingly ignoring a protocol. Protocols satisfy the communication needs of the great majority of behaviors.

Various ad hoc communication involves smaller groups of behaviors. The communication is still open so that arbitrary behaviors can participate, but behaviors must know how to communicate in a language socially agreed upon by concerned behaviors outside of core definitions. Out-of-protocol communication can be made by leaving state in the document tree (in attributes and global variables), by manager behaviors, and by defining further protocols. As well, although all behaviors are presented with semantic events, they typically examine the message to determine whether the event is interesting to them, and since messages are simple strings, it's easy for groups of behaviors to coordinate on a new string, thus in effect spontaneously creating a new ad hoc communication on top of a well known transport.

3.6.1 Example: multipage.

Several document formats are paginated, including scanned paper, DVI, and PDF. Multipage documents share a set of behaviors that provide GUI controls for navigating pages and save and restore annotations between individual pages and a single file on disk.

Multipage behaviors define two attributes held in the document root, `PAGECNT` that holds the total number of pages in a document and `PAGE` that holds the current page number. Media adaptors interpret their specific medium and report the total number of pages, and during the build protocol they inquire for the value of `PAGE` and construct that page. Page numbers are given as positive integers; media adaptors may have to map a more complex numbering scheme, but the page navigation controls is guaranteed that the current page number plus one gives the next page.

Simply by referring to a particular behavior in their hub, multipage document formats can take advantage of per-page annotations that are saved to a common file. All document formats can be annotated with an open-ended set of annotation types, none of which is built in. Neither is the saving and restoring of annotations built in. An annotation manager behavior waits for a `closedDocument` semantic event, at which point it collects annotations writes them to disk, and restores them on a subsequent `openedDocument`. Multipage documents are more complicated since it was desirable to save annotations from all pages in a single file, the better to distribute the set as a unit. Multipage behaviors have socially defined an extension to `openedDocument/closeDocument` for pages; the multipage controls send `openedDocumentPage` and `closeDocumentPage`, and the save/restore annotation behavior responds to these. The pagewise annotation manager takes advantage of state in the Layer object, and stores annotations for pages other than the current page in data trees. This in turn is a state known by the wipe annotations behavior, so it can not only clear annotations for the current page as it can for other documents, but also enables it to wipe all annotations in the document by clearing the layer's data trees.

4 RELATED WORK

Additional related work is considered in [13].

4.1 Mozilla and Other Open Source Projects

As compared to proprietary web browsers, the Mozilla browser [9] seems to offer inventors an inviting playground in which to execute their ideas. As an Open Source project, the browser's source code is available for arbitrary changes. Likewise, Open Source viewers can be found for most but probably not all other document formats. Several researchers have taken this path, extending NCSA Mosaic [11] to experiment with annotations in Stanford's ComMentor [16] and advanced style sheets with Proteus [8], and others have taken advantage of W3C's experimental browser Amaya [18] for constraint-based style sheets [1].

But the situation is not ideal. In the first place, retrofitting a feature into the browser for one document format still neglects all the other document formats one works with regularly. Moveover, in the case of Mozilla, the system is enormous: the current version as this writing (v0.9.1) comprises 3817 C++ files and over 1.6 million lines of code (compared to Multivalent's 287 files and 54,000 lines of code). While it is modularized, there is nevertheless a significant amount to master before one can work on the new idea — which was the reason for working with the system in the first place.

When it comes to the essential point of distribution, Open Source systems are not necessarily an improvement over closed source. Open Source proponents claim that one can simply redistribute the source with changes, and this can be done. In practice, however, such systems are evolving and one would want to track new versions, but revising one's changes to match new mainline source changes is tedious at best. One can contribute the changes back to the main project developers, but they may not be accepted, especially if they are of limited applicability or if they are large and thus hard for the main developers to maintain.

4.2 Document Object Model

The World Wide Web Consortium has defined a Document Object Model (DOM) [19] for the runtime programmatic manipulation of XML and HTML document trees. It similar to the Multivalent document tree in that both have the standard set of tree definitions, and navigation and manipulation functions. DOM continues to evolve and the latest "level" (level 3) defines a model of event propagation, in which "bubble" and "capture" correspond remarkably closely to before and after phases of the low-level events protocols.

At least for the foreseeable future, various strengths of the Multivalent model seem to lie outside of the scope of DOM:

- Behaviors can define new node types, which is crucial for support of non-HTML documents, such as scanned paper and PDF. Since GUI widgets are simply node types, behaviors can cleanly introduce new widgets.
- DOM defines the tree, views, style, events, load and save. The Multivalent Model covers — at a less sharply defined detail — the other parts of what it calls the fundamental document lifecycle: build, format, paint, semantic events.

- DOM is language neutral. In practice, the language is JavaScript, whose sole virtue is that it is the only way to script web pages. As a programming language, Java is vastly better designed.
- JavaScript can be associated with individual pages, whereas one wants some functions to operate on all pages or all pages of some genre. One could conceive of running through a proxy server that spliced arbitrary JavaScript into all pages, but aside from a decided lack of aesthetic appeal, that's awkward and introduces new security risks.
- Since it is difficult to package JavaScript from different sites, it is not surprising that there is little thought given to composing the scripts and mediating conflicts, a situation which Multivalent has addressed with several mechanisms.

5 FUTURE WORK

We plan to finish HTML 3.2 and CSS1 support in the immediate future. We will integrate the Java Media Framework (JMF) [17] for QuickTime and Flash support and Netscape's Rhino [10] JavaScript-in-Java implementation, to complete the Web browsing aspects of the platform.

We plan to take advantage of media adaptors for manual pages, PDF, and other document formats to provide access to text for use in constructing a full-text index, probably implemented by the search engine Lucene [4].

We eagerly welcome third party developers.

6 ACKNOWLEDGEMENT

This research was supported by the Digital Libraries Initiative under grant NSF CA98-17353.

7 REFERENCES

[1] Greg Badros, Alan Borning, Kim Marriott, and Peter Stuckey. "Constraint Cascading Style Sheets for the Web", *Proceedings of the 1999 ACM Symposium on User Interface Software and Technology*.

[2] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton and Tony D. DeRose. Toolglass and Magic Lenses: The See-Through Interface. *Proceedings of SIGGRAPH '93*, Anaheim, California, pages 73-80.

[3] Kenneth P. Brooks. A Two-view Document Editor with User-definable Document Structure, Digital Systems Research Center Technical Report 33, 1988.

[4] Doug Cutting. Lucene, http://www.lucene.com/.

[5] L. Peter Deutsch. Ghostscript, http://www.cs.wisc.edu/~ghost/.

[6] Garth Dickie. IDVI, http://www.geom.umn.edu/java/idvi.

[7] Michael A. Hiltzik. Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age, HarperCollins, 1999.

[8] Philip M. Marden, Jr. and Ethan V. Munson. Multiple Presentations of WWW Documents Using Style Sheets, *Proceedings of NPIV 97, the Workshop on New Paradigms in Information Visualization and Manipulation*, Las Vegas, November 1997.

[9] Mozilla.org. Mozilla, http://www.mozilla.org/

[10] Mozilla.org. Rhino: JavaScript for Java, http://www.mozilla.org/rhino/.

[11] National Center for Supercomputing Applications, Mosaic, http://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/help-about.html

[12] Derek B. Noonburg. xpdf, http://www.foolabs.com/xpdf/

[13] Thomas A. Phelps. "Multivalent Documents: Anytime, Anywhere, Any Type, Every Way User-Improvable Digital Documents and Systems", Ph.D. Dissertation (1998).

[14] Thomas A. Phelps and Robert Wilensky. "Multivalent Annotations", Proceedings of First European Conference on Research and Advanced Technology for Digital Libraries (1997).

[15] Thomas A. Phelps and Robert Wilensky. "Robust Intra-document Locations", *Proceedings of the Ninth World Wide Web Conference*, 15-18 May 2000, Amsterdam.

[16] Martin Roscheisen, Christian Mogensen and Terry Winograd. Beond Browsing: Shared Comments, SOAPs, Trails, and On-line Communities. *Proceedings of the Third World Wide Web Conference: Technology, Tools and Applications*, April 1995, Darmstadt, Germany.

[17] Sun Microsystems. Java Media Framework, http://java.sun.com/products/java-media/jmf/.

[18] World Wide Web Consortium (Irene Vatton et alia). Amaya, http://www.w3.org/Amaya/.

[19] World Wide Web Consortium. Document Object Model, http://www.w3.org/DOM/.